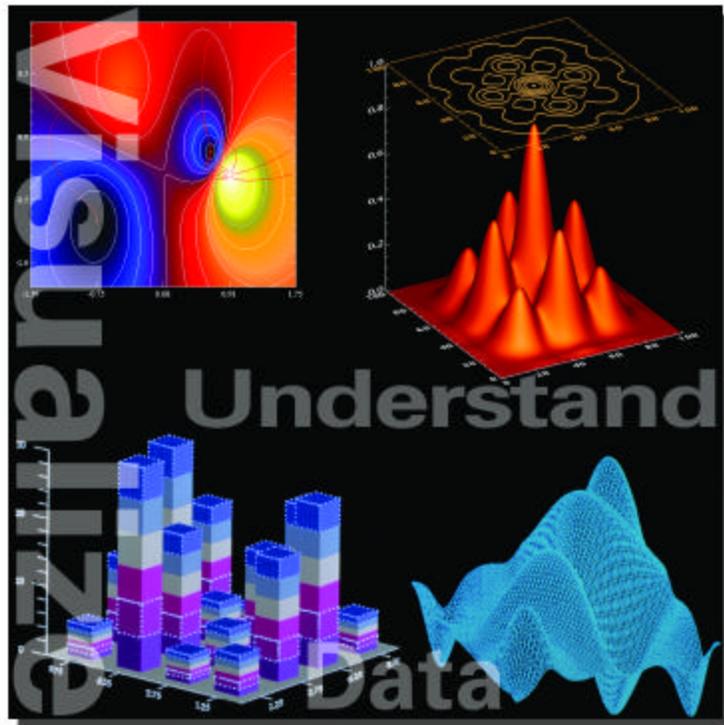




# P V - W A V E 7 . 5<sup>®</sup>



Reference V1, V2, and V3

HELPING CUSTOMERS **SOLVE** COMPLEX PROBLEMS

## Visual Numerics, Inc.

---

Visual Numerics, Inc.  
2500 Wilcrest Drive  
Suite 200  
Houston, Texas 77042-2579  
United States of America  
713-784-3131  
800-222-4675  
(FAX) 713-781-9260  
<http://www.vni.com>  
e-mail: [info@boulder.vni.com](mailto:info@boulder.vni.com)

Visual Numerics, Inc.  
7/F, #510, Sect. 5  
Chung Hsiao E. Rd.  
Taipei, Taiwan 110 ROC  
+886-2-727-2255  
(FAX) +886-2-727-6798  
e-mail: [info@vni.com.tw](mailto:info@vni.com.tw)

Visual Numerics S.A. de C.V.  
Cerrada de Berna 3, Tercer Piso  
Col. Juarez  
Mexico, D.F. C.P. 06600  
Mexico

Visual Numerics, Inc. (France) S.A.R.L.  
Tour Europe  
33 place des Corolles  
Cedex 07  
92049 PARIS LA DEFENSE  
FRANCE  
+33-1-46-93-94-20  
(FAX) +33-1-46-93-94-39  
e-mail: [info@vni-paris.fr](mailto:info@vni-paris.fr)

Visual Numerics International GmbH  
Zettachring 10  
D-70567 Stuttgart  
GERMANY  
+49-711-13287-0  
(FAX) +49-711-13287-99  
e-mail: [info@visual-numeric.de](mailto:info@visual-numeric.de)

Visual Numerics, Inc., Korea  
Rm. 801, Hanshin Bldg.  
136-1, Mapo-dong, Mapo-gu  
Seoul 121-050  
Korea

Visual Numerics International, Ltd.  
Suite 1  
Centennial Court  
East Hampstead Road  
Bracknell, Berkshire  
RG 12 1 YQ  
UNITED KINGDOM  
+01-344-458-700  
(FAX) +01-344-458-748  
e-mail: [info@vniuk.co.uk](mailto:info@vniuk.co.uk)

Visual Numerics Japan, Inc.  
Gobancho Hikari Building, 4th Floor  
14 Gobancho  
Chiyoda-Ku, Tokyo, 102  
JAPAN  
+81-3-5211-7760  
(FAX) +81-3-5211-7769  
e-mail: [vda-spirt@vnij.co.jp](mailto:vda-spirt@vnij.co.jp)

---

© 1990-2001 by Visual Numerics, Inc. An unpublished work. All rights reserved. Printed in the USA.

Information contained in this documentation is subject to change without notice.

IMSL, PV-WAVE, Visual Numerics and PV-WAVE Advantage are either trademarks or registered trademarks of Visual Numerics, Inc. in the United States and other countries.

The following are trademarks or registered trademarks of their respective owners: Microsoft, Windows, Windows 95, Windows NT, Fortran PowerStation, Excel, Microsoft Access, FoxPro, Visual C, Visual C++ — Microsoft Corporation; Motif — The Open Systems Foundation, Inc.; PostScript — Adobe Systems, Inc.; UNIX — X/Open Company, Limited; X Window System, X11 — Massachusetts Institute of Technology; RISC System/6000 and IBM — International Business Machines Corporation; Java, Sun — Sun Microsystems, Inc.; HPGL and PCL — Hewlett Packard Corporation; DEC, VAX, VMS, OpenVMS — Compaq Computer Corporation; Tektronix 4510 Rasterizer — Tektronix, Inc.; IRIX, TIFF — Silicon Graphics, Inc.; ORACLE — Oracle Corporation; SPARCstation — SPARC International, licensed exclusively to Sun Microsystems, Inc.; SYBASE — Sybase, Inc.; HyperHelp — Bristol Technology, Inc.; dBase — Borland International, Inc.; MIFF — E.I. du Pont de Nemours and Company; JPEG — Independent JPEG Group; PNG — Aladdin Enterprises; XWD — X Consortium. Other product names and companies mentioned herein may be the trademarks of their respective owners.

**IMPORTANT NOTICE: Use of this document is subject to the terms and conditions of a Visual Numerics Software License Agreement, including, without limitation, the Limited Warranty and Limitation of Liability.** If you do not accept the terms of the license agreement, you may not use this documentation and should promptly return the product for a full refund. Do not make illegal copies of this documentation. No part of this documentation may be stored in a retrieval system, reproduced or transmitted in any form or by any means without the express written consent of Visual Numerics, unless expressly permitted by applicable law.

## **The Visualizaton Toolkit**

Copyright (c) 1993-1995 Ken Martin, Will Schroeder, Bill Lorensen.

This software is copyrighted by Ken Martin, Will Schroeder and Bill Lorensen. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files. This copyright specifically does not apply to the related textbook "The Visualization Toolkit" ISBN 013199837-4 published by Prentice Hall which is covered by its own copyright.

The authors hereby grant permission to use, copy, and distribute this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. Additionally, the authors grant permission to modify this software and its documentation for any purpose, provided that such modifications are not distributed without the explicit consent of the authors and that existing copyright notices are retained in all copies. Some of the algorithms implemented by this software are patented, observe all applicable patent law.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

---

---

# ***Table of Contents***

## ***Preface*** [i](#)

- What's in this Manual [i](#)
- Conventions Used in this Manual [iii](#)
- Technical Support [iv](#)

## ***Chapter 1: Functional Summary of Routines*** [1](#)

- 3D Visualization Toolkit (VTK) Routines [4](#)
- Array Creation Routines [5](#)
- Array Manipulation Routines [6](#)
- Color Table Manipulation Routines [8](#)
- Concurrent Processing Routines [9](#)
- Coordinate Conversion Routines [9](#)
- Data Connection Routines [9](#)
- Data Conversion Routines [10](#)
- Data Extraction Routines [11](#)
- Date/Time Functions [11](#)
- File Manipulation Routines [12](#)
- General Graphics Routines [13](#)
- General Mathematical Functions [14](#)
- Gridding Routines [16](#)
- HDF Routines [17](#)
- Help and Information Routines [18](#)
- Hypertext Markup Language (HTML) Routines [18](#)
- Image Display Routines [19](#)
- Image IO Routines [20](#)
- Image Processing Routines [20](#)
- Input and Output Routines [22](#)
- Interpolation Routines [23](#)

Mapping Routines	24
Operating System Access Routines	24
Optimization and Regression Routines	25
Plotting Routines	26
Polygon Generation Routines	27
Polygon Manipulation Routines	28
Polygon Rendering Routines	28
Programming Routines	28
Ray Tracing Routines	30
Session Routines	30
Special Mathematical Functions	30
String Processing Routines	30
Table Manipulation Functions	31
Transcendental Mathematical Functions	32
VDA Tools Routines	32
VDA Tools Manager Routines	33
VDA Tools Manager Graphical Element Routines	35
VDA Utilities Routines	36
View Setup Routines	38
Virtual Reality Modeling Language (VRML) Routines	39
Volume Manipulation Routines	39
Volume Rendering Routines	40
WAVE Widgets Routines	40
WAVE Widget Utilities	42
Widget Toolbox Routines	42
Window Routines	44

## ***Chapter 2: Procedure and Function Reference*** 45

ABS Function	46
ACOS Function	47
ADD_EXEC_ON_SELECT Procedure (UNIX)	48
ADDVAR Procedure	49

<b>AFFINE Function</b>	<b>50</b>
<b>ALOG Function</b>	<b>51</b>
<b>ALOG10 Function</b>	<b>52</b>
<b>ASARR Function</b>	<b>53</b>
<b>ASIN Function</b>	<b>55</b>
<b>ASKEYS Function</b>	<b>56</b>
<b>ASSOC Function</b>	<b>58</b>
<b>ATAN Function</b>	<b>60</b>
<b>AVG Function</b>	<b>61</b>
<b>AXIS Procedure</b>	<b>63</b>
<b>BAR2D Procedure</b>	<b>73</b>
<b>BAR3D Procedure</b>	<b>75</b>
<b>BESELI Function</b>	<b>77</b>
<b>BESELJ Function</b>	<b>78</b>
<b>BESELY Function</b>	<b>80</b>
<b>BILINEAR Function</b>	<b>82</b>
<b>BINDGEN Function</b>	<b>85</b>
<b>BLOB Function</b>	<b>86</b>
<b>BLOBCOUNT Function</b>	<b>87</b>
<b>BOUNDARY Function</b>	<b>88</b>
<b>BREAKPOINT Procedure</b>	<b>89</b>
<b>BUILDRESOURCEFILENAME Function</b>	<b>90</b>
<b>BUILD_TABLE Function</b>	<b>93</b>
<b>BYTARR Function</b>	<b>96</b>
<b>BYTE Function</b>	<b>97</b>
<b>BYTEORDER Procedure</b>	<b>100</b>
<b>BYTSCL Function</b>	<b>102</b>
<b>CD Procedure</b>	<b>107</b>
<b>C_EDIT Procedure</b>	<b>109</b>
<b>CENTER_VIEW Procedure</b>	<b>113</b>
<b>CHEBYSHEV Function</b>	<b>115</b>
<b>CHECKFILE Function</b>	<b>116</b>

**CHECK\_MATH Function 118**  
**CINDGEN Function 121**  
**CLOSE Procedure 122**  
**COLOR\_CONVERT Procedure 123**  
**COLOR\_EDIT Procedure 124**  
**COLOR\_PALETTE Procedure 129**  
**COMPILE Procedure 131**  
**COMPLEX Function 135**  
**COMPLEXARR Function 138**  
**CONE Function 139**  
**CONGRID Function 140**  
**CONJ Function 142**  
**CONTOUR Procedure 144**  
**CONTOUR2 Procedure 147**  
**CONTOURFILL Procedure 151**  
**CONVERT\_COORD Function 156**  
**CONV\_FROM\_RECT Function 158**  
**CONVOL Function 160**  
**CONV\_TO\_RECT Function 163**  
**CORRELATE Function 166**  
**COS Function 168**  
**COSH Function 169**  
**COSINES Function 170**  
**CPROD Function 171**  
**CREATE \_ HOLIDAYS Procedure 171**  
**CREATE \_ WEEKENDS Procedure 172**  
**CROSSP Function 174**  
**CURSOR Procedure 175**  
**CURVEFIT Function 179**  
**CYLINDER Function 181**  
**DAY\_OF\_WEEK Function 184**  
**DAY\_OF\_YEAR Function 185**

<b>DBLARR Function</b>	<b>186</b>
<b>DCINDGEN Function</b>	<b>187</b>
<b>DCOMPLEX Function</b>	<b>188</b>
<b>DCOMPLEXARR Function</b>	<b>190</b>
<b>DC_ERROR_MSG Function</b>	<b>191</b>
<b>DC_OPTIONS Function</b>	<b>193</b>
<b>DC_READ_8_BIT Function</b>	<b>194</b>
<b>DC_READ_24_BIT Function</b>	<b>196</b>
<b>DC_READ_CONTAINER Function</b>	<b>199</b>
<b>DC_READ_DIB Function (Windows)</b>	<b>201</b>
<b>DC_READ_FIXED Function</b>	<b>203</b>
<b>DC_READ_FREE Function</b>	<b>218</b>
<b>DC_READ_TIFF Function</b>	<b>231</b>
<b>DC_SCAN_CONTAINER Function</b>	<b>235</b>
<b>DC_WRITE_8_BIT Function</b>	<b>237</b>
<b>DC_WRITE_24_BIT Function</b>	<b>238</b>
<b>DC_WRITE_DIB Function (Windows)</b>	<b>240</b>
<b>DC_WRITE_FIXED Function</b>	<b>242</b>
<b>DC_WRITE_FREE Function</b>	<b>250</b>
<b>DC_WRITE_TIFF Function</b>	<b>256</b>
<b>DEFINE_KEY Procedure</b>	<b>259</b>
<b>DEFROI Function</b>	<b>266</b>
<b>DEFSYSV Procedure</b>	<b>269</b>
<b>DELETE_SYMBOL Procedure (OpenVMS)</b>	<b>271</b>
<b>DEL_FILE Procedure</b>	<b>272</b>
<b>DELFUNC Procedure</b>	<b>273</b>
<b>DELLOG Procedure (OpenVMS)</b>	<b>274</b>
<b>DELPROC Procedure</b>	<b>275</b>
<b>DELSTRUCT Procedure</b>	<b>276</b>
<b>DELVAR Procedure</b>	<b>277</b>
<b>DERIV Function</b>	<b>279</b>
<b>DERIVN Function</b>	<b>281</b>

**DETERM Function 282**  
**DEVICE Procedure 283**  
**DIAG Function 284**  
**DICM\_TAG\_INFO Function 285**  
**DIGITAL\_FILTER Function 286**  
**DILATE Function 289**  
**DINDGEN Function 293**  
**DIST Function 294**  
**DOC\_LIBRARY Procedure (UNIX/OpenVMS) 297**  
**DOUBLE Function 300**  
**DROP\_EXEC\_ON\_SELECT Procedure (UNIX) 302**  
**DT\_ADD Function 303**  
**DT\_COMPRESS Function 304**  
**DT\_DURATION Function 308**  
**DTGEN Function 309**  
**DT\_PRINT Procedure 311**  
**DT\_SUBTRACT Function 312**  
**DT\_TO\_SEC Function 314**  
**DT\_TO\_STR Procedure 316**  
**DT\_TO\_VAR Procedure 319**  
**ENVIRONMENT Function (UNIX/Windows) 322**  
**EOF Function 323**  
**ERASE Procedure 324**  
**ERODE Function 326**  
**ERRORF Function 330**  
**ERRPLOT Procedure 331**  
**EUCLIDEAN Function 333**  
**EXEC\_ON\_SELECT Procedure (UNIX) 334**  
**EXECUTE Function 337**  
**EXIT Procedure 340**  
**EXP Function 341**  
**EXPAND Function 342**

<b>EXPON Function</b>	<b>343</b>
<b>EXTREMA Function</b>	<b>344</b>
<b>FAST_GRID2 Function</b>	<b>346</b>
<b>FAST_GRID3 Function</b>	<b>348</b>
<b>FAST_GRID4 Function</b>	<b>351</b>
<b>FFT Function</b>	<b>353</b>
<b>FILEPATH Function</b>	<b>356</b>
<b>FINDFILE Function</b>	<b>358</b>
<b>FINDGEN Function</b>	<b>359</b>
<b>FINITE Function</b>	<b>360</b>
<b>FIX Function</b>	<b>362</b>
<b>FLOAT Function</b>	<b>364</b>
<b>FLTARR Function</b>	<b>366</b>
<b>FLUSH Procedure</b>	<b>367</b>
<b>FREE_LUN Procedure</b>	<b>368</b>
<b>FSTAT Function</b>	<b>369</b>
<b>FUNCT Procedure</b>	<b>372</b>
<b>GAUSSFIT Function</b>	<b>375</b>
<b>GAUSSINT Function</b>	<b>376</b>
<b>GCD Function</b>	<b>377</b>
<b>GETENV Function</b>	<b>378</b>
<b>GET_KBRD Function</b>	<b>379</b>
<b>GET_LUN Procedure</b>	<b>380</b>
<b>GETNCERR Function</b>	<b>382</b>
<b>GETNCOPTS Function</b>	<b>383</b>
<b>GET_SYMBOL Function (OpenVMS)</b>	<b>385</b>
<b>GREAT_INT Function</b>	<b>386</b>
<b>GRID Function</b>	<b>387</b>
<b>GRIDN Function</b>	<b>388</b>
<b>GRID_2D Function</b>	<b>389</b>
<b>GRID_3D Function</b>	<b>391</b>
<b>GRID_4D Function</b>	<b>393</b>

**GRID\_SPHERE Function 396**  
**GROUP\_BY Function 399**  
**HANNING Function 405**  
**HDFGET24 Function 407**  
**HDFGETANN Function 409**  
**HDFGETFILEANN Function 410**  
**HDFGETNT Function 412**  
**HDFGETR8 Function 414**  
**HDFGETRANGE Function 416**  
**HDFGETSDS Function 417**  
**HDFLCT Procedure 419**  
**HDFPUT24 Function 420**  
**HDFPUTFILEANN Function 422**  
**HDFPUTR8 Function 423**  
**HDFPUTSDS Function 425**  
**HDFSCAN Procedure 427**  
**HDFSETNT Function 428**  
**HDF\_TEST Procedure 429**  
**HELP Procedure 430**  
**HILBERT Function 432**  
**HIST\_EQUAL Function 434**  
**HIST\_EQUAL\_CT Procedure 437**  
**HISTN Function 438**  
**HISTOGRAM Function 440**  
**HLS Procedure 448**  
**HSV Procedure 450**  
**HSV\_TO\_RGB Procedure 451**  
**HTML\_BLOCK Procedure 453**  
**HTML\_CLOSE Procedure 454**  
**HTML\_HEADING Procedure 455**  
**HTML\_HIGHLIGHT Function 456**  
**HTML\_IMAGE Function 457**

HTML_LINK Function	459
HTML_LIST Procedure	460
HTML_OPEN Procedure	463
HTML_PARAGRAPH Procedure	464
HTML_RULE Procedure	466
HTML_SAFE Function	466
HTML_TABLE Procedure	468
HTML_TEXT Procedure	470
IMAGE_COLOR_QUANT Function	474
IMAGE_CONT Procedure	477
IMAGE_CREATE Function	479
IMAGE_DISPLAY Procedure	486
IMAGE_QUERY_FILE Function	488
IMAGE_READ Function	492
IMAGE_WRITE Function	495
IMAGINARY Function	499
IMG_TRUE8 Procedure	500
INDEX_AND Function	502
INDEX_CONV Function	503
INDEX_OR Function	504
INDGEN Function	505
INFO Procedure	506
INTARR Function	509
INTERPOL Function	510
INTERPOLATE Function	513
INTRP Function	513
INVERT Function	514
ISASKEY Function	515
ISHFT Function	516
JACOBIAN Function	518
JOURNAL Procedure	518
JUL_TO_DT Function	519

**KEYWORD\_SET Function 520**  
**LCM Function 522**  
**LEEFILT Function 523**  
**LEGEND Procedure 524**  
**LINDGEN Function 526**  
**LINKNLOAD Function 527**  
**LIST Function 533**  
**LISTARR Function 534**  
**LN03 Procedure (UNIX/OpenVMS) 535**  
**LOADCT Procedure 535**  
**LOADCT\_CUSTOM Procedure 537**  
**LOAD\_HOLIDAYS Procedure 538**  
**LOAD\_OPTION Procedure 539**  
**LOADRESOURCES Procedure 540**  
**LOADSTRINGS Procedure 542**  
**LOAD\_WEEKENDS Procedure 545**  
**LONARR Function 546**  
**LONG Function 547**  
**LUBKSB Procedure 549**  
**LUDCMP Procedure 551**  
**MAP Procedure 556**  
**MAP\_CONTOUR Procedure 565**  
**MAP\_PLOTS Procedure 567**  
**MAP\_POLYFILL Procedure 569**  
**MAP\_REVERSE Procedure 571**  
**MAP\_VELOECT Procedure 572**  
**MAP\_XYOUTS Procedure 574**  
**MAX Function 575**  
**MEDIAN Function 577**  
**MESH Function 580**  
**MESSAGE Procedure 582**  
**MIN Function 584**

MINIMIZE Function	586
MODIFYCT Procedure	587
MOLEC Function	588
MOMENT Function	589
MONTH_NAME Function	590
MOVIE Procedure	591
MPROVE Procedure	593
MSWORD_CGM_SETUP Procedure	594
NAVIGATOR Procedure	595
NEIGHBORS Function	596
N_ELEMENTS Function	597
NINT Function	598
NORMALS Function	600
N_PARAMS Function	601
N_TAGS Function	602
ON_ERROR_GOTO Procedure	613
ON_IOERROR Procedure	614
OPEN Procedures (UNIX/OpenVMS)	615
OPEN Procedures (Windows)	621
OPENURL Procedure	624
OPLOT Procedure	626
OPLOTERR Procedure	628
OPTION_IS_LOADED Function	630
ORDER_BY Function	631
PALETTE Procedure	635
PARAM_PRESENT Function	638
PARSEFILENAME Procedure	640
PIE Procedure	641
PIE_CHART Procedure	646
PLOT Procedures	651
PLOTERR Procedure	657
PLOT_FIELD Procedure	659

**PLOT\_HISTOGRAM Procedure 662**  
**PLOTS Procedure 664**  
**PM Procedure 667**  
**PMF Procedure 669**  
**POINT\_LUN Procedure 670**  
**POLY Function 672**  
**POLY\_2D Function 673**  
**POLY\_AREA Function 676**  
**POLY\_C\_CONV Function 677**  
**POLY\_COUNT Function 679**  
**POLY\_DEV Function 680**  
**POLYFILL Procedure 683**  
**POLYFILLV Function 690**  
**POLY\_FIT Function 691**  
**POLYFITW Function 694**  
**POLY\_MERGE Procedure 696**  
**POLY\_NORM Function 697**  
**POLY\_PLOT Procedure 699**  
**POLYSHADE Function 702**  
**POLY\_SPHERE Procedure 705**  
**POLY\_SURF Procedure 708**  
**POLY\_TRANS Function 709**  
**POLYWARP Procedure 710**  
**POPD Procedure 713**  
**PRIME Function 715**  
**PRINT Procedures 716**  
**PRINTD Procedure 717**  
**PRODUCT Function 718**  
**PROFILE Function 718**  
**PROFILES Procedure 720**  
**PROMPT Procedure 722**  
**PSEUDO Procedure 723**

<b>PUSHD Procedure</b>	<b>724</b>
<b>QUIT Procedure</b>	<b>734</b>
<b>RANDOMU Function</b>	<b>737</b>
<b>RDPIX Procedure</b>	<b>738</b>
<b>READ Procedures</b>	<b>739</b>
<b>READ_XBM Procedure</b>	<b>742</b>
<b>REBIN Function</b>	<b>743</b>
<b>REFORM Function</b>	<b>745</b>
<b>REGRESS Function</b>	<b>747</b>
<b>RENAME Procedure</b>	<b>749</b>
<b>RENDER Function</b>	<b>752</b>
<b>RENDER24 Function</b>	<b>754</b>
<b>REPLICATE Function</b>	<b>756</b>
<b>REPLV Function</b>	<b>757</b>
<b>RESAMP Function</b>	<b>758</b>
<b>RESTORE Procedure</b>	<b>759</b>
<b>RETALL Procedure</b>	<b>760</b>
<b>RETURN Procedure</b>	<b>761</b>
<b>REVERSE Function</b>	<b>762</b>
<b>REWIND Procedure (OpenVMS)</b>	<b>764</b>
<b>RGB_TO_HSV Procedure</b>	<b>764</b>
<b>RM Procedure</b>	<b>765</b>
<b>RMF Procedure</b>	<b>768</b>
<b>ROBERTS Function</b>	<b>769</b>
<b>ROT Function</b>	<b>772</b>
<b>ROTATE Function</b>	<b>775</b>
<b>ROT_INT Function</b>	<b>777</b>
<b>SAVE Procedure</b>	<b>781</b>
<b>SCALE3D Procedure</b>	<b>782</b>
<b>SEC_TO_DT Function</b>	<b>783</b>
<b>SELECT_READ_LUN Procedure (UNIX)</b>	<b>785</b>
<b>SETDEMO Procedure</b>	<b>786</b>

SETENV Procedure (UNIX/Windows) 788  
SETLOG Procedure (OpenVMS) 789  
SETNCOPTS Procedure 790  
SET\_PLOT Procedure 791  
SET\_SCREEN Procedure 793  
SET\_SHADING Procedure 795  
SET\_SYMBOL Procedure (OpenVMS) 798  
SETUP\_KEYS Procedure 798  
SET\_VIEW3D Procedure 800  
SET\_VIEWPORT Procedure 801  
SET\_XY Procedure 803  
SGN Function 805  
SHADE\_SURF Procedure 806  
SHADE\_SURF\_IRR Procedure 812  
SHADE\_VOLUME Procedure 815  
SHIF Function 818  
SHIFT Function 819  
SHOW3 Procedure 823  
SHOW\_OPTIONS Procedure 825  
SIGMA Function 827  
SIN Function 829  
SINDGEN Function 830  
SINH Function 831  
SIZE Function 832  
SKIPF Procedure (OpenVMS) 834  
SLICE Function 835  
SLICE\_VOL Function 835  
SMALL\_INT Function 837  
SMOOTH Function 838  
SOBEL Function 841  
SOCKET\_ACCEPT Function 844  
SOCKET\_CLOSE Procedure 846

<b>SOCKET_CONNECT Function</b>	<b>847</b>
<b>SOCKET_GETPORT Function</b>	<b>848</b>
<b>SOCKET_INIT Function</b>	<b>849</b>
<b>SOCKET_READ Function</b>	<b>851</b>
<b>SOCKET_WRITE Procedure</b>	<b>852</b>
<b>SORT Function</b>	<b>854</b>
<b>SPAWN Procedure (UNIX/OpenVMS)</b>	<b>855</b>
<b>SORTN Function</b>	<b>858</b>
<b>SPAWN Procedure (Windows)</b>	<b>859</b>
<b>SPHERE Function</b>	<b>860</b>
<b>SPLINE Function</b>	<b>862</b>
<b>SQRT Function</b>	<b>864</b>
<b>STDEV Function</b>	<b>865</b>
<b>STOP Procedure</b>	<b>867</b>
<b>STRARR Function</b>	<b>868</b>
<b>STRCOMPRESS Function</b>	<b>869</b>
<b>STRETCH Procedure</b>	<b>870</b>
<b>STRING Function</b>	<b>872</b>
<b>STRJOIN Function</b>	<b>875</b>
<b>STRLEN Function</b>	<b>876</b>
<b>STRLOOKUP Function</b>	<b>877</b>
<b>STRLOWCASE Function</b>	<b>879</b>
<b>STRMATCH Function</b>	<b>881</b>
<b>STRMESSAGE Function</b>	<b>884</b>
<b>STRMID Function</b>	<b>886</b>
<b>STRPOS Function</b>	<b>887</b>
<b>STRPUT Procedure</b>	<b>889</b>
<b>STRSPLIT Function</b>	<b>890</b>
<b>STRSUBST Function</b>	<b>892</b>
<b>STR_TO_DT Function</b>	<b>894</b>
<b>STRTRIM Function</b>	<b>896</b>
<b>STRUCTREF Function</b>	<b>898</b>

**STRUPCASE Function 899**  
**SUM Function 901**  
**SURFACE Procedure 902**  
**SURFACE\_FIT Function 905**  
**SURFR Procedure 907**  
**SVBKS Procedure 910**  
**SVD Procedure 911**  
**SVDFIT Function 913**  
**SYSTIME Function 915**  
**TAG\_NAMES Function 920**  
**TAN Function 921**  
**TANH Function 922**  
**TAPRD Procedure (OpenVMS) 923**  
**TAPWRT Procedure (OpenVMS) 924**  
**TEK\_COLOR Procedure 925**  
**TENSOR Functions 927**  
**THREED Procedure 929**  
**TODAY Function 930**  
**TOTAL Function 931**  
**TQLI Procedure 934**  
**TRANSPOSE Function 936**  
**TRED2 Procedure 938**  
**TRIDAG Procedure 939**  
**TRNLOG Function (OpenVMS) 941**  
**TV Procedure 943**  
**TVCRS Procedure 947**  
**TVLCT Procedure 949**  
**TVRD Function 951**  
**TVSCL Procedure 952**  
**TVSIZE Procedure 956**  
**UNIQUE Function 960**  
**UNIX\_LISTEN Function (UNIX Only) 962**

UNIX_REPLY Function (UNIX Only)	963
UNLOAD_OPTION Procedure	964
UPVAR Procedure	965
USERSYM Procedure	967
USGS_NAMES Function	969
VAR_MATCH Function	971
VAR_TO_DT Function	973
VECTOR_FIELD3 Procedure	975
VEL Procedure	978
VELOVECT Procedure	982
VIEWER Procedure	985
VOL_MARKER Procedure	993
VOL_PAD Function	995
VOL_REND Function	996
VOL_TRANS Function	999
VOLUME Function	1000
VRML_AXIS Procedure	1002
VRML_CAMERA Procedure	1004
VRML_CLOSE Procedure	1005
VRML_CONE Procedure	1006
VRML_CUBE Procedure	1009
VRML_CYLINDER Procedure	1011
VRML_LIGHT Procedure	1014
VRML_LINE Procedure	1015
VRML_OPEN Procedure	1017
VRML_POLY Procedure	1018
VRML_SPHERE Procedure	1020
VRML_SPOTLIGHT Procedure	1022
VRML_SURFACE Procedure	1024
VRML_TEXT Procedure	1025
vtkADDATTRIBUTE Procedure	1028
vtkAXES Procedure	1029

**vtkCAMERA Procedure 1031**  
**vtkCLOSE Procedure 1032**  
**vtkCOLORBAR Procedure 1033**  
**vtkCOMMAND Procedure 1034**  
**vtkERASE Procedure 1035**  
**vtkGRID Procedure 1036**  
**vtkHEDGEHOG Procedure 1037**  
**vtkINIT Procedure 1039**  
**vtkLIGHT Procedure 1040**  
**vtkPLOTS Procedure 1041**  
**vtkPOLYDATA Procedure 1043**  
**vtkPOLYSHADE Procedure 1044**  
**vtkPPMREAD Function 1046**  
**vtkPPMWRITE Procedure 1047**  
**vtkRECTILINEARGRID Procedure 1048**  
**vtkRENDERWINDOW Procedure 1049**  
**vtkSCATTER Procedure 1050**  
**vtkSLICEVOL Procedure 1053**  
**vtkSTRUCTUREDGRID Procedure 1055**  
**vtkSTRUCTUREDPOINTS Procedure 1056**  
**vtkSURFACE Procedure 1057**  
**vtkSURFGEN Procedure 1060**  
**vtkTEXT Procedure 1061**  
**vtkTVRD Function 1062**  
**vtkUNSTRUCTUREDGRID Procedure 1063**  
**vtkWDELETE Procedure 1064**  
**vtkWINDOW Procedure 1065**  
**vtkWRITEVRML Procedure 1067**  
**vtkWSET Procedure 1068**  
**WCOPY Function (Windows) 1070**  
**WDELETE Procedure 1072**  
**WEOF Procedure 1073**

WgAnimateTool Procedure	1073
WgCbarTool Procedure	1079
WgCeditTool Procedure	1083
WgCtTool Procedure	1092
WgIsoSurfTool Procedure	1096
WgMovieTool Procedure	1102
WgOrbit Procedure	1108
WgSimageTool Procedure	1109
WgSliceTool Procedure	1113
WgStripTool Procedure	1119
WgSurfaceTool Procedure	1124
WgTextTool Procedure	1130
WHERE Function	1133
WHEREIN Function	1135
WIN32_PICK_FONT Function	1136
WIN32_PICK_PRINTER Function	1137
WINDOW Procedure	1138
WMENU Function (UNIX/OpenVMS)	1143
WPASTE Function (Windows)	1145
WPRINT Procedure (Windows)	1146
WREAD_DIB Function (Windows)	1148
WREAD_META Function (Windows)	1149
WRITEU Procedure	1151
WRITE_XBM Procedure	1152
WSET Procedure	1153
WSHOW Procedure	1155
WWRITE_DIB Function (Windows)	1156
WWRITE_META Function (Windows)	1157
WzAnimate Procedure	1159
WzBar Procedure	1160
WzBar3D Procedure	1162
WzColorEdit Procedure	1164

WzContour Procedure [1167](#)  
WzExport Procedure [1168](#)  
WzHistogram Procedure [1170](#)  
WzImage Procedure [1172](#)  
WzImport Procedure [1174](#)  
WzMultiView Procedure [1176](#)  
WzPie Procedure [1177](#)  
WzPlot Procedure [1179](#)  
WzPreview Procedure [1180](#)  
WzSurface Procedure [1182](#)  
WzTable Procedure [1184](#)  
WzVariable Procedure [1186](#)  
ZOOM Procedure [1192](#)  
ZROOTS Procedure [1194](#)

### ***Chapter 3: Graphics and Plotting Keywords*** [1197](#)

### ***Chapter 4: System Variables*** [1233](#)

### ***Chapter 5: Software Character Sets*** [1255](#)

### ***Chapter 6: Special Characters*** [1265](#)

### ***Chapter 7: Executive Commands*** [1267](#)

Using Executive Commands [1267](#)

### ***Appendix A: The PV-WAVE HDF Interface*** [A-1](#)

What is the PV-WAVE HDF Interface? [A-1](#)

Example Programs Are Available [A-2](#)

Using the PV-WAVE HDF Functions [A-3](#)

PV-WAVE HDF Base Function Interface [A-6](#)

### ***Appendix B: Output Devices and Window Systems*** [B-1](#)

Window System Features	B-2
CGM Output	B-4
HPGL Output	B-8
PCL Output	B-14
Pixel Map Output	B-17
PostScript Output	B-19
Regis Output	B-34
Tektronix Terminals	B-36
WIN32 Driver	B-39
WMF Driver	B-53
X Window System	B-58
Z-buffer Output	B-86

***Reference Index*** 1



# Preface

The *PV-WAVE Reference* describes the PV-WAVE functions and procedures, keywords, system variables, fonts, special characters, executive commands, and device drivers.

---

## ***What's in this Manual***

**Chapter 1, *Functional Summary of Routines*** — A listing of PV-WAVE functions and procedures arranged into functional groups, such as image processing routines, input/output routines, programming routines, and string processing routines. The basic syntax for each routine is also shown.

**Chapter 2, *Procedure and Function Reference*** — An alphabetically arranged reference for all PV-WAVE procedures and functions. Most descriptions include one or more examples and cross references to related information.

**Chapter 3, *Graphics and Plotting Keywords*** — Describes the keywords that can be used with the graphics and plotting system routines.

**Chapter 4, *System Variables*** — Describes each of the system variables.

**Chapter 5, *Software Character Sets*** — Shows the software character sets provided by PV-WAVE.

**Chapter 6, *Special Characters*** — Describes characters with special interpretation and their function in PV-WAVE.

**Chapter 7, *Executive Commands*** — Describes each of the PV-WAVE executive commands.

**Appendix A, *The PV-WAVE HDF Interface*** — Discusses how to access HDF base and convenience functions from within PV-WAVE.

**Appendix B, *Output Devices and Window Systems*** — Explains how to use the standard graphic output devices and window systems.

**Reference Index** — A subject index with hypertext links to information contained in the Reference.

---

## Conventions Used in this Manual

You will find the following conventions used throughout this manual:

- Code examples appear in this typeface. For example:

```
PLOT, temp, s02, Title = 'Air Quality'
```

- Code comments are shown in this typeface, immediately below the commands they describe. For example:

```
PLOT, temp, s02, Title = 'Air Quality'  
; This command plots air temperature data vs. sulphur  
; dioxide concentration.
```

- Variables are shown in lowercase italics (*myvar*), function and procedure names are shown in uppercase (XYOUTS), keywords are shown in mixed case italic (*XTitle*), and system variables are shown in regular mixed case type (!Version). For better readability, all GUI development routines are shown in mixed case (WwMainMenu).
- A \$ at the end of a line of PV-WAVE code indicates that the current statement is continued on the following line. By convention, use of the continuation character (\$) in this document reflects its syntactically correct use in PV-WAVE. This means, for instance, that *strings* are never split onto two lines without the addition of the string concatenation operator (+). For example, the following lines would produce an error if entered literally in PV-WAVE.

```
WAVE> PLOT, x, y, Title = 'Average $  
Air Temperatures by Two-Hour Periods'  
; Note that the string is split onto two lines; an error  
; message is displayed if you enter a string this way.
```

The correct way to enter these lines is:

```
WAVE> PLOT, x, y, Title = 'Average ' + $  
'Air Temperatures by Two-Hour Periods'  
; This is the correct way to split a string onto two  
; command lines.
```

- Reserved words, such as FOR, IF, CASE, are always shown in uppercase.

---

## **Technical Support**

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

<b>Office Location</b>	<b>Phone Number</b>
Corporate Headquarters Houston, Texas	713-784-3131
Boulder, Colorado	303-939-8920
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-344-458-700

---

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

## **FAX and E-mail Inquiries**

Contact Visual Numerics Technical Support staff by sending a FAX to:

<b>Office Location</b>	<b>FAX Number</b>
Corporate Headquarters	713-781-9260
Boulder, Colorado	303-245-5301
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-4873
Taiwan	+886-2-727-6798
United Kingdom	+44-1-344-458-748

---

or by sending E-mail to:

<b>Office Location</b>	<b>E-mail Address</b>
Boulder, Colorado	support@boulder.vni.com
France	support@vni-paris.fr
Germany	support@visual-numerics.de
Japan	vda-sprt@vnij.co.jp
Korea	support@vni.co.kr
Taiwan	support@vni.com.tw
United Kingdom	support@vniuk.co.uk

---

## Electronic Services

<b>Service</b>	<b>Address</b>
General e-mail	info@boulder.vni.com
Support e-mail	support@boulder.vni.com
World Wide Web	http://www.vni.com
Anonymous FTP	ftp.boulder.vni.com
FTP Using URL	ftp://ftp.boulder.vni.com/VNI/
<b>PV-WAVE</b> Mailing List:	Majordomo@boulder.vni.com
To subscribe include:	subscribe pv-wave YourEmailAddress
To post messages	pv-wave@boulder.vni.com

---

# ***Functional Summary of Routines***

This chapter lists the following groups of related routines:

## ***Arrays***

*Array Creation Routines* on page 5

*Array Manipulation Routines* on page 6

*Interpolation Routines* on page 23

## ***Graphics and Plotting***

*Coordinate Conversion Routines* on page 9

*General Graphics Routines* on page 13

*Gridding Routines* on page 16

*Mapping Routines* on page 24

*Plotting Routines* on page 26

*View Setup Routines* on page 38

*Window Routines* on page 44

*VDA Tools Routines* on page 32

## **GUI Development**

[BUILDDRESOURCEFILENAME](#) *Function*

[Concurrent Processing Routines](#) on page 9

[LOADRESOURCES](#) *Procedure*

[LOADSTRINGS](#) *Procedure*

[WAVE](#) *Widget Utilities* on page 42 (Wg)

---

**NOTE** For more detailed information on the following routines, please refer to the *PV-WAVE Application Developer's Guide*.

---

[VDA Tools Manager Routines](#) on page 33 (Tm)

[VDA Tools Manager Graphical Element Routines](#) on page 35 (Tm)

[VDA Utilities Routines](#) on page 36 (Wo)

[WAVE Widgets Routines](#) on page 40 (Ww)

[Widget Toolbox Routines](#) on page 42 (Wt)

## **Image Processing and Color**

[Color Table Manipulation Routines](#) on page 8

[Image Display Routines](#) on page 19

[Image IO Routines](#) on page 20

[Image Processing Routines](#) on page 20

## **Input/Output**

[Data Connection Routines](#) on page 9

[HDF Routines](#) on page 17

[Input and Output Routines](#) on page 22

## **Internet Enabling (Wave On Web)**

[Hypertext Markup Language \(HTML\) Routines](#) on page 18

[OPENURL](#) *Procedure* on page 624

[Virtual Reality Modeling Language \(VRML\) Routines](#) on page 39

## **Mathematics**

*General Mathematical Functions* on page 14

*Optimization and Regression Routines* on page 25

*Special Mathematical Functions* on page 30

*Transcendental Mathematical Functions* on page 32

## **Programming**

*Data Conversion Routines* on page 10

*Data Extraction Routines* on page 11

*Date/Time Functions* on page 11

*File Manipulation Routines* on page 12

*Help and Information Routines* on page 18

*Operating System Access Routines* on page 24

*Programming Routines* on page 28

*Session Routines* on page 30

*String Processing Routines* on page 30

*Table Manipulation Functions* on page 31

## **Rendering Techniques**

*3D Visualization Toolkit (VTK) Routines* on page 4

*Polygon Generation Routines* on page 27

*Polygon Manipulation Routines* on page 28

*Polygon Rendering Routines* on page 28

*Ray Tracing Routines* on page 30

*Volume Manipulation Routines* on page 39

*Volume Rendering Routines* on page 40

---

## 3D Visualization Toolkit (VTK) Routines

**vtkADDATTRIBUTE**, attributes

Collects point attributes for VTK datasets.

**vtkAXES**

Creates a set of axes.

**vtkCAMERA**

Changes the camera's parameters.

**vtkCOLORBAR**

Adds a color bar legend to a VTK scene using the current PV-WAVE color table.

**vtkCLOSE**

Closes the VTK process.

**vtkCOMMAND**, command

Sends Tcl and VTK commands to the Tcl process.

**vtkERASE** [, background\_color]

Erases the contents of the current VTK window.

**vtkGRID** [, Number=n]

Adds 3D grid lines to a VTK scene.

**vtkHEDGEHOG**, points, vectors, scalars

Creates a HedgeHog (vector) plot.

**vtkINIT**

Initializes the VTK system.

**vtkLIGHT**

Adds a light to a VTK window.

**vtkPLOTS**, points

Adds a polyline.

**vtkPOLYDATA**, points

Passes vertex/polygon lists, lines, points, and triangles to VTK.

**vtkPOLYSHADE**, vertices, polygons

Renders a polygon object.

**vtkPPMREAD** (filename)

Reads a PPM file.

**vtkPPMWRITE** [, window\_index]

Writes the contents of a VTK window to a PPM file.

**vtkRECTILINEARGRID**, Dimensions

Passes data describing a Rectilinear Grid to VTK.

**vtkRENDERWINDOW** [, window\_index]

Renders a VTK window.

**vtkSCATTER**, points

Renders 3D points.

**vtkSLICEVOL**, v, [sx=sx, sy=sy, sz=sz,  
xc=xc, yc=yc, zc=zc]

Creates a sliced 3D volume at specific x, y, z locations.

**vtkSTRUCTUREDGRID**, Dimensions,  
Points

Passes data describing a structured grid to VTK.

**vtkSTRUCTUREDPOINTS**, Dimensions

Passes data describing structured points to VTK.

**vtkSURFACE**, z [,x] [,y]

Renders a surface.

**vtkSURFGEN**, points

Generates a 3D surface from sampled points assumed to lie on a surface.

**vtkTEXT**, string

Adds a text string.

**vtkTVRD** ([window\_index])

Returns the contents of a VTK window as a bitmapped image.

**vtkUNSTRUCTUREDGRID**, Points, Cells,  
Cell\_types

Passes data describing an unstructured grid to VTK.

**vtkWDELETE** [, window\_index]

Closes a VTK window, however it does not shut down the Tcl process.

**vtkWINDOW** [, window\_index]

Creates a VTK window.

**vtkWRITEVRML**, filename [,

WindowID=id, Speed=s]

Creates a Virtual Reality Modeling Language file (VRML .wrl file) from a scene in a VTK window.

**vtkWSET** [, window\_index]

Sets the active VTK window.

---

## **Array Creation Routines**

**ASARR**(key<sub>1</sub>, value<sub>1</sub>, ... key<sub>n</sub>, value<sub>n</sub>)

**ASARR**(keys\_arr, values\_list)

Creates an associative array containing specified variables and expressions.

**BINDGEN** (dim<sub>1</sub> [, dim<sub>2</sub>, ... , dim<sub>n</sub>] )

Returns a byte array with the specified dimensions, setting the contents of the result to increasing numbers starting at 0.

**BYTARR** (dim<sub>1</sub> [, dim<sub>2</sub>, ... , dim<sub>n</sub>] )

Returns a byte vector or array.

**CINDGEN** (dim<sub>1</sub> [, dim<sub>2</sub>, ... , dim<sub>n</sub>] )

Returns a complex single-precision floating-point array.

**COMPLEXARR** (dim<sub>1</sub> [, dim<sub>2</sub>, ... , dim<sub>n</sub>] )

Returns a complex single-precision floating-point vector or array.

**DBLARR** (dim<sub>1</sub>, ... , dim<sub>n</sub>)

Returns a double-precision floating-point vector or array.

**DCINDGEN**(dim1 [, dim2 , ... , dimn])

Returns a double-precision floating-point complex array.

**DCOMPLEXARR** (dim<sub>1</sub> [, dim<sub>2</sub>, ... , dim<sub>n</sub>] )

Returns a double-precision floating-point complex vector or array.

**DIAG**( a )

Makes a diagonal array or extracts the diagonal of an array.

**DINDGEN** (dim<sub>1</sub>, ..., dim<sub>n</sub>)

Returns a double-precision floating-point array with the specified dimensions.

**FINDGEN** (dim<sub>1</sub>, ..., dim<sub>n</sub>)

Returns a single-precision floating-point array with the specified dimensions.

**FLTARR** (dim<sub>1</sub>, ..., dim<sub>n</sub>)

Returns a single-precision floating-point vector or array.

**INDGEN** (dim<sub>1</sub>, ... , dim<sub>n</sub>)

Returns an integer array with the specified dimensions.

**INTARR** (dim<sub>1</sub>, ... , dim<sub>n</sub>)

Returns an integer vector or array.

**LINDGEN** (dim<sub>1</sub>, ... , dim<sub>n</sub>)

Returns a longword integer array with the specified dimensions.

**LIST**(expr<sub>1</sub>, ... expr<sub>n</sub>)

Creates a list array.

**LISTARR**(number\_elements,[value])

Returns a list.

**LONARR** (dim<sub>1</sub>, ... , dim<sub>n</sub>)

Returns a longword integer vector or array.

**MAKE\_ARRAY** ([dim<sub>1</sub>,... , dim<sub>n</sub>] )

Returns an array of specified type, dimensions, and initialization. It provides the ability to create an array dynamically whose characteristics are not known until run time.

**REPLICATE** (value, dim<sub>1</sub>, ..., dim<sub>n</sub>)

Forms an array with the given dimensions, filled with the specified scalar value.

**SINDGEN** (dim<sub>1</sub>, ... , dim<sub>n</sub>)

Returns a string array with the specified dimensions.

**STRARR** (dim<sub>1</sub>, ... , dim<sub>n</sub>)

Returns a string array.

---

## Array Manipulation Routines

### AFFINE(*a, b, [c]*)

Applies an affine transformation to an array.

### ASKEYS(*asarr*)

Obtains the key names for a given associative array.

### AVG (*array [, dim]* )

Standard Library function that returns the average value of an array. Optionally, it can return the average value of one dimension of an array.

### BILINEAR (*array, x, y*)

Standard Library function that creates an array containing values calculated using a bilinear interpolation to solve for requested points interior to an input grid specified by the input array.

### BLOB (*a, i, b*)

Isolates a homogeneous region in an array.

### BLOBCOUNT (*a, b*)

Counts homogeneous regions in an array.

### BOUNDARY (*a,r*)

Computes the boundary of a region in an array.

### CORRELATE (*x, y*)

Standard Library function that calculates a simple correlation coefficient for two arrays.

### CPROD(*a*)

Returns the Cartesian product of some arrays.

### CURVATURES (*s*)

Standard Library function that computes curvatures on a parametrically defined surface.

### DICM\_TAG\_INFO (*filename, image*)

Extracts Digital Imaging and Communications in Medicine (DICOM) tags information from an image associative array.

### DETERM (*array*)

Standard Library function that calculates the determinant of a square, two-dimensional input variable.

### DERIVN(*a, n*)

Differentiates a function represented by an array.

### EUCLIDEAN (*j*)

Standard Library function that transforms the Euclidean metric for a Jacobian  $j = \text{Jacobian}(f)$

### EXPAND(*a, d, i*)

Expands an array into higher dimensions.

### EXTREMA(*array*)

Finds the local extrema in an array.

### HISTN(*d [, axes]*)

Computes an n dimensional histogram.

### HISTOGRAM (*array*)

Returns the density function of an array.

### INDEX\_AND(*array<sub>1</sub>, array<sub>2</sub>*)

Computes the logical AND for two vectors of positive integers.

### INDEX\_CONV(*a, i* )

Converts one-dimensional indices to n-dimensional indices, or n-dimensional indices to 1D indices.

### INDEX\_OR(*array<sub>1</sub>, array<sub>2</sub>*)

Computes the logical OR for two vectors of positive integers.

### INTRP(*a, n, x*)

Interpolates an array along one of its dimensions.

### ISASKEY(*asarr, key*)

Matches a key name in a given associative array.

### JACOBIAN (*f*)

Standard Library function that computes the Jacobian of a function represented by  $n$  m-dimensional arrays

**MAX** (array [, max\_subscript] )

Returns the value of the largest element in an input array.

**MEDIAN** (array [, width] )

Finds the median value of an array, or applies a one- or two- dimensional median filter of a specified width to an array.

**MIN** (array [, min\_subscript] )

Returns the value of the smallest element in array.

**MOMENT**(a, i)

Computes moments of an array.

**NEIGHBORS**(a, i)

Finds the neighbors of specified array elements.

**NORMALS** (j)

Standard Library function that computes unit normals on a parametrically defined surface.

**PADIT**( a, [b] )

Pads an array with variable thickness.

**PRODUCT**(array)

Returns the product of all elements in an array.

**REBIN** (array, dim<sub>1</sub>, ..., dim<sub>n</sub>)

Returns a vector or array resized to the given dimensions.

**REFORM** (array, dim<sub>1</sub>, ... , dim<sub>n</sub>)

Reformats an array without changing its values numerically.

**REPLV**(*vector*, *dim\_vector*, *dim*)

Replicates a vector into an array.

**RESAMP**(*array*, *dim<sub>1</sub>*, ..., *dim<sub>n</sub>*)

Resamples an array to new dimensions.

**REVERSE** (array, dimension)

Standard Library function that reverses a vector or array for a given dimension.

**ROTATE** (array, direction)

Returns a rotated and/or transposed copy of the input array.

**SAME**(*x*, *y*)

Tests if two variables are the same.

**SHIF**(*array*, *dimension*, *shift\_amount*)

Shifts an array along one of its dimensions.

**SHIFT** (array, shift<sub>1</sub>, ... , shift<sub>n</sub>)

Shifts the elements of a vector or array along any dimension by any number of elements.

**SIGMA** (array [, npar, dim] )

Standard Library function that calculates the standard deviation value of an array.

**SLICE**(array, dimension, indices)

Subsets an array along one of its dimensions.

**SMOOTH** (array, width)

Smooths an array with a boxcar average of a specified width.

**SORT** (array)

Sorts the contents of an array.

**SORTN**(a)

Sorts an array of n-tuples.

**STDEV** (array [, mean] )

Standard Library function that computes the standard deviation and (optionally) the mean of the input array.

**TENSOR\_\* Functions**

Compute the generalized tensor product of two arrays.

**TOTAL** (array)

Sums the elements of an input array.

**TRANSPOSE** (array)

Transposes the input array.

**UNIQN**( a )

Finds the unique n-tuples from a set of n-tuples.

**WHERE** (array\_expr [, count ] )

Returns a longword vector containing the one-dimensional subscripts of the nonzero elements of the input array.

## WHEREIN( a, b [,c] )

Find the indices into an array where the values occur in a second array; keywords yield intersection, union, and complement.

---

## Color Table Manipulation Routines

### C\_EDIT [, colors\_out]

Standard Library procedure that lets you interactively create a new color table based on the HLS or HSV color system.

### COLOR\_EDIT [, colors\_out]

Standard Library procedure that lets you interactively create color tables based on the HLS or HSV color system.

### COLOR\_PALETTE

Standard Library procedure that displays the current color table colors and their associated color table indices.

### HLS, ltlo, lthi, stlo, sthi, hue, lp [, rgb]

Standard Library procedure that generates and loads color tables into an image display device based on the HLS color system. The resulting color table is loaded into the display system.

### HSV, vlo, vhi, stlo, sthi, hue, lp [, rgb]

Standard Library procedure that generates and loads color tables into an image display device based on the HSV color system. The final color table is loaded into the display device.

### LOADCT [, table\_number]

Standard Library procedure that loads a predefined color table.

### MODIFYCT, table, name, red, green, blue

Standard Library procedure that lets you replace one of the PV-WAVE color tables (defined in the colors.tbl file) with a new color table.

### PALETTE [, colors\_out]

Standard Library procedure that lets you interactively create a new color table based on the RGB color system.

**PSEUDO**, ltlo, lthi, stlo, sthi, hue, lp [, rgb]

Standard Library procedure that creates a pseudo color table based on the Hue, Lightness, Saturation (HLS) color system.

**STRETCH**, low, high

Standard Library procedure that linearly expands the range of the color table currently loaded to cover an arbitrary range of pixel values.

**TVLCT**, v1, v2, v3 [, start]

Loads the display color translation tables from the specified variables.

**WgCbarTool** [, parent [, shell [, windowid  
[, movedCallback], [, range]]]]

Creates a simple color bar that can be used to view and interactively shift a color table.

**WgCeditTool** [, parent [, shell ]]

Creates a full-featured set of menus and widgets enclosed in a window; this window allows you to edit the values in PV-WAVE color tables in many different ways.

**WgCtTool** [, parent [, shell ]]

Creates a simple widget that can be used interactively to modify a PV-WAVE color table.

---

## **Concurrent Processing Routines**

**ADD\_EXEC\_ON\_SELECT**, lun, command

Adds a single new item to the EXEC\_ON\_SELECT list.

**DROP\_EXEC\_ON\_SELECT**, lun

Drops a single item from the EXEC\_ON\_SELECT list.

**EXEC\_ON\_SELECT**, luns, commands

Registers callback procedures on input for a vector of logical unit numbers (LUNs).

**SELECT\_READ\_LUN**, luns

Waits for input on any list of logical unit numbers.

---

## **Coordinate Conversion Routines**

**CONVERT\_COORD** (points) OR (x [, y, z])

Converts coordinates from one coordinate system to another.

**CONV\_FROM\_RECT** (vec1, vec2, vec3)

Converts rectangular coordinates (points) to polar, cylindrical, or spherical coordinates.

**CONV\_TO\_RECT** (vec1, vec2, vec3)

Converts polar, cylindrical, or spherical coordinates to rectangular coordinates (points).

**POLY\_DEV** (points, winx, winy)

Returns a list of 3D points converted from normal coordinates to device coordinates.

**POLY\_NORM** (points)

Returns a list of 3D points converted from data coordinates to normal coordinates.

**POLY\_TRANS** (points, trans)

Returns a list of 3D points transformed by a 4-by-4 transformation matrix.

---

## **Data Connection Routines**

**DC\_ERROR\_MSG** (status)

Returns the text string associated with the negative status code generated by a "DC" data import/export function that does not complete successfully.

**DC\_OPTIONS** (msg\_level)

Sets the error message reporting level for all "DC" import/export functions.

**DC\_READ\_8\_BIT** (filename, imgarr)

Reads an 8-bit image file.

**DC\_READ\_24\_BIT** (filename, imgarr)

Reads a 24-bit image file.

**DC\_READ\_CONTAINER** (filename, var\_name)  
 Reads a single variable from an HP VEE Container file.

**DC\_READ\_DIB**(filename, imgarr)  
 Reads data from a Device Independent Bitmap (DIB) format file into a variable.

**DC\_READ\_FIXED** (filename, var\_list)  
 Reads fixed-formatted ASCII data using a PV-WAVE format that you specify.

**DC\_READ\_FREE** (filename, var\_list)  
 Reads freely-formatted ASCII files.

**DC\_READ\_TIFF** (filename, imgarr)  
 Reads a Tag Image File Format (TIFF) file.

**DC\_SCAN\_CONTAINER** (filename, num\_variables, start\_records, end\_records)  
 Scans an HP VEE Container file to determine the number and location of defined variables.

**DC\_WRITE\_8\_BIT** (filename, imgarr)  
 Writes 8-bit image data to a file.

**DC\_WRITE\_24\_BIT** (filename, imgarr)  
 Writes 24-bit image data to a file.

**DC\_WRITE\_DIB**(filename, imgarr)  
 Writes image data from a variable to a Device Independent Bitmap (DIB) format file.

**DC\_WRITE\_FIXED** (filename, var\_list, format)  
 Writes the contents of one or more PV-WAVE variables (in ASCII fixed format) to a file using a format that you specify.

**DC\_WRITE\_FREE** (filename, var\_list)  
 Writes the contents of one or more PV-WAVE variables to a file in ASCII free format.

**DC\_WRITE\_TIFF** (filename, imgarr)  
 Writes image data to a file using the Tag Image File Format (TIFF) format.

---

## **Data Conversion Routines**

**BYTE** (expr)  
 Converts an expression to byte data type.

**BYTSCL** (array)  
 Scales and converts an array to byte data type.

**COMPLEX** (real [, imaginary] )  
 Converts an expression to complex data type.

**DCOMPLEX** (expr, offset, dim1 [, dim2, ..., dimn ] )  
 Converts an expression to double-precision complex data type.

**DOUBLE** (expr)  
 Converts an expression to double-precision floating-point data type.

**FIX** (expr)  
 Converts an expression to integer data type.

**FLOAT** (expr)  
 Converts an expression to single-precision floating-point data type.

**LONG** (expr)  
 Converts an expression to longword integer data type.

**NINT** (x)  
 Converts input to the nearest integer.

**STRING** (expr<sub>1</sub>, ... , expr<sub>n</sub>)  
 Converts the input parameters to characters and returns a string expression.

---

## Data Extraction Routines

**BYTE** (expr, offset [, dim1, ... , dimn] )

Extracts data from an expression and places it in a byte scalar or array.

**COMPLEX** (expr, offset, dim1  
[, dim2, ... , dimn ] )

Extracts data from an expression and places it in a complex scalar or array.

**DCOMPLEX** (expr, offset, dim1  
[, dim2, ... , dimn ] )

Extracts data from an expression and places it in a complex scalar or array.

**DOUBLE** (expr, offset, dim1 [, ..., dimn ] )

Extracts data from an expression and places it in a double- precision floating-point scalar or array.

**FIX** (expr, offset, dim1 [, ..., dimn ] )

Extracts data from an expression and places it in a integer scalar or array.

**FLOAT** (expr, offset, dim1 [, ..., dimn ] )

Extracts data from an expression and places it in a single- precision floating-point scalar or array.

**LONG** (expr, offset, dim1 [, ... , dimn ] )

Extracts data from an expression and places it in a longword integer scalar or array.

---

## Date/Time Functions

**CREATE\_HOLIDAYS**, dt\_list

Creates the system variable !Holiday\_List.

**CREATE\_WEEKENDS**, day\_names

Creates the system variable !Weekend\_List.

**DAY\_NAME** (dt\_var)

Returns a string array containing the name of the day of the week for each day in a Date/ Time variable.

**DAY\_OF\_WEEK** (dt\_var)

Returns an array of integers containing the day of the week for each date in a Date/Time variable.

**DAY\_OF\_YEAR** (dt\_var)

Returns an array of integers containing the day of the year for each date in a Date/Time variable.

**DT\_ADD** (dt\_value)

Increment the values in a Date/Time variable by a specified amount.

**DT\_COMPRESS** (dt\_array)

Removes holidays and weekends from the Julian day portion of Date/Time variables.

**DT\_DURATION** (dt\_value\_1, dt\_value\_2)

Determines the elapsed time between two Date/Time variables.

**DT\_PRINT**, dt\_var

Prints the values of PV-WAVE Date/Time variables in a readable manner.

**DT\_SUBTRACT** (dt\_value)

Decrements the values in a Date/Time variable by a specified amount.

**DT\_TO\_SEC** (dt\_value)

Converts PV-WAVE Date/Time variables to double-precision variables containing the number of seconds elapsed from a base date.

**DT\_TO\_STR**, dt\_var, [, dates] [, times]

Converts PV-WAVE Date/Time variables into string data.

**DT\_TO\_VAR**, dt\_value

Converts a PV-WAVE Date/Time variable to regular numerical data.

**DTGEN** (dt\_start, dimension)

Returns an array of PV-WAVE Date/Time variables beginning from a specified date and incremented by a specified amount.

**JUL\_TO\_DT** (julian\_day)

Converts a Julian day number to a PV-WAVE Date/Time variable.

**LOAD\_HOLIDAYS**

Passes the value of the !Holiday\_List system variable to the Date/Time routines.

**LOAD\_WEEKENDS**

Passes the value of the !Weekend\_List system variable to the Date/Time routines.

**MONTH\_NAME** (dt\_var)

Returns a string or array of strings containing the names of the months contained in a Date/Time variable.

**SEC\_TO\_DT** (num\_of\_seconds)

Converts any number of seconds into PV-WAVE Date/Time variables.

**STR\_TO\_DT** (date\_strings [, time\_strings] )

Converts date and time string data to PV-WAVE Date/Time variables.

**TODAY** ()

Returns a Date/Time variable containing the current system date and time.

**VAR\_TO\_DT** (yyyy, mm, dd, hh, mn, ss)

Converts scalars or arrays of scalars representing dates and times into PV-WAVE Date/Time variables.

---

## **File Manipulation Routines**

**CLOSE** [, unit<sub>1</sub>, ... , unit<sub>n</sub>]

Closes the specified file units.

**EOF** (unit)

Tests the specified file unit for the end-of-file condition.

**FINDFILE** (file\_specification)

Returns a string array containing the names of all files matching a specified file description.

**FLUSH**, unit<sub>1</sub>, ..., unit<sub>n</sub>

Causes all buffered output on the specified file units to be written.

**FREE\_LUN**, unit<sub>1</sub>, ..., unit<sub>n</sub>

Deallocates file units previously allocated with GET\_LUN.

**FSTAT** (unit)

Returns an expression containing status information about a specified file unit.

**GET\_LUN**, unit

Allocates a file unit from a pool of free units.

**OPENR**, unit, filename [, record\_length]

OPENR (OPEN Read) opens an existing file for input only.

**OPENU**, unit, filename [, record\_length]

OPENU (OPEN Update) opens an existing file for input and output.

**OPENURL**, url

Opens a file on the internet to be accessed using PV=WAVE. 1

**OPENW**, unit, filename [, record\_length]

OPENW (OPEN Write) opens a new file for input and output.

**POINT\_LUN**, unit, position

Allows the current position of the specified file to be set to any arbitrary point in the file.

---

## General Graphics Routines

### CPROD(*a*)

Returns the Cartesian product of some arrays.

### CURSOR, *x*, *y* [, *wait*]

Reads the position of the interactive graphics cursor from the current graphics device.

### DERIVN(*a*, *n*)

Differentiates a function represented by an array.

### DEVICE

Provides device-dependent control over the current graphics device (as specified by the SET\_PLOT procedure).

### EMPTY

Causes all buffered output for the current graphics device to be written.

### ERASE [, *background\_color*]

Erases the display surface of the currently active window.

### EXPON(*a*, *b*)

Performs general exponentiation.

### FACTOR(*i*)

Returns the prime factorization of an integer greater than 1.

### GCD(*i*)

Returns the greatest common divisor of some integers greater than 0.

### GREAT\_INT(*values*)

Greatest Integer Function. Standard Library function that returns the greatest integer less than or equal to the passed value. Also known as the Floor Function.

### IMAGE\_CONT, *array*

Standard Library procedure that overlays a contour plot onto an image display of the same array.

### LCM(*i*)

Returns the least common multiple of some integers greater than 1.

### MOVIE, *images* [, *rate*]

Standard Library procedure that shows a cyclic sequence of images stored in a three-dimensional array.

### PLOTS, *x* [, *y* [, *z*]]

Plots vectors or points on the current graphics device in either two or three dimensions.

### PRIME(*value*)

Returns all positive primes less than or equal to a scalar input.

### PRODUCT(*array*)

Returns the product of all elements in an array.

### PROFILE (*image*)

Standard Library function that extracts a profile from an image.

### PROFILES, *image*

Standard Library procedure that lets you interactively draw row or column profiles of the image displayed in the current window. The profiles are displayed in a new window, which is deleted when you exit the procedure.

### RDPIX, *image* [, *x0*, *y0*]

Standard Library procedure that displays the X, Y, and pixel values at the location of the cursor in the image displayed in the currently active window.

### SCALE3D

Standard Library procedure that scales a three-dimensional unit cube into the viewing area.

### SET\_PLOT, *device*

Specifies the device type used by PV-WAVE graphics procedures.

### SGN(*x*)

Returns the sign of passed values.

### SHOW3, array

Standard Library procedure that displays a two-dimensional array as a combination contour, surface, and image plot. The resulting display shows a surface with an image underneath and a contour overhead.

### SMALL\_INT(x)

Smallest Integer Function. Standard Library function that returns the smallest integer greater than or equal to the passed value. Also known as Ceiling Function.

### T3D

Standard Library procedure that accumulates one or more sequences of translation, scaling, rotation, perspective, or oblique transformations and stores the result in the system variable !P.T.

### THREED, array [, space]

Standard Library procedure that plots a two-dimensional array as a pseudo three-dimensional plot on the currently selected graphics device.

### TVCRS [, on\_off]

Manipulates the cursor within a displayed image, allowing it to be enabled and disabled, as well as positioned.

### XYOUTS, x, y, string

Draws text on the currently selected graphics device starting at the designated data coordinate.

### ZOOM

Expands and displays part of an image (or graphic plot) from the current window in a second window.

---

## General Mathematical Functions

### ABS (x)

Returns the absolute value of x.

### AVG (array [, dim] )

Standard Library function that returns the average value of an array. Optionally, it can return the average value of one dimension of an array.

### BILINEAR (array, x, y)

Standard Library function that creates an array containing values calculated using a bilinear interpolation to solve for requested points interior to an input grid specified by the input array.

### CHECK\_MATH ( [print\_flag, message\_inhibit] )

Returns and clears the accumulated math error status.

### CONJ (x)

Returns the complex conjugate of the input variable.

### CONVOL (array, kernel [, scale\_factor] )

Convolve an array with a kernel (or another array).

### CORRELATE (x, y)

Standard Library function that calculates a simple correlation coefficient for two arrays.

### CROSSP (v<sub>1</sub>, v<sub>2</sub>)

Standard Library function that returns the cross product of two three-element vectors.

### CURVATURES (s)

Standard Library function that computes curvatures on a parametrically defined surface.

### CURVEFIT (x, y, wt, parms, [sigma] )

Standard Library function that performs a nonlinear least-squares fit to a function of an arbitrary number of parameters.

**DERIV** ( [x,] y)

Standard Library function that calculates the first derivative of a function in x and y.

**DERIVN** (a, n)

Differentiates a function represented by an array.

**DETERM** (array)

Standard Library function that calculates the determinant of a square, two-dimensional input variable.

**EUCLIDEAN** (j)

Standard Library function that transforms the Euclidean metric for a Jacobian  $j = \text{Jacobian}(f)$

**EXPON** (a, b)

Performs general exponentiation.

**FFT** (array, direction)

Returns the Fast Fourier Transform for the input variable.

**FINITE** (x)

Returns a value indicating if the input variable is finite or not.

**GAUSSFIT** (x, y [, coefficients] )

Standard Library function that fits a Gaussian curve through a data set.

**HILBERT** (x [, d] )

Standard Library function that constructs a Hilbert transformation matrix.

**IMAGINARY** (complex\_expr)

Returns the imaginary part of a complex number.

**INVERT** (array [, status] )

Returns an inverted copy of a square array.

**ISHFT** (p<sub>1</sub>, p<sub>2</sub>)

Performs the bit shift operation on bytes, integers, and longwords.

**JACOBIAN** (f)

Standard Library function that computes the Jacobian of a function represented by  $n$   $m$ -dimensional arrays

**LUBKSB**, a, index, b

Solves the set of  $n$  linear equations  $Ax = b$ . (LUBKSB must be used with the procedure LUDCMP to do this.)

**LUDCMP**, a, index, d

Replaces an  $n$ -by- $n$  matrix, a, with the LU decomposition of a row-wise permutation of itself.

**MPROVE**, a, alud, index, b, x

Iteratively improves the solution vector, x, of a linear set of equations,  $Ax = b$ . (You must call the LUDCMP procedure before calling MPROVE.)

**NORMALS** (j)

Standard Library function that computes unit normals on a parametrically defined surface.

**POLY** (x, coefficients)

Standard Library function that evaluates a polynomial function of a variable.

**POLY\_AREA** (x, y)

Standard Library function that returns the area of an  $n$ -sided polygon, given the vertices of the polygon.

**POLY\_FIT** (x, y, deg [, yft, ybd, sig, mat] )

Standard Library function that fits an  $n$ -degree polynomial curve through a set of data points using the least-squares method.

**POLYFITW** (x, y, wt, deg [, yft, ybd, sig, mat] )

Standard Library function that fits an  $n$ -degree polynomial curve through a set of data points using the least-squares method.

**RANDOMN** (seed [, dim<sub>1</sub>, ... , dim<sub>n</sub>] )

Returns one or more normally distributed floating-point pseudo-random numbers with a mean of zero and a standard deviation of 1.

**RANDOMU** (seed [, dim<sub>1</sub>, ... , dim<sub>n</sub>] )

Returns one or more uniformly distributed floating-point pseudo-random numbers over the range  $0 < Y < 1.0$ .

**REGRESS** (x, y, wt [, yf, a0, sig, ft, r, rm, c] )  
Standard Library function that fits a curve to data using the multiple linear regression method.

**SIGMA** (array [, npar, dim] )  
Standard Library function that calculates the standard deviation value of an array. (Optionally, it can also calculate the standard deviation over one dimension of an array as a function of the other dimensions.)

**SOBEL** (image)  
Performs a Sobel edge enhancement of an image.

**SPLINE** (x, y, t [, tension] )  
Standard Library function that performs a cubic spline interpolation.

**STDEV** (array [, mean] )  
Standard Library function that computes the standard deviation and (optionally) the mean of the input array.

**SUM** (array, dim)  
Sums an array of n dimensions over one of its dimensions.

**SURFACE\_FIT** (array, degree)  
Standard Library function that determines the polynomial fit to a surface.

**SVBKS**, u, w, v, b, x  
Uses “back substitution” to solve the set of simultaneous linear equations  $Ax = b$ , given the u, w, and v arrays created by the SVD procedure from the matrix a.

**SVD**, a, W [, u [, v]]  
Performs a singular value decomposition on a matrix.

**SVDFIT** (x, y, m)  
Standard Library function that uses the singular value decomposition method of least-squares curve fitting to fit a polynomial function to data.

**TOTAL** (array)  
Sums the elements of an input array.

**TQLI**, d, e, z  
Uses the QL algorithm with implicit shifts to determine the eigenvalues and eigenvectors of a real, symmetric, tridiagonal matrix.

**TRED2**, a [, d [, e]]  
Reduces a real, symmetric matrix to tridiagonal form, using Householder’s method.

**TRIDAG**, a, b, c, r, u  
Solves tridiagonal systems of linear equations.

**ZROOTS**, a, roots [, polish]  
Finds the roots of the m-degree complex polynomial, using Laguerre’s method.

---

## **Gridding Routines**

**FAST\_GRID2** (points, grid\_x)  
Returns a gridded, 1D array containing Y values, given random X,Y coordinates (this function works best with dense data points).

**FAST\_GRID3** (points, grid\_x, grid\_y)  
Returns a gridded, 2D array containing Z values, given random X, Y, Z coordinates (this function works best with dense data points).

**FAST\_GRID4** (points, grid\_x, grid\_y, grid\_z)  
Returns a gridded, 3D array containing intensity values, given random 4D coordinates (this function works best with dense data points).

**GRID\_2D** (points, grid\_x)  
Returns a gridded, 1D array containing Y values, given random X,Y coordinates (this function works best with sparse data points).

### GRID\_3D (points, grid\_x, grid\_y)

Returns a gridded, 2D array containing Z values, given random X, Y, Z coordinates (this function works best with sparse data points).

### GRID\_4D (points, grid\_x, grid\_y, grid\_z)

Returns a gridded, 3D array containing intensity values, given random 4D coordinates (this function works best with sparse data points).

### GRIDN(*d*, *i*)

Grids *n* dimensional data.

### GRID\_SPHERE (points, grid\_x, grid\_y)

Returns a gridded, 2D array containing radii, given random longitude, latitude, and radius values.

### INTERPOLATE(*d*, *x*)

Interpolates scattered data at scattered locations.

---

## HDF Routines

### GETNCERR ( [errstr,] )

Retrieves the current value of the “ncerr” variable as discussed in the error section of the *NetCDF User's Guide*.

### GETNCOPTS ( )

Retrieves the current value of the ncopts variable as discussed in the error section of the *NetCDF User's Guide*.

### HDFGET24 (filename, image)

Obtains an HDF Raster 24 image.

### HDFGETANN (filename, tag, ref)

Obtains HDF object (e.g., an SDS, Raster 8 image, etc.) annotations, either a label or a description.

### HDFGETFILEANN (filename)

Obtains an HDF file annotation, either label or description.

### HDFGETNT (type)

Obtains the HDF number type (i.e., data type) and descriptive number type string for the current HDF Scientific Data Set.

### HDFGETR8 (filename, image, palette)

Obtains an HDF Raster 8 image and associated palette.

### HDFGETRANGE (maxvalue, minvalue)

Gets the maximum and minimum range for the current HDF Scientific Data Set.

### HDFGETSDS (filename, data)

Gets an HDF Scientific Data Set.

### HDFLCT, palette

Loads an HDF palette as a PV-WAVE color table.

### HDFPUT24 (filename, image)

Puts an HDF Raster 24 image into an HDF file.

### HDFPUTFILEANN (filename)

Inserts HDF file labels and file descriptions (annotations) into a file.

### HDFPUTR8 (filename, image)

Writes an 8 bit image to an HDF file.

### HDFPUTSDS (filename, data)

Writes a Scientific Data Set to an HDF file.

### HDFSCAN, filename

Scans an HDF file and prints a simple list of file contents by HDF object type.

### HDFSETNT (data)

Computes and sets the HDF number type (i.e., data type) and descriptive number type string for the specified data array.

### HDF\_STARTUP

A batch file used to initialize the HDF interface. See PV-WAVE Reference Appendix A, “The PV-WAVE HDF Interface” for more information.

### HDF\_TEST

Runs the PV-WAVE HDF test suite.

**SETNCOPTS**, `new_ncopts`

Sets the value of the `ncopts` variable and defines the level of error reporting for the `netCDF` functions as discussed in the error section of the *NetCDF User's Guide*.

---

## Help and Information Routines

**DOC\_LIBRARY** [, `name`]

Standard Library procedure that extracts header documentation for user-written PV-WAVE procedures and functions.

**HELP**

Starts the online help system.

**INFO**, `expr1`, ... , `exprn`

Displays information on many aspects of the current PV-WAVE session.

---

## Hypertext Markup Language (HTML) Routines

**HTML\_BLOCK**, `text`

Writes out a specifically formatted "block" of HTML text.

**HTML\_CLOSE**

Closes an HTML file, after end-tagging major elements.

**HTML\_HEADING**, `text`

Creates a heading, with a level specification.

**HTML\_HIGHLIGHT**([`str1`, `str2`, ... , `strn`], [`tag1`, `tag2`, ... , `tagn`])

Allows for all the basic textual highlighting elements in HTML.

**HTML\_IMAGE**(`url`)

Returns a string or an array of strings containing a reference or references to image URL(s)

**HTML\_LINK**(`url`, `text`)

Sets up links to Uniform Resource Locations (URLs).

**HTML\_LIST**, [`list_item1`, ..., `list_itemn`]

Generates HTML code for all types of lists.

**HTML\_OPEN**

Opens the output HTML file, writes out the basic HTML information and sets an HTML output file information variable, `hinfo`.

**HTML\_PARAGRAPH**, `text`

Defines an HTML paragraph.

**HTML\_RULE**

Inserts a horizontal-line separator.

**HTML\_SAFE**(`str`)

Escapes special characters so that the HTML displays them as intended, rather than using them for format tagging. The escapes codes are for: `<`, `>`, `&`, and `"`.

**HTML\_TABLE**, `table_text`

Writes out an HTML table.

---

## Image Display Routines

### **C\_EDIT** [, colors\_out]

Standard Library procedure that lets you interactively create a new color table based on the HLS or HSV color system.

### **COLOR\_CONVERT**, i<sub>0</sub>, i<sub>1</sub>, i<sub>2</sub>, o<sub>0</sub>, o<sub>1</sub>, o<sub>2</sub>, keyword

Converts colors to and from the RGB color system and either the HLS or HSV systems.

### **COLOR\_EDIT** [, colors\_out]

Standard Library procedure that lets you interactively create color tables based on the HLS or HSV color system.

### **COLOR\_PALETTE**

Standard Library procedure that displays the current color table colors and their associated color table indices.

### **HIST\_EQUAL\_CT** [, image]

Standard Library procedure that uses an input image parameter, or the region of the display you mark, to obtain a pixel distribution histogram. The cumulative integral is taken and scaled, and the result is applied to the current color table.

### **HLS**, ltlo, lthi, stlo, sthi, hue, lp [, rgb]

Standard Library procedure that generates and loads color tables into an image display device based on the HLS color system. The resulting color table is loaded into the display system.

### **HSV**, vlo, vhi, stlo, sthi, hue, lp [, rgb]

Standard Library procedure that generates and loads color tables into an image display device based on the HSV color system. The final color table is loaded into the display device.

### **LOADCT** [, table\_number]

Standard Library procedure that loads a predefined PV-WAVE color table.

### **LOADCT\_CUSTOM** [, table\_number]

Loads a predefined custom color table.

### **MODIFYCT**, table, name, red, green, blue

Standard Library procedure that lets you replace one of the PV-WAVE color tables (defined in the colors.tbl file) with a new color table.

### **PALETTE** [, colors\_out]

Standard Library procedure that lets you interactively create a new color table based on the RGB color system.

### **PSEUDO**, ltlo, lthi, stlo, sthi, hue, lp [, rgb]

Standard Library procedure that creates a pseudo color table based on the Hue, Lightness, Saturation (HLS) color system.

### **STRETCH**, low, high

Standard Library procedure that linearly expands the range of the color table currently loaded to cover an arbitrary range of pixel values.

### **TV**, image [, position]

Displays images without scaling the intensity.

### **TVCRS** [, on\_off]

Manipulates the cursor within a displayed image, allowing it to be enabled and disabled, as well as positioned.

### **TVLCT**, v<sub>1</sub>, v<sub>2</sub>, v<sub>3</sub> [, start]

Loads the display color translation tables from the specified variables.

### **TVRD** (x<sub>0</sub>, y<sub>0</sub>, n<sub>x</sub>, n<sub>y</sub> [, channel ])

Returns the contents of the specified rectangular portion of a displayed image.

### **TVSCL**, image [, x, y [, channel ]]

### **TVSCL**, image [, position]

Scales the intensity values of an input image into the range of the image display, usually from 0 to 255, and outputs the data to the image display at the specified location.

### **TVSIZE**, image [, x, y [, channel ]]

### **TVSIZE**, image [, position]

D displays images at the current or specified size and device resolution.

---

## Image IO Routines

**DICM\_TAG\_INFO** (filename, image)

Extracts Digital Imaging and Communications in Medicine (DICOM) tags information from an image associative array.

**IMAGE\_CHECK**(image)

Determines that the input variable is an associative array in image format. The function also checks to make sure that all fields in the image associative array are present.

**IMAGE\_COLOR\_QUANT**( image  
[, n\_colors] )

Quantizes a 24-bit image to 8-bit pseudo color.

**IMAGE\_CREATE**(pixel\_array)

Creates an associative array in image format. See the *Discussion* section for detailed information on the image format.

**IMAGE\_DISPLAY**, image

Displays an image.

**IMAGE\_QUERY\_FILE**, filename

Return the type of a specified image file.

**IMAGE\_READ**(filename)

Reads image files and returns an associative array in image format.

**IMAGE\_WRITE**(filename, image)

Writes PV=WAVE graphics to a specified file type.

**READ\_XBM**, file, image

Reads the contents of an X-bitmap (XBM) file into a PV=WAVE variable.

**WRITE\_XBM**, file, image

Writes an image to an X-bitmap (XBM) file.

---

## Image Processing Routines

**AFFINE**(a, b, [c])

Applies an affine transformation to an array.

**BLOB**(a, i, b)

Isolates a homogeneous region in an array.

**BLOBCOUNT**(a, b)

Counts homogeneous regions in an array.

**BOUNDARY**(a,r)

Computes the boundary of a region in an array.

**CONGRID** (image, col, row)

Standard Library function that shrinks or expands an image or array.

**CONVOL** (array, kernel [, scale\_factor] )

Convolve an array with a kernel (or another array).

**DEFROI** (sizeX, sizeY [, xverts, yverts] )

Standard Library function that defines an irregular region of interest within an image by using the image display system and the mouse.

**DERIVN**(a, n)

Differentiates a function represented by an array.

**DIGITAL\_FILTER** (flow, fhigh, gibbs,  
nterm)

Standard Library function that constructs finite impulse response digital filters for signal processing.

**DILATE** (image, structure [, x0, y0] )

Implements the morphologic dilation operator for shape processing.

**DIST** (n)

Standard Library function that generates a square array in which each element equals the euclidean distance from the nearest corner.

**ERODE** (image, structure [, x0, y0] )

Implements the morphologic erosion operator for shape processing.

**FFT** (array, direction)

Returns the Fast Fourier Transform for the input variable.

**HANNING** (col [, row] )

Standard Library function that implements a window function for Fast Fourier Transform signal or image filtering.

**HIST\_EQUAL** (image)

Standard Library function that returns a histogram-equalized image or vector.

**HISTOGRAM** (array)

Returns the density function of an array.

**IMAGE\_CONT**, array

Standard Library procedure that overlays a contour plot onto an image display of the same array.

**LEEFILT** (image [, n, sigma] )

Standard Library function that performs image smoothing by applying the Lee Filter algorithm.

**MEDIAN** (array [, width] )

Finds the median value of an array, or applies a one- or two- dimensional median filter of a specified width to an array.

**MOMENT** ( $a, i$ )

Computes moments of an array.

**MOVIE**, images [, rate]

Standard Library procedure that shows a cyclic sequence of images stored in a three-dimensional array.

**NEIGHBORS**( $a, i$ )

Finds the neighbors of specified array elements.

**POLY\_2D** (array, coeff<sub>x</sub>, coeff<sub>y</sub> [, interp  
[, dim<sub>x</sub> ,..., dim<sub>y</sub>]])

Performs polynomial warping of images.

**POLYFILLV** (x, y, sx, sy [, run\_length] )

Returns a vector containing the subscripts of the array elements contained inside a specified polygon.

**POLYWARP**, xd, yd, xin, yin, deg, xm, ym

Standard Library procedure that calculates the coefficients needed for a polynomial image warping transformation.

**PROFILE** (image)

Standard Library function that extracts a profile from an image.

**PROFILES**, image

Standard Library procedure that lets you interactively draw row or column profiles of the image displayed in the current window. The profiles are displayed in a new window, which is deleted when you exit the procedure.

**RDPIX**, image [, x0, y0]

Standard Library procedure that displays the X, Y, and pixel values at the location of the cursor in the image displayed in the currently active window.

**REBIN** (array, dim<sub>1</sub>, ..., dim<sub>n</sub>)

Returns a vector or array resized to the given dimensions.

**REFORM** (array, dim<sub>1</sub>, ... , dim<sub>n</sub>)

Reformats an array without changing its values numerically.

**RESAMP** (array, dim<sub>1</sub>, ..., dim<sub>n</sub>)

Resamples an array to new dimensions.

**ROBERTS** (image)

Performs a Roberts edge enhancement of an image.

**ROT** (image, ang [, mag, xctr, yctr] )

Standard Library function that rotates and magnifies (or demagnifies) a two-dimensional array.

**ROT\_INT** (image, ang [, mag, xctr, yctr] )  
Standard Library function that rotates and magnifies (or demagnifies) an image on the display screen.

**ROTATE** (array, direction)  
Returns a rotated and/or transposed copy of the input array.

**SHIFT** (array, shift<sub>1</sub>, ... , shift<sub>n</sub>)  
Shifts the elements of a vector or array along any dimension by any number of elements.

**SHOW3**, array  
Standard Library procedure that displays a two-dimensional array as a combination contour, surface, and image plot. The resulting display shows a surface with an image underneath and a contour overhead.

**SMOOTH** (array, width)  
Smooths an array with a boxcar average of a specified width.

**SOBEL** (image)  
Performs a Sobel edge enhancement of an image.

**TRANSPOSE** (array)  
Transposes the input array.

**ZOOM**  
Standard Library procedure that expands and displays part of an image (or graphic plot) from the current window in a second window.

---

## **Input and Output Routines**

**ASSOC** (unit, array\_structure [, offset])  
Associates an array structure with a file, allowing random access input and output.

**BYTEORDER**, variable<sub>1</sub>, ... , variable<sub>n</sub>  
Converts integers between host and network byte ordering. Can also be used to swap the order of bytes within both short and long integers.

**GET\_KBRD** (wait)  
Returns the next character available from standard input (PV-WAVE file unit 0).

**LN03** [, filename]  
Standard Library procedure that opens or closes an output file for LN03 graphics output. The file can then be printed on an LN03 printer.

**PRINT**, expr<sub>1</sub>, ... , expr<sub>n</sub>  
PRINT performs output to the standard output stream (PV-WAVE file unit -1).

**PRINTF**, unit, expr<sub>1</sub>, ... , expr<sub>n</sub>  
PRINTF requires the output file unit to be specified.

**READ**, var<sub>1</sub>, ..., var<sub>n</sub>  
Read input from the standard input stream into PV-WAVE variables.

**READF**, unit, var<sub>1</sub>, ... , var<sub>n</sub>  
Read input from a file into PV-WAVE variables.

**READU**, unit, var<sub>1</sub>, ... , var<sub>n</sub>  
READU reads binary (unformatted) input from a specified file. (No processing of any kind is done to the data.)

**REWIND**, unit  
(OpenVMS Only) Rewinds the tape on the designated PV-WAVE tape unit.

**SKIPF**, unit, files  
(OpenVMS Only) Skips files on the designated magnetic tape unit.

**SKIPF**, unit, records, r

(OpenVMS Only) Skips records on the designated magnetic tape unit.

**TAPRD**, array, unit [, byte\_reverse]

(OpenVMS Only) Reads the next record on the selected tape unit into the specified array.

**TAPWRT**, array, unit [, byte\_reverse]

(OpenVMS Only) Writes data from the input array to the selected tape unit.

**WPRINT** [, window\_index]

(Microsoft Windows Only) Prints the contents of a specified window.

**WREAD\_DIB** ( [window\_index] )

(Microsoft Windows Only) Loads a Device Independent Bitmap (DIB) from a file into a graphics window.

**WREAD\_META** ( [window\_index] )

(Microsoft Windows Only) Loads a Windows metafile (WMF) into a graphics window.

**WRITEU**, unit, expr<sub>1</sub>, ... , expr<sub>n</sub>

Writes binary (unformatted) data from an expression into a file.

**WWRITE\_DIB** ( [window\_index] )

(Microsoft Windows Only) Saves the contents of a graphics window to a file as a Device Independent Bitmap (DIB).

**WWRITE\_META** ( [window\_index] )

(Microsoft Windows Only) Saves the contents of a graphics window to a file as a Windows metafile (WMF).

---

## **Interpolation Routines**

**AFFINE**(a, b, [c])

Applies an affine transformation to an array.

**BILINEAR** (array, x, y)

Standard Library function that creates an array containing values calculated using a bilinear interpolation to solve for requested points interior to an input grid specified by the input array.

**GRIDN**(d, i)

Grids n dimensional data.

**INTERPOLATE**(d, x)

Interpolates scattered data at scattered locations.

**INTERPOL**(v, n)

Performs a linear interpolation of a vector using a regular grid.

**INTERPOL**(v, x, u)

Performs a linear interpolation of a vector using an irregular grid.

**INTRP**(a, n, x)

Interpolates an array along one of its dimensions.

**REBIN** (array, dim<sub>1</sub>, ..., dim<sub>n</sub>)

Returns a vector or array resized to the given dimensions.

**RESAMP**(array, dim<sub>1</sub>, ..., dim<sub>n</sub>)

Resamples an array to new dimensions.

**SPLINE** (x, y, t [, tension] )

Standard Library function that performs a cubic spline interpolation.

---

## Mapping Routines

### MAP

Plots a map.

### MAP\_CONTOUR, z [, x, y]

Draws a contour plot from longitude/latitude data stored in a 2D array.

### MAP\_PLOTS, x, y

Plots vectors or points (specified as longitude/latitude data) on the current map projection.

### MAP\_POLYFILL, x, y

Fills the interior of a region of the display enclosed by an arbitrary 2D polygon.

### MAP\_REVERSE, x, y, lon, lat

Converts output from routines like CURSOR and WtPointer from device, normal, or data coordinates to longitude and latitude coordinates.

### MAP\_VELOVECT, u, v, [, x, y]

Draws a two-dimensional velocity field plot on a map, with each directed arrow indicating the magnitude and direction of the field.

### MAP\_XYOUTS, x, y, string

Draws text on the currently selected graphics device starting at the designated map coordinate.

### USGS\_NAMES ( [name] )

Queries a database containing names, FIPS codes, and longitude/latitude values for cities and towns in the United States.

Plots a map.

---

## Operating System Access Routines

### CALL\_UNIX (p<sub>1</sub> [, p<sub>2</sub>, ... , p<sub>30</sub>])

(UNIX Only) Lets a PV-WAVE procedure communicate with an external routine written in C.

### CD [, directory]

Changes the current working directory.

### DELETE\_SYMBOL, name

(OpenVMS Only) Deletes a DCL (Digital Command Language) interpreter symbol from the current process.

### DEL\_FILE, filename

Deletes a specified file on your system.

### DELLOG, logname

(OpenVMS Only) Deletes a logical name.

### ENVIRONMENT ( )

(UNIX Only) Returns a string array containing all the UNIX environment strings for the PV-WAVE process.

### GETENV (name)

Returns the specified equivalence string from the environment of the PV-WAVE process.

### GET\_SYMBOL (name)

(OpenVMS Only) Returns the value of an OpenVMS DCL interpreter symbol as a scalar string.

### LINKLOAD (object, symbol [, param<sub>1</sub>, ..., param<sub>n</sub>])

Provides simplified access to external routines in shareable images.

### POPD

Standard Library procedure that pops a directory from the top of a last-in, first-out directory stack.

## PRINTD

Standard Library procedure that lists the directories located in the directory stack, and the current working directory.

## PUSHD [, directory]

Standard Library procedure that pushes a directory onto the top of a last-in, first-out directory stack.

## SETENV, environment\_expr

(UNIX Only) Adds or changes an environment string in the process environment.

## SETLOG, logname, value

(OpenVMS Only) Defines a logical name.

## SET\_SYMBOL, name, value

(OpenVMS Only) Defines a DCL interpreter symbol for the current process.

## SPAWN [, command [, result]]

(UNIX/OpenVMS) Spawns a child process to execute a given command.

## SPAWN [, command [, result]]

(Windows) Spawns a child process to execute a given command.

## SYSTIME (param)

Returns the current system time as either a string or as the number of seconds elapsed since January 1, 1970.

## TRNLOG (logname, value)

(OpenVMS Only) Searches the OpenVMS name tables for a specified logical name and returns the equivalence string (s) in a PV-WAVE variable.

## WEOF, unit

(OpenVMS Only) Writes an end-of-file mark on the designated unit at the current position.

## printer\_name = WIN32\_PICK\_PRINTER( )

Displays a Windows printer dialog.

## font\_name = WIN32\_PICK\_FONT( )

Displays a Windows common font dialog.

---

# Optimization and Regression Routines

## CURVEFIT (x, y, wt, parms, [sigma] )

Standard Library function that performs a nonlinear least-squares fit to a function of an arbitrary number of parameters.

## GAUSSFIT (x, y [, coefficients] )

Standard Library function that fits a Gaussian curve through a data set.

## MINIMIZE(f, l, u, g, i, y )

Minimizes a real valued function of n real variables.

## POLY\_FIT (x, y, deg [, yft, ybd, sig, mat] )

Standard Library function that fits an n-degree polynomial curve through a set of data points using the least-squares method.

## POLYFITW (x, y, wt, deg [, yft, ybd, sig, mat] )

Standard Library function that fits an n-degree polynomial curve through a set of data points using the least-squares method.

## REGRESS (x, y, wt [, yf, a0, sig, ft, r, rm, c] )

Standard Library function that fits a curve to data using the multiple linear regression method.

## SVDFIT (x, y, m)

Standard Library function that uses the singular value decomposition method of least-squares curve fitting to fit a polynomial function to data.

---

## Plotting Routines

### AXIS [[[, x], y], z]

Draws an axis of the specified type and scale at a given position.

### BAR, x [,y]

Plots a 2D bar graph that can include stacked and grouped bars, as well as various color and fill pattern options.

### BAR2D, x [,y]

Creates a two-dimensional bar plot.

### BAR3D, z

Creates a three-dimensional bar plot.

### CONTOUR, z [, x, y]

Draws a contour plot from data stored in a rectangular array.

### CONTOUR2, z [, x, y]

Draws a contour plot from data stored in a rectangular array.

### CONTOURFILL, filename, z [, x, y]

Standard Library procedure that fills both open and closed contours with specified colors or patterns.

### CURSOR, x, y [, wait]

Reads the position of the interactive graphics cursor from the current graphics device.

### ERRPLOT [, points], low, high

Standard Library procedure that overplots error bars over a previously-drawn plot.

### GRID (xtmp, ytmp, ztmp)

Standard Library function that generates a uniform grid from irregularly-spaced data.

### IMAGE\_CONT, array

Standard Library procedure that overlays a contour plot onto an image display of the same array.

### O PLOT, x [, y]

Plots vector data over a previously drawn plot.

### O PLOTERR, x, y, error [, psym]

Standard Library procedure that overplots symmetrical error bars on any plot already output to the display device.

### PIE, data [, labels]

Displays data as a pie chart.

### PIE\_CHART, data, [xcenter, ycenter, radius]

Creates a pie chart with colors, text labels, exploded slices and/or shadows.

### PLOT, x [, y]

PLOT produces a simple XY plot.

### PLOT\_HISTOGRAM, variable

Plots a histogram.

### PLOT\_IO, x [, y]

PLOT\_IO produces an XY plot with logarithmic scaling on the Y axis.

### PLOT\_OI, x [, y]

PLOT\_OI produces an XY plot with logarithmic scaling on the X axis.

### PLOT\_OO, x [, y]

PLOT\_OO produces an XY plot with logarithmic scaling on both the X and Y axes.

### PLOTERR, [x,] y, error

Standard Library procedure that plots data points with accompanying symmetrical error bars.

### PLOT\_FIELD, u, v

Standard Library procedure that plots a two-dimensional velocity field.

### PLOTS, x [, y [, z]]

Plots vectors or points on the current graphics device in either two or three dimensions.

### POLYFILL, x [, y [, z]]

Fills the interior of a region of the display enclosed by an arbitrary two- or three-dimensional polygon.

### POLYSHADE (vertices, polygons)

Constructs a shaded surface representation of one or more solids described by a set of polygons.

### PROFILE (image)

Standard Library function that extracts a profile from an image.

### PROFILES, image

Standard Library procedure that lets you interactively draw row or column profiles of the image displayed in the current window. The profiles are displayed in a new window, which is deleted when you exit the procedure.

### SCALE3D

Standard Library procedure that scales a three-dimensional unit cube into the viewing area.

### SET\_SHADING

Modifies the light source shading parameters affecting the output of SHADE\_SURF and POLYSHADE.

### SHADE\_SURF, z [, x, y]

Standard Library procedure that creates a shaded surface representation of a regular or nearly regular gridded surface, with shading from either a light source model or from a specified array of intensities.

### SHADE\_SURF\_IRR, z, x, y

Creates a shaded-surface representation of a semiregularly gridded surface, with shading from either a light source model or from a specified array of intensities.

### SHOW3, array

Standard Library procedure that displays a two-dimensional array as a combination contour, surface, and image plot. The resulting display shows a surface with an image underneath and a contour overhead.

### SURFACE, z [, x, y]

Draws the surface of a two-dimensional array projected into two dimensions, with hidden lines removed.

### T3D

Standard Library procedure that accumulates one or more sequences of translation, scaling, rotation, perspective, or

oblique transformations and stores the result in the system variable !P.T.

### THREED, array [, space]

Standard Library procedure that plots a two-dimensional array as a pseudo three-dimensional plot on the currently selected graphics device.

### USERSYM, x [, y]

Lets you create a custom symbol for marking plotted points.

### VEL, u, v

Standard Library procedure that draws a graph of a velocity field with arrows pointing in the direction of the field. The length of an arrow is proportional to the strength of the field at that point.

### VELOVECT, u, v [, x, y]

Standard Library procedure that draws a two-dimensional velocity field plot, with each directed arrow indicating the magnitude and direction of the field.

---

## **Polygon Generation Routines**

### POLY\_SPHERE, radius, px, py, vertex\_list, polygon\_list

Generates the vertex list and polygon list that represent a sphere.

### POLY\_SURF, surf\_dat, vertex\_list, polygon\_list, pg\_num

Generates a 3D vertex list and a polygon list, given a 2D array containing Z values.

### SHADE\_VOLUME, volume, value, vertex, poly

Given a 3D volume and a contour value, produces a list of vertices and polygons describing the contour surface (also known as an iso-surface).

---

## **Polygon Manipulation Routines**

**POLY\_C\_CONV** (polygon\_list, colors)

Returns a list of colors for each polygon, given a polygon list and a list of colors for each vertex.

**POLY\_COUNT** (polygon\_list)

Returns the total number of polygons contained in a polygon list.

**POLY\_MERGE**, vertex\_list1, vertex\_list2, polygon\_list1, polygon\_list2, vert, poly, pg\_num

Merges two vertex lists and two polygon lists together so that they can be rendered in a single pass.

---

## **Polygon Rendering Routines**

**MOLEC**(filename)

Creates an image of a ball and stick molecular model.

**POLY\_PLOT**, vertex\_list, polygon\_list, pg\_num, winx, winy, fill\_colors, edge\_colors, poly\_opaque

Renders a given list of polygons.

**POLYSHADE** (vertices, polygons)

**POLYSHADE** (x, y, z, polygons)

Constructs a shaded surface representation of one or more solids described by a set of polygons.

**RENDER** (object1, ..., objectn)

Generates a ray-traced rendered image from one or more predefined objects.

**RENDER24**(b)

Generates a ray-traced rendered 24-bit image of m objects.

---

## **Programming Routines**

**ADDVAR**, name, local

Creates a variable on the \$MAIN\$ program level and binds a local variable to it.

**BREAKPOINT**, file, line

Lets you insert and remove breakpoints in programs for debugging.

**CHECKFILE**( filename )

Determines if a file can be read from or written to.

**CHECK\_MATH** ([print\_flag, message\_inhibit])

Returns and clears the accumulated math error status.

**DEFINE\_KEY**, key [, value]

Programs a keyboard function key with a string value, or with a specified action.

**DEFSYSV**, name, value [, read\_only]

Creates a new system variable initialized to the specified value.

**DELFUNC**, function<sub>i</sub> ,..., function<sub>n</sub>

Deletes one or more compiled functions from memory.

**DELPROC**, procedure<sub>i</sub> ,..., procedure<sub>n</sub>

Deletes one or more compiled procedures from memory.

**DELSTRUCT**, structure<sub>i</sub> ,..., structure<sub>n</sub>

Deletes one or more named structure definitions from memory.

**DELVAR**, v<sub>1</sub>, ... ,v<sub>n</sub>

Deletes variables from the main program level.

**EXIT**

Exits PV-WAVE and returns you to the operating system.

**FINITE** (x)

Returns a value indicating if the input variable is finite or not.

## HAK

Standard Library procedure that lets you implement a “hit any key to continue” function.

## KEYWORD\_SET (expr)

Tests if an input expression has a nonzero value.

## MESSAGE, text

Issues error and informational messages using the same mechanism employed by built-in PV-WAVE routines.

## N\_ELEMENTS (expr)

Returns the number of elements contained in any expression or variable.

## N\_PARAMS ( )

Returns the number of non-keyword parameters used in calling a PV-WAVE procedure or function.

## N\_TAGS (expr)

Returns the number of structure tags contained in any expression.

## ON\_ERROR, n

Determines the action taken when an error is detected inside a PV-WAVE user-written procedure or function.

## ON\_ERROR\_GOTO, label

Specifies a statement to jump to if an error occurs in the current procedure.

## ON\_IOERROR, label

Specifies a statement to jump to if an I/O error occurs in the current procedure.

## PARAM\_PRESENT (parameter)

Tests if a parameter was actually present in the call to a procedure or function.

## PARSEFILENAME (pathname)

Extracts specified parts of a full file pathname.

## RENAME, variable, new\_name

Renames a PV-WAVE variable.

## RETALL

Issues RETURNS from nested routines. Used primarily to recover from errors in user-written procedures and functions.

## RETURN [, expr]

Returns control to the caller of a user-written procedure or function.

## SAME(x, y)

Tests if two variables are the same.

## SIZE (expr)

Returns a vector containing size and type information for the given expression.

## STOP [, expr<sub>1</sub>,... , expr<sub>n</sub>]

Stops the execution of a running program or batch file, and returns control to the interactive mode.

## STRMESSAGE (errno)

Returns the text of the error message specified by the input error number.

## STRUCTREF ({structure})

Returns a list of all existing references to a structure.

## TAG\_NAMES (expr)

Returns a string array containing the names of the tags in a structure expression.

## UPVAR, name, local

Accesses a variable that is not on the current program level.

## VAR\_MATCH( )

Standard Library function that scans for PV-WAVE variables that match the given criteria.

## WAIT, seconds

Suspends execution of a PV-WAVE program for a specified period.

---

## Ray Tracing Routines

### CONE ( )

Defines a conic object that can be used by the RENDER function.

### CYLINDER ( )

Defines a cylindrical object that can be used by the RENDER function.

### MESH (vertex\_list, polygon\_list)

Defines a polygonal mesh object that can be used by the RENDER function.

### RENDER (object1, ..., objectn)

Generates a ray-traced rendered image from one or more predefined objects.

### RENDER24(b)

Generates a ray-traced rendered 24-bit image of m objects.

### SPHERE ( )

Defines a spherical object that can be used by the RENDER function.

### VOLUME (voxels)

Defines the volumetric data that can be used by the RENDER function.

---

## Session Routines

### INFO, expr<sub>1</sub>, ..., expr<sub>n</sub>

Displays information on many aspects of the current PV-WAVE session.

### JOURNAL [, param]

Provides a record of an interactive session by saving in a file all text entered from the terminal in response to a prompt.

### RESTORE [, filename]

Restores the PV-WAVE objects saved in a file by the SAVE procedure.

### SAVE [, var<sub>1</sub>, ..., var<sub>n</sub>]

Saves variables in a file for later recovery by RESTORE.

---

## Special Mathematical Functions

### BESELI (x [, n])

Calculates the Bessel I function for the input parameter.

### BESELJ (x [, n])

Calculates the Bessel J function for the input parameter.

### BESELY (x [, n])

Calculates the Bessel Y function for the input parameter.

### ERRORF (x)

Calculates the standard error function of the input variable.

### GAMMA (x)

Calculates the gamma function of the input variable.

### GAUSSINT (x)

Evaluates the integral of the Gaussian probability function.

---

## String Processing Routines

### STRCOMPRESS (string)

Compresses the white space in an input string.

### STRJOIN(expr [, sep])

Concatenates all of the elements of a string array into a single scalar string.

### STRLEN (expr)

Returns the length of the input parameter.

### STRLOOKUP([name])

Queries, creates, saves, or modifies a string server database.

### STRLOWCASE (string)

Converts a copy of the input string to lowercase letters.

**STRMATCH**(string, expr [, registers])

Matches a specified string to an existing regular expression.

**STRMID** (expr, first\_character, length)

Extracts a substring from a string expression.

**STRPOS** (object, search\_string [, position] )

Searches for the occurrence of a substring within an object string, and returns its position.

**STRPUT**, destination, source [, position]

Inserts the contents of one string into another.

**STRSPLIT**(expr, pattern)

Splits a string into an array of tokens (substrings).

**STRSUBST**(expr, pattern, repl)

Performs string substitution (search and replace).

**STRTRIM** (string [, flag] )

Removes extra blank spaces from an input string.

**STRUPCASE** (string)

Converts a copy of the input string to uppercase letters.

---

## **Table Manipulation Functions**

**BUILD\_TABLE** ( ' var<sub>i</sub> [alias], ..., var<sub>n</sub> [alias] ' )

Creates a table from one or more vectors (one-dimensional arrays).

**GROUP\_BY**(in\_table, 'sum\_column [alias] [ASC | DESC]')

Performs summary (aggregate) functions to groups of rows in a PV-WAVE table variable.

**ORDER\_BY**(in\_table, 'col\_1 [ASC | DESC] [, col\_2 [ASC | DESC]] ... [, col\_n [ASC | DESC]]')

Sorts the rows in a PV-WAVE table variable to create a new table.

**QUERY\_TABLE** ( table, ' [Distinct] \* | col<sub>i</sub> [alias] [, ..., col<sub>n</sub> [alias]] [Where cond] [Group By col<sub>g</sub><sub>i</sub> [,... col<sub>g</sub><sub>n</sub>]] | [Order By col<sub>o</sub><sub>i</sub> [direction][,...,col<sub>o</sub><sub>n</sub> [direction]]] ' )

Subsets a table created with the BUILD\_TABLE function.

**UNIQUE** (vec)

Returns a vector (one-dimensional array) containing the unique elements from another vector variable.

---

## Transcendental Mathematical Functions

### ACOS (x)

Returns the arc-cosine of x.

### ALOG (x)

Returns the natural logarithm of x.

### ALOG10 (x)

Returns the logarithm to the base 10 of x.

### ASIN (x)

Returns the arcsine of x.

### ATAN (x [, y] )

Calculates the arctangent of the input variable (s).

### COS (x)

Calculates the cosine of the input variable.

### COSH (x)

Calculates the hyperbolic cosine of the input variable.

### EXP (x)

Raises e to the power of the value of the input variable.

### SIN (x)

Returns the sine of the input variable.

### SINH (x)

Returns the hyperbolic sine of the input variable.

### SQRT (x)

Calculates the square root of the input variable.

### TAN (x)

Returns the tangent of the input variable.

### TANH (x)

Returns the hyperbolic tangent of the input variable.

---

## VDA Tools Routines

### Navigator

Starts the Navigator.

### WzAnimate, var

Starts a VDA Tool used for animating a sequence of images.

### WzColorEdit [, var1[, var2, var3]]

Starts a VDA Tool used for editing the image and plot color tables used in other VDA Tools.

### WzContour, var

Starts a VDA Tool used for plotting contours.

### WzExport, var

Starts a VDA Tool used for exporting a PV=WAVE variable to an external file in a specified format.

### WzHistogram, var

Starts a VDA Tool used for plotting a histogram.

### WzImage, var

Starts a VDA Tool used for displaying image data.

### WzImport [, var<sub>1</sub>, var<sub>2</sub>, ... , var<sub>n</sub>]

Starts a VDA Tool used for importing data into PV=WAVE.

### WzMultiView

Starts a VDA Tool used to display multiple plots.

### WzPlot, var<sub>1</sub> [, var<sub>2</sub>, ... , var<sub>n</sub>]

Starts a VDA Tool used for 2D plotting.

### WzPreview [, filename]

Starts a VDA Tool used to view an ASCII file's contents and select which parts of the file are to be read in as PV=WAVE variables.

### WzSurface, z [, x, y]

Starts a VDA Tool used for surface plots.

[WzTable](#), var

Starts a VDA Tool used for creating an editable 2D array of cells containing string data.

[WzVariable](#)

Starts a VDA Tool used for viewing and exporting variables.

---

## ***VDA Tools Manager Routines***

---

**NOTE** For detailed information on the following routines, please refer to the *PV-WAVE Application Developer's Guide*.

---

**TmAddSelectedVars**, tool\_name, var\_name

Adds selected variables from a VDA Tool to the list of selected variables in the Tools Manager.

**TmAddVar**, tool\_name, var\_name

Adds a variable to a VDA Tool.

**TmAxis**, tool\_name

Adds axes to a VDA Tool.

**TmCodeGen**, string

Writes a specified string to the code generation file.

**TmCopy**, tool\_name

Copies the selected graphical elements from the specified VDA Tool to the clipboard.

**TmCut**, tool\_name

Cuts the selected graphical elements from the specified VDA Tool and moves them to the clipboard.

**TmDelVar**, tool\_name [, var\_names]

Removes variables from a VDA Tool.

**TmDelete**, tool\_name

Permanently deletes the selected graphical elements from the specified VDA Tool.

**TmDeselectVars**

Clears the current list of selected variables.

**TmDynamicDisplay**, indices

Displays selected data in all active VDA Tools, provided that the VDA Tools can display the related variable(s).

**TmEndCodeGen**

Closes the file in which generated code is written.

**TmEnumerateAttributes**(tool\_name, item)

Obtains the attributes for a specified graphical element, variable, or other item in a VDA Tool.

**TmEnumerateItems**(tool\_name)

Obtains the items defined for a specified VDA Tool.

**TmEnumerateMethods**(tool\_name)

Obtains the methods that were set for a VDA Tool.

**TmEnumerateSelectedVars**( )

Returns the names of variables on the selected variables list.

**TmEnumerateToolNames**( )

Returns all the registered VDA Tool names.

**TmEnumerateVars**(tool\_name)

Returns all the variables associated with an instance of a VDA Tool.

**TmExecuteMethod**, tool\_name,  
method\_name

Executes a method that was set by TmSetMethod.

**TmExport**, variable\_names,  
destination\_tool\_names

Exports \$MAIN\$-level variables to specified VDA Tools or to all currently active VDA Tools.

**TmExportSelection**, destination\_tool\_names

Exports the contents of the variable selection list to specified VDA Tools.

**TmGetAttribute**(tool\_name, item, attr\_name)  
Returns the value that was set for an attribute in a VDA Tool instance.

**TmGetMessage**( [message\_file], message\_code)  
Loads a string resource file into the resource database and extracts a message string from the database.

**TmGetMethod**(tool\_name, method\_name)  
Returns the data structure of the specified method.

**TmGetTop**(tool\_name)  
Gets the top-level widget ID for a VDA Tool.

**TmGetUniqueToolName**(tool\_name)  
Returns a unique name for a particular instance of a specified VDA Tool.

**TmGetVarMainName**(tool\_name, local\_variable)  
Returns the \$MAIN\$ level name of a variable.

**TmInit**  
Initializes the VDA Tools Manager layer.

**TmList**(tool\_name)  
Creates a list item.

**TmListAppend**, tool\_name, list\_name, item  
Adds a new item at the end of the specified list.

**TmListClear**, tool\_name, list\_name  
Resets a specified list to its initial state, clearing all previously defined items.

**TmListDelete**, tool\_name, list\_name [, pos]  
Deletes an item in the specified list.

**TmListDestroy**, tool\_name, list\_name  
Clears all items and destroys the list.

**TmListExtend**, tool\_name, list\_name  
Extends the specified list by adding empty items.

**TmListGetMethod**(tool\_name, list\_name, method\_name)  
Returns the procedure name associated with the specified list method name.

**TmListInsert**, tool\_name, list\_name, item, pos  
Inserts a new item into the specified list.

**TmListReplace**, tool\_name, list\_name, item, pos  
Replaces an item in a list with a new item.

**TmListRetrieve**(tool\_name, list\_name)  
Gets the items currently set in the specified list.

**TmListSetMethod**, tool\_name, list\_name, method\_name, method  
Sets the method procedure name for a specific list method.

**TmPaste**, tool\_name  
Pastes the graphical elements from the clipboard to the specified VDA Tool.

**TmRegister**, unique\_name, topShell  
Registers a VDA Tool with the Tools Manager.

**TmRestoreTemplate**(tool\_name, filename)  
Restores a saved VDA Tool template.

**TmRestoreTools**(filename)  
Restores the VDA Tools that were saved with the TmSaveTools procedure.

**TmSaveTools**, filename [, tool\_names]  
Saves the specified VDA Tools in a file.

**TmSetAttribute**(tool\_name, item, attr\_name, attr\_value)  
Set an attribute for an item in the given VDA Tool.

**TmSetMethod**, tool\_name, method\_name, method\_call  
Sets a method for a given VDA Tool.

TmStartCodegen, filename  
Opens a file into which PV-WAVE code is written.

TmUnregister, tool\_name  
Removes the specified VDA Tool from the Tools Manager registry.

---

## ***VDA Tools Manager Graphical Element Routines***

---

**NOTE** For detailed information on the following routines, please refer to the *PV-WAVE Application Developer's Guide*.

---

TmAddGrael, tool\_name, grael\_name  
Adds a graphical element to the graphical element list for the specified instance of a VDA Tool.

TmAddSelectedGrael, tool\_name, grael\_name  
Adds a graphical element to the graphical element selection list.

TmAxis, tool\_name  
Adds axes to a VDA Tool.

TmBitmap, tool\_name, bitmap\_name  
Adds a bitmap (2D array) to a VDA Tool.

TmBottomGrael, tool\_name, grael\_name  
Sets the specified graphical element to be on the bottom of the display list (displayed behind the other graphical elements).

TmDelGrael, tool\_name, grael\_name  
Removes a specified graphical element from the list of graphical elements associated with a VDA Tool instance.

TmDelSelectedGraels, tool\_name, grael\_name  
Deletes a graphical element from the list of selected graphical elements.

TmEnumerateGraelMethods(tool\_name, grael\_name)  
Obtain a list of all the methods set for a graphical element in a specified VDA Tool.

TmEnumerateGraels(tool\_name)  
Returns all the graphical elements that were set for a given VDA Tool.

TmEnumerateSelectedGraels(tool\_name)  
Obtains a list of graphical elements or other items currently on the graphical items selection list.

TmExecuteGraelMethod, tool\_name, grael\_name, method\_name  
Executes a method for a graphical method based on the method name.

TmGetGraelMethod(tool\_name, grael\_name, method\_name)  
Obtains the data structure for the specified method.

TmGetGraelRectangle(tool\_name, grael\_name)  
Returns the rectangular boundary of a graphical element.

TmGetUniqueGraelName(tool\_name, grael\_name)  
Obtains a unique name based on the name of the specified graphical element.

TmGroupGraels(tool\_name, grael\_names)  
Groups a number of selected graphical elements as one graphical element with a unique name.

TmLegend, tool\_name  
Adds a legend to a VDA Tool. The exact size and position of the legend is determined interactively by the user.

TmLine, tool\_name  
Adds a line to a VDA Tool. The exact length and position of the line is determined interactively by the user.

TmRect, tool\_name

Adds a rectangle to a VDA Tool. The exact size and position of the rectangle is determined interactively by the user.

TmSetGraelMethod, tool\_name, grael\_name, method\_name, method\_value

Sets the name of the method procedure for a given method name and graphical element.

TmSetGraelRectangle, tool\_name, grael\_name, rectangle

Sets the selection rectangle for a graphical element, or a set of graphical elements.

TmText, tool\_name

Adds text to a VDA Tool. The position of the text and the text itself are determined interactively by the user.

TmTopGrael, tool\_name, grael\_name

Sets the specified graphical element to be at the top of the display list (displayed in front of other graphical elements).

TmUngroupGraels, tool\_name, group\_name

Ungroups a group of graphical elements.

---

## VDA Utilities Routines

---

**NOTE** For detailed information on the following routines, please refer to the *PV-WAVE Application Developer's Guide*.

---

WoAddButtons, toolname, buttons

Adds a bank of buttons to a button bar.

WoAddMessage, toolname, message\_key

Adds a message to a message area created by WoMessage.

WoAddStatus, toolname, status\_key

Display a message in the status bar of a VDA Tool.

WoBuildResourceFilename(file)

Returns the full path name for a specified resource file.

WoButtonBar(parent, toolname, [buttons])

Creates a predefined, two-row button bar that can be included in a VDA Tool.

WoButtonBarSet, toolname, descriptor, setting

Changes the setting of a button in a button bar.

WoButtonBarSetSensitivity, toolname, descriptor, sensitivity

Sets the sensitivity of one or more buttons on a button bar.

WoCheckFile(file)

Confirms if a file is readable or writable.

WoColorButton(parent)

Creates a button that brings up a color table dialog box used to set colors in a VDA Tool. The button has an associated color pixmap that reflects the currently selected color.

WoColorButtonGetValue(wid)

Gets the currently selected color index from a color button created by WoColorButton.

- WoColorButtonSetValue(wid, color)**  
Sets the current color index for a color button created by **WoColorButton**, and updates the color button's color pixmap.
- WoColorConvert(color)**  
Convert from a long RGB value to an index into the current color table, or from an index in the current color table to an RGB value.
- WoColorGrid(parent)**  
Creates a grid of color squares from the current color table.
- WoColorGridGetValue(wid, index, num\_values)**  
Gets the color indices for a range of colors in a color grid.
- WoColorGridSetValue, wid, index, color**  
Sets the color indices for a range of colors in the color grid.
- WoColorWheel(tool\_name, color\_index, value\_changed\_cb)**  
Creates a color wheel that can be used to modify a single color in the current color table.
- WoConfirmClose, wid, tool\_name**  
Displays a dialog box requiring the user to confirm a window close action.
- WoDialogStatus, toolname, status**  
Saves or restores the status of a dialog box by saving or restoring the state of its widgets as stored in the Tools Manager.
- WoFontOptionsMenu(parent, toolname)**  
Creates an option menu with the standard list of software (vector-drawn) fonts found in PV-WAVE.
- WoFontOptionsMenuGetValue(wid)**  
Gets the software font command for the currently selected font.
- WoFontOptionsMenuSetValue, wid, font**  
Sets the current font and updates the font option menu.
- WoGenericDialog(parent, topLayout [,callback])**  
Creates a generic dialog box for use in VDA Tools.
- WoGetToolNameFromTitle(window\_title)**  
Gets the unique name of a VDA tool given the unique window title of the VDA Tool.
- WoGetUniqueWindowTitle(primary, secondary)**  
Given a window title, adds a numeric suffix to make the title unique.
- WoLabeledText(parent, label\_names, verify\_callback)**  
Creates a group of aligned text widgets (widgets with a label and a text field).
- WoLinestyleOptionsMenu(parent, toolname)**  
Creates an option menu for selecting linestyles.
- WoLinestyleOptionsMenuGetValue(wid)**  
Gets the currently selected linestyle.
- WoLineStyleOptionsMenuSetValue, wid, linestyle**  
Sets the option menu to a specified linestyle.
- WoLoadResources, file**  
Loads resources and strings from a file for VDA tools.
- WoLoadStrings, file**  
Loads strings from a resource file for use by the VDA tools.
- WoMenuBar(parent, toolname [,menus])**  
Create a menu bar for a VDA Tool.
- WoMenuBarSetSensitivity, toolname, pane\_index, item\_index, sensitivity**  
Sets the sensitivity of one or more items in a menu.
- WoMenuBarSetToggle, tool\_name, pane\_index, item\_index, value**  
Sets the status of a menu toggle button.

**WoMessage**(parent, toolname)  
Creates a message area for a VDA Tool

**WoSaveAsPixmap**, tool\_name, varname  
Saves graphics from a specified VDA Tool as a pixmap.

**WoSetCursor**, tool\_name  
Changes the cursor for a VDA Tool.

**WoSetToolIcon**, tool\_name, icon  
Assigns a pixmap to be the icon for a VDA Tool.

**WoSetWindowTitle**, tool\_name, window\_title  
Specifies a unique title for a VDA Tool window.

**WoStatus**(parent, toolname)  
Create a status bar for a VDA Tool.

**WoVariableOptionsMenu**(parent, toolname)  
Creates an option menu containing the names of all of the variables associated with the current tool.

**WoVariableOptionsMenuGetValue**(wid)  
Gets the currently selected variable name from an option menu that was created with the **WoVariableOptionsMenu** function.

**WoVariableOptionsMenuSetValue**, wid, value  
Sets the current selection in the variable option menu.

---

## **View Setup Routines**

### **CENTER\_VIEW**

Sets system viewing parameters to display data in the center of the current window (a convenient way to set up a 3D view).

**SET\_VIEW3D**, viewpoint, viewvector, perspective, izoom, viewup, viewcenter, winx, winy, xr, yr, zr

Generates a 3D view, given a view position and a view direction.

### **T3D**

Standard Library procedure that accumulates one or more sequences of translation, scaling, rotation, perspective, or oblique transformation and stores the results in the system variable !P.T.

**VIEWER**, win\_num, xsize, ysize, size\_fac, xpos, ypos, colors, retain, xdim, ydim, zdim

Lets users interactively define a 3D view, a slicing plane, and multiple cut-away volumes for volume rendering. (Creates a View Control and a View Orientation window in which to make these definitions.)

---

## Virtual Reality Modeling Language (VRML) Routines

**VRML\_AXIS**, origin [, length, range]

Adds an axis to a VRML world.

**VRML\_CAMERA**, position

Positions a VRML camera for rendering the VRML view.

**VRML\_CLOSE**

Closes the VRML file.

**VRML\_CONE**

Creates a VRML cone.

**VRML\_CUBE**

Positions a VRML cube in the world.

**VRML\_CYLINDER**

Positions a VRML cylinder in the world.

**VRML\_LIGHT**, position

Sets up the light source for a VRML world.

**VRML\_LINE**, x, y, z

Creates a VRML line object.

**VRML\_OPEN**

Opens a VRML file and writes out header information consistent with VRML formatting.

**VRML\_POLY**, vlist, plist

Creates a VRML polyline node, based on PV-WAVE's variables for vertex list and polygon list.

**VRML\_SPHERE**

Creates a sphere in a VRML world.

**VRML\_SPOTLIGHT**, position

Creates a VRML spotlight.

**VRML\_SURFACE**, z [, x, y]

Creates a VRML surface node based on PV-WAVE-type variables.

**VRML\_TEXT**, text

Creates a VRML text object in an open VRML file.

---

## Volume Manipulation Routines

**AFFINE**(a, b, [c])

Applies an affine transformation to an array.

**BLOB**(a, i, b)

Isolates a homogeneous region in an array.

**BLOBCOUNT**(a, b)

Counts homogeneous regions in an array.

**BOUNDARY**(a,r)

Computes the boundary of a region in an array.

**DERIVN**(a, n)

Differentiates a function represented by an array.

**MOMENT**(a, i)

Computes moments of an array.

**NEIGHBORS**(a, i)

Finds the neighbors of specified array elements.

**RESAMP**(array, dim<sub>1</sub>, ..., dim<sub>n</sub>)

Resamples an array to new dimensions.

**SLICE\_VOL** (volume, dim, cut\_plane)

Returns a 2D array containing a slice from a 3D volumetric array.

**VOL\_PAD** (volume, pad\_width)

Returns a 3D volume of data padded on all six sides with zeroes.

**VOL\_TRANS** (volume, dim, trans)

Returns a 3D volume of data transformed by a 4-by-4 matrix.

---

## Volume Rendering Routines

**RENDER** (object1, ..., objectn)

Generates a ray-traced rendered image from one or more predefined objects.

**VECTOR\_FIELD3**, vx, vy, vz, n\_points

Plots a 3D vector field from three arrays.

**VOL\_MARKER**, vol, n\_points

Displays colored markers scattered throughout a volume.

**VOL\_REND** (volume, imgx, imgy)

Renders volumetric data in a translucent manner.

---

## WAVE Widgets Routines

**NOTE** For detailed information on the following routines, please refer to the *PV-WAVE Application Developer's Guide*.

---

**WwAlert**(parent, label [, answers])

Creates a modal (blocking) or modeless (non-blocking) popup alert box containing a message and optional control buttons.

**WwAlertPopdown**, wid

Destroys an alert box.

**WwButtonBox** (parent, labels, callback)

Creates a horizontally or vertically oriented box containing push buttons.

**WwCallback**(wid, callback, reason, client\_data)

Adds or removes a WAVE Widgets callback.

**WwCommand** (parent, enteredCallback, doneCallback)

Creates a command window.

**WwControlsBox** (parent, labels, range, changedCallback)

Creates a box containing sliders.

**WwDialog** (parent, label, OKCallback, CancelCallback, HelpCallback)

Creates a blocking or nonblocking dialog box.

**WwDrawing** (parent, windowid, drawCallback, wsize, dsize)

Creates a drawing area, which allows users to display graphics generated by PV-WAVE.

**WwFileSelection** (parent, OKCallback, CancelCallback, HelpCallback)

Creates a file selection widget, which lets the user display the contents of directories and select files.

**WwGenericDialog**(parent, layout [, labels] [, callback])

Creates a generic dialog box that can be filled with custom widgets.

**WwGetButton**(event)

Obtains the index of a pressed or released button passed as an event structure by a WAVE Widgets event handler.

**WwGetKey**(event)

Obtains the ASCII value of a pressed or released key passed as an event structure by a WAVE Widgets event handler.

**WwGetPosition**(event)

Obtains the coordinates of a selected point inside a widget. The selected point coordinates are passed in an event structure by a WAVE Widgets event handler.

**WwGetValue** (widget)

Returns a specific value for a given widget.

**WwHandler**(wid, handler [, mask] [, userdata])

Adds or removes a WAVE Widgets event handler from a widget.

- WwInit** (app\_name, appclass\_name, workarea [, destroyCallback])  
Initializes the WAVE Widgets environment, opens the display, creates the first top-level shell, and creates a layout widget.
- WwLayout** (parent)  
Creates a layout widget that is used to control the arrangement of other widgets.
- WwList** (parent, items, selectedCallback, defaultCallback)  
Creates a scrolling list widget.
- WwListUtils**(wid [, param1[, param2]])  
Manages the contents of a list widget.
- WwLoop**  
Handles the dispatching of events and calling of PV-WAVE callbacks.
- WwMainWindow** (parent, workarea, [destroyCallback])  
Creates a top-level window and a layout widget.
- WwMenuBar** (parent, items)  
Creates a menu bar.
- WwMenuItem** (parent, item, value [, callback])  
Adds, modifies, or deletes specified menu items.
- WwMessage** (parent, label, OKCallback, CancelCallback, HelpCallback)  
Creates a blocking or nonblocking message box.
- WwMultiClickHandler**(wid, handler [, userdata])  
Adds or removes a multi-click event handler.
- WwOptionMenu** (parent, label, items)  
Creates an option menu.
- WwPickFile**(parent [, HelpCallback ] )  
Creates a modal file selection dialog that blocks until a file name has been selected.
- WwPopupMenu** (parent, items)  
Creates a popup menu.
- WwPreview**, parent, confirmCallback, clearCallback  
Creates an ASCII data preview widget.
- WwPreviewUtils**(wid [, param1, param2, param3])  
Manages the contents of a preview widget.
- WwRadioBox** (parent, labels, callback)  
Creates a box containing radio buttons.
- WwResource**([resvar])  
Queries, creates, saves, or modifies the widget resource database.
- WwSeparator**( parent )  
Creates a horizontal or vertical line that separates components in a graphical user interface.
- WwSetCursor**(wid, cursor)  
Sets the cursor for a widget.
- WwSetValue** (widget, [value])  
Sets the specified value for a given widget.
- WwTable** (parent, callback [, variable])  
Creates an editable 2D array of cells containing string data, similar to a spreadsheet.
- WwTableUtils**(wid [, param1, ..., param9])  
Manages the contents of a table widget.
- WwText** (parent, verifyCallback)  
Creates a text widget that can be used for both single-line text entry or as a full text editor. In addition, this function can create a static text label.
- WwTimer**(time, timer\_proc [, userdata])  
Registers a WAVE Widgets timer procedure.
- WwToolBox** (parent, labels, callback)  
Creates an array of graphic buttons (icons).

---

## WAVE Widget Utilities

**WgAnimateTool**, image\_data [, parent  
[, shell ]]  
Creates a window for animating a sequence of images.

**WgCbarTool** [, parent [, shell [, windowid  
[, movedCallback], [, range]]]]  
Creates a simple color bar that can be used to view and interactively shift a PV-WAVE color table.

**WgCeditTool** [, parent [, shell ]]  
Creates a full-featured set of menus and widgets enclosed in a window; this window allows you to edit the values in PV-WAVE color tables in many different ways.

**WgCtTool** [, parent [, shell ]]  
Creates a simple widget that can be used interactively to modify a PV-WAVE color table.

**WgIsoSurfTool**, surface\_data [, parent  
[, shell ]]  
Creates a window with a built-in set of controls; these controls allow you to easily view and modify an iso-surface taken from a three-dimensional block of data.

**WgOrbit**, vertices, polygons, parent, shell  
Creates an interactive window for viewing objects.

**WgMovieTool**, image\_data [, parent [, shell  
[, windowid [, rate]]]]  
Creates a window that cycles through a sequence of images.

**WgSimageTool**, image\_data [, parent  
[, shell ]]  
Creates two windows: 1) a scrolling image window and 2) an optional smaller window that shows a reduced view of the entire image.

**WgSliceTool**, block\_data [, parent  
[, last\_slice [, shell ]]  
Creates a window with a built-in set of controls; these controls allow you to easily select and view “slices” from a three-dimensional block of data.

**WgStripTool** [, x, y1, y2, ... , y10, parent  
[, shell ]]  
Creates a window that displays data in a style that simulates a real-time, moving strip chart.

**WgSurfaceTool**, surface\_data [, parent  
[, shell ]]  
Creates a surface window with a built-in set of controls: these controls allow you to interactively modify surface parameters and view the result of those modifications.

**WgTextTool** [, parent [, shell ]]  
Creates a scrolling window for viewing text from a file or character string.

---

## Widget Toolbox Routines

**NOTE** For detailed information on the following routines, please refer to the *PV-WAVE Application Developer’s Guide*.

**WtAddCallback** (widget, reason, callback  
[, client\_data ] )  
Registers a PV-WAVE callback routine for a given widget.

**WtAddHandler** (widget, eventmask, handler  
[, client\_data ] )  
Registers the X event handler function for a given widget.

**WtClose** (widget)  
Closes the current Xt (Motif) session, and destroys all children of the top-level widget created in **WtInIt**. This routine can also be used to destroy additional widget trees.

**WtCreate** (name, class, parent [, argv] )  
Creates a widget or shell instance specified by widget class.

**WtCursor** (function, widget [, index] )  
Sets or changes the cursor.

**WtGet** (widget [, resource] )  
Retrieves widget resources.

**WtInit** (app\_name, appclass\_name [, Xserverargs ...] )  
Initializes the Widget Toolbox and the Xt toolkit, opens the display, and creates the first top-level shell.

**WtInput** (function [, parameters] )  
Registers a PV-WAVE input source handler procedure.

**WtList** (function, widget [, parameters] )  
Controls the characteristics of scrolling list widgets.

**WtLookupString**(event)  
Maps a KeyPress or KeyRelease event to its KeyEvent structure (and optionally, to its Keysym) when a user presses a key.

**WtLoop**  
Handles the dispatching of events and calling of PV-WAVE callback routines.

**WtMainLoop**  
Handles the dispatching of events.

**WtPointer** (function, widget [, parameters] )  
The pointer utility function.

**WtPreview**(action, widget)  
Handles utility functions for the preview widget (XvnPreview).

**WtProcessEvent**( )  
Handles the dispatching of a Widget Toolbox event.

**WtResource**([resvar])  
Queries, creates, saves, or modifies the widget resource database.

**WtSet** (widget, argv)  
Sets widget resources.

**WtTable** (function, widget [, parameters] )  
Modifies an xbaeMatrix class widget.

**WtTimer** (function, params, [client\_data] )  
Registers a callback function for a given timer.

**WtWorkProc** (function, parameters)  
Registers a PV-WAVE work procedure for background processing.

---

## **Window Routines**

### **WCOPY** ( [window\_index] )

(Microsoft Windows Only) Copies the contents of a graphics window onto the Clipboard.

### **WDELETE** [, window\_index]

Deletes the specified window.

### **WINDOW** [, window\_index]

Creates a window for the display of graphics or text.

### **WMENU** (strings)

Displays a menu inside the current window whose choices are given by the elements of a string array and which returns the index of the user's response.

### **WPASTE** ( [window\_index] )

(Microsoft Windows Only) Pastes the contents of the Clipboard into a graphics window.

### **WPRINT** [, window\_index]

(Microsoft Windows Only) Prints the contents of a specified window.

### **WREAD\_DIB** ( [window\_index] )

(Microsoft Windows Only) Loads a Device Independent Bitmap (DIB) from a file into a graphics window.

### **WREAD\_META** ( [window\_index] )

(Microsoft Windows Only) Loads a Windows metafile (WMF) into a graphics window.

### **WSET** [, window\_index]

Used to select the current window to be used by the graphics and imaging routines.

### **WSHOW** [, window\_index [, show]]

Exposes or hides the designated window. It does not automatically make the designated window the active window.

### **WWRITE\_DIB** ( [window\_index] )

(Microsoft Windows Only) Saves the contents of a graphics window to a file as a Device Independent Bitmap (DIB).

### **WWRITE\_META** ( [window\_index] )

(Microsoft Windows Only) Saves the contents of a graphics window to a file as a Windows metafile (WMF).

## ***Procedure and Function Reference***

This chapter contains detailed descriptions of the procedures and functions distributed with PV-WAVE. Most of these system procedures and functions are proprietary. However, you have access to the source code for some routines—such routines are called Standard Library procedures and functions.

### **Standard Library Routines**

Standard Library procedures and functions are designated as such in their descriptions. The code for these routines can be found in:

(UNIX)        <wavedir>/lib/std  
(OpenVMS)   <wavedir>:[LIB.STD]  
(Windows)   <wavedir>\lib\std

Where <wavedir> is the main PV-WAVE directory.

### **Users' Library Routines**

Additional routines that have been contributed by PV-WAVE users comprise the Users' Library. For the names of these routines, list the Users' Library in:

(UNIX)        <wavedir>/lib/user  
(OpenVMS)   <wavedir>:[LIB.USER]  
(Windows)   <wavedir>\lib\user

Where <wavedir> is the main PV-WAVE directory.

Users' Library routines are not covered in the *PV-WAVE Reference*; use the documentation available in the .pro source file for each routine.

For more information, see the *PV-WAVE Programmer's Guide*.

---

## ABS Function

Returns the absolute value of  $x$ .

### Usage

$result = ABS(x)$

### Input Parameters

$x$  — The value that is evaluated. May be of any dimension.

### Returned Value

$result$  — The absolute value of  $x$ .

### Keywords

None.

### Discussion

ABS is defined by:

$$f(x) = |x|$$

If  $x$  is an array, the result has the same dimension. Each element of the returned array contains the absolute value of the corresponding element in the input array.

When  $x$  is a complex number, the result is the magnitude of the complex number:

$$result_i = \sqrt{Real_i^2 + Imaginary_i^2}$$

When  $x$  has a data type of complex, the result is double-precision floating-point. All other data types produce a result with the same data type as  $x$ .

### Example

```
x = [-1, 2, 3, -4, 5]
PRINT, ABS(x)
      1      2      3      4      5
x = COMPLEX(4, 3)
PRINT, ABS(x)
      5.0000000
```

### See Also

See [General Mathematical Functions](#) on page 14.

---

## ACOS Function

Returns the arc-cosine of  $x$ .

### Usage

$result = ACOS(x)$

### Input Parameters

$x$  — The cosine of the desired angle. Cannot be of a complex data type and must be in the range  $-1 \leq x \leq 1$ .

### Returned Value

*result* — Arc-cosine of  $x$ .

### Keywords

None.

### Discussion

The inverse cosine function, or arc-cosine, denoted by  $\cos^{-1}$ , is defined by:

$$y = \cos^{-1}x$$

if and only if

$$\cos y = x$$

where

$$-1 \leq x \leq 1 \text{ and } 0 \leq y \leq \pi$$

The parameter  $x$  can be an array, with the result having the same data type where each element contains the arc-cosine of the corresponding element from  $x$ .

When  $x$  is of double-precision floating-point data type, the result is of the same type. All other data types are converted to single-precision floating-point and yield a floating-point result. The result is an angle, expressed in radians, whose cosine is  $x$ .

Values generated by ACOS range between 0 and  $\pi$ .

## Example

```
x = ACOS(1)
PRINT, x
0
```

## See Also

[COS](#)

For a list of other transcendental functions, see [Chapter 1, \*Functional Summary of Routines\*](#).

---

## ***ADD\_EXEC\_ON\_SELECT Procedure (UNIX)***

Adds a single new item to the EXEC\_ON\_SELECT list.

### ***Usage***

```
ADD_EXEC_ON_SELECT, lun, command
```

### ***Input Parameters***

*lun* — Logical unit number.

*command* — Procedure name.

### ***Description***

A new logical unit number and associated command is added to the EXEC\_ON\_SELECT list. This procedure is designed to be called from an EXEC\_ON\_SELECT callback procedure.

## See Also

[DROP\\_EXEC\\_ON\\_SELECT](#), [EXEC\\_ON\\_SELECT](#), [SELECT\\_READ\\_LUN](#)

---

## ***ADDVAR Procedure***

Creates a variable on the \$MAIN\$ program level and binds a local variable to it.

### **Usage**

*ADDVAR, name, local*

### **Input Parameters**

*name* — A string containing the name of a variable to create on the \$MAIN\$ program level.

### **Output Parameters**

*local* — The name of the local variable that you want to bind to the variable *name* on the \$MAIN\$ program level.

### **Keywords**

None.

### **Example**

This example shows how `ADDVAR` is used to pass a variable from inside a procedure to the \$MAIN\$ program level.

```
PRO test_addvar
    ; Create a scalar variable inside a procedure, then use ADDVAR to pass it to
    ; the top-level procedure $MAIN$.
    ADDVAR, 'sclvar', local
    local = 1.2345
    local = local + 1.
    PRINT, local
END
```

Now, at the `WAVE>` prompt, do the following:

```
test_advar
INFO, /Traceback
    % At $MAIN$.
    ; Verify that you are now on the $MAIN$ program level.
```

```
INFO, /Variables
; This INFO command verifies that the scalar created inside the
; procedure now exists on the $MAIN$ program level.
SCLVAR          FLOAT      =      2.23450
```

## See Also

[DELVAR](#), [UPVAR](#)

---

## AFFINE Function

Standard Library function that applies an affine transformation to an array.

### Usage

*result* = AFFINE(*a*, *b*, [*c*])

### Input Parameters

*a* — An n-dimensional array.

*b* — An invertible (n,n) array.

*c* — An n-element vector (optional).

### Returned Value

*result* — An array representing *a* (and of the same dimensions as *a*) under the coordinate transformation  $y = b \# x + c$ , where *y* and *x* are coordinates for the *result* and for *a*, respectively, which differ from array index coordinates by a simple translation to the array centroid.

### Keywords

None.

### Example

See `wave/lib/user/examples/affine_ex`.

## See Also

[ROT](#), [ROTATE](#), [ROT\\_INT](#)

---

## **ALOG Function**

Returns the natural logarithm of  $x$ .

### **Usage**

*result* = ALOG( $x$ )

### **Input Parameters**

$x$  — The expression that is  $> 0$  which is evaluated. This expression can be an array.

### **Returned Value**

*result* — The natural logarithm of  $x$ .

### **Keywords**

None.

### **Discussion**

ALOG is defined as:

$$y = \log_e x$$

Double-precision floating-point and complex values return a result with the same data type. All other data types are converted to single-precision floating-point and yield a floating-point result.

ALOG handles complex numbers in the following way:

$$Alog(x) \equiv Complex(\log_e(|x|), arctan(x))$$

### **Examples**

```
x = ALOG(10)
```

```
PRINT, x
```

```
2.30259
```

```
x = ALOG(1)
```

```
PRINT, x
```

```
0
```

### **See Also**

[ALOG10](#)

For a list of other transcendental functions, see [Chapter 1, Functional Summary of Routines](#).

---

## **ALOG10 Function**

Returns the logarithm to the base 10 of  $x$ .

### **Usage**

*result* = ALOG10( $x$ )

### **Input Parameters**

$x$  — The expression that is  $> 0$  which is evaluated. This expression can be an array.

### **Returned Value**

*result* — The base 10 logarithm for  $x$ .

### **Keywords**

None.

### **Discussion**

ALOG10 is defined by:

$$y = \log_{10}x$$

Double-precision floating-point and complex values return a result with the same data type. All other data types are converted to single-precision floating-point and yield a floating-point result.

ALOG10 handles complex numbers in the following way:

$$Alog10(x) = Complex(\log_{10}(|x|), \arctan(x))$$

### **Examples**

```
x = ALOG10(10)
```

```
PRINT, x
```

```
1
```

```
x = ALOG10(100)
```

```
PRINT, x
```

2

```
x = ALOG10(50)
PRINT, x
      1.69897
```

## See Also

[ALOG](#)

For a list of other transcendental functions, see [Chapter 1, \*Functional Summary of Routines\*](#).

---

## ASARR Function

Creates an associative array containing specified variables and expressions.

### Usage

*result* = ASARR(*key*<sub>1</sub>, *value*<sub>1</sub>, ... *key*<sub>*n*</sub>, *value*<sub>*n*</sub>)

*result* = ASARR(*keys\_arr*, *values\_list*)

### Input Parameters

*key*<sub>*i*</sub> — One or more strings, each containing the key name for an element of the associative array.

*value*<sub>*i*</sub> — Expressions or variables used to set the values of the associative array elements.

*keys\_arr* — A Fstring array containing one or more key names for elements of the associative array.

*values\_list* — A variable of type list containing expressions or variables used to set the values of the associative array elements. The LIST function is used to create list variables.

### Returned Value

*result* — A new associative array containing the specified elements and their associated names.

## Keywords

None.

## Discussion

An associative array is an array of elements (variables or expressions), each with a unique name. The names act like array subscripts; they let you access the elements of the associative array. An associative array is a distinct data type in PV-WAVE. You can use a method similar to array subscripting to reference the elements of an associative array.

## Example

This example demonstrates how to create an associative array. The INFO and PRINT commands are used to show the contents of the array.

```
asar1 = ASARR('byte', 1B, 'float', 2.2, 'string', '3.3', $
            'struct', {,a:1, b:lindgen(2)})
            ; Create the associative array by specifying the array elements (key names
            ; and values) as separate parameters. Note that each element is of a
            ; different data type.

asar2 = ASARR(['byte', 'float', 'string', 'struct'], $
            LIST(1B, 2.2, '3.3', {,a:1, b:lindgen(2)}))
            ; Create an associative array that is equivalent to the previous one, only
            ; this time the input parameters consist of a string array of key names
            ; and a list array of values.

INFO, asar1, /Full
            ; Show information on the associative array asar1.

ASAR1 AS. ARR = Associative Array(4)
byte  BYTE = 1
struct  STRUCT = ** Structure $1, 2 tags, 12  length:
    A INT    1
    B LONG   Array(2)
float    FLOAT = 2.20000
string   STRING = '3.3'
PRINT, asar1
            ; Print the contents of the associative array asar1.

{'byte' 1 'struct' { 1 0 1} 'float' 2.20000 'string'3.3 }
PRINT, asar2
            ; Print the contents of the associative array asar2.
```

```
{'byte' 1 'struct' {1 0 1} 'float' 2.20000 'string' 3.3 }  
; The contents of the second associative array are the same as the first.
```

## See Also

[ASKEYS](#), [ISASKEY](#), [LIST](#)

---

## ASIN Function

Returns the arcsine of  $x$ .

### Usage

*result* = ASIN( $x$ )

### Input Parameters

$x$  — The sine of the desired angle. Cannot be a complex data type and must be in the range of  $(-1 \leq x \leq 1)$ .

### Returned Value

*result* — The arcsine of  $x$ .

### Keywords

None.

### Discussion

The inverse sine function, or arcsine, denoted by  $\sin^{-1}$ , is defined by:

$$y = \sin^{-1}x$$

if and only if

$$\sin y = x$$

where

$$-1 \leq x \leq 1 \text{ and } -\pi/2 \leq y \leq \pi/2$$

The parameter  $x$  can be an array, with the result having the same data type as  $x$ , where each element contains the arcsine of the corresponding element from  $x$ .

When  $x$  is of double-precision floating-point data type, the result is of the same type. All other data types are converted to single-precision floating-point and yield a floating-point result. The result is an angle, expressed in radians, whose sine is  $x$ .

Values generated by ASIN range between  $-\pi/2$  and  $\pi/2$ .

## Example

```
x = ASIN(0)
PRINT, x
0
```

## See Also

[SIN](#)

For a list of other transcendental functions, see [Chapter 1, \*Functional Summary of Routines\*](#).

---

## ASKEYS Function

Obtains the key names for a given associative array.

### Usage

```
result = ASKEYS(asarr)
```

### Input Parameters

*asarr* — The name of an associative array.

### Returned Value

*result* — A string containing the key names in the given associative array. If the array is empty, an empty string is returned.

### Keywords

None.

## Discussion

A key name is the name associated with an element in an associative array. ASKEYS returns the key names of the elements in an associative array. If the associative array is empty, an empty string is returned. To create an associative array, use the ASARR function.

## Example

ASKEYS is used to obtain the key names in an associative array. The names are used to reference the values of the array to replace the original values with new integer values. The INFO command is then used to show the modified contents of the associative array.

```
asar = ASARR('byte', 1B, 'float', 2.2, 'string', '3.3', $
           'struct', {,a:1, b:lindgen(2)})
keys = ASKEYS(asar)
PRINT, keys
      byte struct float string
      ; These key names appear later in the output of the INFO command.
FOR I = 0, N_ELEMENTS(asar) - 1 DO $
      asar(keys(i)) = i + 10
INFO, asar, /Full
      ; Show the replaced values.
ASAR1AS. ARR = Associative Array(4)
      byte INT = 10
      struct INT = 11
      float INT = 12
      string INT = 13
```

## See Also

[ASARR](#), [ISASKEY](#), [LIST](#)

---

## ASSOC Function

Associates an array definition with a file, allowing random access input and output.

### Usage

```
result = ASSOC(unit, array_definition [, offset])
```

### Input Parameters

*unit* — The file unit to associate with *array\_definition*.

*array\_definition* — An expression that defines the data type and dimensions of the associated data.

*offset* — The offset in the file to the start of the data in the file. For stream files and RMS block mode files, this offset is given in bytes. For RMS record-oriented files, this offset is specified in records.

---

**TIP** The *offset* parameter is useful for skipping past descriptive header blocks in files.

---

### Returned Value

*result* — A variable that associates the array definition with the file.

### Keywords

None.

### Discussion

ASSOC provides a basic method of random access input/output. The associated variable (the one storing the association) is created by assigning the result of ASSOC to a variable. This variable provides a means for mapping a file into vectors or arrays of a specified type and size.

---

**UNIX USERS** ASSOC does not work with UNIX FORTRAN binary files.

---

## Example 1

Assume you have a binary file, `image_file.img`, with five 512-by-512 byte images and a 1024-byte header:

```
OPENR 1, 'image_file.img'
    ; Open the file.
aimage = ASSOC(1, BYTARR(512, 512), 1024)
image1 = aimage(0)
    ; Read the first image.
image5 = aimage(4)
    ; Read the fifth image.
TVSCL, aimage(2)
    ; Display the third image.
fft_image = FFT(aimage(1), -1)
    ; Do an FFT function on the second image.
arow = ASSOC(1, BYTARR(512), 1024)
row100 = arow(99)
    ; Read the 100th row.
PLOT, arow(512)
    ; Plot the first row in the second image.
```

## Example 2

For another example showing how to transfer data into an associated variable, see the *PV-WAVE Programmer's Guide*.

## See Also

[OPEN \(UNIX/OpenVMS\)](#), [OPEN \(Windows\)](#)

---

## ATAN Function

Returns the arctangent of the input.

### Usage

```
result = ATAN(y [, x])
```

### Input Parameters

*y* — The tangent of the desired angle.

*x* — If the second argument is supplied, ATAN returns the angle whose tangent is equal to  $y/x$ . If both arguments are zero, the result is undefined.

### Returned Value

*result* — The angle, in radians, whose tangent is  $y$  (or, optionally  $y/x$ ).

### Discussion

If two parameters are supplied, the angle whose tangent is equal to  $y/x$  is returned. The range of ATAN is between  $-\pi/2$  and  $\pi/2$  for the single argument case and between  $-\pi$  and  $\pi$  if two arguments are given. If  $y$  or  $x$  are double-precision floating, the result of ATAN is also double precision. Arguments are not allowed to be complex. All other types are converted to single-precision floating point and yield floating-point results.

### Example

```
PRINT, !radeg*atan(2,3)
      33.6901
; This result is the angle (in degrees) whose tangent is 2/3.
```

### See Also

[ACOS](#), [ASIN](#), [COS](#), [SIN](#), [TAN](#)

For a list of other transcendental functions, see [Chapter 1, Functional Summary of Routines](#).

---

## AVG Function

Standard Library function that returns the average value of an array. Optionally, it can return the average value of one dimension of an array.

### Usage

*result* = AVG(*array* [, *dim*])

### Input Parameters

*array* — The array that is averaged. This array may be any data type except string.

*dim* — (optional) The specific dimension of *array* that will be averaged. Must be a number that is in the range  $0 \leq dim < n$ , where  $n$  is the number of dimensions in *array*.

### Returned Value

*result* — The average value of *array*. If *dim* is not specified, *result* will be of floating-point type; otherwise, it will be of the same data type as *array*.

If *dim* is specified, *result* is an array containing the average values for all elements of the specified dimension.

### Keywords

None.

### Discussion

AVG is defined as:

$$f(x) = \frac{\sum_{n=1}^n x_n}{n}$$

The optional parameter *dim* allows you to find the average values for one dimension of *array* rather than the whole array. The first dimension in the array is denoted by 0, the second dimension by 1, and so on.

If the dimension you specify is not valid for *array*, the input array is returned as the result.

## Example 1

```
array = INTARR(3, 4)
array(*, 0) = [5, 7, 9]
array(*, 1) = [2, 8, 5]
array(*, 2) = [3, 4, 8]
array(*, 3) = [3, 3, 3]
PRINT, AVG(array)
    5.00000
PRINT, AVG(array, 0)
    7    5    5    3
PRINT, AVG(array, 1)
    3    5    6
```

## Example 2

When *AVG* is called with the dimension parameter, the result is an array with the dimensions of the input array, except for the dimension specified. In this case, each element of the result is the average of the corresponding vector in the input array. For example, if *Array* has dimensions of (3, 4, 5), then the command

```
avg_dim = AVG(array, 1)
```

is equivalent to these commands:

```
avg_dim = FLTARR(3, 5)
FOR j = 0,4 DO BEGIN
    FOR i = 0,2 DO BEGIN
        avg_dim(i,j) = TOTAL(array(i,*,j)) / 4.
    ENDFOR
ENDFOR
```

## See Also

[MAX](#), [MEDIAN](#), [MIN](#), [SQRT](#), [STDEV](#)

---

## AXIS Procedure

Draws an axis of the specified type and scale at a given position.

### Usage

AXIS [[[,  $x$ ],  $y$ ],  $z$ ]

### Input Parameters

$x$ ,  $y$ , and  $z$  — (optional) Scalars giving the coordinates of the new AXIS.

### Keywords

**XAxis** — Specifies how the  $x$ -axis is to be drawn:

- 0 Draws an axis under the plot window, with the tick marks pointing up.
- 1 Draws an axis over the window, with tick marks pointing down.

**YAxis** — Specifies how the  $y$ -axis is to be drawn:

- 0 Draws a  $y$ -axis at the left of the plot window, with tick marks pointing to the right.
- 1 Draws a  $y$ -axis at the right of the plot window, with tick marks pointing to the left.

**ZAxis** — Specifies how the  $z$ -axis is to be drawn:

- 1 Lower-right
- 2 Lower-left
- 3 Upper-left
- 4 Upper-right

Additional keywords let you control many aspects of the plot's appearance. Additional plotting keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

Channel

Position

[XYZ]Range

Charsize	Save	[XYZ]Style
Charthick	Subtitle	[XYZ]Tickformat
Clip	T3d	[XYZ]Ticklen
Color	Thick	[XYZ]Tickname
Data	Tickformat	[XYZ]Ticks
Device	Ticklen	[XYZ]Tickv
Font	Title	[XYZ]Title
Gridstyle	[XY]Axis	[XYZ]Type
Noclip	[XYZ]Charsize	YLabelCenter
Nodata	[XYZ]Gridstyle	YNozero
Noerase	[XYZ]Margin	ZAxis
Normal	[XYZ]Minor	ZValue

---

**Save** — Indicates that the scaling to and from data coordinates established by the call to `AXIS` is to be saved in the appropriate axis system variable, `!X`, `!Y`, or `!Z`. If not present, the scaling is not changed.

## Discussion

If no coordinates are specified, the axis is drawn in its default position as given by the `XAxis`, `YAxis` or `ZAxis` keyword. When drawing an  $x$ -axis, the  $x$ -coordinate is ignored. Similarly, the  $y$  and  $z$  parameters are ignored when drawing their respective axes.

The new scale is saved for use by subsequent overplots if the `Save` keyword is present.

## Example

The following example shows how the `AXIS` procedure can be used with normal or polar plots to draw axes through the origin dividing the plot window into four quadrants:

```
theta = FINDGEN(361) * !Dtor
PLOT, /Polar, XStyle=4, YStyle=4, Title='Nine-Leaved Rose', $
      5 * (COS(9 * theta), theta
```

```
    ; Make a polar plot, suppressing the x- and y-axes with the  
    ; XStyle and YStyle keywords.
```

```
WAIT, 2
```

```
AXIS, 0, 0, XAxis=0, /Data
```

```
    ; Draw an x-axis through data y-coordinate of 0. Because the  
    ; XAxis keyword has a value of 0, the tick marks point down.
```

```
WAIT, 2
```

```
AXIS, 0, 0, 0, YAxis=0, /Data
```

```
    ; Similarly, draw the y-axis through data x-coordinate of 0.
```

## See Also

### [PLOT](#)

For more information and an illustration, see the *PV-WAVE User's Guide*.

---

## BAR Procedure

Plots a 2D bar graph that can include stacked and grouped bars, as well as various color and fill pattern options.

### Usage

BAR, [ *x*, ] *y*

### Input Parameters

*x* — (optional) An array of values to plot along the *x*-axis. The values *y*(*i*) are plotted from *x*(*i*) to *x*(*i*+1).

*y* — An array of values to plot along the *y*-axis (or the *x*-axis if */Horizontal* is specified).

If *y* is a 2D array and the *Stack* keyword is not set, then the first dimension is construed as a group and the second dimension as a bar value.

If *y* is a 2D array and the *Stack* keyword is set, then the first dimension is construed as a stack and the second dimension as a bar value.

If *y* is a 3D array, then the first dimension is construed as a group, the second as the stack, and the third as a bar in a stack in a group. The *Stack* keyword is not relevant.

---

**NOTE** Each of these plotting options is discussed in the *Examples* section.

---

### Keywords

**Barmin** — Sets the value to draw the bars down to. (Default: 0)

**DrawLegendBox** — If nonzero, a box is drawn around the legend. (Default: no box)

**Endpoints** — If set, *y* is a 2D array containing endpoints (minimum and maximum values) and *Grouped* and *Stacked* keywords are not available.

**FillColors** — Specifies a 1D array of color index values. These values specify the colors with which the bars are filled. By default, the bars are filled with solid color. If either *FillSpacing* or *FillOrientation* are used, the bars are filled with lines

instead of solid color. In this case, *FillColors* specifies the line colors. (Default: no fill color)

***FillLinestyle*** — An integer or integer array specifying the style of fill lines. If set to a scalar, all bars are filled with the same linestyle. If set to an array, the linestyle of each bar is mapped, sequentially, to the value of each array index. (Default: !P.Linestyle)

---

**NOTE** This keyword has no effect unless either *FillSpacing* or *FillOrientation* are used.

---

Valid linestyle indices are shown in the following table:

---

Index	X Windows Style	Windows Style
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

---

***FillOrientation*** — A floating point scalar or array specifying the orientation of fill lines, in degrees, counterclockwise from the horizontal. If set to a scalar, the orientation of fill lines in all bars is the same. If set to an array, the orientation of each bar is mapped, sequentially, to the value of each array index. (Default: Horizontal lines)

***FillSpacing*** — A floating point scalar or array specifying the space, in centimeters, between fill lines. If set to a scalar, the spacing between lines in all bars is the same. If set to an array, the spacing for each bar is mapped, sequentially, to the value of each array index. (Default: five times the value of the *FillThick* keyword, converted to centimeters.)

***FillThick*** — A floating point scalar or array specifying the thickness of fill lines. If set to a scalar, all bars are filled with lines of the same thickness. If set to an array, the thicknesses are mapped, sequentially, to the value of each array index. A thickness of 1 is normal, two is double-wide, and so on. (Default: !P.Thick)

---

**NOTE** This keyword has no effect unless either *FillSpacing* or *FillOrientation* are used.

---

***Horizontal*** — If nonzero, the *y* values are shown on the *x*-axis. If zero, the *x* values are shown on the *y*-axis.

***LegendBoxColor*** — An integer specifying the color of the legend box.

***LegendCharSize*** — A floating-point scalar specifying the size of text in the legend. (Default: 1.0).

***LegendLabels*** — An array of strings used to label individual bars. The number of strings in the array must correspond to the number of individual bars.

***LegendPosition*** — A four-element floating point array specifying the position of the legend in normal coordinates.

***LegendTextColor*** — An integer specifying the color of the text in the legend.

***LineColors*** — Specifies a 1D array of color indices.

***OutlineColor*** — An integer specifying the color index to use for the color of the outline for the bars (Default: black)

***Stacked*** — If the input array is a 2D array and the *Stacked* keyword is set, then the first dimension of the input array is construed as a stack and the second dimension as a bar value. See the *Examples* section for examples of stacked and grouped bar charts.

***Width*** — Sets the width of each bar. When set to 1, bars touch each other. When set to 0.5, bars are separated by the width of a bar. (Default: 0.8)

***XTickName*** — An array of strings specifying the names of tick marks. If the bar chart represents simple bars or stacked bars, the tick name corresponds to each one. If the bar chart represents groups of bars or stacked bars, the *XTickName* correspond to each group.

***YTickName*** — An array of strings, for the *y*-axis (or *x*-axis if the *Horizontal* keyword is specified), which specifies both the number of major tick marks (with no minor tick mark) and their labeling.

Additional BAR keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

---

[XY]Range	Noerase	[XY]Title
Position	Title	YLabelCenter

---

## Discussion

The  $x$  and  $y$  variables must be simple numeric types. The dependent axis only shows major tick marks. Tick marks are placed in the center, to the left, and to the right of individual bars or groups of bars. Only the center tick is labeled.

Either the *FillLineStyle* or *FillOrientation* keyword (or both) must be specified to fill the bars with lines.

A stacked bar is a bar that depicts more than one value, where each value is shown on top of the previous value. A group is a set of two or more related bars appearing next to one another on the chart. It is possible to create grouped bars that are also stacked. Refer to the following examples for information on producing stacked and grouped bar charts.

The BAR2D procedure also draws bar graphs; however, it does not permit stacked and grouped bars.

## Examples

This section includes three example plots. First, data is defined for the examples. Then the following bar plots are created:

- A horizontal bar chart with a legend. The bars are filled with colors and patterns.
- A chart of grouped bars with a legend. The bars are filled with colors and patterns.
- A chart of stacked and grouped bars. This plot includes a legend and pattern-filled bars.

### ***Define Data for the Examples***

```
; The following expressions create an array of data to plot (the
; bar values) and tick names for the x and y-axes.
simple = [ 10, 20, 10, 40]
XTickNames=["Quarter 1", "Quarter 2", "Quarter 3", "Quarter 4"]
YTickNames=["$ 0000", "$ 1000", "$ 2000", "$ 3000"]
;
; The following expression creates a 2D array specifying data for
; four groups of bars containing three sets of bars per group.
;
group1 = [ [ 100, 200, 100], [200, 150, 100], [400, 200, 100], $
```

```

    [100, 110, 120]]
;
; The following expression creates a 3D array specifying data for
; two groups of bars containing three stacks with two values per
; stack.
;
group2 = [ [ [10, 20], [30, 40], [100, 60]], $
          [ [30, 10], [50, 50], [60, 40]]]

```

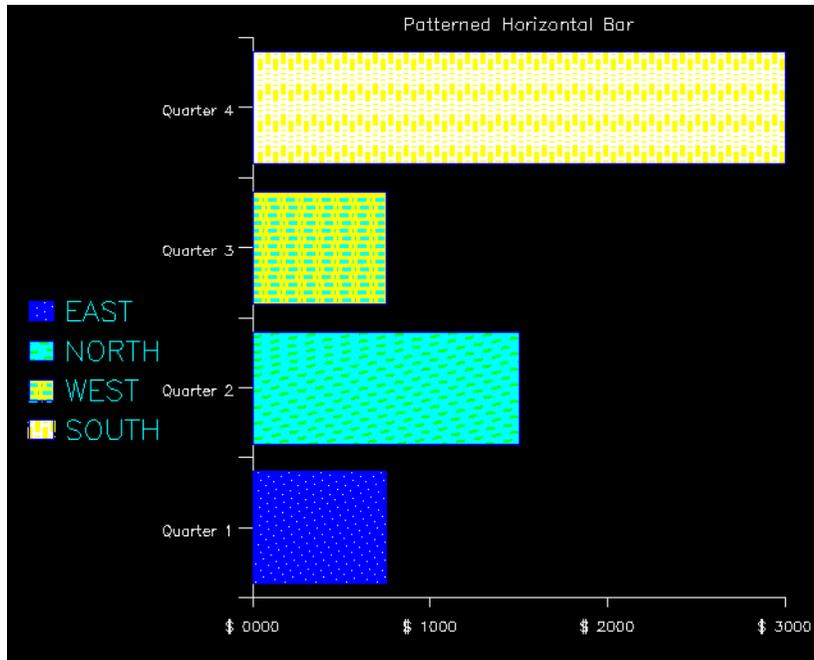
### ***Example 1: Horizontal Bars***

Note that the patterns inside the bars are created by filling the bars with lines of various styles and thicknesses.

```

BAR, simple, XTickName=XTickNames, YTickName=YTickNames, $
    FillOrientation=[30, 10, 0, 90], LineCol=[1,3,5,7], Outline=4, $
    Fillcolors=[4, 5, 7, 1], Filllinestyle=[1,2,3,4], $
    Fillthick=[1,2,3,4], $
    LegendLabel=["EAST", "NORTH", "WEST", "SOUTH"], $
    LegendTextColor=5, LegendCharSize=2, $
    LegendPosition=[0.00, 0.3, 0.22, 0.6], $
    Position=[0.3, 0.1, 0.95, 0.95], $
    /Horizontal, Title="Patterned Horizontal Bar"

```

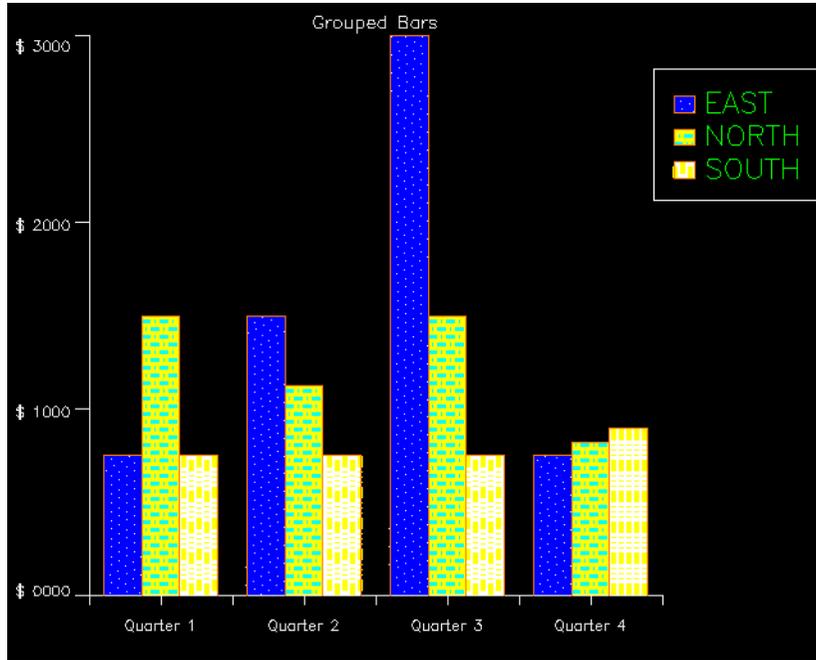


**Figure 2-1** A horizontal bar chart with a legend. The bars are filled with colors and lines.

### ***Example 2: Grouped Bars***

In this example, a 2D array is used to create a chart of grouped bars.

```
BAR, group1, XTickName=XTickNames, YTickName=YTickNames, $
  FillOrientation=[30, 0, 90], LineCol=[1,5,7], Outline=8, $
  Fillcolors=[4, 7, 1], FillLinestyle=[1,3,4], $
  fillthick=[1,3,4], $
  LegendLabel=["EAST", "NORTH", "SOUTH"], LegendTextColor=3, $
  LegendCharSize=2, /DrawLegendBox, $
  LegendPosition=[0.79, 0.7, 0.99, 0.9], $
  Position=[0.1, 0.1, 0.8, 0.95], $
  Title="Grouped Bars"
```

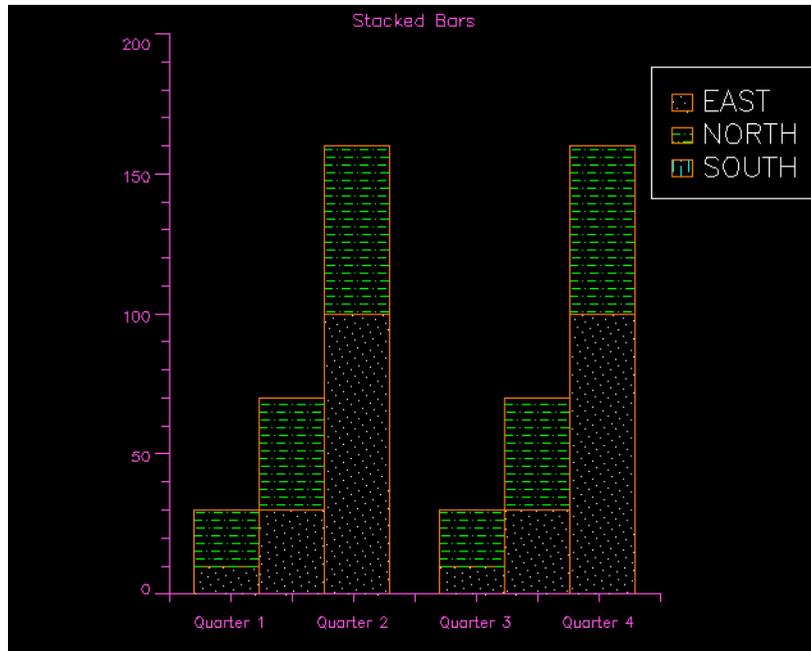


**Figure 2-2** A bar chart with grouped bars and a legend. The bars are filled with colors and line.

### ***Example 3: Stacked Bars***

In this example, a 3D array is used to create a chart of grouped and stacked bars.

```
BAR, group1, XTickName=XTickNames, $
  FillOrientation=[30, 0, 90], LineCol=[1,5,7], Outline=8, $
  filllinestyle=[1,3,4] , $
  LegendLabel=["EAST", "NORTH", "SOUTH"], LegendTextColor=1, $
  LegendCharSize=2.0, /DrawLegendBox,$
  LegendPosition=[0.79, 0.7, 0.99, 0.9], $
  Position=[0.2, 0.1, 0.8, 0.95], $
  /Stacked, Title="Stacked Bars"
```



**Figure 2-3** A bar chart with grouped and stacked bars. This plot includes a legend and line filled bars.

## See Also

[BAR2D](#), [BAR3D](#)

---

## ***BAR2D Procedure***

Creates a 2D bar graph.

### **Usage**

`BAR2D, [x, ] y`

### **Input Parameters**

**x** — (optional) Specifies a 1D array of *x* values. (Default: the index number in the *y* array)

*y* — Specifies a 1D array of *y* values.

## Keywords

**ColumnColors** — An array of color indices specifying the colors to use for the column bars.

**Outline** — If nonzero, each bar is outlined. (Default: no outlining)

Additional BAR keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

---

Background	Normal	[XY]Minor
Charsize	Position	[XY]Range
Charthick	Subtitle	[XY]Style
Color	Thick	[XY]Ticklen
Data	Ticklen	[XY]Tickname
Device	Title	[XY]Ticks
Font	[XY]Charsize	[XY]Title
Gridstyle	[XY]Gridstyle	YLabelCenter
Linestyle	[XY]Margin	

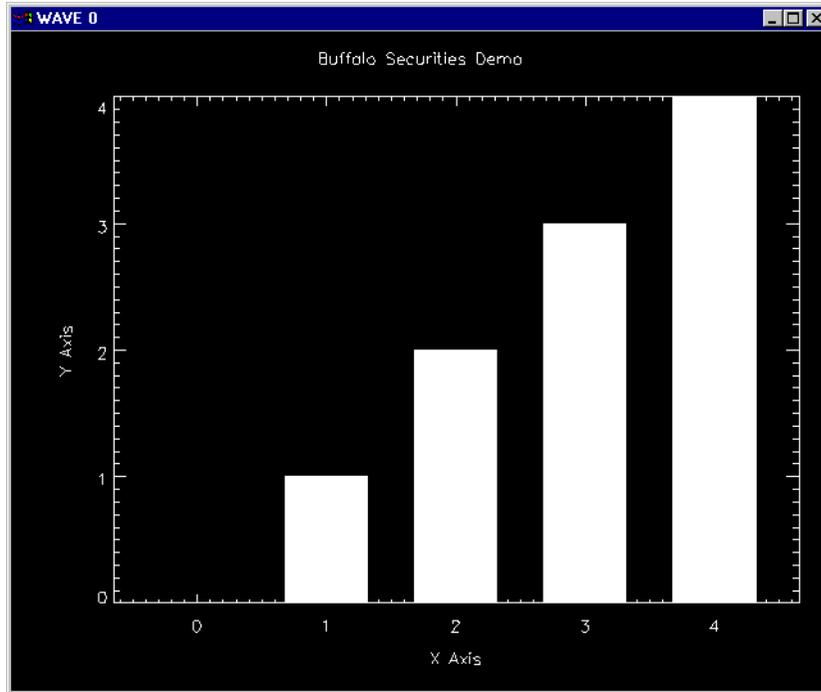
---

## Discussion

This procedure creates a simple bar chart. To create more complex bar charts that include grouped and stacked bars, legends, and pattern-filled bars, see the BAR procedure.

## Examples

```
y = FINDGEN(5)
TEK_COLOR
BAR2D, y, Title= 'Buffalo Securities Demo', Xtitle = 'X Axis', $
    Ytitle = 'Y Axis'
```



**Figure 2-4** A simple 2D bar chart.

## See Also

[BAR](#), [BAR3D](#)

---

## ***BAR3D Procedure***

Creates a 3D bar graph.

### **Usage**

`BAR3D, z`

### **Input Parameters**

`z` — A 2D array of `z` values.

## Keywords

**ColumnColors** — An array of color indices specifying the colors to use for column bars.

**Noshade** — If nonzero, turns off the shading of each bar. (Default: bars are shaded)

**Outline** — If nonzero, each bar is outlined. (Default: no outlining)

**RowColors** — An array of color indices specifying the colors to use for row bars.

Other BAR3D keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

---

<a href="#">Ax</a>	<a href="#">Font</a>	<a href="#">Ticklen</a>	<a href="#">[XYZ]Ticklen</a>
<a href="#">Az</a>	<a href="#">Gridstyle</a>	<a href="#">Title</a>	<a href="#">[XYZ]Tickname</a>
<a href="#">Background</a>	<a href="#">Horizontal</a>	<a href="#">[XYZ]Charsize</a>	<a href="#">[XYZ]Ticks</a>
<a href="#">Charsize</a>	<a href="#">Linestyle</a>	<a href="#">[XYZ]Gridstyle</a>	<a href="#">[XYZ]Title</a>
<a href="#">Charthick</a>	<a href="#">Normal</a>	<a href="#">[XYZ]Margin</a>	<a href="#">YLabelCenter</a>
<a href="#">Color</a>	<a href="#">Position</a>	<a href="#">[XYZ]Minor</a>	<a href="#">ZAxis</a>
<a href="#">Data</a>	<a href="#">Subtitle</a>	<a href="#">[XYZ]Range</a>	<a href="#">ZValue</a>
<a href="#">Device</a>	<a href="#">Thick</a>	<a href="#">[XYZ]Style</a>	

---

## Discussion

The  $z$  parameter is a 2D array of elevation values. The 3D effect is established by modifying the colortable to create darker color values for use on the top and left sides of the bars. By default, the bars are displayed vertically (upward).

## Examples

```
xx = DIST(5)
TEK_COLOR
BAR3D, xx, Title= 'Buffalo Securities Demo', Xtitle = 'X Axis', $
    Ytitle = 'Y Axis', Ztitle = 'Z Axis'
```

## See Also

[BAR](#), [BAR2D](#)

---

## **BESELI Function**

Calculates the Bessel I function for the input parameter.

### **Usage**

*result* = BESELI(*x* [, *n*])

### **Input Parameters**

*x* — The expression that is evaluated.

*n* — (optional) An integer  $\geq 0$ . (Default: 0)

### **Returned Value**

*result* — The Bessel I function for *x*, having the same dimensions as *x*.

### **Keywords**

None.

### **Discussion**

The Bessel I function is one of a mathematical series that arise in solving differential equations for systems with cylindrical symmetry. The Bessel series can be useful in communications and signal processing, since they give the relative amplitude of the spectral components of a frequency-modulated carrier wave.

The Bessel I function is similar to the Bessel J function, except that it is evaluated for imaginary parameters.

BESELI is a numerical approximation to the solution of the differential equation for an imaginary *x*:

$$x^2 * y'' + x * y' - (x^2 + n^2) * y = 0 \quad n \geq 0$$

The BESELI function is a solution of the first kind of (modified) Bessel functions of order *n*. The general solution of the above differential equation using the BESELI function can be shown in the following ways for arbitrary constants A and B:

$$y = A * \text{BESELI}(x, n) + B * \text{BESELI}(x, -n)$$

; Solution for  $n \neq 0, 1, 2, \dots$

$y = A * \text{BESELI}(x, n) + B * \text{BESELK}(x, n)$   
; Solution for all n.

or

$$y = A \cdot \text{BESELI}(x, n) + B \cdot \text{BESELI}(x, n) \cdot \int \frac{dx}{x \cdot (\text{BESELI}(x, n))^2}$$

; Solution for all n.

Note that BESELK may be generated from the BESELI function.

## See Also

[BESELJ](#), [BESELY](#)

For a synopsis of all the Bessel functions, see *Mathematical Handbook of Formulas and Tables*, by Murray R. Spiegel, McGraw-Hill Book Company, New York, 1968.

For sample usage of the Bessel functions in physics, see *Boundary Value Problems*, Second Edition, edited by David L. Powers, Academic Press, New York, 1979, pp. 213-216.

---

## BESELJ Function

Calculates the Bessel J function for the input parameter.

### Usage

*result* = BESELJ(*x* [, *n*])

### Input Parameters

*x* — The expression that is evaluated.

*n* — (optional) An integer. (Default: 0)

### Returned Value

*result* — The Bessel J function for *x*. It is a floating-point data type, with the same dimensions as *x*.

## Keywords

None.

## Discussion

The Bessel J function is one of a mathematical series that arise in solving differential equations for systems with cylindrical symmetry. The Bessel series can be useful in communications and signal processing, since they give the relative amplitude of the spectral components of a frequency-modulated carrier wave.

Bessel J is a Bessel function of the first order, and has a finite limit as  $x$  approaches zero.

BESELJ is a numerical approximation to the solution of the differential equation for a real  $x$ :

$$x^2 * y'' + x * y' + (x^2 - n^2) * y = 0 \quad n \geq 0$$

The BESELJ function is a solution of the first kind of Bessel functions of order  $n$ . The general solution of the above differential equation using the BESELJ function can be shown in the following ways for arbitrary constants A and B:

$$y = A * \text{BESELJ}(x, n) + B * \text{BESELJ}(x, -n)$$

; Solution for  $n \neq 0, 1, 2, \dots$

$$y = A * \text{BESELJ}(x, n) + B * \text{BESELY}(x, n)$$

; Solution for all  $n$ .

or

$$y = A \cdot \text{BESELJ}(x, n) + B \cdot \text{BESELJ}(x, n) \cdot \int \frac{dx}{x \cdot (\text{BESELJ}(x, n))^2}$$

; Solution for all  $n$ .

---

**UNIX USERS** Under UNIX, BESELJ uses the  $j_0(3M)$ ,  $j_1(3M)$ , and  $j_n(3M)$  functions from the UNIX math library. For details about any of these functions, refer to its UNIX man page.

---

## See Also

[BESELI](#), [BESELY](#)

For a synopsis of all the Bessel functions, see *Mathematical Handbook of Formulas and Tables*, by Murray R. Spiegel, McGraw-Hill Book Company, New York, 1968.

For sample usage of the Bessel functions in physics, see *Boundary Value Problems*, Second Edition, edited by David L. Powers, Academic Press, New York, 1979, pp. 213-216.

---

## **BESELY Function**

Calculates the Bessel Y function for the input parameter.

### **Usage**

*result* = BESELY(*x* [, *n*])

### **Input Parameters**

*x* — The expression that is evaluated. This expression must be > 0.

*n* — (optional) An integer. (Default: 0)

### **Returned Value**

*result* — The Bessel Y function for *x*, having the same dimensions as *x*.

### **Keywords**

None.

### **Discussion**

The Bessel Y function is one of a mathematical series that arise in solving differential equations for systems with cylindrical symmetry. The Bessel series can be useful in communications and signal processing, since they give the relative amplitude of the spectral components of a frequency-modulated carrier wave.

Bessel Y is a Bessel function of the second order. Unlike the Bessel J function, it has no finite limit as *x* approaches zero.

BESELY is a numerical approximation to the solution of the differential equation for a real *x*:

$$x^2 * y'' + x * y' + (x^2 - n^2) * y = 0 \quad n \geq 0$$

The BESELY function is a solution of the second kind of Bessel functions of order  $n$ . The general solution of the above differential equation using the BESELJ function is as follows:

$$BESELY(x, n) = \frac{BESELJ(x, n)\cos(n\pi) - BESELJ(x, -n)}{\sin(n\pi)}$$

when  $n \neq 0, 1, 2, \dots$

and

$$BESELY(x, n) = \lim_{p \rightarrow n} \frac{BESELJ(x, p)\cos(p\pi) - BESELJ(x, -p)}{\sin(p\pi)}$$

when  $n = 0, 1, 2, \dots$

---

**UNIX USERS** Under UNIX, BESELY uses the  $j_0$  (3M),  $j_1$  (3M), and  $j_n$  (3M) functions from the UNIX math library. For details about any of these functions, refer to its UNIX man page.

---

## See Also

[BESELI](#), [BESELJ](#)

For a synopsis of all the Bessel functions, see *Mathematical Handbook of Formulas and Tables*, by Murray R. Spiegel, McGraw-Hill Book Company, New York, 1968.

For sample usage of the Bessel functions in physics, see *Boundary Value Problems*, Second Edition, edited by David L. Powers, Academic Press, New York, 1979, pp. 213-216.

---

## **BILINEAR Function**

Standard Library function that creates an array containing values calculated using a bilinear interpolation to solve for requested points interior to an input grid specified by the input array.

### **Usage**

*result* = BILINEAR(*array*, *x*, *y*)

### **Input Parameters**

***array*** — The array that is interpolated. The array must be a two-dimensional floating-point array with dimensions (*n*, *m*).

***x*** — A floating-point array containing the *x* subscripts of *array* (see *Discussion*). Must satisfy the following conditions:

$$0 \leq \min(x) < n$$

$$0 < \max(x) \leq n$$

***y*** — A floating-point array containing the *y* subscripts of *array* (see *Discussion*). Must satisfy the following conditions:

$$0 \leq \min(y) < m$$

$$0 < \max(y) \leq m$$

### **Returned Value**

***result*** — A two-dimensional floating-point array (*n*, *m*) containing the results of the bilinear interpolation for the requested points.

If *x* is of dimension *i* and *y* is of dimension *j*, the result has dimensions (*i*, *j*). In other words, both *x* and *y* will be converted to (*i*, *j*) dimensions. If you want the result to have dimensions (*i*, *j*), then *x* can be either *FLTARR*(*i*) or *FLTARR*(*i*, *j*). This is also true for *y*.

### **Keywords**

None.

## Discussion

Given a two-dimensional input array, BILINEAR uses the specified set of reference points to compute each element of an output array with a bilinear interpolation algorithm.

The array  $x/y$  contains the X/Y subscripts of the elements in *array* that are used for the interpolation:

- If  $x$  is a one-dimensional array, the same subscripts are used in each row of the output array.
- If  $x$  is a two-dimensional array, different X subscripts may be used on each row of the output array.
- If  $y$  is a one-dimensional array, the same subscripts are used in each column of the output array.
- If  $y$  is a two-dimensional array, different Y subscripts may be used on each column of the output array.

Note that specifying  $x$  and  $y$  as two-dimensional arrays allows you to independently define the X and Y location of each point to be interpolated from the original array.

---

**TIP** Using two-dimensional arrays for  $x$  and  $y$  with BILINEAR increases the speed of the algorithm. If  $x$  and  $y$  are one-dimensional, they are converted to two-dimensional arrays before they are returned by the function. This permits them to be reused in subsequent calls to BILINEAR, thereby saving time.

Conversely, BILINEAR can be time consuming for large, one-dimensional arrays.

---

## Example 1

```
array = FLTARR(3,3)
array(1, 1) = 1
      ; Create an array that is all zeros except for a center value of 1.
x = [.1, .2]
y = [.1, .4, .7, .9]
      ; Find the values where x = .1, .2 and y = .1, .4, .7, .9, knowing that
      ; when x = 1 and y = 1, the value in the array is 1, but at all other
      ; points it is zero.

PRINT, BILINEAR(array, x, y)
      0.0100000    0.0200000
      0.0400000    0.0800000
      0.0700000    0.1400000
      0.0900000    0.1800000
```

## Example 2

```
a = DIST(100)
original = SHIFT(SIN(a/5)/EXP(a/50),50,50)
    ; Create original data.

LOADCT, 5
    ; Load color table 5.

TVSCL, original
    ; Display data.

b = FINDGEN(100)
    ; Make an array of linear values from 0 to 99.

PLOT, b
    ; Look at b.

x = b^2 / 100.0
    ; Create exponentially "warped" arrays to be used for spacing on the x-axis.

OPLOT, x
    ; Look at x; it is non-linear.

y = x
    ; Set y equal to x.

result = BILINEAR(original, x, y)
    ; Perform bilinear interpolation from "original" to "result" based
    ; on the (non-linear) spacing characteristics of the indices in x and y.

ERASE
TVSCL, result
    ; Note that the original data has been interpolated in the upper right
    ; corner of "result" due to the non-linearity of the x- and y-axis arrays.
```

## See Also

[CONGRID](#), [INTERPOL](#), [SPLINE](#)

---

## ***BINDGEN Function***

Returns a byte array with the specified dimensions, setting the contents of the result to increasing numbers starting at 0.

### **Usage**

*result* = BINDGEN(*dim*<sub>1</sub> [, *dim*<sub>2</sub>, ... , *dim*<sub>*n*</sub>])

### **Input Parameters**

*dim*<sub>*i*</sub> — The dimensions of the result array. May be any scalar expression. Up to eight dimensions can be specified.

### **Returned Value**

*result* — An initialized byte array. If the resulting array is treated as a one-dimensional array, then its initialization is given by the following:

$$\begin{aligned} \text{array}(i) &= \text{BYTE}(i \text{ MOD } 256) \\ \text{for } i &= 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right). \end{aligned}$$

### **Keywords**

None.

### **Discussion**

Each element of the result array is set to the value of its one-dimensional subscript.

### **Example**

```
a = BINDGEN(4, 2)
      ; Create a byte array.
INFO, a
A      BYTE      = Array(4, 2)
PRINT, a
  0    1    2    3
  4    5    6    7
```

## See Also

[BYTARR](#), [BYTE](#), [CINDGEN](#), [DINDGEN](#), [FINDGEN](#), [INDGEN](#), [LINDGEN](#), [SINDGEN](#)

---

## **BLOB Function**

Standard Library function that isolates a homogeneous region in an array.

### **Usage**

*result* = BLOB(*a*, *i*, *b*)

### **Input Parameters**

*a* — An array of *n* dimensions.

*i* — A vector of *n* integers giving a seed element for the region.

*b* — A two-element vector giving bounds for values in the region.

### **Returned Value**

*result* — An (*m*,*n*) array of *m* *n*-dimensional indices into *a*. *result* defines the region containing *i* whose values lie in the range [*b*(0),*b*(1)]. If no such region exists then *result* is returned as -1.

### **Keywords**

*k* — A positive integer (less than or equal to *n*) controlling connectivity: two cells are connected if they share a common boundary point, and if their centroids are within the square root of *k* of each other. *k* = 1 by default, which implies connected cells share a common face.

### **Example**

See `wave/lib/user/examples/blob_grow.pro`

## See Also

[BLOBCOUNT](#), [BOUNDARY](#), [NEIGHBORS](#)

---

## **BLOBCOUNT Function**

Standard Library function that counts homogeneous regions in an array.

### **Usage**

```
result = BLOBCOUNT(a, b)
```

### **Input Parameters**

*a* — An array of *n* dimensions.

*b* — A two-element vector of bounds for values in a region.

### **Returned Value**

*result* — A list in which each element defines a distinct region whose values lie in the range [b(0),b(1)]. *result*(*j*) is a (m(*j*),*n*) array of m(*j*) *n*-dimensional indices into *a*. If no such regions exist, then *result* is returned as -1.

### **Keywords**

*k* — A positive integer (less than or equal to *n*) controlling connectivity: two cells are connected if they share a common boundary point, and if their centroids are within the square root of *k* of each other. *k* = 1 by default, which implies that connected cells share a common face.

### **Example 1**

```
a = ( image_read(!data_dir+'vni_small.tif') ) ( 'pixels' )
a = bytscl( resamp(a,500,500) ) & tv, a
r = blobcount( a, [255,255] )
for i = 0, n_elements(r)-1 do a(index_conv(a,r(i))) = 50 & tv, a
```

### **Example 2**

See wave/lib/user/examples/blobcount\_ex.pro

### **See Also**

[BLOB](#), [BOUNDARY](#), [NEIGHBORS](#)

---

## **BOUNDARY Function**

Standard Library function that computes the boundary of a region in an array.

### **Usage**

*result* = BOUNDARY(*a*,*r*)

### **Input Parameters**

*a* — An array of *n* dimensions.

*r* — A vector of indices defining the region of *a*.

### **Returned Value**

*result* — A vector of indices defining the boundary of *r*.

### **Keywords**

*k* — A positive integer (less than or equal to *n*) defining connectivity. A boundary element of *r* is an element of *r* with neighbors not in *r*; two array cells are neighbors if they share a common boundary point and their centroids are within the square root of *k* of each other. *k* = 1 by default, which implies neighbors share a common face.

### **Examples**

```
a = indgen( 5, 4 ) & pm, a
print, fix(boundary(a, [1,2,3,6,7,8,12,13]))
print, fix(boundary(a, [1,2,3,6,7,8,12,13],k=2))
a = bytscl( dist(500) ) & r = where( 150 le a and a le 200 )
a( boundary(a,r) ) = 0 & tv, a
```

### **See Also**

[BLOB](#), [BLOBCOUNT](#), [NEIGHBORS](#)

---

## **BREAKPOINT Procedure**

Lets you insert and remove breakpoints in programs for debugging.

### **Usage**

BREAKPOINT, *file*, *line*

### **Input Parameters**

*file* — The name of the source file in which to insert the breakpoint.

*line* — Specify either a line number (integer) or a procedure/function name (string). If you specify a line number, the breakpoint is set at that line. If you specify a procedure or function name, the breakpoint is set at the beginning of the procedure or function.

### **Keywords**

*Allclear* — Removes all currently set breakpoints.

*Clear* — Removes the breakpoint specified by its index, or by the *file* and *line* parameters. If only one input parameter is specified, it is interpreted as an index identifying the currently set breakpoint. If two input parameters are specified, they are interpreted as the *file* and *line* of the currently set breakpoint.

*Set* — Sets a breakpoint in the specified *file* at the specified *line* number.

### **Discussion**

A breakpoint causes program execution to stop after the designated statement is executed. Breakpoints are specified using the source file name and line number. You can insert breakpoints in programs without editing the source file.

Once a breakpoint has stopped execution, use `.CON` to continue execution.

Use `INFO, /Breakpoint` to display the breakpoint table, which gives the index, module, line number, and file location of each breakpoint.

### **Examples**

To clear a breakpoint:

```
BREAKPOINT, /Clear, 3
```

```
    ; Clear the breakpoint with index 3.  
BREAKPOINT, /Clear, 'test.pro', 8  
    ; Clear the breakpoint corresponding to the statement in the file  
    ; test.pro, line number 8.
```

To set a breakpoint at line 23, in the source file `xyz.pro`:

```
BREAKPOINT, 'xyz.pro', 23  
  
or  
  
BREAKPOINT, /Set, 'xyz.pro', 23
```

## See Also

[CHECK\\_MATH](#), [INFO](#), [ON\\_ERROR](#), [STOP](#)

---

## ***BUILDRESOURCEFILENAME* Function**

Standard library routine that returns the full pathname for a specified resource file.

### **Usage**

```
resource_file = BUILDRESOURCEFILENAME(file)
```

### **Input Parameter**

*file* — The name of the resource file.

### **Returned Value**

*resource\_file* — A string containing the resource file path.

### **Keywords**

***Appdir*** — A string that specifies the application directory name. This is the directory in which the application searches for resource files, string resource files, and icon files. (Default: 'vdatools')

***Subdir*** — A string specifying a resource file subdirectory. (Default: !Lang, whose default string is 'american')

## Discussion

By default, the function looks for *file* first in directories specified by the environment variable `WAVE_RESPATH`.

---

**UNIX USERS** The `WAVE_RESPATH` environment variable is a colon-separated list of directories, similar to the `WAVE_PATH` environment variable in **PV-WAVE**. If not found in a `WAVE_RESPATH` directory, the directory `<wavedir>/xres/!Lang/vdatools` is searched, where `<wavedir>` is the main **PV-WAVE** directory and `!Lang` represents the value of the `!Lang` system variable (`!Lang` default is `'american'`).

---

**OpenVMS USERS** The `WAVE_RESPATH` logical is a comma-separated list of directories and text libraries, similar to the `WAVE_PATH` logical in **PV-WAVE**. If not found in a `WAVE_RESPATH` directory, the directory `<wavedir>:[XRES.!Lang.VDATOOLS]` is searched, where `<wavedir>` is the main **PV-WAVE** directory and `!Lang` represents the value of the `!Lang` system variable (`!Lang` default is `'american'`).

---

**Windows USERS** The `WAVE_RESPATH` environment variable is a semicolon-separated list of directories, similar to the `WAVE_PATH` environment variable in **PV-WAVE**. If not found in a `WAVE_RESPATH` directory, the directory `<wavedir>\xres\!Lang\vdatools` is searched, where `<wavedir>` is the main **PV-WAVE** directory and `!Lang` represents the value of the `!Lang` system variable (`!Lang` default is `'american'`).

---

If *Subdir* alone is specified, the file is searched for in:

**(UNIX)** `<wavedir>/xres/subdir/vdatools`  
**(OpenVMS)** `<wavedir>:[XRES.SUBDIR.VDATOOLS]`  
**(Windows)** `<wavedir>\xres\subdir\vdatools`

Where `<wavedir>` is the main **PV-WAVE** directory.

If only *Appdir* is specified, the application searches for resources in the following directory:

**(UNIX)** `<wavedir>/xres/!Lang/appdir`  
**(OpenVMS)** `<wavedir>:[XRES.!Lang.APPDIR]`  
**(Windows)** `<wavedir>\xres\!Lang\appdir`

Where `<wavedir>` is the main **PV-WAVE** directory.

If both *Subdir* and *Appdir* are specified, the application searches for resources in the following directory:

**(UNIX)**        <wavedir>/xres/subdir/appdir

**(OpenVMS)** <wavedir>:[XRES.SUBDIR.APPDIR]

**(Windows)** <wavedir>\xres\subdir\appdir

Where <wavedir> is the main PV-WAVE directory.

If the file is not already in the resource database, the full pathname is returned.

## Example

The following commands are taken from the code for a VDA Tool called WzMyVDA. The full pathname of the resource file for WzMyVDA is returned and is passed to the *Resource* keyword of WwInit.

```
...
resource_file = BUILDRESOURCEFILENAME('wzmyvda.ad')
top = WwInit('WzMyVDA', 'VDATools', layout, $
    'DestroyCB', Shell_name = 'WzMyVDA', $
    Layout_name = 'toolArea', $
    Title = unique_name, /Form, $
    ConfirmClose = 'ConfirmClose', $
    Resource = resource_file, $
    Userdata = unique_name)
...
```

## See Also

[LOADRESOURCES](#), [LOADSTRINGS](#)

For information on environment variables and logicals used with PV-WAVE, see the *PV-WAVE Programmer's Guide*.

---

## ***BUILD\_TABLE* Function**

Creates a table from one or more vectors (one-dimensional arrays).

### **Usage**

```
result = BUILD_TABLE(' var1 [alias], ..., varn [alias] ' )
```

### **Input Parameters**

*var*<sub>*i*</sub> — A vector (one-dimensional array) variable. If additional vectors are specified, they must contain the same number of elements as *var*<sub>*i*</sub>. The input variable(s) can be of any data type.

*alias* — (optional) Specifies a new name for the table column. By default, the input variable's name is used.

### **Returned Value**

*result* — A table containing *n* columns, where *n* is equal to the number of input variables.

### **Input Keywords**

*In\_Structure* — A scalar string expression that specifies the name of a PV-WAVE structure to use to create the result table. This structure can either be defined by the user, or obtained from the *Out\_Structure* keyword from a previous BUILD\_TABLE call. If user-defined, the tag definitions of this structure must meet the requirements for PV-WAVE table variables. If no value is specified by *In\_Structure*, PV-WAVE creates a new named structure, based on the types of the column variables specified in the parameter string. The purpose of this keyword is to allow you to append new rows to an existing table variable.

### **Output Keywords**

*Out\_Structure* — A string variable which receives the name of the PV-WAVE structure that was used to create the result table. The purpose of this keyword is to allow you to append rows to the result table with subsequent calls to BUILD\_TABLE. In this scenario, *Out\_Structure* is used to save the name of the structure created during the first BUILD\_TABLE call. This same structure name is used (with the *In\_Structure* keyword) to append rows to the result table.

## Discussion

Once created, you can subset the table using the `QUERY_TABLE` function. Each vector must have the same number of elements. If not, an error message is displayed and the table is not created.

A table is built from vector (one-dimensional array) variables only. You cannot include expressions in the `BUILD_TABLE` function. For example, The following `BUILD_TABLE` call is *not* allowed:

```
result = BUILD_TABLE('EXT(0:5), COST(0:5)')
```

However, you can achieve the desired results by performing the array subsetting operations first, then using the resulting variables in `BUILD_TABLE`. For example:

```
EXT = EXT(0:5)
COST = COST(0:5)
result = BUILD_TABLE('EXT, COST')
```

In addition, you cannot include scalars or multidimensional-array variables in `BUILD_TABLE`.

---

**NOTE** `ASC` and `DESC` are reserved words (used by `QUERY_TABLE` for direction) and thus are not allowed to be used as variable names or aliases.

---

## Example 1

The following command creates a table consisting of eight columns of data. The columns are created from data read into `PV-WAVE` and placed into vector variables.

```
phone_data = BUILD_TABLE('DATE, TIME, ' + $
    'DUR, INIT, EXT, COST, AREA, NUMBER')
```

Here is a portion of the resulting table:

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	093200	21.40	TAC	311	5.78	215	2155554242
901002	094700	1.05	BWD	358	0	303	5553869
901002	094700	17.44	EBH	320	4.71	214	2145559893
901002	094800	16.23	TDW	289	0	303	5555836
901002	094800	1.31	RLD	248	.35	617	6175551999

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901003	091500	2.53	DLH	332	.68	614	6145555553
901003	091600	2.33	JAT	000	0	303	555344
901003	091600	.35	CCW	418	.27	303	5555190
901003	091600	1.53	SRB	379	.41	212	2125556618

You can use the INFO command to view the new table structure, for example:

```
INFO, /Structure, phone_data
** Structure TABLE_0, 8 tags, 40 length:

DATE          LONG          901002
TIME          LONG          93200
DUR           FLOAT        21.4000
INIT          STRING        'TAC'
EXT           LONG          311
COST          FLOAT        5.78000
AREA          LONG          215
NUMBER        STRING        2155554242
```

The *Structure* keyword is used in this example because tables are represented in PV-WAVE as an array of structures.

The QUERY\_TABLE function can be used to retrieve information from this table. For example:

```
res = QUERY_TABLE(phone_data, ' * Where COST > 1.0')
```

This query produces a new table containing only the rows where the cost is greater than one dollar.

## Example 2

This example demonstrates the use of the optional *alias* parameter. This parameter lets you specify new names for the columns of the table. By default, the names of the input variables are used as column names.

```
phone_data1 = BUILD_TABLE('DATE Call_Date,' + $
    TIME Call_Time, DUR Call_Length,' + $
    'INIT, EXT, COST Charge, AREA Area_Code,' + $
    'NUMBER Phone_Number')
```

The structure of this table reflects the new column names:

```
INFO, /structure, phone_data
** Structure TABLE_0, 8 tags, 40 length:

CALL_DATE          LONG          901002
CALL_TIME          LONG          93200
CALL_LENGTH        FLOAT         21.4000
INIT               STRING         'TAC'
EXT                LONG          311
CHARGE             FLOAT         5.78000
AREA_CODE          LONG          215
PHONE_NUMBER       STRING         2155554242
```

## See Also

[GROUP\\_BY](#), [ORDER\\_BY](#), [QUERY\\_TABLE](#), [UNIQUE](#)

For more information on `BUILD_TABLE`, see ,

For information on reading data into variables, see .

---

## ***BYTARR Function***

Returns a byte vector or array.

### **Usage**

```
result = BYTARR(dim1[, dim2, ... , dimn])
```

### **Input Parameters**

*dim*<sub>*i*</sub> — The dimensions of the array. This may be any scalar expression, and can have up to eight dimensions specified.

### **Returned Value**

*result* — A one-dimensional or multi-dimensional byte array.

## Keywords

*Nozero* — Normally, BYTARR sets every element of the result to zero. If *Nozero* is nonzero, this zeroing is not performed, thereby causing BYTARR to execute faster.

## Examples

```
a = BYTARR(5)
PRINT, a
      0  0  0  0  0
b = BYTARR(2, 3, 5, 7)
INFO, b
      B  BYTE  = ARRAY(2, 3, 5, 7)
```

## See Also

[BINDGEN](#), [BYTE](#), [BYTEORDER](#), [BYTSCL](#), [DBLARR](#),  
[COMPLEXARR](#), [FLTARR](#), [INTARR](#), [LONARR](#),  
[MAKE\\_ARRAY](#), [STRARR](#)

---

## BYTE Function

Converts an expression to byte data type.

Extracts data from an expression and places it in a byte scalar or array.

## Usage

*result* = BYTE(*expr*)

This form is used to convert data.

*result* = BYTE(*expr*, *offset* [, *dim*<sub>1</sub>, ... , *dim*<sub>*n*</sub>])

This form is used to extract data.

## Input Parameters

To convert data:

*expr* — The expression to be converted.

To extract data:

*expr* — The expression from which to extract data.

*offset* — The offset, in bytes, from the beginning of *expr* to where the extraction is to begin. If present, causes BYTE to extract data, not convert it.

*dim<sub>i</sub>* — (optional) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

## Returned Value

For data conversion:

**result** — A copy of *expr* converted to byte data type. The result has the same size and structure (scalar or array) as *expr*.

For extracting data:

**result** — A copy of only part of *expr*—the part that is defined by the *offset* and *dim* input parameters. The result has the size and structure of the specified dimensions and is of the byte data type. If no dimensions are specified, the result is scalar.

## Keywords

None.

## Discussion

BYTE can be useful in a variety of applications — for example, in hexadecimal math, when you want to be certain that you are working with a byte value to ensure that any comparison you make is valid.

If *expr* is of type string, each character is converted to its ASCII value and placed into a vector. In other words, each vector element is the ASCII character code of the corresponding character in the string.

If *expr* is not of type string, then *expr* is converted to byte data type. The result is *expr* modulo 256.

---

**TIP** Use BYTSCL to convert *expr* to byte data type using scaling rather than modulo.

---

---

**CAUTION** If the values of *expr* are within the range of a long integer, but outside the range of the byte data type (0 to +255), a misleading result occurs without an

accompanying message. For example, `BYTE (256)` erroneously results in 0. If the values of *expr* are outside the range of a long integer data type, an error message may be displayed.

In addition, `PV-WAVE` does not check for overflow during conversion to byte data type. The values in *expr* are simply converted to long integers and the low 8 bits are extracted.

---

### Example 1

```
a = BYTE('01abc')
INFO, a
      A   BYTE   = Array(5)
PRINT, a
      48   49   97   98   99
```

### Example 2

```
a = BYTE(1.2)
PRINT, a
      1
```

### Example 3

```
a = BYTE(-1)
PRINT, a
      255
; The calculated result is 255 (bytes are modulo 256).
```

### See Also

[BINDGEN](#), [BYTARR](#), [BYTEORDER](#), [BYTSCL](#), [COMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#)

For more information on using this function to extract data, see the *PV-WAVE Programmer's Guide*.

---

## **BYTEORDER Procedure**

Converts integers between host and network byte ordering. This procedure can also be used to swap the order of bytes within both short and long integers.

### **Usage**

BYTEORDER, *variable*<sub>1</sub>, ..., *variable*<sub>*n*</sub>

### **Input Parameters**

*variable*<sub>*i*</sub> — A variable (see *Discussion* below).

### **Output Parameters**

*variable*<sub>*i*</sub> — A variable (see *Discussion* below).

### **Keywords**

*Htonl* — Host to network, longwords.

*Htons* — Host to network, short integers.

*Lswap* — Longword swap. Always swaps the order of the bytes within each longword. For example, the four bytes within a longword are changed from (B<sub>0</sub>, B<sub>1</sub>, B<sub>2</sub>, B<sub>3</sub>) to (B<sub>3</sub>, B<sub>2</sub>, B<sub>1</sub>, B<sub>0</sub>).

*Ntohl* — Network to host, longwords.

*Ntohs* — Network to host, short integers.

*Sswap* — Short word swap. Always swaps the bytes within short integers. The even and odd numbered bytes are interchanged.

### **Discussion**

BYTEORDER is most commonly used when dealing with binary data from non-native architectures that may have different byte ordering. An easier solution to use when reading or writing this sort of data is the XDR format, as explained in the *PV-WAVE Programmer's Guide*.

The size of the parameter, in bytes, must be evenly divisible by two for short integer swaps, and by four for long integer swaps. BYTEORDER operates on both scalars

and arrays. The parameter must be a variable, not an expression or constant, and may not contain strings.

---

**NOTE** The contents of *variable<sub>i</sub>* are overwritten by the result.

---

Network byte ordering is big endian. This means that multiple byte integers are transmitted beginning with the most significant byte.

## Examples

```
a = '1234'X
    ; Form a hexadecimal value that can be easily interpreted. Note that
    ; the hex value "12" is in the high-order byte, and "34" is in the low order byte.
```

```
b = a
    ; Remember that b will be overwritten by BYTEORDER.
```

```
BYTEORDER, b, /Sswap
```

```
PRINT, Format='(ZZ9)', a, b
```

```
1234      3412
```

; The result shows that the high- and low-order bytes in b have been switched.

```
a = '12345678'XL
```

```
b = a
```

```
BYTEORDER, b, /Lswap
```

```
PRINT, Format='(ZZ9)', a, b
```

```
12345678      78563412
```

; Bytes in b are swapped as expected, whereas in hexadecimal  
; format, two digits represent a single byte.

## See Also

[BYTE](#)

---

## **BYTSCL Function**

Scales and converts an array to byte data type.

### **Usage**

*result* = BYTSCL(*array*)

### **Input Parameters**

*array* — The array to be scaled and converted to byte data type.

### **Returned Value**

*result* — A copy of *array* whose values have been scaled and converted to bytes.

### **Keywords**

*Max* — The maximum value of *array* elements to be considered. If *Max* is not specified, *array* is searched for its largest value.

*Min* — The minimum value of *array* elements to be considered. If *Min* is not specified, *array* is searched for its smallest value.

*Top* — The maximum value of the scaled result. (Default: 255)

### **Discussion**

BYTSCL can be used in a variety of applications — for example, to compress the gray levels in an image to suit the levels supported by the particular hardware you are using. It can also be used to increase or reduce the contrast of an image by expanding or restricting the number of gray levels used.

BYTSCL linearly scales all values of *array* that lie in the range ( $Min \leq x \leq Max$ ) into the range ( $0 \leq x \leq Top$ ). The result has the same number of dimensions as the original array.

If the values of *array* are outside this range ( $Min \leq x \leq Max$ ), BYTSCL maps all values of *array* < *Min* to zero, and maps all values of *array* > *Max* to *Top* (255 by default).

## Example 1

To scale an array of floats to byte values, you might enter:

```
arr = FINDGEN(100)
byt = BYTSCL(arr, Max=50.0)
PRINT, SIZE(byt)
      1          100          1          100
PRINT, byt
  0  5 10 15 20 25 30 35 40 45 50 56 61 66 71 76 81
 86 91
 96 101 107 112 117 122 127 132 137 142 147 152 158 163 168 173
178 183 188
193 198 203 209 214 219 224 229 234 239 244 249 255 255 255 255
255 255 255
255 255 255 255 255 255 255 255 255 255 255 255 255 255 255
255 255 255
255 255 255 255 255
255 255 255 255 255
```

## Example 2

This example uses the BYTSCL function to enhance the contrast of an image. The image is stored in a byte array, b. The argument

```
BYTSCL(b, Min = 50, Max = 70)
```

in the second call to the TV procedure scales the values of b so all bytes with a value less than or equal to 50 are set to 0, and all bytes with a value greater than or equal to 70 are set to 255. All bytes with a value between 50 and 70 are scaled to lie in the range {0...255}.

```
OPENR, unit, FILEPATH('whirlpool.img', Subdir = 'data'), /Get_Lun
      ; Open the file galaxy.dat for reading.
```

```
b = BYTARR(512,512)
      ; Retrieve the first galaxy image, which is stored as a 256-by-256 byte array.
```

```
READU, unit, b
```

```
FREE_LUN, unit
```

```
!Order = 1
```

```
LOADCT, 3
```

```
WINDOW, 0, Xsize = 1024, Ysize = 512
```

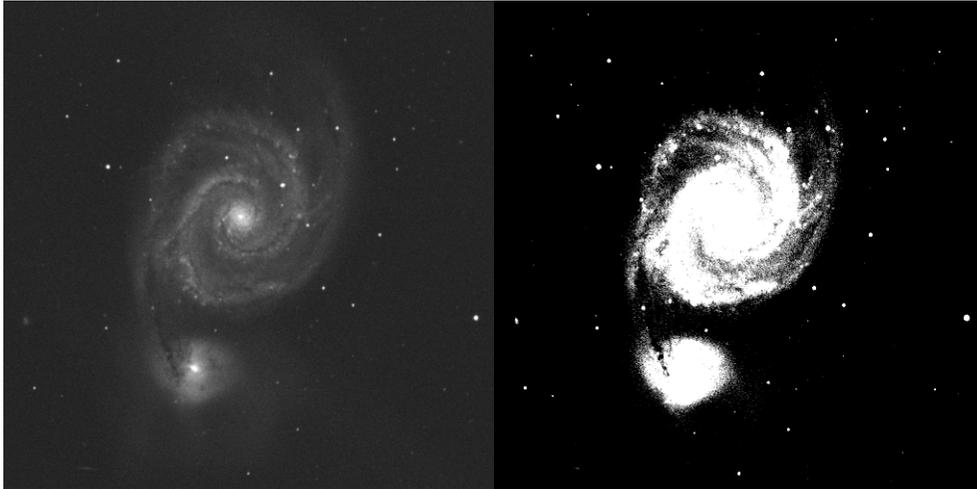
```
      ; Load the red temperature color table and create a window big
      ; enough for two images.
```

```
TV, b, 0
```

```
; Display the image, without any contrast enhancement, at left side of window.
```

```
TV, BYTSCCL(b, Min = 50, Max = 70), 1
```

```
; Display the contrast enhanced image at right side of window.
```



**Figure 2-5** Galaxy image before (left) and after (right) contrast enhancement.

## See Also

[BYTE](#), [BYTARR](#), [BINDGEN](#)

---

## **CALL\_UNIX Function (UNIX)**

Lets a PV-WAVE procedure communicate with an external routine written in C.

### **Usage**

*result* = CALL\_UNIX(*p*<sub>1</sub> [, *p*<sub>2</sub>, ... , *p*<sub>30</sub>])

### **Input Parameters**

*p*<sub>*i*</sub> — A variable of any type. At least one parameter must be passed, but there can be up to 30 parameters. If the external routine does not require any parameters, the value of *p* must be zero.

### **Returned Value**

*result* — A user-defined variable or data type to be returned from the external program. The returned variable cannot be -1, since -1 is reserved to indicate failure.

### **Keywords**

**Close** — If present and nonzero, causes PV-WAVE to close *Unit* at the end of CALL\_UNIX. (If *Unit* is not specified, *Close* has no effect.)

**Hostname** — A string that identifies the node name of the host on which the called external program is executing. If not specified, the default value of “localhost” is used.

**Procedure** — A string with a maximum length of 40 characters. Can say anything, but its intended use is to control program flow in the external routine.

**Program** — An integer identifier that enables the C routine `w_listen` to match a particular call to CALL\_UNIX to a particular external routine. Must be greater than or equal to zero. The default value is zero. *Program* is intended to allow more than one external routine to be called by CALL\_UNIX.

**Unit** — An integer used to reference an RPC socket:

- If *Unit* is zero, *Unit* is returned with a valid unit number.
- If *Unit* is nonzero, the value specified by *Unit* is used.
- If *Unit* is not specified, an RPC socket is reopened with each call to CALL\_UNIX.

By specifying *Unit*, the overhead of opening an RPC socket each time is saved. In most cases, however, the overhead is not noticeable.

**User** — A string with a maximum length of 40 characters. Can say anything, but its intended use is for controlling access to the external routine.

**Timeout** — An integer that indicates the maximum time, in seconds, that PV-WAVE will wait for the external routine to finish. The default value is 60 seconds. If the external routine requires more than 60 seconds to execute, *Timeout* must be specified. There is no value to indicate an infinite amount of time.

## Discussion

CALL\_UNIX sends parameters to another process that is running the external C routine.

The external routine uses the following C routines:

- `w_listen` to connect with the process running PV-WAVE
- `w_get_par` to actually get the parameters
- `w_send_reply`, `w_smp1_reply`, or `w_cmpnd_reply` to send values and parameters back to PV-WAVE.

For information on these C routines, see the *PV-WAVE Application Developer's Guide*.

If an error occurs in a call to CALL\_UNIX, -1 is returned. ON\_IOERROR can also be used to catch CALL\_UNIX errors.

## Example

---

**NOTE** For information on these C routines, see the *PV-WAVE Application Developer's Guide*.

---

## See Also

[ON\\_IOERROR](#), [UNIX\\_LISTEN](#), [UNIX\\_REPLY](#)

---

## **CD Procedure**

Changes the current working directory.

### **Usage**

CD [, *directory*]

### **Input Parameters**

*directory* — (optional) If specified, this parameter is a string specifying the path of the new working directory. If it is specified as a null string, the working directory is changed to the user's home directory.

If this parameter is not specified, no directory change is made and the current directory remains the working directory.

### **Keywords**

**Current** — Creates a variable that stores the current directory name. You can store the name of the current working directory and change the working directory in a single statement:

```
CD, new_dir, Current=old_dir
```

The variable `old_dir` contains the name of the working directory before the change to `new_dir`.

### **Discussion**

Initially, the working directory is the directory from which you started PV-WAVE.

This procedure changes the working directory for the current PV-WAVE session and any child processes started during the session after the change is made. It does not affect the working directory of the process that started PV-WAVE. Therefore, when you exit PV-WAVE, you will be in the directory you were in when you started.

The PUSH, POP, and PRINTD procedures, which maintain a directory stack and call CD to change directories, provide a convenient interface to CD.

## Examples

### **UNIX**

On a UNIX system, to change the current working directory to `/usr/home/mydata`, enter the following at the `WAVE>` prompt:

```
CD, '/usr/home/mydata'
```

To move to the home directory, enter the following:

```
CD, ''
```

### **OpenVMS**

On an OpenVMS system, to change the current working directory to `SYS$SYSDEVICE:[MYDATA]`, enter the following at the `WAVE>` prompt:

```
CD, 'SYS$SYSDEVICE:[MYDATA]'
```

To move to the home directory, enter the following:

```
CD, ''
```

### **Windows**

To change the current directory to `D:\user\home\mydata`, enter the following at the `WAVE>` prompt:

```
CD, 'D:\user\home\mydata'
```

## See Also

[!Dir](#), [!Path](#), [FILEPATH](#), [POPD](#), [PRINTD](#), [PUSHD](#)

---

## ***C\_EDIT Procedure***

Standard Library procedure that lets you interactively create a new color table based on the HLS or HSV color system.

### **Usage**

`C_EDIT [, colors_out]`

### **Input Parameters**

None.

### **Output Parameters**

*colors\_out* — (optional) Contains the color values of the final color table in the form of a two-dimensional array that has the number of colors in the color table as the first dimension and the integer 3 as the second dimension.

The values for red are stored in the first row, the values for green are stored in the second row, and those for blue in the third row; in other words:

*red* = `colors_out(*, 0)`

*green* = `colors_out(*, 1)`

*blue* = `colors_out(*, 2)`

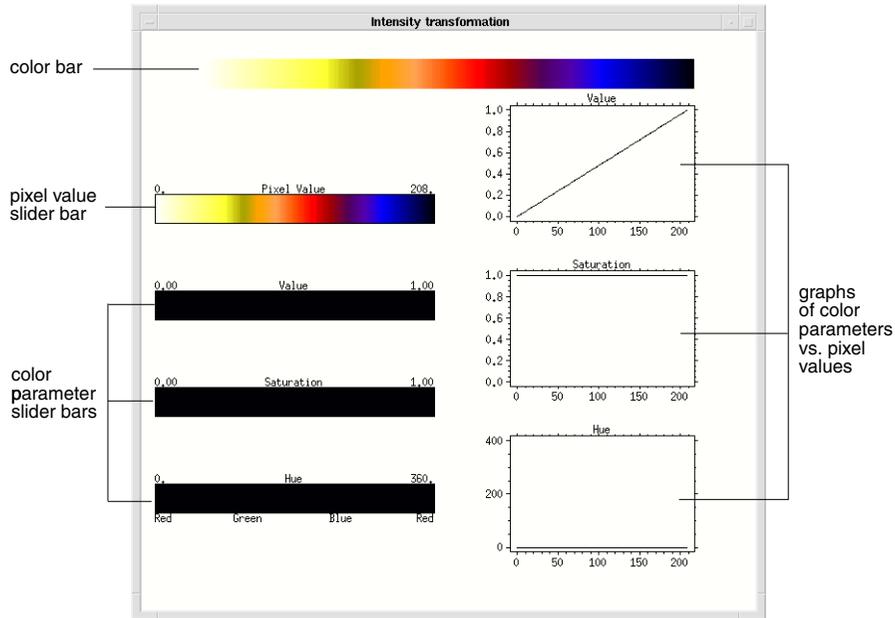
### **Keywords**

*Hls* — If set to 1, indicates the HLS (hue, lightness, saturation) color system should be used.

*Hsv* — If set to 1, indicates the HSV (hue, saturation, value) color system should be used. This is the default.

### **Discussion**

`C_EDIT` works only on displays with window systems. It creates an interactive window that lets you use the mouse to create a new color table. This window is shown in .



**Figure 2-6** The C\_EDIT window lets you use the mouse to create a new color table based on either the HLS or HSV color system.

C\_EDIT not only changes the colors displayed in the window that it creates, it also changes the colors in other windows so that you can watch different anomalies rise out of your data.

C\_EDIT is similar to the COLOR\_EDIT procedure, except that the color wheel has been replaced by two additional slider bars. This allows better control of HSV colors near zero percent saturation.

The C\_EDIT window contains the following items:

- **Color Bar** — Displays the current color table. It is updated as changes are made to the color table. (When C\_EDIT is initialized, it sets the current color table to red.)
- **Pixel Value Slider Bar** — Used to set tie points.
- **Color Parameter Slider Bars** — Used to adjust the values for the three color parameters of value (or lightness), saturation, and hue.
- **Graphs** — Plot the current values of the three color system parameters against pixel value. These graphs are updated as tie points are selected, and the color table is changed.

To use C\_EDIT:

- Adjust the three color-parameter slider bars by dragging the left mouse button within each bar until you reach the first color you want in your color table.
- On the **Pixel Value** slider bar, click with the left mouse button at the position where you want that particular color to be. The range on this bar begins at 0 and ends with the maximum value for your color table.

A small tie point then appears indicating the exact point where this color will occur in the color table. The values in the color table are interpolated between the tie points.

- If you need to erase the tie point, click on it with the middle mouse button.

---

**Windows USERS** If you have a two-button mouse, use <Alt> in combination with the left mouse button to erase the tie point.

---

The color-system parameter graphs on the right of the window and the color bar at the top of the window are updated whenever a tie point is created or removed.

- Create your second color by again adjusting the three color-parameter slider bars and entering the corresponding tie point in the **Pixel Value** slider bar.  
Repeat this step until you have finished creating all the colors you want in your color table.
- Use the right mouse button to exit the procedure.

### ***Sample Usage***

Assume that the HSV values associated with the default tie points are (0, 1, 0) for the 0 pixel value and (1, 1, 0) for the 255 pixel value. Suppose a new tie point at pixel value 100 is selected after setting the HSV values to (.8, .5, 0). Then the Hue values are interpolated between 0 and .8 and assigned to pixel values 0 to 100, and interpolated between .8 and 1 and assigned to pixel values 101 to 255. The Saturation values are interpolated between 1 and .5, and between .5 and 1, and assigned to the same pixel values. The Value quantities remain unchanged in this example.

You may select as many tie points as desired, with the understanding that each tie point is associated with the color system parameters in effect when the selection is made.

Note that when the HSV color system is being used, a Value of 1.0 is maximum brightness of the selected hue. In the HLS color system, a Lightness of 0.5 is the

maximum brightness of a chromatic hue; 0.0 is black, and 1.0 is bright white. Also, in the HLS system, which models a double-ended cone, the Saturation has no effect at the extreme ends of the cone (i.e., Lightness equals 0 or 1).

## Example 1

```
TVSCL, DIST(200)
C_EDIT, rgb_array
```

— User modifies the color table and exits the procedure. —

```
SAVE, filename = 'my_colortable', rgb_array
LOADCT, 5
RESTORE, 'my_colortable'
rgb_array = REFORM(rgb_array, $
    N_ELEMENTS(rgb_array)/3,3)
TVLCT, rgb_array(*,0),rgb_array(*,1), $
    rgb_array(*,2)
```

## Example 2

```
TVSCL, DIST(200)
C_EDIT
```

— User modifies the color table and exits the procedure. —

```
TVLCT, r, g, b, /Get
SAVE, filename = 'my_colortable_2', r, g, b
LOADCT, 8
RESTORE, 'my_colortable_2'
TVLCT, r, g, b
```

## See Also

[COLOR\\_CONVERT](#), [COLOR\\_EDIT](#), [COLOR\\_PALETTE](#), [HLS](#), [HSV](#), [LOADCT](#), [MODIFYCT](#), [PALETTE](#), [PSEUDO](#), [STRETCH](#), [TVLCT](#), [WgCbarTool](#), [WgCeditTool](#), [WgCtTool](#)

For more background information about color systems, see the *PV-WAVE User's Guide*.

For an excellent discussion of the HSV and HLS color systems, see *Computer Graphics: Principles and Practice*, by Foley, Van Dam, Feiner, and Hughes, Second Edition, Addison Wesley Publishing Company, Reading, MA, 1990.

---

## ***CENTER\_VIEW Procedure***

Sets system viewing parameters to display data in the center of the current window (a convenient way to set up a 3D view).

### **Usage**

`CENTER_VIEW`

### **Parameters**

None.

### **Keywords**

***Ax*** — The angle, in degrees, at which to rotate the data around the  $x$ -axis. (Default:  $-60.0$ )

***Ay*** — The angle, in degrees, at which to rotate the data around the  $y$ -axis. (Default:  $0.0$ )

***Az*** — The angle, in degrees, at which to rotate the data around the  $z$ -axis. (Default:  $30.0$ )

***Persp*** — The perspective projection distance. If *Persp* is  $0.0$  (the default), then parallel projection is set.

***Winx*** — The  $x$  size of the plot window in device coordinates. (Default:  $640$ )

***Winy*** — The  $y$  size of the plot window in device coordinates. (Default:  $512$ )

***Xr, Yr, Zr*** — Two-element vectors.  $Xr(0)$ ,  $Yr(0)$ , and  $Zr(0)$  contain the minimum  $x$ ,  $y$ , and  $z$  values, respectively, in the data to be plotted.  $Xr(1)$ ,  $Yr(1)$ , and  $Zr(1)$  contain the maximum values for this data. The default is  $[-1.0, 1.0]$  for each of *Xr*, *Yr*, and *Zr*.

***Zoom*** — The magnification factor. The default is  $[0.5, 0.5, 0.5]$ .

If *Zoom* contains one element, then the view is zoomed equally in the  $x$ ,  $y$ , and  $z$  dimensions.

If *Zoom* contains three elements, then the view is scaled by  $Zoom(0)$  in the  $x$  direction,  $Zoom(1)$  in the  $y$  direction, and  $Zoom(2)$  in the  $z$  direction.

## Discussion

CENTER\_VIEW sets the system 3D viewing transformation and conversion factors from data coordinates to normal coordinates so that data is displayed in the center of the current window. The correct aspect ratio of the data is preserved even if the plot window is not square.

---

**NOTE** The data is rotated  $A_z$  degrees about the  $z$ -axis first,  $A_y$  degrees about the  $y$ -axis second, and  $A_x$  degrees about the  $x$ -axis last.

---

---

**CAUTION** This procedure sets the system variables !P.T, !P.T3D, !X.S, !Y.S, and !Z.S, overriding any values you may have previously set. (These system variables are described in [Chapter 4, System Variables](#).)

---

## Example

```
PRO f_gridemo4
    ; This program shows 4D gridding of dense data and a cut-away view
    ; of a block of volume data.

points = RANDOMU(s, 4, 1000)
    ; Generate random data to be used for shading.

ival = FAST_GRID4(points, 32, 32, 32)
ival = BYTSCL(ival)
    ; Grid the generated data.

block = BYTARR(30, 30, 30)
block(*, *, *) = 255
block = VOL_PAD(block, 1)
    ; Pad the data with zeroes.

block(0:16, 0:16, 16:31) = 0
    ; Cut away a portion of the block array by setting the elements to zero.

WINDOW, 0, Colors=128
LOADCT, 3
CENTER_VIEW, Xr=[0.0, 31.0], Yr=[0.0, 31.0], $
    Zr=[0.0, 31.0], Ax=(-60.0), Az=45.0, $
    Zoom=0.6
    ; Set up the viewing window and load the color table. (The
    ; indices for the 32-by-32-by-32 volume we are viewing go
    ; from 0 to 31.)

SET_SHADING, Light=[-1.0, 1.0, 0.2]
    ; Change the direction of the light source for shading.
```

```

SHADE_VOLUME, block, 1, vertex_list, $
    polygon_list, Shades=ival, /Low
    ; Compute the 3D contour surface.

img1 = POLYSHADE(vertex_list, polygon_list, /T3d)
    ; Render the cut-away block with light source shading.

img2 = POLYSHADE(vertex_list, polygon_list, Shades=ival, /T3d)
    ; Render the cut-away block shaded by the gridded data.

TVSCL, (FIX(img1) + FIX(img2))
    ; Display the resulting composite image of the light source-shaded
    ; block and data-shaded image of the block.

END

```

For other examples, see the following demonstration programs: `grid_demo4`, `grid_demo5`, `sphere_demo1`, `sphere_demo2`, `sphere_demo3`, `vol_demo2`, `vol_demo3`, and `vol_demo4` in these directories:

```

(UNIX)      <wavedir>/demo/arl
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows) <wavedir>\demo\arl

```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[SET\\_VIEW3D](#)

---

## CHEBYSHEV Function

Standard Library function that implements the forward and reverse Chebyshev polynomial expansion of a set of data.

### Usage

*result* = CHEBYSHEV(*data*, *ntype*)

### Input Parameters

*data* — The input data (either the original dataset or the Chebyshev polynomial expansion, depending upon *ntype*).

*ntype* — The numeric type to be returned:

- 1 To return the set of Chebyshev polynomials.
- +1 To return the original data.

## Returned Value

*result* — The numeric type specified by *ntype*.

## Keywords

None.

## Discussion

CHEBYSHEV uses a straightforward implementation of the recursion formula. If you use discontinuous data, the result is subject to round-off error.

---

## CHECKFILE Function

Determines if a file can be read from or written to.

## Usage

*status* = CHECKFILE(*filename*)

## Input Parameters

*filename* — A string containing the name of a file. If a pathname is not included, the function looks in the current directory for the file.

## Returned Value

*status* — A value indicating if the file can be used for the given operation.

- 1 Indicates the file can be used for the specified operation.
- 0 Indicates the file cannot be used.

## Keywords

**FullName** — (UNIX Only) A string containing the expanded filename is returned. Constructs such as `~user` and `$ENV_VAR` are expanded.

**Is\_Dir** — Returns a 1 if *filename* is a directory.

**Read** — If specified and nonzero, the function verifies that the file is readable.

---

**NOTE** Either the *Read* keyword or the *Write* keyword must be specified.

---

**Size** — Returns the size of the file in bytes.

**Write** — If specified and nonzero, the function verifies that the file is writable.

## Discussion

You must supply either the *Read* or *Write* keyword. If neither of these keywords is supplied, the CHECKFILE function returns 0.

## Example

```
status = CHECKFILE(!Data_dir + 'head.img', /Read)
PRINT, status
      1

status = CHECKFILE(!Data_dir + 'new_head.img', /Write)
PRINT, status
      1

status = CHECKFILE(!Data_dir + 'head_not.img', /Read)
PRINT, status
      0
      ; Check the status of a file.

status = CHECKFILE(!Dir, /Read, Is_Dir = isdir)
PRINT, status
      1

PRINT, isdir
      1
      ; Determine if a directory exists.
```

```
status = CHECKFILE(!Data_dir + 'head.img', Size = sz)
PRINT, sz
      262144
      ; Check the size of a file.
```

## See Also

WoCheckFile in the *PV-WAVE Application Developer's Guide*

---

## CHECK\_MATH Function

Returns and clears the accumulated math error status.

### Usage

```
result = CHECK_MATH([print_flag, message_inhibit])
```

### Input Parameters

*print\_flag* — (optional) If present and nonzero, indicates an error message is to be printed if any accumulated math errors exist. Otherwise, no messages are printed.

*message\_inhibit* — (optional) Disables or enables the printing of math error exception error messages when they are detected. By default, these messages are enabled. Set *message\_inhibit* to 1 to inhibit, and 0 to re-enable.

When the interpreter exits to the interactive mode, error messages are printed for accumulated math errors that were suppressed but not cleared.

### Returned Value

*result* — An integer indicating the accumulated math error status since the last call or issuance of the interactive prompt. (See the Discussion section below for a list of values.)

---

**CAUTION** On machines that do not implement the IEEE standard for floating-point math, CHECK\_MATH does not properly maintain an accumulated error status.

---

## Keywords

**Trap** — Controls how floating-point traps are handled:

- If set to 0, no error messages are printed except the final accumulated error status.
- If set to 1 (the default), traps are enabled and programs are allowed to continue after floating-point errors. The first eight floating-point error exceptions issue messages. Subsequent errors are silent.

If a floating-point error occurs which is not logged, the accumulated floating-point error status is printed when PV-WAVE returns to the interactive mode.

---

**NOTE** Trap handling is machine dependent. Some machines won't work properly with traps enabled, while others don't allow disabling traps.

---

## Discussion

The result of CHECK\_MATH is 0 if no math errors have occurred since the last call or issuance of the interactive prompt. Other error status values as follows, where each binary bit represents an error:

Value	Condition
0	No errors detected since the last interactive prompt or call to CHECK_MATH.
1	Integer divide by zero.
2	Integer overflow.
16	Floating-point divide by zero.
32	Floating-point underflow.
64	Floating-point overflow.
128	Floating-point operand error. An illegal operand was encountered, such as a negative operand to the SQRT or ALOG functions; or an attempt to convert to integer a number whose absolute value is greater than $2^{31} - 1$ .

---

**CAUTION** Not all machines detect all errors.

---

## Example

```
a = [1.0, 1.0, 2.0]
    ; Array a will not fail as divisor.

b = [1.0, 0.0, 2.0]
    ; The second element in array b should cause a divide-by-zero error.

junkstatus = CHECK_MATH(1, 0, Trap=1)
    ; Clear the previous error status and print error messages if an error exists.

c = 1.0 / a
status = CHECK_MATH(0, 0)
PRINT, a, c, status
    1.00000      1.00000      2.00000
    1.00000      1.00000      0.500000
        0

d = 1.0 / b
    ; Cause an integer divide-by-zero error.
    % Program caused arithmetic error:
    % Floating divide by 0
    % Detected at $MAIN$ .

status = CHECK_MATH(0, 0)
PRINT, b, d, status
    1.00000      0.00000      2.00000
    1.00000      Inf        0.500000
        16
```

## See Also

[FINITE](#), [ON\\_ERROR](#), [RETURN](#), [STOP](#)

For additional information on error handling, see the *PV-WAVE Programmer's Guide*.

---

## **CINDGEN Function**

Returns a complex single-precision floating-point array.

### **Usage**

$result = CINDGEN(dim_1 [, dim_2, \dots, dim_n])$

### **Input Parameters**

$dim_i$ — The dimensions of the result. The dimensions may be any scalar expression, and up to eight dimensions may be specified.

### **Returned Value**

**result** — An initialized complex array with real and imaginary parts of type single precision, floating point. If the resulting array is treated as a one-dimensional array, then its initialization is given by the following:

$$array(i) = COMPLEX(i, 0)$$

$$\text{for } i = 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right)$$

### **Keywords**

None.

### **Example**

```
c = CINDGEN(4)
INFO, c
      C              COMPLEX      = Array(4)
PRINT, c
(      0.00000,      0.00000)
(      1.00000,      0.00000)
(      2.00000,      0.00000)
(      3.00000,      0.00000)
```

## See Also

[BINDGEN](#), [COMPLEX](#), [COMPLEXARR](#), [DCINDGEN](#), [DINDGEN](#),  
[FINDGEN](#), [INDGEN](#), [LINDGEN](#), [SINDGEN](#)

---

## CLOSE Procedure

Closes the specified file units.

### Usage

CLOSE [, *unit*<sub>1</sub>, ... , *unit*<sub>*n*</sub>]

### Input Parameters

*unit*<sub>*i*</sub> — (optional) The file units to close.

### Keywords

*All* — If present and nonzero, closes all file units and frees any file units that were allocated via GET\_LUN.

*Files* — If present and nonzero, closes all file units between 1 and 99. File units greater than 99, which are associated with the GET\_LUN and FREE\_LUN procedures, are not affected.

### Discussion

All open files are closed and deallocated when you exit PV-WAVE.

### Example

```
OPENW, 1, 'test'  
PRINTF, 1, 'Example Text'  
CLOSE, 1
```

## See Also

[FREE\\_LUN](#), [GET\\_LUN](#), [OPEN \(UNIX/OpenVMS\)](#), [OPEN \(Windows\)](#),  
[READ](#), [WRITEU](#)

---

## ***COLOR\_CONVERT Procedure***

Converts colors to and from the RGB color system, and either the HLS or HSV systems.

### **Usage**

`COLOR_CONVERT,  $i_0, i_1, i_2, o_0, o_1, o_2, keyword$`

### **Input Parameters**

$i_0, i_1, i_2$  — The input color triple(s). May be either scalars or arrays of the same length.

### **Output Parameters**

$o_0, o_1, o_2$  — The variables to receive the result. Their structure is copied from the input parameters.

### **Keywords**

One of the following *keywords* is required:

***CMY\_RGB*** — Convert from CMY (cyan, magenta, yellow) to RGB (red, green, blue).

***HLS\_RGB*** — Convert from HLS (hue, lightness, saturation) to RGB .

***HSV\_RGB*** — Convert from HSV (hue, saturation, value) to RGB.

***RGB\_CMY*** — Convert from RGB to CMY.

***RGB\_HLS*** — Convert from RGB to HLS.

***RGB\_HSV*** — Convert from RGB to HSV.

### **Discussion**

RGB and CMY values are bytes in the range of 0 to 255.

Hue is a floating-point number measured in degrees, from 0.0 to 360.0; a hue of 0.0 degrees is the color red, green is 120.0 degrees, and blue is 240.0 degrees.

Saturation, lightness, and value are floating-point numbers in the range of 0.0 to 1.0.

Note that when RGB values are the same during an RGB to HSV conversion, the saturation is set to 0.0 and the hue is undefined.

## Example

```
COLOR_CONVERT, 255, 255, 0, h, s, v, /RGB_HSV
; Converts the RGB color triple (0, 255, 255), which is the color
; yellow at full intensity and saturation, to the HSV system.

PRINT, h, s, v
60.00000    1.00000    1.00000
; The resulting hue in the variable h is 60 degrees. The saturation
; and value (s and v) are set to 1.0.
```

## See Also

[COLOR\\_EDIT](#), [HLS](#), [HSV](#), [HSV\\_TO\\_RGB](#), [LOADCT](#),  
[MODIFYCT](#), [PALETTE](#), [PSEUDO](#), [RGB\\_TO\\_HSV](#), [STRETCH](#), [TVLCT](#),  
[WgCeditTool](#)

For more background information about color systems, see the *PV-WAVE User's Guide*.

For a discussion of the various color systems, see *Computer Graphics: Principles and Practice*, by Foley, Van Dam, Feiner, and Hughes, Second Edition, Addison Wesley Publishing Company, Reading, MA, 1990, pp. 585-596.

---

## ***COLOR\_EDIT Procedure***

Standard Library procedure that lets you interactively create color tables based on the HLS or HSV color system.

### Usage

```
COLOR_EDIT [, colors_out]
```

### Input Parameters

None.

## Output Parameters

*colors\_out* — (optional) Contains the color values of the final color table in the form of a two-dimensional array that has the number of colors in the color table as the first dimension and the integer 3 as the second dimension.

The values for red are stored in the first row, the values for green are stored in the second row, and those for blue in the third row; in other words:

$$red = colors\_out(*, 0)$$
$$green = colors\_out(*, 1)$$
$$blue = colors\_out(*, 2)$$

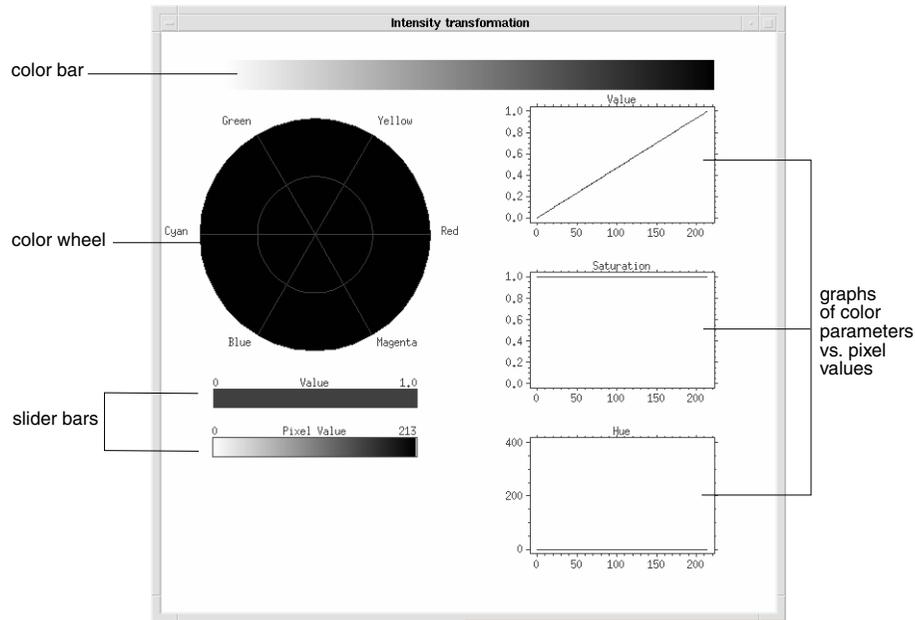
## Keywords

*HLS* — If set to 1, indicates the HLS (hue, lightness, saturation) color system should be used.

*HSV* — If set to 1, indicates the HSV (hue, saturation, value) color system should be used. (Default: 1)

## Discussion

COLOR\_EDIT creates an interactive window that lets you use the mouse to create a new color table. This window is shown in .



**Figure 2-7** The COLOR\_EDIT window lets you use the mouse to create a new color table based on either the HLS or HSV color system.

COLOR\_EDIT not only changes the colors displayed in the window that it creates, it also changes the colors in other windows so that you can watch different anomalies rise out of your data.

---

**TIP** If you need greater control of HSV colors near zero percent saturation, use the C\_EDIT procedure.

---

The COLOR\_EDIT window contains the following items:

- **Color Bar** — Displays the current color table. It is updated as changes are made to the color table. (When COLOR\_EDIT is initialized, it sets the current color table to red.)
- **Color Wheel** — Lets you simultaneously select the hue (position from the azimuth of the wheel) and saturation (distance from the center of the wheel) with the cursor.
- **Slider Bars** — Use the top bar to select either the value (HSV system) or the lightness (HLS system) parameter, depending on the system in use. Use the

bottom bar to select the pixel value that will become a tie point (explained below).

- **Graphs** — Plots the current values of the three color system parameters versus pixel value. These graphs are updated as tie points are selected and the color table is changed.

To use COLOR\_EDIT:

- Adjust the **Value/Lightness** slider bar and color wheel by dragging the left mouse button within each until you reach the first color you want in your color table.
- On the **Pixel Value** slider bar, click with the left mouse button at the position where you want that particular color to be. The range on this bar begins at 0 and ends with the maximum value for your color table.)

A small *tie point* then appears indicating the exact point where this color will occur in the color table. The values in the color table are interpolated between the tie points.

- If you need to erase the tie point, simply click on it with the middle mouse button.

---

**Windows USERS** If you have a two-button mouse, use <Alt> in combination with the left mouse button to erase the tie point.

---

The color system parameter graphs on the right of the window and the color bar at the top of the window are updated whenever a tie point is created or removed.

- Create your second color by again adjusting the **Value/Lightness** slider bar and color wheel and entering the corresponding tie point in the **Pixel Value** slider bar.

Repeat this step until you have finished creating all the colors you want in your color table.

- Use the right mouse button to exit the procedure.

For more information on using interactive color table procedures, see [Sample Usage on page 111](#).

## Example 1

```
TVSCL, FINDGEN(256, 256)
COLOR_EDIT, rgb_array
```

— User modifies the color table and exits the procedure. —

```
SAVE, filename='my_colortable', rgb_array
LOADCT, 2
RESTORE, 'my_colortable'
rgb_array=REFORM(rgb_array, N_ELEMENTS(rgb_array)/3, 3)
TVLCT, rgb_array(*, 0), rgb_array(*, 1), rgb_array(*, 2)
```

## Example 2

```
TVSCL, FINDGEN(256, 256)
COLOR_EDIT
```

— User modifies the color table and exits the procedure. —

```
TVLCT, r, g, b, /get
SAVE, filename='my_colortable_2', r, g, b
LOADCT, 8
RESTORE, 'my_colortable_2'
TVLCT, r, g, b
```

## See Also

[C\\_EDIT](#), [COLOR\\_CONVERT](#), [COLOR\\_PALETTE](#), [HLS](#), [HSV](#), [LOADCT](#), [MODIFYCT](#), [PALETTE](#), [PSEUDO](#), [STRETCH](#), [TVLCT](#), [WgCbarTool](#), [WgCeditTool](#), [WgCtTool](#)

For additional background information about color systems, see the *PV-WAVE User's Guide*.

For an excellent discussion of the HSV and HLS color systems, see *Computer Graphics: Principles and Practice*, by Foley, Van Dam, Feiner, and Hughes, Second Edition, Addison Wesley Publishing Company, Reading, MA, 1990.

---

## ***COLOR\_PALETTE Procedure***

Standard Library procedure that displays the current color table colors and their associated color table indices.

### **Usage**

COLOR\_PALETTE

### **Parameters**

None.

### **Keywords**

None.

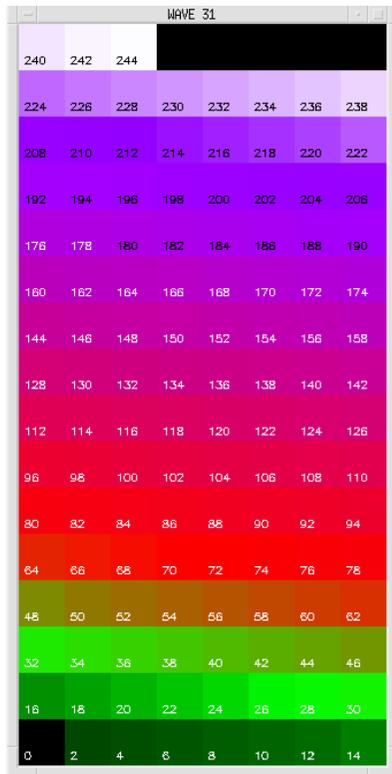
### **Discussion**

COLOR\_PALETTE works only on displays with window systems. It displays the current color table in a new window, along with the corresponding numerical values or color table indices, thereby letting you visually determine the color associated with a particular color index. This window (Motif version only) is shown in .

---

**Windows USERS** The total number of colors that can appear in the COLOR\_PALETTE window is 236, which reflects the current value of !D.N\_Colors. The black cells in the upper-right corner of the window represent colors that are not available to PV-WAVE because they have been reserved by Windows.

---



**Figure 2-8** The COLOR\_PALETTE window (Motif version). This window displays every other color in the current color table, along with the corresponding numerical value or color table index. The black cells in the upper-right corner of the window represent colors that are not available to PV=WAVE because they have been reserved by another application, such as the window manager.

## Example 1

```

b = FINDGEN(37)
x = b * 10
y = SIN(x * !Dtor)
    ; Create an array containing the values for a sine function from 0 to 360 degrees.
PLOT, x, y, XRange=[0,360], XStyle=1, YStyle=1
    ; Plot data and set the range to be exactly 0 to 360.
COLOR_PALETTE
    ; Put up a window containing a display of the current color table and
    ; its associated color indices.
TEK_COLOR
    ; Load a predefined color table that contains 32 distinct colors.

```

```

POLYFILL, x, y, Color=6
POLYFILL, x, y/2, Color=3
POLYFILL, x, y/6, Color=4
    ; Fill in areas under the curve with different colors.
z = COS(x * !Dtor)
    ; Create an array containing the values for a COS function from 0 to 360 degrees.
OPLOT, x, z/8, Linestyle=2, Color=5
    ; Plot the cosine data on top of the sine data.

```

## Example 2

```

OPENR, lun, !Data_dir + 'head.img', /Get_lun
image = BYTARR(512, 512)
READU, lun, image
LOADCT, 0
TVSCL, image
COLOR_PALETTE
LOADCT, 5
LOADCT, 3

```

## See Also

[COLOR\\_CONVERT](#), [COLOR\\_EDIT](#), [MODIFYCT](#), [PALETTE](#), [WgCeditTool](#)

For more information about the number of colors that are displayed in the palette, see the *PV-WAVE User's Guide*.

---

## COMPILE Procedure

Saves compiled user-written procedures and functions in a file.

### Usage

```
COMPILE, routine1[ , ..., routinen ]
```

### Input Parameters

*routine<sub>i</sub>*— A string containing the name of the compiled function or procedure that you want to save.

## Keywords

**All** — If nonzero, all currently compiled user-written functions and procedures are saved.

**Filename** — Specifies the name of a file in which to save specified compiled routines. By default, a file named *routine.cpr* is saved in the current working directory.

**Verbose** — If present and nonzero, prints a message for each saved function and procedure.

## Discussion

The COMPILE procedure saves compiled routines in a format (XDR) that is recognized by all the platforms on which PV-WAVE runs.

When a compiled routine is called in a PV-WAVE application, the directories in the !Path system variable are searched for a .cpr file with the same name as the called routine. If the .cpr file is found, it is loaded and immediately executed. If a .cpr file is not found, PV-WAVE searches !Path for a .pro file with the same name. If the .pro file is found, it is executed instead.

With a special runtime license, saved compiled applications can be executed from the operating system level using the runtime mode flag. For example:

```
wave -r filename
```

The -r flag signifies “runtime” mode. It is possible to set an environment variable so that the -r flag is not needed.

To do this enter the following command:

```
(UNIX)      setenv WAVE_FEATURE_TYPE RT
```

```
(OpenVMS)  DEFINE WAVE_FEATURE_TYPE RT
```

```
(Windows)  set WAVE_FEATURE= RT
```

Then, you can execute compiled routines from the operating system prompt by entering:

```
wave filename
```

Note that you do not use the .cpr extension when you execute compiled routines from the operating system prompt.

---

**NOTE** To execute a runtime mode application, you must have a runtime license. Without a runtime license for PV-WAVE, you will be unable to start PV-WAVE in

runtime mode as described in this section. For information on obtaining a runtime license for PV-WAVE, please contact Visual Numerics.

---

## Example 1

This example demonstrates how to save a single compiled procedure. Assume the following procedure is saved in the current working directory in the file `log_plot.pro`:

```
PRO log_plot
  x = FLTARR(256)
  x(80:120) = 1
  freq = FINDGEN(256)
  freq = freq < (256-freq)
  fil = 1. / (1+(freq / 20) ^2)
  PLOT_IO, freq, ABS(FFT(X,1)), Xtitle = $
    'Relative Frequency', Ytitle = 'Power', $
    Xstyle = 1
  OPLOT, freq, fil
  WAIT, 3
  WDELETE

END
```

Now start PV-WAVE. At the `WAVE>` prompt, compile the procedure with `.RUN`, and save the compiled procedure in a file using the `COMPILE` procedure. Then, delete the compiled procedure from memory and run the compiled procedure that is stored in the file.

```
.RUN log_plot
  ; Compile the procedure.

COMPILE, 'log_plot'
  ; Save the compiled procedure.

$ls log_plot*
  log_plot.cpr  log_plot.pro
  ; The file log_plot.cpr is created. This file contains the compiled
  ; procedure.

DELPROC, 'log_plot'
  ; Delete the log_plot procedure that is currently in memory.

log_plot
  ; Run the compiled procedure log_plot.cpr.

EXIT
  ; Exit PV-WAVE, and return to the operating system prompt.
```

At the system prompt, enter the following:

```
% wave -r -nohome log_plot
```

This command runs the compiled procedure from the operating system prompt. The `-r` flag signifies “runtime” mode.

## Example 2

This example demonstrates how several files from a single application can be compiled and saved in one file. This simple application creates a Command Tool widget and includes three separate callback routines.

Place the following code in a file called `comtool.pro`, and save the file in the current working directory:

Widget commands:

```
PRO ComTool
  top=WwInit('example2', 'Examples', layout)
  button=WwButtonBox(layout, 'Command', 'CbuttonCB')
  status=WwSetValue(top, /Display)
  WwLoop
END
```

Callback routines:

```
PRO CbuttonCB, wid, data
  command = WwCommand(wid, 'CommandOK', $
    'CommandDone', Position=[300,300], $
    Title = 'Command Entry Window')
END
```

```
PRO CommandOK, wid, shell
  value = WwGetValue(wid)
  PRINT, value
END
```

```
PRO CommandDone, wid, shell
  status = WwSetValue(shell, /Close)
END
```

Now start **PV-WAVE** and enter the following commands at the `WAVE>` prompt:

```
.RUN comtool
; Compile the application.
COMPILE, 'ComTool', 'CbuttonCB', 'CommandOK', 'CommandDone', $
  Filename = 'comtool'
```

```
        ; Save the compiled procedures in the file comtool.cpr.
$ls comtool*
    comtool.cpr    comtool.pro
        ; The file comtool.cpr is created. This file contains the compiled
        ; procedures.

EXIT
```

At the system prompt, enter the following:

```
% wave -r comtool
```

This command runs the compiled comtool application from the operating system command line. Note that the filename must be the same as the main procedure in the file for the application to run successfully from the operating system prompt. The `-r` flag signifies “runtime” mode.

## See Also

[RESTORE](#), [SAVE](#)

See the *PV-WAVE Programmer’s Guide* for more information about runtime mode.

---

## COMPLEX Function

Converts an expression to complex data type.

Extracts data from an expression and places it in a complex scalar or array.

### Usage

*result* = COMPLEX(*real* [, *imaginary*])

This form is used to convert data.

*result* = COMPLEX(*expr*, *offset*, [*dim*<sub>1</sub>, *dim*<sub>2</sub>, ... , *dim*<sub>*n*</sub>])

This form is used to extract data.

### Input Parameters

To convert data:

*real* — Scalar or array to be used as the real part of the complex result.

*imaginary* — (optional) Scalar or array to be used as the imaginary part of the complex result. If not present, the imaginary part of the result is zero.

To extract data:

*expr* — The expression to be converted, or from which to extract data.

*offset* — The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim<sub>i</sub>* — (optional) The dimensions of the result. This parameter may be any scalar expression, and up to eight dimensions may be specified.

## Returned Value

If converting:

*result* — The *result* is a complex data type with the size and structure determined by the size and structure of *real* and *imaginary* input parameters. If either or both of the parameters are arrays, *result* will be an array, following the same rules as standard PV-WAVE operators.

If extracting:

*result* — The *result* is a complex data type with the size and structure determined by the size and structure of the *dim<sub>i</sub>* parameters. If no dimensions are specified, the result is scalar.

## Keywords

None.

## Discussion

COMPLEX is used primarily to convert data to complex data type. If *real* is of type string and if the string does not contain a valid floating-point value (thereby making it impossible to convert), then PV-WAVE returns 0 and displays a notice.

Otherwise, *expr* is converted to complex data type. The ON\_IOERROR procedure can be used to establish a statement to jump to in the case of such errors.

If only one parameter is supplied, the imaginary part of the result is 0; otherwise, it is set by the *imaginary* parameter. Parameters are first converted to single-precision floating-point.

---

**NOTE** If three or more parameters are supplied, COMPLEX extracts fields of data from *expr*, rather than performing conversion.

---

## Example

```
real = INDGEN(5)
b = COMPLEX(real)
INFO, b
      B              COMPLEX   = Array(5)
PRINT, b
      (      0.00000,      0.00000)
      (      1.00000,      0.00000)
      (      2.00000,      0.00000)
      (      3.00000,      0.00000)
      (      4.00000,      0.00000)
img = INTARR(5) + 6
c = COMPLEX(real, img)
INFO, c
      C              COMPLEX   = Array(5)
PRINT, c
      (      0.00000,      6.00000)
      (      1.00000,      6.00000)
      (      2.00000,      6.00000)
      (      3.00000,      6.00000)
      (      4.00000,      6.00000)
d = COMPLEX(real, 7)
INFO, d
      D              COMPLEX   = Array(5)
PRINT, d
      (      0.00000,      7.00000)
      (      1.00000,      7.00000)
      (      2.00000,      7.00000)
      (      3.00000,      7.00000)
      (      4.00000,      7.00000)
e = COMPLEX(7, img)
INFO, e
      E              COMPLEX   = Array(5)
PRINT, e
      (      7.00000,      6.00000)
      (      7.00000,      6.00000)
      (      7.00000,      6.00000)
      (      7.00000,      6.00000)
      (      7.00000,      6.00000)
```

## See Also

[BYTE](#), [COMPLEXARR](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#)

For more information on using this function to extract data, see the *PV-WAVE Programmer's Guide*.

---

## COMPLEXARR Function

Returns a complex single-precision floating-point vector or array.

### Usage

```
result = COMPLEXARR(dim1 [, dim2, ... , dimn])
```

### Input Parameters

*dim<sub>i</sub>* — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### Returned Value

*result* — A complex single-precision floating-point vector or array.

### Keywords

*Nozero* — If *Nozero* is nonzero, the normal zeroing (see *Discussion*) is not performed, thereby causing COMPLEXARR to execute faster.

### Discussion

Normally, COMPLEXARR sets every element of the result to zero.

### Example

```
c = COMPLEXARR(4)
INFO, c
      C              COMPLEX      = Array(4)
PRINT, c
      (      0.00000,      0.00000)
      (      0.00000,      0.00000)
```

```
(      0.00000,      0.00000)
(      0.00000,      0.00000)
```

## See Also

[BYTARR](#), [CINDGEN](#), [DBLARR](#), [FLTARR](#), [INTARR](#), [LONARR](#)

---

## CONE Function

Defines a conic object that can be used by the RENDER function.

### Usage

*result* = CONE()

### Parameters

None.

### Returned Value

*result* — A structure that defines a conic object.

### Keywords

**Color** — A 256-element double-precision floating-point vector containing the color (intensity) coefficients of the object. (Default: `Color(*) = 1.0`)

**Decal** — A 2D array of bytes whose elements correspond to indices into the arrays of material properties.

**Kamb** — A 256-element double-precision floating-point vector containing the ambient (flat shaded) coefficients. (Default: `Kamb(*) = 0.0`)

**Kdiff** — A 256-element double-precision floating-point vector containing the diffuse reflectance coefficients. (Default: `Kdiff(*) = 1.0`)

**Ktran** — A 256-element double-precision floating-point vector containing the specular transmission coefficients. (Default: `Ktran(*) = 0.0`)

**Radius** — A double-precision floating-point number that corresponds to a scaling factor in the range [0...1]. *Radius* is multiplied by the upper radius at  $Z = +0.5$  to give the lower radius at  $Z = -0.5$ . (Default: `Radius = 0.0`)

**Transform** — A 4-by-4 double-precision floating-point array containing the local transformation matrix whose default is the identity matrix.

## Discussion

CONE is used by the RENDER function to render conic objects, such as caps on axes. By default, it is centered at the origin with a height of 1.0, and has an upper radius of 0.5 (at  $Z = +1/2$ ) and a lower radius of 0 (at  $Z = -1/2$ ).

To change the upper radius, use the *Scale* keyword with the T3D procedure.

To change the lower radius, use the *Radius* keyword. For example, *Radius=0.5* corresponds to a conic object whose lower radius is one-half of the upper radius, while *Radius=0.0* corresponds to a point whose lower radius is 0 (a conic that ends in a point).

To change the dimensions and orientation of a CONE, use the *Transform* keyword.

## Example

```
T3D, /Reset, Rotate=[90, 0., 0]
c = CONE(Radius=0.33, Transform=!P.T)
TVSCL, RENDER(c)
```

## See Also

[CYLINDER](#), [MESH](#), [RENDER](#), [SPHERE](#), [VOLUME](#)

For more information, see the *PV-WAVE User's Guide*.

---

## CONGRID Function

Standard Library function that shrinks or expands an image or array.

### Usage

```
result = CONGRID(image, col, row)
```

### Input Parameters

*image* — The two-dimensional image to resample. Can be of any data type except string.

*col* — The number of columns to be in the resulting image.

*row* — The number of rows to be in the resulting image.

## Returned Value

*result* — The resampled image or array.

## Keywords

*Interp* — Specifies the interpolation method to be used in the resampling:

If zero, uses the nearest neighbor method.

If nonzero, uses the bilinear interpolation method.

## Discussion

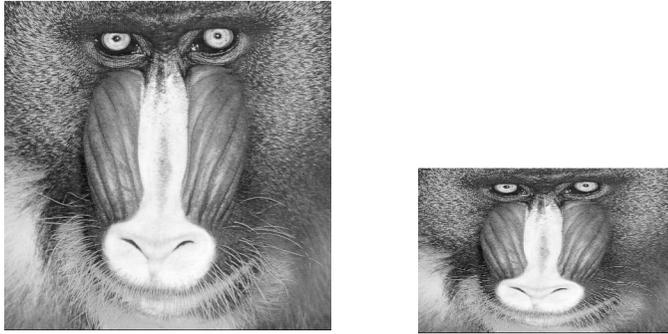
CONGRID shrinks or expands the number of elements in *image* by interpolating values at intervals where there might not have been values before. The resulting image is of the same data type as the input image.

The nearest neighbor interpolation method is not linear, because new values that are needed are merely set equal to the nearest existing value of *image*. Therefore, when increasing the image size, the result may appear as individual blocks. For more information, see the *PV-WAVE User's Guide*.

## Example 1

The following commands demonstrate what the mandril image looks like before and after resizing:

```
OPENR, lun, !Data_dir + 'mandril.img', /Get_lun
mandril_img = BYTARR(512,512)
READU, lun, mandril_img
new_image = CONGRID(mandril_img, 400, 256)
TVSCL, mandril_img
ERASE
TVSCL, new_image
```



**Figure 2-9** CONGRID has been used to shrink this 512-by-512 mandrill image to one measuring 400-by-256.

### Example 2

```
x = DIST(100)
new_x = CONGRID(x, 500, 200)
TVSCL, x
ERASE
TVSCL, new_x
```

### See Also

[BILINEAR](#), [REBIN](#)

---

## CONJ Function

Returns the complex conjugate of the input variable.

### Usage

```
result = CONJ(x)
```

### Input Parameters

**x** — The variable that is evaluated. The variable can be a single or double-precision complex scalar or array.

## Returned Value

*result* — The complex conjugate of  $x$ .

## Keywords

None.

## Discussion

If  $x$  is single-precision complex, the result is single-precision complex. If  $x$  is double-precision complex, the result is double-precision complex.

CONJ is defined as:

$$f(i, j) \equiv (i, -j)$$

where  $i$  represents the real part of  $x$ , and  $j$  represents the imaginary part of  $x$ .

If  $x$  is an array, the result has the same structure, with each element containing the complex conjugate of the corresponding element of  $x$ .

## Example

```
p = COMPLEX(0, 1)
PRINT, p
      (  0.00000,      1.00000)
PRINT, CONJ(p)
      (  0.00000,     -1.00000)
```

## See Also

[COMPLEX](#), [COMPLEXARR](#), [DCOMPLEX](#), [DCOMPLEXARR](#),  
[IMAGINARY](#)

---

## CONTOUR Procedure

Draws a contour plot from data stored in a rectangular array.

### Usage

CONTOUR, z [, x, y]

### Input Parameters

*z* — A 2D array containing the values that make up the contour surface.

*x* — (optional) A vector or 2D array specifying the *x*-coordinates for the contour surface.

*y* — (optional) A vector or 2D array specifying the *y*-coordinates for the contour surface.

### Keywords

The CONTOUR keywords let you control many aspects of the contour plot's appearance. These keywords are listed in the following table. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

Background	Gridstyle	Title
Channel	Levels	[XYZ]Charsize
Charsize	Max_Value	[XYZ]Gridstyle
Charthick	NLevels	[XYZ]Margin
Clip	Noclip	[XYZ]Minor
Color	Nodata	[XYZ]Range
C_Annotation	Noerase	[XYZ]Style
C_Charsize	Normal	[XYZ]Tickformat
C_Charthick	Overplot	[XYZ]Ticklen
C_Colors	Path_Filename	[XYZ]Tickname
C_Labels	Position	[XYZ]Ticks
C_Linestyle	Spline	[XYZ]Tickv

C_Thick	Subtitle	[XYZ>Title
Data	T3d	[XYZ>Type
Device	Thick	YLabelCenter
Follow	Tickformat	ZAxis
Font	Ticklen	ZValue

---

## Discussion

If the  $x$  and  $y$  parameters are provided, the contour is plotted as a function of the X,Y locations specified by their contents. Otherwise, the contour is generated as a function of the array index of each element of  $z$ .

If  $x$  is a vector, each element of  $x$  specifies the  $x$ -coordinate for a column of  $z$ . For example,  $X(0)$  specifies the  $x$ -coordinate for  $Z(0, *)$ . If the  $x$  parameter is a 2D array, each element of  $x$  specifies the  $x$ -coordinate of the corresponding point in  $z$  ( $x_{ij}$  specifies the  $x$ -coordinate for  $z_{ij}$ ).

If  $y$  is a vector, each element of  $y$  specifies the  $y$ -coordinate for a row of  $z$ . If the  $y$  parameter is a 2D array, each element of  $y$  specifies the  $y$ -coordinate of the corresponding point in  $z$  ( $y_{ij}$  specifies the  $y$ -coordinate for  $z_{ij}$ ).

CONTOUR draws contours using one of two different methods:

- The first method, used by default, examines each array cell and draws all contours emanating from that cell before proceeding to the next cell. This method is efficient in terms of computer resources, but does not allow contour labeling.
- The second method searches for each contour line and then follows the line until it reaches a boundary or closes. This method gives better-looking results with dashed line styles, and allows contour labeling, but requires more computer time. It is used if any of the following keywords is specified: *C\_Annotation*, *C\_Charsize*, *C\_Charthick*, *C\_Labels*, *Follow*, *Spline*, or *Path\_Filename*.

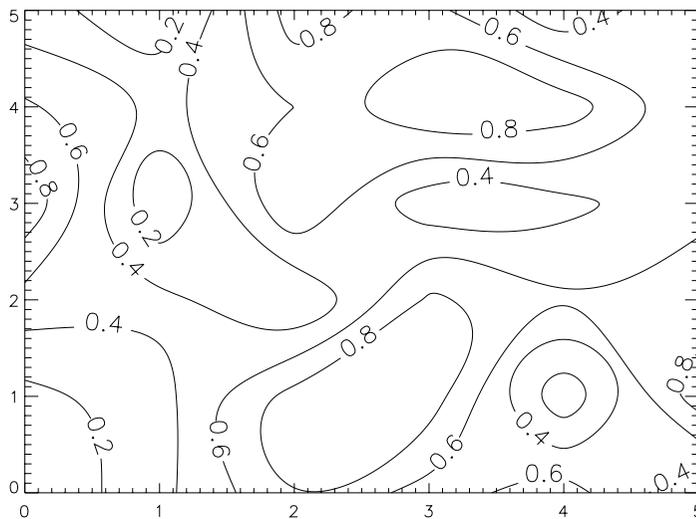
Although these two methods both draw correct contour maps, differences in their algorithms can cause small differences in the resulting graph.

## Example

In the example below, a contour plot of random data is plotted. The random data is generated with the PV-WAVE:IMSL Statistics RANDOMOPT procedure. The *Spline* keyword causes the contours to be smoothed using cubic splines. The vector

assigned to the *Levels* keyword specifies the levels at which contours are desired. The vector of 1's assigned to the *C\_Labels* keyword specifies that all contour levels should be labeled. The *C\_Charsize* keyword is used to increase the size of the labels.

```
RANDOMOPT, Set = 1257
z = REFORM(RANDOM(36), 6, 6)
    ; Create a 6-by-6 array of random numbers.
CONTOUR, z, /Spline, $
    Levels = [0.2, 0.4, 0.6, 0.8], $
    C_Labels = [1, 1, 1, 1], C_Charsize = 1.5
    ; Create a contour plot from the random data.
```



**Figure 2-10** Contour plot of random data.

## See Also

[CONTOUR2](#), [IMAGE\\_CONT](#), [SHOW3](#), [SURFACE](#)

For more information, see

---

## CONTOUR2 Procedure

Draws a contour plot from data stored in an array.

### Usage

CONTOUR2,  $z$  [,  $x$ ,  $y$ ]

### Input Parameters

$z$  — A 1D or 2D array containing values of the dependent variable  $z=z(x, y)$ .

$x$  — A 1D or 2D array containing values of the first independent variable. If  $z$  is 1D, then  $x$  is a required 1D input variable with the same number of elements as  $z$ . If  $z$  is 2D, then  $x$  is an optional 1D or 2D input.

$y$  — A 1D or 2D array containing values of the second independent variable. If  $z$  is 1D, then  $y$  is a required 1D input variable with the same number of elements as  $z$ . If  $z$  is 2D, then  $y$  is an optional 1D or 2D input.

### Keywords

***C\_Fillcolors*** — Specifies an array of color indices used to fill the contour intervals.

***Fill*** — Fills contour intervals with color. This keyword can have the following values:

- 0 No fill (Default)
- 1 Filled intervals with contour lines
- 2 Filled intervals with no contour lines

---

**NOTE** If *Fill* is specified, labeling is disabled. See *Example 2* for information on creating a filled contour plot with labels.

---

***Frequency*** — A floating-point value  $> 0.0$  that determines how frequently labels are printed along the contour. If  $z$  is an  $m$ -by- $n$  array, the default *Frequency* value is:  $\text{MIN}(m, n) / 3 . 5$ . If  $z$  is a 1D array of length  $m$ , the default *Frequency* value is:  $\text{SQRT}(m) / 3 . 5$ .

***Label\_style*** — An integer specifying the contour label fill style. Possible values are:

- 0 Do not print labels (Default)
- 1 Print labels and fill them with the background color. This option takes effect if any other contour label keywords are specified (*C\_Annotation*, *C\_Charsize*, *C\_Labels*, *C\_Charthick*, *Frequency*).
- 2 Print labels with a transparent background.
- 3 Print labels with the fill colors specified by *C\_Fillcolors*.

The CONTOUR2 keywords let you control many aspects of the contour plot's appearance. For a description of each keyword, see [Chapter 3, \*Graphics and Plotting Keywords\*](#).

Background	Gridstyle	[XYZ]Charsize
Channel	Levels	[XYZ]Gridstyle
Charsize	Max_Value	[XYZ]Margin
Charthick	NLevels	[XYZ]Minor
Clip	Noclip	[XYZ]Range
Color	Nodata	[XYZ]Style
C_Annotation	Noerase	[XYZ]Tickformat
C_Charsize	Normal	[XYZ]Ticklen
C_Charthick	Overplot	[XYZ]Tickname
C_Colors	Position	[XYZ]Ticks
C_Labels	Subtitle	[XYZ]Tickv
C_Linestyle	T3d	[XYZ]Title
C_Thick	Thick	[XYZ]Type
Data	Tickformat	YLabelCenter
Device	Ticklen	ZAxis
Font	Title	ZValue

---

## Discussion

CONTOUR2 is an implementation of an algorithm developed by Dr. Albrecht Preusser, "Computing area filling contours for surfaces defined by piecewise polynomials", *Computer Aided Geometric Design* 3 (1986), pp. 267-279. For more information, see the following Web page:

[www.fhi-berlin.mpg.de/~grz/pub/preusser.html](http://www.fhi-berlin.mpg.de/~grz/pub/preusser.html)

CONTOUR2 provides functionality that CONTOUR does not. CONTOUR accepts only gridded data: the  $x$  and  $y$  arrays must define a curvilinear coordinate system. CONTOUR2 places no such restriction on  $x$  and  $y$ , and thus accepts scattered data as well as gridded data.

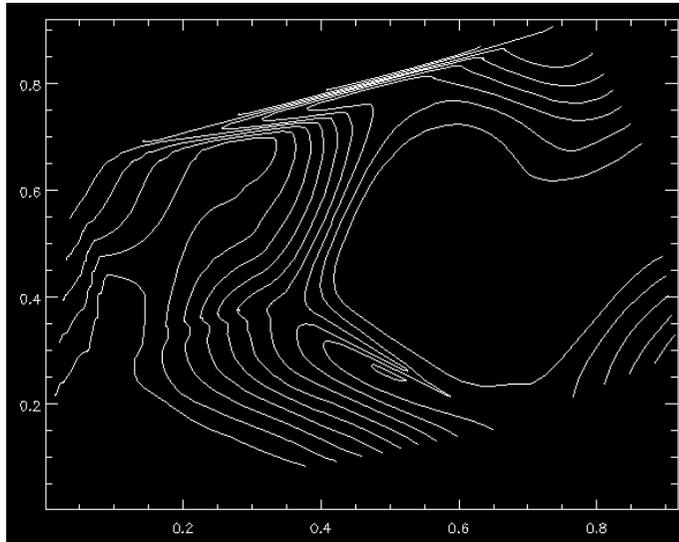
For scattered data  $z$ ,  $x$ , and  $y$  are 1D arrays of the same length.

For gridded data  $z$  is a 2D ( $m$  by  $n$ ) array, while  $x$  and  $y$  can be 2D, 1D, or undefined. If  $x$  and  $y$  are 2D then they are of dimensions  $m$  by  $n$ , and  $z(i,j)$  corresponds to the point ( $x(i,j)$ ,  $y(i,j)$ ). If  $x$  and  $y$  are 1D then they are of lengths  $m$  and  $n$  respectively, and  $z(i,j)$  corresponds to the point ( $x(i)$ ,  $y(j)$ ). If  $x$  and  $y$  are undefined then they default to  $x = \text{FINDGEN}(m)$  and  $y = \text{FINDGEN}(n)$ .

## Example 1

In the example below, randomly scattered data is contoured.

```
seed0=0 & seed1=2 & seed2=5
z = RANDOMU(seed0, 20)
x = RANDOMU(seed1, 20)
y = RANDOMU(seed2, 20)
CONTOUR2, z, x, y, Nlevels=10, /Xstyle, /Ystyle
```



**Figure 2-11** Randomly scattered data is contoured.

## Example 2

In this example, a filled contour plot with labels is plotted. Labeling is not active for filled plots, so we'll generate the filled plot first, then plot the contour lines with labels over the filled plot using the *NoErase* keyword. We'll slightly enlarge the size and thickness of the contour labels without affecting the axis text by using the *C\_Charsize* and *C\_Charthick* keywords.

```
TEK_COLOR
    ; Define a color table.

z = DIST(5)
colorindex=[20, 21, 22]
CONTOUR2, z, /XStyle, /YStyle, NLevels=5, Fill=2, $
    C_Fillcolors=colorindex
    ; Generate a filled plot with no contour lines.
    ; Fill contours with colors defined in the colorindex array.
CONTOUR2, z, /XStyle, /YStyle, NLevels=5, Label_style=3, $
    /NoErase, C_Colors=0, C_Fillcolors=colorindex, $
    C_Charsize=1, C_Charthick=1
    ; Fill contour labels with colors defined in the colorindex array.
```

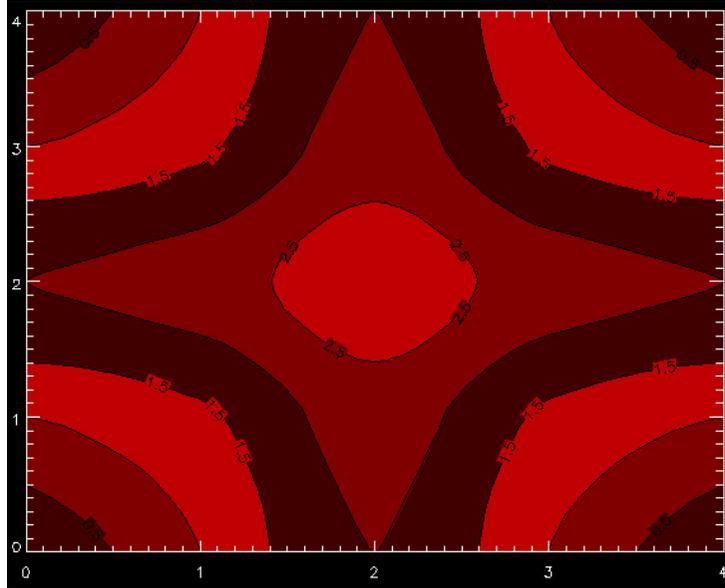


Figure 2-12 Contour plot with contour labels and fill color.

### See Also

[CONTOUR](#), [IMAGE\\_CONT](#), [SHOW3](#), [SURFACE](#)

For more information on contour plots, see the *PV-WAVE User's Guide*.

---

## **CONTOURFILL Procedure**

Standard Library procedure that fills both open and closed contours with specified colors or patterns.

### Usage

CONTOURFILL, *filename*, *z* [, *x*, *y*]

### Input Parameters

*filename* — The name of the file containing the contour paths. This file is created using the CONTOUR procedure with the *Path\_Filename* keyword.

*z* — A 2D array used to generate the contour surface. This array is the same as the one used by CONTOUR.

*x* — (optional) A vector specifying the *x*-coordinates used to generate the contours. This vector is the same as the one used by CONTOUR.

*y* — (optional) A vector specifying the *y*-coordinates used to generate the contours. This array is the same as the one used by CONTOUR.

## Keywords

**Color\_Index** — If present, specifies an array containing the color indices to be used in the plot. Element *i* of this array contains the color of contour level number *i* – 1. Element 0 contains the background color. There must be one more color index than there are number of contour levels.

If not present, the contour colors span the range of available colors.

**Delete\_File** — If present, deletes *filename* after the CONTOURFILL procedure finishes.

**Pattern** — A 3D array containing the patterns used to fill the various contour levels. Each pattern is an *n*-by-*m* rectangular array of pixels. (See the description of the *Pattern* graphics keyword in [Chapter 3, Graphics and Plotting Keywords](#), for an example.)

If *NP* number of patterns are specified, *Pattern* will be dimensioned (*n*, *m*, *NP*). The patterns are used to fill the various contour levels. If there are more levels than patterns, the patterns will be cyclically repeated.

**XRange** and **YRange** — The desired data range of the *x* and *y*-axes, specified as a two-element vector. The first element is the axis minimum, and the second is the maximum. PV-WAVE will frequently round this range. You must use the *XRange* and *YRange* keywords with CONTOURFILL if:

- The *XRange* and *YRange* keywords are used in the CONTOUR procedure call that is used to generate input for CONTOURFILL, and
- *XRange* and *YRange* are different from the array bounds of the *z* parameter (the contour surface data), or the minimum and maximum of the *x* and *y* parameters, when given.

Defaults: When the *z* parameter has  $\text{dim}_1 = nx$  and  $\text{dim}_2 = ny$ ,  $XRange=[0, nx-1]$  and  $YRange=[0, ny-1]$ . When *x* and *y* parameters are given,  $XRange=[MIN(x), MAX(x)]$  and  $YRange=[MIN(y), MAX(y)]$ .

---

**TIP** For best results, the *XRange* and *YRange* keywords used with CONTOURFILL should match the ones used with CONTOUR.

---

## Discussion

CONTOURFILL can be used with CONTOUR to fill the area between the contour lines with a solid color or a user-defined pattern. The procedure closes the open contours as long as the *z* (and *x* and *y*, if used in the calling sequence) parameter is specified in exactly the same manner as was used with the CONTOUR procedure.

---

**TIP** If you are plotting a large data set, use the EMPTY procedure to be sure that all buffered output is written to the current graphics device.

---

---

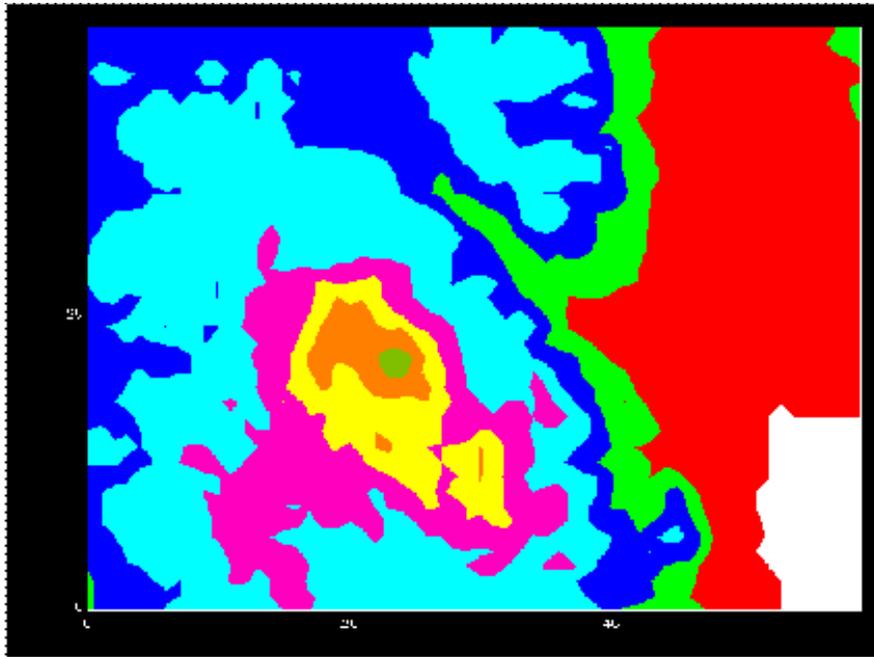
**NOTE** CONTOURFILL creates a temporary file named *filename+1*, so you must have write permission in the directory where *filename* exists.

---

## Example 1

This example creates a contour plot of the Pike's Peak elevation demo file, with the area in between the contour lines filled with a solid color.

```
OPENR, 1, !Data_dir + 'pikeselev.dat'
pikes = FLTARR(60, 40)
READF, 1, pikes
      ; Read in the data file.
TEK_COLOR
      ; Load a color table.
CONTOUR, pikes, Levels=[5,6,7,8,9,10,11,12,13,14,15]*1000, $
      Path='path.dat', XStyle=1, YStyle=1
      ; Contour the data and store the results in file path.dat.
CONTOURFILL, 'path.dat', pikes, Color_Index=INDGEN(12)
      ; Display the contour plot with contours filled with solid colors.
```



**Figure 2-13** Contour plot of Pike's Peak elevation filled with solid colors.

The following commands fill in the area in between the contour lines with a user-defined pattern.

```
pat1 = BYTARR(3, 3)
pat1(1, *) = 255
pat1(*, 1) = 255
      ; Create the first pattern, a cross pattern.

pat2 = BYTARR(3, 3)
FOR i = 0, 2 DO pat2(i, i) = 255
      ; Create the second pattern, a diagonal pattern.

pat3 = BYTARR(3, 3)
      ; Create the third pattern, a solid fill of color zero.

pat4 = REPLICATE(255b, 3, 3)
      ; Create the fourth pattern, a solid fill of color 255.

pat5 = BYTARR(3, 3)
FOR i = 0, 2 DO pat5(2-i, i) = 255
      ; Create the fifth pattern, a backwards diagonal pattern.
```

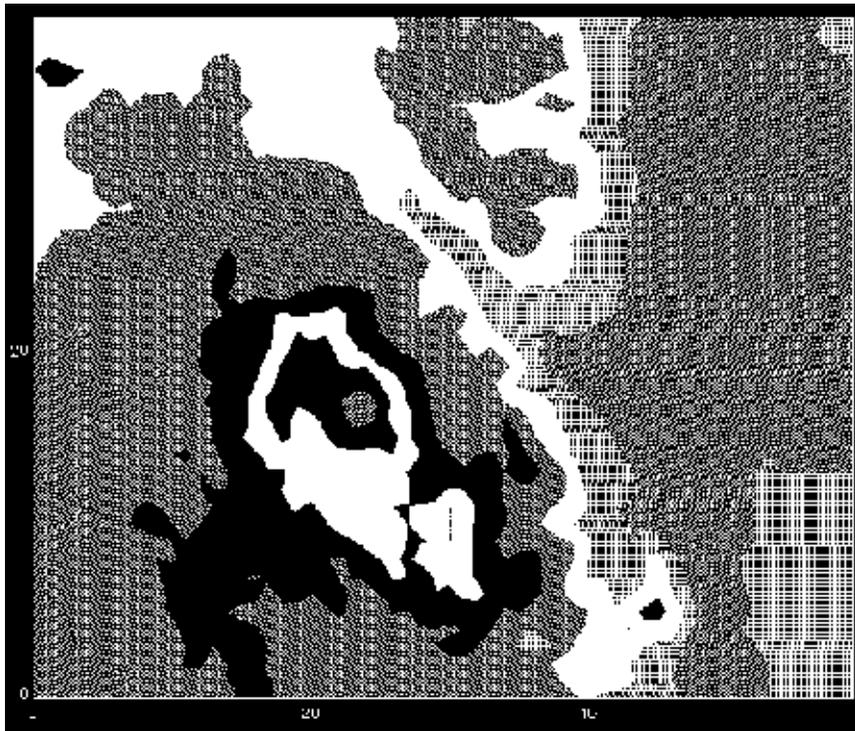
```

pat3d = BYTARR(3, 3, 5)
      ; Create a 3D array in which to store the patterns.

pat3d(*, *, 0) = pat1
pat3d(*, *, 1) = pat2
pat3d(*, *, 2) = pat3
pat3d(*, *, 3) = pat4
pat3d(*, *, 4) = pat5
      ; Store the patterns in the array named pat3d.

CONTOURFILL, 'path.dat', pikes, Pattern = pat3d, /Delete_File
      ; Display the contour plot with the contours lines filled with
      ; the pattern.

```



**Figure 2-14** Pattern-filled contour map of Pike's Peak elevation.

## Example 2

Instead of using CONTOURFILL to create color-filled contour plots, a similar result can be achieved by loading a color table with the TEK\_COLOR and then using a command of the form

```
TV, BYTSCL(array, Top=n)
```

where  $n + 1$  equals the number of contour levels to be colored.

In other words, the previous example could be displayed using the commands:

```
TEK_COLOR  
pikes=REBIN(pikes, 600, 400)  
TV, BYTSCL(pikes, Top = 10)
```

Some of the advantages of using this technique to create color-filled contour plots, instead of the CONTOURFILL procedure, are:

- Easier access to image processing routines that allow you to quickly analyze your data, such as: DEFROI, HISTOGRAM, and PROFILES.
- You don't have to create a temporary file in your directory.

## See Also

[CONTOUR](#), [CONTOUR2](#), [POLYFILL](#), [TEK\\_COLOR](#)

---

## **CONVERT\_COORD Function**

Converts coordinates from one coordinate system to another.

### Usage

```
result = CONVERT_COORD(points)
```

```
result = CONVERT_COORD(x, y [, z])
```

### Input Parameters

*points* — A (2,  $n$ ) or (3,  $n$ ) array of points (or vertices) to convert.

*x* — A scalar or vector parameter providing the  $x$ -coordinates to be converted.

*y* — A scalar or vector parameter providing the  $y$ -coordinates to be converted.

*z* — (optional) If present, a scalar or vector parameter providing the  $z$ -coordinates of the points to be converted.

## Returned Value

*result* — An array of the converted coordinates.

## Keywords

*T3d* — If set, the conversion uses T3d coordinates.

One of the following keywords may be used to specify the input coordinate system. If no input keyword is used, the function defaults to the data coordinate system.

*Data* — Specifies that the input coordinates are based on the data coordinate system.

*Device* — Specifies that the input coordinates are based on the device coordinate system.

*Normal* — Specifies that the input coordinates are based on the normal coordinate system.

One of the following keywords may be used to specify the output coordinate system. If no output keyword is used, the coordinates are converted to the data coordinate system.

*To\_Data* — Converts to the data coordinate system.

*To\_Device* — Converts to the device coordinate system.

*To\_Normal* — Converts to the normal coordinate system.

## Discussion

The CONVERT\_COORD procedure converts among the data, device, and normalized coordinate systems for the currently active window and plot.

A valid data coordinate system must be established before you can convert to or from data coordinates; you may use the PLOT procedure to establish this coordinate system.

## Example

```
xdata = [.1, .2, .5, .8, .9, .5]
ydata = [.3, .6, .9, .6, .3, .1]

PLOT, xdata, ydata
      ; Establish data coordinate system
```

```

point = CONVERT_COORD(0.5, 0.5, /Normal, /To_Data)
      ; Find data coordinate for the center of the window.

PRINT, 'X coord = ', point(0)

PRINT, 'Y coord = ', point(1)

PLOTS, point(0), point(1), Symsize = 5.0, Psym = -1
      ; Print the coordinates and plot a "+" symbol at the center of the window.

```

## See Also

[PLOT](#), [PLOTS](#)

System Variables: [!X.S](#)

For more information, see *Coordinate Conversion* and *Three Graphics Coordinate Systems* in the *PV-WAVE User's Guide*.

## ***CONV\_FROM\_RECT Function***

Converts rectangular coordinates (points) to polar, cylindrical, or one of two spherical coordinate systems: mathematical or global.

### Usage

```
result = CONV_FROM_RECT(vec1, vec2, vec3)
```

### Input Parameters

*vec1* — A 1D array containing the *x* rectangular coordinates.

*vec2* — A 1D array containing the *y* rectangular coordinates.

*vec3* — A 1D array containing the *z* rectangular coordinates. For polar coordinates, set *vec3* to the scalar value 0.

### Returned Value

*result* — By default, global spherical coordinates are returned with (0, \*) containing the longitude, (1, \*) containing the latitude, and (2, \*) containing the radii. Note that latitude angles are given with respect to the horizontal axis or equator.

If the *Sphere* keyword is present and nonzero, mathematical spherical coordinates are returned as in the default case, except that the latitude angles are given with respect to the vertical, or polar, axis.

If the *Polar* keyword is present and nonzero, then a `FLOAT(2, n)` array is returned with (0, \*) containing the angles and (1, \*) containing the radii.

If the *Cylin* keyword is present and nonzero, then a `FLOAT(3, n)` array is returned with (0, \*) containing the angles, (1, \*) containing the radii, and (2, \*) containing the *Z* values.

## Keywords

***Cylin*** — Specifies that cylindrical coordinates are to be returned.

***Degrees*** — If present and nonzero, causes the returned coordinates to be in degrees instead of radians.

***Global*** — If present and nonzero, the function returns global longitude and latitude angles. The longitude angles are the horizontal angles on the Earth's globe, where the angles east of the Greenwich meridian are positive, and angles to the west are negative. The latitude angles are vertical angles rotated with respect to the equator. They are positive in the northern hemisphere and negative in the southern hemisphere. By default, the function returns these global latitude and longitude values; this keyword can be used, however, to add clarity to the function call.

***Polar*** — Specifies that polar coordinates are to be returned.

***Sphere*** — If present and nonzero, the function returns a spherical coordinate system where the vertical angle is rotated with respect to the vertical (or polar) axis instead of the horizontal axis. The horizontal angles and radii are the same as in the global spherical case. This system is based on the set of conversion equations in the CRC Standard Mathematical Tables.

## See Also

[CONV\\_TO\\_RECT](#)

For more information, see the *PV-WAVE User's Guide*.

---

## CONVOL Function

Convolves an array with a kernel (or another array).

### Usage

*result* = CONVOL(*array*, *kernel* [, *scale\_factor*])

### Input Parameters

***array*** — The array to be convolved. The array can be of any data type except string.

***kernel*** — The array used to convolve each value in *array*. The dimensions of *kernel* must be smaller than those of *array*, but they can be of any data type except string. (If a string array is used, PV-WAVE will attempt to convert it and then issue an error message.)

***scale\_factor*** — (optional) A scaling factor that reduces each output value by the specified factor. The *scale\_factor* parameter can be used with integer and byte type data only. (Default: 1.0)

### Returned Value

***result*** — The convolved array of the same data type and dimensions as *array*.

### Keywords

***Center*** — Specifies how the kernel is to be centered. If nonzero or not specified, centers the kernel over each array data point. If explicitly set to zero, centers the kernel a half kernel width to the left of each array data point.

***Edge*** — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

    'zero' — Sets the border of the output image to zero. (Default)

    'copy' — Copies the border of the input image to the output image.

***Zero\_Negatives*** — If set, all negative values in the result are set to zero.

### Discussion

Convolution is a general process that can be used in smoothing, signal processing, shifting, edge detection, and other filtering functions. Therefore, it is often used in conjunction with other functions, such as DIGITAL\_FILTER, SMOOTH, and SHIFT.

---

**TIP** When using CONVOL with image data, make sure the data has been first converted to floating-point type.

---

The *kernel* parameter is an array whose dimensions describe the size of the neighborhood surrounding the value in *array* that is analyzed. The *kernel* also includes values that give a weighting to each point in its array. These weightings determine the average that is the value in the output array. If *kernel* is not of the same type as *array*, a copy is made and converted into the same type before being used.

Using the *scale\_factor* parameter allows you to simulate fractional kernel values and avoid overflow with byte parameters.

In many signal and image processing applications, it is useful to center a symmetric kernel over the data, to align the result with the original array. The *Center* keyword controls the alignment of the *kernel* with the *array* and the ordering of the kernel elements.

### **Sample Usage**

In the convolution of any two functions,  $r(t)$  and  $s(t)$ , for most applications function  $s$  is typically a signal or data stream, which goes on indefinitely in time, while  $r$  is a response function, typically a peaked function that falls to zero in both directions from its maximum.

In terms of CONVOL parameters,  $s$  corresponds to *array* and  $r$  corresponds to *kernel*. The effect of convolution is to smear the signal  $s(t)$  in time according to the “recipe” provided by the response function  $r(t)$ .

### **One-Dimensional Convolution**

For the example below, assume the following equation:

$$R = \text{CONVOL}(A, K, S)$$

where  $A$  is an  $n$ -element vector,  $K$  is an  $m$ -element vector ( $m < n$ ), and  $S$  is the scale factor.

- If the *Center* keyword is set to 0, the results are as follows.

When  $t \geq m - 1$ , then:

$$R_t = (1/S) \sum_{i=0}^{m-1} A_{t-i} K_i$$

Otherwise,  $R_t = 0$ .

- If the *Center* keyword is omitted or set to 1, the results are shown as follows.

When  $(m-1)/2 \leq t \leq n-1-(m-1)/2$ , then:

$$R_t = (1/S) \sum_{i=0}^{m-1} A_{t+i-m/2} K_i$$

Otherwise,  $R_t = 0$ .

### **Two-Dimensional Convolution**

For the second example, assume the same equation:

$$R = \text{CONVOL}(A, K, S)$$

where  $A$  is an  $m$ -by- $n$  element array,  $K$  is an  $l$ -by- $l$  element kernel,  $S$  is the scale factor, and the result  $R$  is an  $m$ -by- $n$  element array.

- If the *Center* keyword is set to 0, the results are as follows.

When  $t \geq l-1$  and  $u \geq l-1$ , then:

$$R_{t,u} = (1/S) \sum_{i=0}^{l-1} \sum_{j=0}^{l-1} A_{t-i,u-j} K_{i,j}$$

Otherwise,  $R_{t,u} = 0$ .

- The centered two-dimensional case is similar, except the  $t-i$  and  $u-j$  subscripts are replaced by  $t+i-l/2$  and  $u+j-l/2$ .

### **Example**

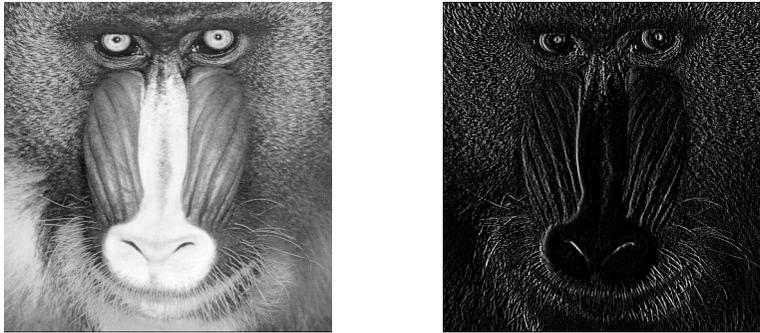
This example demonstrates what a 512-by-512 mandrill image looks like before and after applying the CONVOL function. The following parameters were used:

```
result = CONVOL(mandrill_img, kernel, /Center)
```

where `kernel` is a 3-by-3 array. This array has the following value:

$$\begin{bmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{bmatrix}$$

This kernel value represents a commonly-used algorithm for edge enhancement.



**Figure 2-15** The CONVOL function has been used to enhance the edges of this 512-by-512 mandrill image. In other words, after CONVOL is applied, the dark colors change quickly to light ones.

### See Also

[DIGITAL\\_FILTER](#), [ROBERTS](#), [SHIFT](#), [SMOOTH](#), [SOBEL](#)

For more information on displaying images, see the *PV-WAVE User's Guide*.

For a signal processing example, see the [DIGITAL\\_FILTER Example](#) section.

---

## ***CONV\_TO\_RECT Function***

Converts polar, cylindrical, or spherical (mathematical or global) coordinates to rectangular coordinates (points).

### **Usage**

*result* = CONV\_TO\_RECT(*vec1*, *vec2*, *vec3*)

### **Input Parameters**

*vec1* — A 1D array containing the polar (longitude) angles.

*vec2* — A 1D array containing the latitude angles, unless the *Polar* or *Cylin* keywords are present and nonzero. If either keyword is specified, then *vec2* should contain the radii.

*vec3* — A 1D array containing the radii for spherical coordinates, unless *Polar* or *Cylin* keywords are present and nonzero. If *Polar* is specified, then *vec3* should be the scalar value 0 (it is ignored). If *Cylin* is specified, then *vec3* should contain the *z* values.

## Returned Value

**result** — If the *Polar* keyword is present and nonzero, then a `FLOAT(2, n)` array is returned with (0, \*) containing the *x*-coordinates and (1, \*) containing the *y*-coordinates.

If *Polar* is zero (or not present), then a `FLOAT(3, *)` array is returned with (0, \*) containing the *x*-coordinates, (1, \*) containing the *y*-coordinates, and (2, \*) containing the *z*-coordinates.

## Keywords

**Cylin** — Specifies that the input coordinates are cylindrical.

**Degrees** — If present and nonzero, causes the input coordinates to be in degrees instead of radians.

**Global** — If present and nonzero, causes the input coordinates to be in global longitude and latitude angles. The longitude angles are the horizontal angles on the Earth's globe, where the angles east of the Greenwich meridian are positive, and angles to the west are negative. The latitude angles are vertical angles rotated with respect to the equator. They are positive in the northern hemisphere and negative in the southern hemisphere. By default, the function expects these global latitude and longitude values; this keyword can be used, however, to add clarity to the function call.

**Polar** — Specifies that the input coordinates are polar.

**Sphere** — If present and nonzero, causes the input coordinates to be in a spherical coordinate system where the vertical angle is rotated with respect to the vertical (or polar) axis instead of the horizontal axis. The horizontal angles and radii are the same as in the global spherical case. This system is based on the set of conversion equations in the CRC Standard Mathematical Tables.

## Examples

```
PRO vol_demo1
    ; This program displays a 3D fluid flow vector field with random
    ; starting points for the vectors.

volx = 17
voly = 17
volz = 59
    ; Specify the size of the volumes.

winx = 500
winy = 700
    ; Specify the window size.
```

```

flow_axial = FLTARR(volx, voly, volz)
OPENR, 1, !Data_Dir + 'cfd_axial.dat', /Xdr
READU, 1, flow_axial
CLOSE, 1
flow_radial = FLTARR(volx, voly, volz)
OPENR, 1, !Data_Dir + 'cfd_radial.dat', /Xdr
READU, 1, flow_radial
CLOSE, 1
flow_tangent = FLTARR(volx, voly, volz)
OPENR, 1, !Data_Dir + 'cfd_tangent.dat', /Xdr
READU, 1, flow_tangent
CLOSE, 1
    ; Read in the data as cylindrical coordinates.
flow_pressure = FLTARR(volx, voly, volz)
OPENR, 1, !Data_Dir + 'cfd_pressure.dat', /Xdr
READU, 1, flow_pressure
CLOSE, 1
    ; Read in the data to be used for the vector color.
points = CONV_TO_RECT(flow_tangent(*), $
    flow_radial(*), flow_axial(*), /Cylin, /Degrees)
    ; Convert the data from cylindrical coordinates to Cartesian
    ; coordinates.
flow_x = FLTARR(volx, voly, volz)
flow_y = FLTARR(volx, voly, volz)
flow_z = FLTARR(volx, voly, volz)
flow_x(*) = points(0, *)
flow_y(*) = points(1, *)
flow_z(*) = points(2, *)
    ; Split the points array into three 2D arrays to abstract the x, y, z
    ; values from the converted data.
T3D, /Reset
T3D, Translate=[-0.5, -0.5, -0.5]
T3D, Scale=[0.9, 0.9, 0.9]
T3D, Rotate=[0.0, 0.0, -30.0]
T3D, Rotate=[-60.0, 0.0, 0.0]
T3D, Translate=[0.5, 0.5, 0.5]
    ; Set up the transformation matrix for the view.
WINDOW, 0, XSize=winx, YSize=winy, $
    XPos=256, YPos=128, Colors=128, $
    Title='3D Velocity Vector Field'
LOADCT, 4

```

```

; Set up the viewing window and load the color table.
VECTOR_FIELD3, flow_x, flow_y, flow_z, 1000, $
    Max_Length=2.5, Vec_Color=flow_pressure, $
    Min_Color=32, Max_Color=127, $
    Axis_Color=100, Mark_Symbol=2, $
    Mark_Color=90, Mark_Size=0.5, Thick=2
; Plot the converted data as a vector field.
END

```

For another example, see the `vec_demo2` demonstration program in

```

(UNIX)      <wavedir>/demo/ar1
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows) <wavedir>\demo\ar1

```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[CONV\\_FROM\\_RECT](#)

For more information, see the *PV-WAVE User's Guide*.

## ***CORRELATE Function***

Standard Library function that calculates a simple correlation coefficient for two arrays.

### **Usage**

*result* = CORRELATE(*x*, *y*)

### **Input Parameters**

*x* — The X array for which the correlation coefficient is calculated. Can be of any data type except string and it must be of the same data type and have the same number of elements as *y*.

*y* — The Y array for which the correlation coefficient is calculated. Can be of any data type except string and it must be of the same data type and have the same number of elements as *x*.

### **Returned Value**

*result* — The simple product-moment correlation coefficient for *x* and *y*.

## Keywords

None.

## Discussion

CORRELATE calculates the product-moment correlation coefficient of the two arrays that are supplied.

Correlation can be characterized as the probability that values (i.e., the two input arrays) are related. In other words, it measures whether the events in one population are likely to have produced effects in another population. A result of 1.0 indicates a high correlation, while a result of 0.0 indicates no correlation whatsoever.

### Example 1

```
scores_1 = [95,76,60,88,91,97,68,75,82,85]
scores_2 = [93,77,62,87,90,97,67,77,80,86]
scores_corr = CORRELATE(scores_1, scores_2)
PRINT, scores_corr
      .993408
```

### Example 2

```
sample_1 = RANDOMU(seed, 128, 128)
sample_2 = RANDOMU(seed, 128, 128)
samples_corr = CORRELATE(sample_1, sample_2)
PRINT, samples_corr
      0.00
```

### Example 3

```
x = DIST(200)
y = x
exact_corr = CORRELATE(x, y)
PRINT, exact_corr
      1.0000
```

---

## ***COS Function***

Calculates the cosine of the input variable.

### **Usage**

*result* = COS(*x*)

### **Input Parameters**

*x* — The angle for which the cosine is desired, specified in radians.

### **Returned Value**

*result* — The trigonometric cosine of *x*.

### **Keywords**

None.

### **Discussion**

If *x* is of double-precision floating-point or complex data type, COS yields a result of the same type. All other types yield a single-precision floating-point result.

COS handles complex numbers in the following manner:

$$\cos(x) = \text{complex}(\cos(i)\cosh(r), -\sin(r)\sinh(-i))$$

where *r* and *i* are the real and imaginary parts of *x*. If *x* is an array, the result of COS has the same dimensions (size and shape) as *x*, with each element containing the cosine of the corresponding element of *x*.

### **Example**

```
x = [-60, -30, 0, 30, 60]
PRINT, COS(x * !Dtor)
      0.500000 0.866025 1.00000 0.866025  0.500000
```

### **See Also**

[ACOS](#), [ASIN](#), [ATAN](#), [COSH](#), [COSINES](#), [SIN](#), [TAN](#)

For a list of other transcendental functions, see *Transcendental Mathematical Functions* in Chapter 1.

---

## ***COSH Function***

Calculates the hyperbolic cosine of the input variable.

### **Usage**

*result* = COSH(*x*)

### **Input Parameters**

*x* — The angle, in radians, that is evaluated.

### **Returned Value**

*result* — The hyperbolic cosine of *x*.

### **Keywords**

None.

### **Discussion**

COSH is defined by:

$$\cosh(x) \equiv (e^x + e^{-x})/2$$

If *x* is of double-precision floating-point data type, or of complex type, COSH yields a result of the same type. All other data types yield a single-precision floating-point result.

If *x* is an array, the result of COSH has the same dimensions, with each element containing the hyperbolic cosine of the corresponding element of *x*.

### **Example**

```
x = [0.3, 0.5, 0.7, 0.9]
```

```
PRINT, COSH(x)
```

```
1.04534      1.12763      1.25517      1.43309
```

### **See Also**

[COS](#), [SINH](#), [TANH](#)

For a list of other transcendental functions, see *Transcendental Mathematical Functions* in Chapter 1.

---

## ***COSINES Function***

Standard Library basis function that can be used by the SVDFIT function.

### **Usage**

*result* = COSINES(*x*, *m*)

### **Input Parameters**

*x* — A vector of data values with *n* elements.

*m* — The number of terms in the basis function.

### **Returned Value**

*result* — An *n*-by-*m* array, such that:

`result(i, j) = COS(j * x(i))`

### **Keywords**

None.

### **Discussion**

COSINES consists simply of the following two lines:

```
FUNCTION COSINES, x, m
RETURN, COS(x # FINDGEN(m))
```

### **See Also**

[SVDFIT](#)

---

## ***CPROD Function***

Standard Library function that returns the Cartesian product of some arrays.

### **Usage**

*result* = CPROD(*a*)

### **Input Parameters**

*a* — A list of *n* arrays.

### **Returned Value**

*result* — An (*m*,*n*) array where *result*(*i*,\*) is an element of the Cartesian product of the *n* arrays in *a*, and where *result*(\*,*j*) contains only elements from *a*(*j*); *result* is ordered so that *result*(\*,*j*) cycles through the elements of *a*(*j*) in order, and does so faster than *result*(\*,*j*+1) cycles through the elements of *a*(*j*+1).

### **Keywords**

None.

### **Example**

```
pm, cprod( list( [0,1], [0,1,2], [0,1,2,3] ) )
```

---

## ***CREATE \_ HOLIDAYS Procedure***

Standard Library procedure that creates the system variable !Holiday\_List, which is used in calculating Date/Time compression.

### **Usage**

CREATE\_HOLIDAYS, *dt\_list*

### **Input Parameters**

*dt\_list* — A Date/Time variable containing one or more days to be specified as holidays.

## Keywords

None.

## Discussion

The result is stored in the system variable !Holiday\_List, a 50-element Date/Time array. !Holiday\_List is used in calculating Date/Time compressions for functions that take the *Compress* keyword. For instance, the functions DT\_SUBTRACT, DT\_ADD, and DT\_DURATION can take the *Compress* keyword which, if set, will exclude holidays from their results. In addition, the PLOT procedure uses the *Compress* keyword to exclude holidays from a plot.

## Example1

The following commands define !Holiday\_List to contain the dates for Christmas and New Years:

```
holidays=STR_TO_DT(['12-25-92', '1-1-92'], date_fmt=1)
CREATE_HOLIDAYS, holidays
```

## Example 2

```
CREATE_HOLIDAYS, STR_TO_DT('04-july-1992', $
    Date_Fmt=4)
```

## See Also

[CREATE\\_WEEKENDS](#), [DT\\_COMPRESS](#), [LOAD\\_HOLIDAYS](#),  
[STR\\_TO\\_DT](#)

---

## **CREATE\_WEEKENDS Procedure**

Standard Library procedure that creates the system variable !Weekend\_List, which is used in calculating Date/Time compression.

### Usage

```
CREATE_WEEKENDS, day_names
```

### Input Parameters

*day\_names* — A string or string array containing the weekend names.

## Keywords

None.

## Discussion

The result is stored in the system variable `!Weekend_List`, a seven-element integer array. The values in `!Weekend_List` are either ones or zeros, where 1 represents a weekend and 0 represents a weekday. The first element of `!Weekend_List` represents Sunday, and the last represents Saturday. `!Weekend_List` is used in calculating Date/Time compressions for functions that take the *Compress* keyword.

For instance, the routines `DT_SUBTRACT`, `DT_ADD` and `DT_DURATION`, can use the *Compress* keyword which, if set, excludes weekends from their results. In addition, the `PLOT` procedure uses the *Compress* keyword to remove weekends from a plot.

The values in the input string *day\_names* must match or be a substring of strings in the `!Day_Names` system variable. By default, `!Day_Names` contains:

```
PRINT, !Day_Names
      Sunday Monday Tuesday Wednesday Thursday
      Friday Saturday
```

Thus, `day_names = ['Sat', 'Sun']` is a valid assignment. If all days of the week are set to weekends, an error results.

## Example

```
CREATE_WEEKENDS, 'Sat'
      ; Defines Saturday as a weekend.

PRINT, !Weekend_List
      0  0  0  0  0  0  1
      ; The first element in the array !Weekend_List represents
      ; Sunday. The last represents Saturday. Weekend days have
      ; a value of 1.
```

## See Also

[CREATE\\_HOLIDAYS](#), [DT\\_COMPRESS](#), [LOAD\\_WEEKENDS](#)

---

## **CROSSP Function**

Standard Library function that returns the cross product of two three-element vectors.

### **Usage**

*result* = CROSSP(*v*<sub>1</sub>, *v*<sub>2</sub>)

### **Input Parameters**

*v*<sub>1</sub> — The first operand of the cross product. This parameter must be a three-element vector.

*v*<sub>2</sub> — The second operand of the cross product. Must be a three-element vector.

### **Returned Value**

*result* — A three-element floating-point vector containing the cross product of *v*<sub>1</sub> and *v*<sub>2</sub>.

### **Keywords**

None.

### **Discussion**

The cross product of two arrays is commonly used in a variety of applications. It is defined as:

$$v_1 \times v_2 = \begin{bmatrix} i & j & k \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix}$$

or

$$v_1 \times v_2 = (b_1c_2 - b_2c_1)i + (c_1a_2 - c_2a_1)j + (a_1b_2 - a_2b_1)k$$

### **Example**

*v*<sub>1</sub> = [2, 1, -2]

```
v2 = [4, -1, 3]
result = CROSSP(v1, v2)
PRINT, result
      1   -14   -6
```

---

## ***CURSOR Procedure***

Reads the position of the interactive graphics cursor from the current graphics device.

### **Usage**

`CURSOR, x, y [, wait]`

### **Input Parameters**

*wait* — (optional) An integer specifying when `CURSOR` returns. This parameter may be used interchangeably with the five keywords listed below that specify the type of wait.

<b>Value</b>	<b>Keyword</b>	<b>Action</b>
0	<i>Nowait</i>	Return immediately.
1	<i>Wait</i>	Return if button pressed (the default value).
2	<i>Change</i>	Return if button pressed, changed, or pointer moved.
3	<i>Down</i>	Return when button down transition is detected.
4	<i>Up</i>	Return when button up transition is detected.

---

**UNIX and OpenVMS USERS** Not all wait modes work with all display devices. Many devices, such as Tektronix terminals, do not have the ability to return immediately, and so always wait. In addition, not all types of waiting are available for devices that do not have the ability to sense transitions or states.

---

### **Output Parameters**

*x* — A named variable to receive the cursor's current column.

*y* — A named variable to receive the cursor's current row.

## Keywords

**Change** — Waits for pointer movement or button down within the currently selected window.

**Data** — If present and nonzero, causes the values placed into  $x$  and  $y$  to be in data coordinates (the default).

**Device** — If present and nonzero, causes the values placed into  $x$  and  $y$  to be in device coordinates.

**Down** — Waits for a button down transition within the currently selected window.

**Normal** — If present and nonzero, causes the values placed into  $x$  and  $y$  to be in normalized coordinates.

**Nowait** — Reads the pointer position and button status and return immediately. If the pointer is not within the currently selected window, the device coordinates  $-1$ ,  $-1$  are returned.

**Up** — Waits for a button up transition within the current window.

**Wait** — Waits for a button to be depressed within the currently selected window. If a button is already pressed, returns immediately.

## Discussion

CURSOR enables the graphic cursor on the device and waits for the operator to position it. On devices that have a mouse, CURSOR normally waits until a mouse button is pressed. If no mouse is present, CURSOR waits for a key on the keyboard to be pressed.

---

**NOTE** Not all graphics devices have interactive cursors.

---

The system variable !Err is set to the button status; if no mouse is present, it is set to the ASCII code of the key. Each mouse button is assigned a bit in !Err—bit 0 is the left-most button, bit 1 the next, and so on.

Thus, for a three-button mouse, !Err will contain the values 1 – 7, depending upon which button or combination of buttons was pushed. For example, the left button produces a value of 1, the middle button 2, and the right button 4, while pressing the left and right buttons together produce the value 5.

The system variable !Mouse contains the X and Y position of the mouse, the mouse button status, and a date/time stamp. The mouse position is given in device

coordinates. The button status appears as 1 – 7; these values are contained in the !Err system variable. The date/time stamp may not be available on all systems.

Since the values returned are, by default, in data coordinates, if no data coordinate system has been previously established, then calling CURSOR without specifying either the *Normal* or *Device* keywords will result in an error and procedural execution will be halted.

## Example 1

```
WINDOW, XSize=512, YSize=512
CURSOR, x, y, /Normal
    ; This returns the normalized coordinates of the point selected in
    ; the graphics window when a button is pressed. The button press
    ; is the default event activation, and not overtly specified.
```

## Example 2

In this example, PLOTS and CURSOR are used interactively in a loop to build a sketch pad. While the cursor is in the graphics window and a button is held down, CURSOR returns the device coordinates of the cursor. The PLOTS procedure draws a line segment between the previously returned cursor position and the current cursor position.

```
PRO sketch
false = 0
true = 1
window, 0
XYOUTS, 2, 2, "QUIT", Size = 2, /Device
    ; Create a quit button in the window.
PLOTS, [0, 48, 48], [20, 20, 0], /Device
first = true
REPEAT BEGIN
    CURSOR, xnew, ynew, /Device
        ; Get cursor position, placing the x-coordinate in xnew, and
        ; the y-coordinate in ynew.
    IF (xnew LE 48) AND (ynew LE 20) THEN STOP
        ; If cursor position is within the QUIT button, then stop.
    IF first THEN BEGIN
        xold = xnew
        yold = ynew
        first = false
    ENDIF
        ; First time through loop, set xold and yold to be the same
```

```

        ; as xnew and ynew.
PLOTS, [xold, xnew], [yold, ynew], /Device
        ; Plot a line segment from (xold, yold) to (xnew, ynew).
xold = xnew
yold = ynew

ENDREP UNTIL FALSE
END

```

## See Also

[!Err](#), [!Mouse](#), [TVCRS](#)

## ***CURVATURES Function***

Standard Library function that computes curvatures on a parametrically defined surface.

### Usage

$c = \text{curvatures}(s)$

### Input Parameters

$s$  — A 3-element list of 2-dimensional arrays of dimensions  $d$ .

### Returned Value

$c$  — A 2-element list of 2-dimensional arrays of dimensions  $d$ , where  $c(0)$  defines the distribution of minimum curvature and  $c(1)$  defines the distribution of maximum curvature.

### Keyword

$x$  - A 2-element list of vectors defining the independent variables. By default,  $x(i) = \text{findgen}(d(i))$

### Example

See `wave/lib/user/examples/curvatures_ex.pro`.

## See Also

[EUCLIDEAN](#), [JACOBIAN](#), [NORMALS](#)

---

## ***CURVEFIT* Function**

Standard Library function that performs a nonlinear least-squares fit to a function of an arbitrary number of parameters.

### **Usage**

*result* = CURVEFIT(*x*, *y*, *wt*, *parms*, [*sigma*])

### **Input Parameters**

*x* — A vector containing the independent *x*-coordinates of the input data points. There are *n* elements in the vector.

*y* — A vector containing the dependent *y*-coordinates of the input data points. Must have the same number of elements as the *x* input parameter.

*wt* — The vector of weighting factors for determining the weighting of the least-squares fit (see *Discussion*). The vector must be the same size as the *x* input parameter.

*parms* — A six-element vector containing the parameters of the fitted function. On input, it should contain the initial estimates of each parameter.

### **Output Parameters**

*parms* — On output, contains the calculated parameters of the fitted function.

If *parms* is supplied as a double-precision variable, the calculations are performed in double-precision accuracy. Otherwise, the calculations are performed in single-precision accuracy.

*sigma* — (optional) A vector containing the standard deviations for the parameters in *parms*.

### **Returned Value**

*result* — A vector containing the calculated *y*-coordinates of the fitted function.

## Keywords

None.

## Discussion

CURVEFIT uses a nonlinear least-squares method to fit an arbitrary function in which the partial derivatives are known or can be approximated. This is in contrast to linear least-squares fitting methods that would require their fitting functions to be linear in their coefficients.

The initial estimates of *parms* should be as close to the actual values as possible or the solution may not converge. CURVEFIT performs iterations of the fitting function until the chi-squared value for the goodness of fit changes by less than 0.1 percent, or until 20 iterations are reached.

---

**TIP** These initial estimates for *parms* can be calculated from the result of the POLY\_FIT function when it is used to fit a straight line through data.

---

The function to be fit must be defined and called with FUNCT.

CURVEFIT is modified from the program CURFIT found in *Data Reduction and Error Analysis for the Physical Sciences*, by Philip Bevington, McGraw-Hill, New York, 1969. It combines a gradient search with an analytical solution developed from linearizing the fitting function. This method is termed a “gradient-expansion algorithm.”

### **Weighting Factor**

Weighting is useful when you want to correct for potential errors in the data you are fitting to a curve. The weighting factor, *wt*, adjusts the parameters of the curve so that the error at each point of the curve is minimized.

*wt* can have any value, as long as its size is correct. Some possible ways to weight a curve are suggested below (where *i* is an index into *y*, the vector of Y values):

- ✓ For statistical weighting, use  $wt = 1/y_i$
- ✓ For instrumental weighting, use  $wt = 1/(\text{Std dev of } y_i)$
- ✓ For no weighting, use  $wt = 1$
- **Statistical Weighting** — Statistical weighting is useful when you arrived at your dependent values by measuring a number of discrete events with respect to the independent variable, such as counting the number of cars passing through an intersection over 10-minute intervals.

- **Instrumental Weighting** — Instrumental weighting is useful when you are measuring things from a scale, such as length, mass, voltage, or current, and you suspect that unequal errors have been introduced into the data by the measuring device. For example, if an ohm meter has three different scales (one for 0 to 1 ohm, one for 2 to 99 ohms, and one for 100 ohms or more), the weighting factor would be the same for each measurement taken with the same scale.

---

**TIP** In most cases, you would use a different weighting factor for each scale or instrument that was used to measure your original data.

---

- **No Weighting** — If you feel that fluctuations in your data are due to instrument error but that the uncertainties of the measuring device used are equal for all the data collected, you would probably specify no weighting ( $wt = 1$ ).

## Example

For an example, refer to the `gaussfit.pro` file in the Standard Library.

## See Also

[FUNCT](#), [GAUSSFIT](#), [POLY\\_FIT](#)

---

## ***CYLINDER Function***

Defines a cylindrical object that can be used by the `RENDER` function.

### Usage

*result* = `CYLINDER()`

### Parameters

None.

### Returned Value

*result* — A structure that defines a cylinder object.

## Keywords

**Color** — A 256-element double-precision floating-point vector containing the color (intensity) coefficients of the object. (Default: `Color (*) = 1 . 0`)

**Decal** — A 2D array of bytes whose elements correspond to indices into the arrays of material properties.

**Kamb** — A 256-element double-precision floating-point vector containing the ambient (flat shaded) coefficients. (Default: `Kamb (*) = 0 . 0`)

**Kdiff** — A 256-element double-precision floating-point vector containing the diffuse reflectance coefficients. (Default: `Kdiff (*) = 1 . 0`)

**Ktran** — A 256-element double-precision floating-point vector containing the specular transmission coefficients. (Default: `Ktran (*) = 0 . 0`)

**Transform** — A 4-by-4 double-precision floating-point array containing the local transformation matrix whose default is the identity matrix.

## Discussion

A CYLINDER is used by the RENDER function to render cylindrical objects, such as for molecular modeling (symbolizing bonds), or for generating axes and 3D lines. It is defined as having a radius of 0.5 and being centered at the origin with a height of 0.5 in the +z direction and 0.5 in the -z direction.

To change the dimensions and orientation of a CYLINDER, use the *Transform* keyword.

## Example

```
T3D, /Reset, Rotate=[90, 0., 0]
c = CYLINDER(Transform=!P.T)
TVSCL, RENDER (c)
```

## See Also

[CONE](#), [MESH](#), [RENDER](#), [SPHERE](#), [VOLUME](#)

For more information, see the *PV-WAVE User's Guide*.

---

## ***DAY\_NAME Function***

Standard Library procedure that returns a string array or string constant containing the name of the day of the week for each day in a Date/Time variable.

### **Usage**

```
result = DAY_NAME(dt_var)
```

### **Input Parameters**

*dt\_var* — A date/time variable.

### **Returned Value**

*result* — A string array or string constant containing the name of the day of the week for each input Date/Time value.

### **Keywords**

None.

### **Discussion**

The names of the days of the week are string values taken from the system variable !Day\_Names.

### **Example**

```
date = TODAY( )  
day = DAY_NAME(date)  
PRINT, day  
    Monday
```

### **See Also**

[DAY\\_OF\\_WEEK](#), [DAY\\_OF\\_YEAR](#), [MONTH\\_NAME](#)

For more information on Date/Time data, see the *PV-WAVE User's Guide*.

---

## ***DAY\_OF\_WEEK Function***

Returns an array of integers containing the day of the week for each date in a Date/Time variable.

### **Usage**

```
result = DAY_OF_WEEK(dt_var)
```

### **Input Parameters**

*dt\_var* — A Date/Time variable.

### **Returned Value**

*result* — The day of the week expressed as an integer. Day 0 is Sunday and day 6 is Saturday.

### **Keywords**

None.

### **Example**

Assume that you have a Date/Time variable, `date`, for April 13, 1992. To find out which day of the week this date is, enter:

```
day = DAY_OF_WEEK(date)
PRINT, day
      1
      ; The day is a Monday.
```

### **See Also**

[DAY\\_NAME](#), [DAY\\_OF\\_YEAR](#), [MONTH\\_NAME](#)

For more information on Date/Time data, see the *PV-WAVE User's Guide*.

---

## **DAY\_OF\_YEAR Function**

Returns an array of integers containing the day of the year for each date in a Date/Time variable.

### **Usage**

```
result = DAY_OF_YEAR(dt_var)
```

### **Input Parameters**

*dt\_var* — A Date/Time variable.

### **Returned Value**

*result* — An array of integers representing the day of the year for each date in the input variable.

### **Keywords**

None.

### **Discussion**

The result falls in a range between 1 and 365 (or 366 if it is a leap year).

### **Example**

```
today = TODAY()  
      ; Create a Date/Time variable.  
  
daynumber = DAY_OF_YEAR(today)  
PRINT, daynumber  
      106
```

### **See Also**

[DAY\\_NAME](#), [DAY\\_OF\\_WEEK](#), [MONTH\\_NAME](#)

For more information on Date/Time data, see the *PV-WAVE User's Guide*.

---

## ***DBLARR Function***

Returns a double-precision floating-point vector or array.

### **Usage**

*result* = DBLARR(*dim*<sub>1</sub>, ... , *dim*<sub>*n*</sub>)

### **Input Parameters**

*dim*<sub>*i*</sub>— The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### **Returned Value**

*result* —A double-precision floating-point vector or array.

### **Keywords**

*Nozero* — If *Nozero* is nonzero, the normal zeroing is not performed, causing DBLARR to execute faster.

### **Discussion**

Normally, DBLARR sets every element of the result to zero.

### **Example**

```
r = DBLARR(3, 3)
PRINT, r
  0.0000000  0.0000000  0.0000000
  0.0000000  0.0000000  0.0000000
  0.0000000  0.0000000  0.0000000
```

### **See Also**

[BYTARR](#), [COMPLEXARR](#), [DCOMPLEXARR](#), [DINDGEN](#), [DOUBLE](#),  
[FLTARR](#), [INTARR](#), [LONARR](#), [MAKE\\_ARRAY](#)

---

## DCINDGEN Function

Returns a double-precision floating-point complex array.

### Usage

$result = DCINDGEN(dim_1 [, dim_2, \dots, dim_n])$

### Input Parameters

$dim_i$ — The dimensions of the result. The dimensions may be any scalar expression, and up to eight dimensions may be specified.

### Returned Value

**result** — An initialized complex array with real and imaginary parts of type double precision, floating point. If the resulting array is treated as a one-dimensional array, then its initialization is given by the following:

$$array(i) = DCOMPLEX(i, 0)$$

$$\text{for } i = 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right)$$

### Keywords

None.

### Example

```
c = DCINDGEN(4)
INFO, c
C          DOUBLE COMPLEX   = Array(4)
PRINT, c
(  0.0000000,  0.0000000)
(  1.0000000,  0.0000000)
(  2.0000000,  0.0000000)
(  3.0000000,  0.0000000)
```

### See Also

[COMPLEX](#), [COMPLEXARR](#), [DCOMPLEX](#), [DCOMPLEXARR](#)

---

## DCOMPLEX Function

Converts an expression to double-precision complex data type.

Extracts data from an expression and places it in a complex scalar or array.

### Usage

$result = DCOMPLEX(real [, imaginary])$

This form is used to convert data.

$result = DCOMPLEX(expr, offset, [dim_1, dim_2, \dots, dim_n])$

This form is used to extract data.

### Input Parameters

To convert data:

*real* — Scalar or array to be used as the real part of the complex result.

*imaginary* — (optional) Scalar or array to be used as the imaginary part of the complex result. If not present, the imaginary part of the result is zero.

To extract data:

*expr* — The expression to be converted, or from which to extract data.

*offset* — The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim<sub>i</sub>* — (optional) The dimensions of the result. This parameter may be any scalar expression, and up to eight dimensions may be specified.

### Returned Value

If converting:

*result* — The *result* is a double-precision complex data type with the size and structure determined by the size and structure of *real* and *imaginary* input parameters. If either or both of the parameters are arrays, *result* will be an array, following the same rules as standard PV-WAVE operators.

If extracting:

*result* — The *result* is a complex data type with the size and structure determined by the size and structure of the *dim<sub>i</sub>* parameters. If no dimensions are specified, the result is scalar.

## Keywords

None.

## Discussion

DCOMPLEX is used primarily to convert data to complex data type. If *real* is of type string and if the string does not contain a valid floating-point value (thereby making it impossible to convert), then PV-WAVE returns 0 and displays a notice. Otherwise, *expr* is converted to complex data type. The ON\_IOERROR procedure can be used to establish a statement to jump to in the case of such errors.

If only one parameter is supplied, the imaginary part of the result is 0; otherwise, it is set by the *imaginary* parameter. Parameters are first converted to double-precision floating-point.

---

**NOTE** If three or more parameters are supplied, DCOMPLEX extracts fields of data from *expr*, rather than performing conversion.

---

## Example

```
real = DINDGEN(5)
b = DCOMPLEX(real)
INFO, b
      B                DOUBLE COMPLEX   = Array(5)
PRINT, b
      (      0.0000000,      0.0000000)
      (      1.0000000,      0.0000000)
      (      2.0000000,      0.0000000)
      (      3.0000000,      0.0000000)
      (      4.0000000,      0.0000000)
img = INTARR(5) + 6
c = DCOMPLEX(real, img)
INFO, c
      C                DOUBLE COMPLEX   = Array(5)
PRINT, c
      (      0.0000000,      6.0000000)
      (      1.0000000,      6.0000000)
      (      2.0000000,      6.0000000)
      (      3.0000000,      6.0000000)
      (      4.0000000,      6.0000000)
d = DCOMPLEX(real, 7)
```

```

INFO, d
      D                      DOUBLE COMPLEX   = Array(5)
PRINT, d
      (      0.0000000,      7.0000000)
      (      1.0000000,      7.0000000)
      (      2.0000000,      7.0000000)
      (      3.0000000,      7.0000000)
      (      4.0000000,      7.0000000)
e = DCOMPLEX(7, img)
INFO, e
      E                      DOUBLE COMPLEX   = Array(5)
PRINT, e
      (      7.0000000,      6.0000000)
      (      7.0000000,      6.0000000)
      (      7.0000000,      6.0000000)
      (      7.0000000,      6.0000000)
      (      7.0000000,      6.0000000)

```

## See Also

[BYTE](#), [COMPLEXARR](#), [DCOMPLEXARR](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LONG](#)

For more information on using this function to extract data, see the *PV-WAVE Programmer's Guide*.

## ***DCOMPLEXARR Function***

Returns a double-precision floating-point complex vector or array.

### Usage

```
result = DCOMPLEXARR(dim1 [, dim2, ... , dimn)
```

### Input Parameters

*dim*<sub>*i*</sub> — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### Returned Value

*result* — A complex double-precision floating-point vector or array.

## Keywords

*Nozero* — If *Nozero* is nonzero, the normal zeroing (see *Discussion*) is not performed, thereby causing DCOMPLEXARR to execute faster.

## Discussion

Normally, DCOMPLEXARR sets every element of the result to zero.

## Example

```
c = DCOMPLEXARR(4)
INFO, c
      C                DOUBLE COMPLEX    = Array(4)
PRINT, c
      (      0.0000000,      0.0000000)
      (      0.0000000,      0.0000000)
      (      0.0000000,      0.0000000)
      (      0.0000000,      0.0000000)
```

## See Also

[BYTARR](#), [CINDGEN](#), [DBLARR](#), [DCOMPLEX](#), [DOUBLE](#), [FLTARR](#),  
[INTARR](#), [LONARR](#)

---

## ***DC\_ERROR\_MSG Function***

Returns the text string associated with the negative status code generated by a “DC” data import/export function that does not complete successfully.

## Usage

```
msg_str = DC_ERROR_MSG(status)
```

## Input Parameters

*status* — The error message number returned by any of the "DC" functions. Must be integer.

## Returned Value

*msg\_str* — The test string that corresponds to the value of *status*. Returns a string; the string is an empty (null) string if *status* is greater than or equal to 0 (zero).

## Keywords

None.

## Discussion

When *status* has a value less than 0 (zero), it indicates a "DC" function error condition, such as an invalid filename, or an unexpected end-of-file.

Because the error message number *status* includes both the error number and an ID number that corresponds to the "DC" function that produced the error, both the function name and the specific error are described in the message string returned by DC\_ERROR\_MSG.

## Example

The following statements read a file containing 8-bit image data. Depending on the status returned by DC\_READ\_8\_BIT, either an error message is written or the image is displayed in a window the exact size of the image:

```
status = DC_READ_8_BIT('mongo.img', mongo, XSize=xs, YSize=ys)
; Use DC_READ_8_BIT to read the image file.
```

```
IF (status LT 0) THEN BEGIN
    msg_str = DC_ERROR_MSG(status)
    ; Obtain the error message if status has a negative value.
    PRINT, msg_str
    ; Print the error message.
ENDIF ELSE BEGIN
    WINDOW, XSize=xs, YSize=ys
    ; Define a window the right size to hold the image.
    TVSCL, mongo
    ; Display the image inside the window.
ENDELSE
```

## See Also

[DC\\_OPTIONS](#)

---

## ***DC\_OPTIONS* Function**

Sets the error message reporting level for all “DC” import/export functions.

### **Usage**

*status* = DC\_OPTIONS(*msg\_level*)

### **Input Parameters**

*msg\_level* — The error message reporting level. Allowed values are:

- 0 No messages. All “DC” functions operate in a silent mode.
- 1 Error messages (messages that indicate a “DC” function has failed).
- 2 Error message plus warning messages (messages that indicate the “DC” function did something, but possibly not what the user expected).
- 3 Error message plus warning messages plus informational messages. All levels of error messages are reported.

Each level of message reporting includes all error message reporting levels with a lower value, as well. For example, Level 3 includes both Level 2 and Level 1 messages.

### **Returned Value**

*status* — The value returned by DC\_OPTIONS; expected values are:

- < 0 Indicates an error, such as an invalid value for *msg\_level*.
- 0 Indicates a successful interpretation of *msg\_level*.

### **Keywords**

None.

## Discussion

By default, all messages are sent to LUN -2, the standard error stream (`stderr` for UNIX and `SYS$ERROR` for OpenVMS).

If you are using `DC_OPTIONS` with an error message reporting level of 0, messages are not automatically sent to the standard error stream, but status codes are still being generated by the various “DC” functions. These status codes can be used as input to `DC_ERROR_MSG`; this is the way to obtain the corresponding error message string, a string that you can then process or display in any way that you choose to.

## See Also

[DC\\_ERROR\\_MSG](#)

---

## ***DC\_READ\_8\_BIT Function***

Reads an 8-bit image file.

### Usage

*status* = `DC_READ_8_BIT(filename, imgarr)`

### Input Parameters

*filename* — A string containing the pathname and filename of the 8-bit image file.

### Output Parameters

*imgarr* — The byte array into which the 8-bit image data is read.

### Returned Value

*status* — The value returned by `DC_READ_8_BIT`; expected values are:

- < 0    Indicates an error, such as an invalid filename.
- 0      Indicates a successful read.

## Keywords

*XSize* — The width (size in the *x* direction) of *imgarr*. *XSize* is computed and output if *imgarr* is not explicitly dimensioned. *XSize* is returned as an integer.

*YSize* — The height (size in the *y* direction) of *imgarr*. *YSize* is computed and output if *imgarr* is not explicitly dimensioned. *YSize* is returned as an integer.

## Discussion

DC\_READ\_8\_BIT handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done reading the data.

If the dimensions of the byte array *imgarr* are not known, DC\_READ\_8\_BIT makes a “best guess” about the width and height of the image. It guesses by checking the number of bytes in the file and comparing that number to the number of bytes associated with the following common image sizes:

Image Width	Image Height
640	480
640	512
128	128
256	256
512	512
1024	1024

If no match is found, DC\_READ\_8\_BIT assumes that the image is square, and returns *XSize* and *YSize* as the square root of the number of bytes in the file.

---

**NOTE** You do not need to explicitly dimension *imgarr*, but if your image data is not one of the standard sizes shown above, you will get more predictable results if you dimension *imgarr* yourself.

---

## Example

If `still_life.img` is a 640-by-480 image file, the function call:

```
status = DC_READ_8_BIT('still_life.img', $
    s_life, XSize=xdim, YSize=ydim)
```

reads the binary data in the file `still_life.img` and transfers it to a variable named `s_life`. It also returns `xdim=640` and `ydim=480`, since these keywords were provided in the function call.

On the other hand, if `still_life.img` is a 200-by-350 image file, the values returned are `xdim=264` and `ydim=264`. The keyword results `xdim` and `ydim` are computed by taking the square root of the number of bytes in the file. This conversion is done because 200-by-350 is not a “common” image size for which `DC_READ_8_BIT` checks.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_WRITE\\_8\\_BIT](#)

For more information on input and output of image data, see the *PV-WAVE Programmer's Guide*.

---

## DC\_READ\_24\_BIT Function

Reads a 24-bit image file.

### Usage

```
status = DC_READ_24_BIT(filename, imgarr)
```

### Input Parameters

*filename* — A string containing the pathname and filename of the 24-bit image file.

### Output Parameters

*imgarr* — The byte array into which the 24-bit image data is read. Must be a 3-dimensional byte array. Either the first or last dimension of the array is 3; see the *Discussion* section for more details.

### Returned Value

*status* — The value returned by `DC_READ_24_BIT`; expected values are:

- < 0 Indicates an error, such as an invalid filename.
- 0 Indicates a successful read.

## Keywords

**Org** — Organization (in the file) of the 24-bit image data. Allowed values are:

- 0 Pixel interleaving (RGB triplets).
- 1 Image interleaving (separate planes).

If not provided, 0 (pixel interleaving) is assumed.

**XSize** — The width (size in the *x*-direction) of *imgarr*. The width is computed and returned in *XSize* if *imgarr* is not explicitly dimensioned. *XSize* is returned as an integer.

**YSize** — The height (size in the *y*-direction) of *imgarr*. The height is computed and returned in *YSize* if *imgarr* is not explicitly dimensioned. *YSize* is returned as an integer.

## Discussion

DC\_READ\_24\_BIT handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done reading the data.

When choosing the value for the *Org* keyword, be sure to select an organization that matches the file, even if it is the opposite of that used in the variable. In other words, if the data in the file is pixel interleaved, specify *Org=0*, and if the data is image interleaved, specify *Org=1*.

The way the data is read into the variable depends primarily on the dimensions that the variable was given when it was created. Consequently, an image interleaved file can be read into a pixel interleaved variable, and vice versa. So, if you want the data in the variable organized differently than it was organized in the file, pre-dimension the import variable before calling DC\_READ\_24\_BIT. Dimension the variable with a width *w* and a height *h* that matches those shown in the table later in this section.

### ***Dimensionality of the Import Variable***

If the dimensions of the byte array *imgarr* are not known, `DC_READ_24_BIT` makes a “best guess” about the width, height, and depth of the image. It guesses by checking the number of bytes in the file and comparing that number to the number of bytes associated with the following common image sizes:

<b>Image Width</b>	<b>Image Height</b>	<b>Image Depth</b>
640	480	3
640	512	3
128	128	3
256	256	3
512	512	3
1024	1024	3

If no match is found, `DC_READ_24_BIT` resorts to assuming that the image is square, and returns *XSize* and *YSize* as the square root of the number of bytes in the file, divided by 3.

**PV-WAVE** uses the following guidelines to dimension *imgarr*:

<b>Interleaving Method</b>	<b>Dimensions of Image Variable</b>
Pixel Interleaving	Dimension <i>imgarr</i> as $3 \times w \times h$ , where <i>w</i> and <i>h</i> are the width and length of the image in pixels.
Image Interleaving	Dimension <i>imgarr</i> as $w \times h \times 3$ , where <i>w</i> and <i>h</i> are the width and length of the image in pixels.

**NOTE** You do not need to explicitly dimension *imgarr*, but if your image data is not one of the standard sizes (e.g., 3-by-512-by-512 or 640-by-480-by-3), you will get more predictable results if you dimension *imgarr* yourself.

### **Example**

If the file `harpoon.img` contains a 786432 byte 24-bit image-interleaved image, the function call:

```
status = DC_READ_24_BIT('harpoon.img', $
```

```
H24_image, Org=1, XSize=xdim, YSize=ydim)
```

reads the file `harpoon.img`, creates a 512-by-512-by-3 image-interleaved byte array named `H24_image`, and returns `xdim` and `ydim` as 512.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_WRITE\\_24\\_BIT](#)

For more information about 24-bit (binary) data and for more information about image interleaving options, see the *PV-WAVE Programmer's Guide*.

---

**Windows USERS** For an example showing how to use `DC_READ_FREE` to import data from a Microsoft Excel spreadsheet, see the *PV-WAVE Programmer's Guide*.

---

---

## ***DC\_READ\_CONTAINER Function***

Reads a single variable from an HP VEE Container file.

### **Usage**

```
status = DC_READ_CONTAINER(filename, var_name)
```

### **Input Parameters**

*filename* — A string containing the path name and filename of the Container file.

### **Output Parameters**

*var\_name* — The PV-WAVE variable into which the Container file data is read. The appropriate type and dimension of *var\_name* is set based on the data found in the Container file.

### **Returned Value**

*status* — A value returned by `DC_READ_CONTAINER` indicating the success or failure of the container file read operation as follows:

< 0     Indicates an error, such as an invalid filename or incorrect file format.

0 Indicates a successful read.

## Keywords

***Double\_Complex*** — All HP-VEE complex data types (Complex and PComplex) are pairs of double-precision floating point values. However, prior to PV-WAVE 7.0, the only complex data type supported by PV-WAVE was single precision. For compatibility with versions of PV-WAVE prior to version 7.0, this function still returns Complex and PComplex as single precision complex by default. Specify the *Double\_Complex* keyword to retain the precision of the HP-VEE data types and return a PV-WAVE double-precision complex variable.

***End\_record*** — (scalar) Indicates the ending location of the desired variable within a multi-variable Container file. The keyword isn't required, if only one variable is described within the Container file. Valid values for *End\_record* can be obtained by calling DC\_SCAN\_CONTAINER.

***Extent*** — Returns the extent value for the variable described in the Container file, if it exists.

***Mapping*** — Returns the mapping value for the variable described in the Container file, if it exists.

***Start\_record*** — (scalar) Indicates the starting location of the desired variable within a multi-variable Container file. The keyword isn't required, if only one variable is described within the Container file. Valid values for *Start\_record* can be obtained by calling DC\_SCAN\_CONTAINER.

## Discussion

HP VEE is Hewlett-Packard's *Visual Engineering Environment*, a graphical programming language for creating test systems and solving engineering problems.

DC\_READ\_CONTAINER enables you to import data into PV-WAVE from HP VEE. The Container file format is a proprietary HP ASCII file format which contains a header description of the enclosed data. PV-WAVE reads this header information and creates a PV-WAVE variable of the appropriate type and dimension to hold the enclosed data.

An HP VEE Container file is created in HP VEE by using the **Write Container** transaction in the **To File** object. Please refer to your HP VEE documentation for more information.

An HP VEE Container file contain one or more variable descriptions (see DC\_SCAN\_CONTAINER for a description of how to read Container files with multiple variables).

## Example

In this example, *sine* is an undefined variable. DC\_READ\_CONTAINER resizes the variable *sine* to fit the container data.

```
status = DC_READ_CONTAINER(!Data_dir+'hpvee_sine.con', sine)
INFO
    SINE          FLOAT      = Array(256)
WzPlot, sine
    ; View the sine data.
```

## See Also

[DC\\_SCAN\\_CONTAINER](#)

---

## DC\_READ\_DIB Function (Windows)

Reads data from a Device Independent Bitmap (DIB) format file into a variable.

### Usage

```
status = DC_READ_DIB(filename, imgarr)
```

### Input Parameters

*filename* — (string) The pathname and filename of the DIB file.

### Output Parameters

*imgarr* — The variable into which the DIB image data is read. May be an array of any dimension and type; *imgarr*'s data type is changed to byte and then *imgarr* is redimensioned using information in the DIB file.

### Returned Value

*status* — The value returned by DC\_READ\_DIB; expected values are:

- < 0 Indicates an error, such as an invalid filename.
- 0 Indicates a successful read.

## Output Keywords

***Colormap*** — Used to specify a variable in which to place the colormap stored with the DIB image. *Colormap* is returned as a 2D array of long integers.

***ColorsUsed*** — Returns the number of colors used by the bitmap image (long).

***Compression*** — Returns the compression style used in the DIB image. Valid values are:

- 0 None (no compression)
- 1 Run-length encoded format for bitmaps with 8 bits per pixel
- 2 Run-length encoded format for bitmaps with 4 bits per pixel

***ImageHeight*** — Returns the DIB image height (long).

***ImageWidth*** — Returns the DIB image width (long).

***ImportantColors*** — Returns the number of colors that are important for the image to be displayed as it was saved (long).

***XResolution*** — Returns the number of pixels per meter in the *x* direction (long).

***YResolution*** — Returns the number of pixels per meter in the *y* direction (long).

## Discussion

Device Independent Bitmap (DIB) is a bitmap format that is useful for transporting graphics and color table information between different devices and applications in the Windows environment. DIB files can be produced by graphics applications such as Microsoft Image Editor, Microsoft Paintbrush, and PV-WAVE.

DC\_READ\_DIB enables you to import DIB images into variables. It handles: 1) opening the file, 2) assigning it a logical unit number (LUN), 3) closing the file when you are finished reading the data, and 4) automatically redimensioning the input variable.

---

**TIP** To read the contents of a DIB file directly into a graphics window without the intermediate step of having the data placed in a variable, use the function `WREAD_DIB`.

---

## Example

Assume that a file called `head.bmp` contains DIB data that was exported either from PV-WAVE or from another application. To read this data directly into a variable `heading`, enter:

```
status = DC_READ_DIB('head.bmp', heading, $
    Imagewidth=xsize, Imagelength=ysize, Colormap=colors)
```

The dimensions of the image array are returned in the variables `xsize` and `ysize`. The DIB image colormap is returned in the 2D array variable `colors`.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_WRITE\\_DIB](#), [WREAD\\_DIB](#), [WWRITE\\_DIB](#)

For more information on input and output of DIB and metafile images, see the *PV-WAVE Programmer's Guide*.

---

## ***DC\_READ\_FIXED Function***

Reads fixed-formatted ASCII data using a format that you specify.

### Usage

```
status = DC_READ_FIXED(filename, var_list)
```

### Input Parameters

**filename** — A string containing the pathname and filename of the file containing the data.

### Output Parameters

**var\_list** — The list of variables into which the data is read. Include as many variable names in *var\_list* as you want to be filled with data, up to a maximum of 2048. Note that variables of type structure are not supported. An exception to this

is the !DT, or date/time, structure. It is possible to transfer date/time data using this routine.

---

**NOTE** The variables in the *var\_list* do not need to be predefined unless multiple data types exist in the data file. An example of a file with multiple data types is:

```
08/04/1994 10:00:00 23.00 -94.00 11.00
```

Since the above example contains date/time and float data types, all of the variables holding this data will need to be declared before the DC\_READ\_FIXED function is called.

---

## Returned Value

*status* — The value returned by DC\_READ\_FIXED; expected values are:

- < 0 Indicates an error, such as an invalid filename or an I/O error.
- 0 Indicates a successful read.

## Keywords

**Bytes\_Per\_Rec** — A long integer that specifies how many characters comprise a single record in the input data file; use only with column-oriented files. If not provided, each line of data in the file is treated as a new record. For more details about when to use the *Bytes\_Per\_Rec* keyword, see [Example 5 on page 214](#).

**Column** — A flag that signifies *filename* is a column-organized file.

**Dt\_Template** — An array of integers indicating the date/time templates that are to be used for interpreting date/time data. Positive numbers refer to date templates; negative numbers refer to time templates. For more details, see [Example 6 on page 215](#). To see a complete list of date/time templates, see the *PV-WAVE Programmer's Guide*.

**Filters** — An array of one-character strings that PV-WAVE should check for and filter out as it reads the data. A character found on the keyboard can be typed; a special character not found on the keyboard is specified by ASCII code. For more details, see [Example 2 on page 212](#).

**Format** — A string containing the C- or FORTRAN-like format statement that will be used to read the data. The format string must contain at least one format code

that transfers data; FORTRAN formats must be enclosed in parentheses. If not provided, a C format of %1f is assumed.

**Ignore** — An array of strings; if any of these strings are encountered, PV-WAVE skips the entire record and starts reading data from the next line. Any string is allowed, but the following three strings have special meanings:

\$BLANK_LINES	Skip all blank lines; this prevents those lines from being interpreted as a series of zeroes.
\$TEXT_IN_NUMERIC	Skip any line where text is found in a numeric field.
\$BAD_DATE_TIME	Skip any line where invalid date/time data is found.

For an example showing how to use the *Ignore* keyword, see [Example 7 on page 216](#).

**Miss\_Str** — An array of strings that may be present in the data file to represent missing data. If not provided, PV-WAVE does not check for missing data as it reads the file. For an example showing how to use the *Miss\_Str* keyword, see DC\_READ\_FREE, [Example 3 on page 227](#).

**Miss\_Vals** — An array of integer or floating-point values, each of which corresponds to a string in *Miss\_Str*. As PV-WAVE reads the input data file, occurrences of strings that match those in *Miss\_Str* are replaced by the corresponding element of *Miss\_Vals*.

**Nrecs** — Number of records to read. If not provided or if set equal to zero (0), the entire file is read. For more information about records, see [Physical Records vs. Logical Records on page 208](#).

**Nskip** — Number of physical records in the file to skip before data is read. If not provided, or set equal to zero (0), no records are skipped.

**Resize** — An array of integers indicating the variables in *var\_list* that can be resized based on the number of records detected in the input data file. Values in *Resize* should be in the range:

$$1 \leq \text{Resize}_n \leq \#\_of\_vars\_in\_var\_list$$

For an example showing how to use the *Resize* keyword, see [Example 4 on page 213](#).

**Row** — A flag that signifies *filename* is a row-organized file. If neither *Row* nor *Column* is present, *Row* is the default.

## Discussion

DC\_READ\_FIXED is capable of interpreting either FORTRAN-or C-style formats, and is very adept at reading column-oriented data files. Also, DC\_READ\_FIXED handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done reading the data.

If neither the *Row* or *Column* keywords are provided, the file is assumed to be organized by rows. If both keywords are used, the *Row* keyword is assumed.

---

**NOTE** This function can be used to read data into date/time structures, but not into any other kind of structures.

---

## String Resources Used By This Function

Upon execution, the DC\_READ\_FIXED function examines two strings in a string resource file. These strings, described below, allow you to control how the function handles binary files.

The string resource file is:

**(UNIX)** <wavedir>/xres/!Lang/kernel/dc.ads

**(OpenVMS)** <wavedir>:[XRES.!Lang.KERNEL]DC.ADS

**(Windows)** <wavedir>\xres\!Lang\kernel\dc.ads

Where <wavedir> is the main PV-WAVE directory.

The strings that are examined are DC\_binary\_check and DC\_allow\_chars.

DC\_binary\_check — This string can be set to the values True or False. If set to True, the data file is checked for the presence of binary characters before the file is read. If binary characters are found, the file is not read. If this string is set to False, no binary character checking is performed. (Default: True)

For example, to turn off binary checking, set the string as follows in the dc.ads file:

```
DC_binary_check: False
```

DC\_allow\_chars — This string lets you specify additional characters to allow in the check for binary files. Before a file is read, the first several lines are checked for the presence of non-printable characters. If non-printable characters are found, the file is considered to be a binary file and the file is not read. By default, all

printable characters in the system locale are allowed. Characters may be specified either by entering them directly or numerically by three digit decimal values by preceding them with a “\” (backslash).

For example, to allow characters 165 and 220, set the string as follows in the `dc.ads` file:

```
DC_allow_chars: \165\220
```

## How the Data is Transferred into Variables

As many as 2048 variables can be included in the input argument *var\_list*. You can use the continuation character (\$) to continue the function call onto additional lines, if needed. Any undeclared variables in *var\_list* are assumed to have a data type of float (single-precision floating-point).

As data is being transferred into multi-dimensional variables, those variables are treated as collections of scalar variables, meaning the first subscript of the import variable varies the fastest. For two-dimensional import variables, this implies that the column index varies faster than the row index. In other words, data is transferred into a two-dimensional import variable one row at a time. For more details about reading column-oriented data into multi-dimensional variables, see [Example 4 on page 227](#) (in the DC\_READ\_FREE function description).

The format string is processed from left to right. Record terminators and format codes are processed until no variables are left in the variable list or until an error occurs. In a FORTRAN format string, when a slash record terminator (/) is encountered, the rest of the current input record is ignored, and the next input record is read.

Format codes that transfer data are matched with the next available variable (or element of a multi-dimensional variable) in the variable list *var\_list*. Data is read from the file and formatted according to the format code. If the data from the file does not agree with the format code, or the format code does not agree with the type of the variable, a type conversion is performed. If no type conversion is possible, an error results and a nonzero status is returned.

Once all variables in the variable list have been filled with data, DC\_READ\_FIXED stops reading data, and returns a status code of zero (0). This is true even if there are format codes in *Format* that did not get used. Even if an error occurs, and *status* is nonzero, the data that has been read successfully (prior to the error) is returned in the *var\_list* variables.

---

**TIP** If an error does occur, use the PRINT command to view the contents of the variables to see where the last successfully read value occurs. This will enable you to isolate the portion of the file in which the error occurred.

---

If the format string does not contain any format codes that transfer data, an error occurs and a nonzero status is returned. The format codes that PV-WAVE recognizes are listed in Appendix A of the *PV-WAVE Programmer's Guide*. If a format code that does not transfer data is encountered, it is processed as discussed in that appendix.

### ***Format Reversion when Reading Data***

If the last closing parenthesis of the format string is reached and there are still unfilled variables remaining, *format reversion* occurs. In format reversion, the current record is terminated, a new one is read, and format string processing reverts to the first group repeat specification that does not have an explicit repeat count. If the format does not contain a group repeat specification, format processing reverts to the initial opening parenthesis of the format string.

For more information about format reversion and group repeat specifications, see the *PV-WAVE Programmer's Guide*.

## **Physical Records vs. Logical Records**

In an ASCII text file, the end-of-line is signified by the presence of either a CTRL-J or a CTRL-M character, and a *record* extends from one end-of-line character to the next. However, there are actually two kinds of records:

- ✓ physical records
- ✓ logical records

For column-oriented files, the amount of data in a *physical record* is often sufficient to provide exactly one value for each variable in *var\_list*, and then it is a *logical record*, as well. For row-oriented files, the concept of logical records is not relevant, since data is merely read as contiguous values separated by delimiters, and the end-of-line is merely interpreted as another delimiter.

---

**NOTE** The *Nrecs* keyword counts by logical records, if they have been defined. The *Nskip* keyword, on the other hand, counts by physical records, regardless of any logical record size that has been defined.

---

## Changing the Logical Record Size

You can use the *Bytes\_Per\_Rec* keyword to explicitly define a different logical record size, if you wish. However, in most cases, you do not need to provide this keyword. For an example of when to use the *Bytes\_Per\_Rec* keyword, see [Example 5 on page 214](#).

---

**NOTE** By default, DC\_READ\_FIXED considers the physical record to be one line in the file, and the concept of a logical record is not needed. But if you *are* using logical records, the physical records in the file must all be the same length. The *Bytes\_Per\_Rec* keyword can be used only with column-oriented data files.

---

## Filtering and Substitution While Reading Data

If you want certain characters filtered out of the data as it is read, use the *Filters* keyword to specify these characters. Each character (or sequence of digits that represents the ASCII code for a character) must be enclosed with single quotes. For example, either of the following is a valid specification:

' , ' or ' 44 '

Furthermore, the two specifications shown above are equivalent to one another. For more examples of using the *Filters* keyword, see [Example 2 on page 212](#), or DC\_READ\_FREE, [Example 4 on page 227](#).

Characters that match one of the values in *Filters* are treated as if they aren't even there; in other words, these characters are not treated as data and do not contribute to the size of the logical record, if one has been defined using the *Bytes\_Per\_Rec* keyword.

---

**NOTE** If you want to supply multi-character strings instead of individual characters, you can do this with the *Ignore* keyword. However, keep in mind that a character that matches *Filters* is simply discarded, and filtering resumes from that point, while a string that matches *Ignore* causes that entire line to be skipped.

---

So if you are reading a data file that contains a value such as `#$*10.00**`, but you don't want the entire line to be skipped, filter the characters individually with *Filters* = `['#', '$', '*']`, instead of collectively with *Ignore* = `['#$*', '**']`.

## **Missing Data Substitution**

PV-WAVE expects to substitute a value from *Miss\_Vals* whenever it encounters a string from *Miss\_Str* in the data. Consequently, if the number of elements in *Miss\_Str* does not match the number of elements in *Miss\_Vals*, a nonzero status is returned and no data is read. The maximum number of values permitted in *Miss\_Str* and *Miss\_Vals* is 10.

If the end of the file is reached before all variables are filled with data, the remainder of each variable is set to *Miss\_Vals*(0) if it was specified, or 0 (zero) if *Miss\_Vals* was not specified. In this case, *status* is returned with a value less than zero to signify an unexpected end-of-file condition.

## **Reading Row-Oriented Files**

If you include the *Row* keyword, each variable in *var\_list* is completely filled before any data is transferred to the next variable.

The dimensionality of the last variable in *var\_list* can be unknown; a variable of length *n* is created, where *n* is the number of values remaining in the file. All other variables in *var\_list* must be pre-dimensioned.

If you include the *Resize* keyword with the call to DC\_READ\_FIXED, the last variable can be redimensioned to match the actual number of values that were transferred to the variable during the read operation.

If you are interested in an illustration showing what row-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

## **Reading Column-Oriented Files**

If you include the *Column* keyword, DC\_READ\_FIXED views the data files as a series of columns, with a one-to-one correspondence between columns in the file and variables in the variable list. In other words, one value from the first record of the file is transferred into each variable in *var\_list*, then another value from the next record of the file is transferred into each variable in *var\_list*, and so forth, until all the data in the file has been read, or until the variables are completely filled with data.

If a variable in *var\_list* is undefined, a floating-point variable of length *n* is created, where *n* is the number of records read from the file. To get a similar effect in an existing variable, include the *Resize* keyword with the function call.

All variables specified with the *Resize* keyword are redimensioned to the same length — the length of the longest column of data in the file. The variables that

correspond to the shortest columns in the file will have one or more values added to the end; either *Miss\_Vals*(0) if it was specified, or 0 (zero) if *Miss\_Vals* was not specified.

If you are interested in an illustration demonstrating what column-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

### ***Multi-dimensional Variables***

The following table shows how column-oriented data in a file is read into multi-dimensional variables:

<b>Dimensions of Variable</b>	<b>How Data is Read From the File (If Variable is Pre-dimensioned)</b>
One-dimensional ( $1 \times n$ )	One value read from each record of file (repeated $n$ times)
Two-dimensional ( $m$ columns by $n$ rows)	$m$ values read from each record of file (repeated $n$ times)
Three-dimensional ( $m \times n \times p$ )	$m$ values read from each record of file (repeated $n$ times) (entire process repeated $p$ times)
$q$ -dimensional ( $m \times n \times p \times q$ )	$m$ values read from each record of file (repeated $n$ times) (above process repeated $p$ times) (entire process repeated $q$ times)

You can combine one- and two-dimensional variables in *var\_list*, as long as the second dimension of the two-dimensional variable matches the dimension of the one-dimensional variable. For example, with two variables, `var1(50)` and `var2(2, 50)`, one column of data will be transferred to `var1` and two columns of data will be transferred to `var2`.

**NOTE** If you want to intermingle multi-dimensional variables in *var\_list*, you must be sure that the product of all dimensions (excluding the first dimension) of each variable is equal. For example, you can combine two-, three-, and four-dimensional variables in *var\_list* if the variables have dimensions like these:

```
Var1      2-by-30
Var2      2-by-15-by-2
```

Var3        2-by-10-by-3  
Var4        2-by-3-by-2-by-5

---

## Example 1

The function call:

```
status=DC_READ_FIXED('results.wp', /Column, $
    unit1, unit2, unit3, run_total, Ignore= $
    ["Total", "-----", "$TEXT_IN_NUMERIC", $
    "$BLANK_LINES"], Format="(F7.2,5X)")
```

reads the data from file `results.wp` and places the data into four variables: `unit1`, `unit2`, `unit3`, and `run_total`.

Because the variables were not predefined, all data is interpreted as single-precision floating-point data, and all variables are treated as resizable one-dimensional arrays. Any blank lines or strings specified with the *Ignore* keyword (in this example, “Total” and “-----”) are ignored. Also, any line with non-numeric characters in a numeric field is ignored.

## Example 2

The function call:

```
status = DC_READ_FIXED('yields.doc', intake, $
    chute, conveyor, crusher, /Column, $
    Filter=['/', ':', ','], $
    Format="(F7.2, 8X, F6.4, 3X)", $
    Ignore=["$BLANK_LINES"])
```

reads data from the file `yields.doc` and places the data into four variables: `intake`, `chute`, `conveyor`, and `crusher`.

Because the variables were not predefined, all data is interpreted as single-precision floating-point data, and all variables are treated as resizable one-dimensional arrays. Any extraneous characters (in this example, “/”, “:”, and “;”) are discarded because the *Filter* keyword is provided. Also, all totally blank lines in the file are ignored.

## Example 3

The data file shown below is a fixed-formatted ASCII file named `simple.dat`. The ‘.’ characters in `simple.dat` represent blank spaces:

```
...1...2...3...4...5
...6...7...8...9..10
..11..12..13..14..15
..16..17..18..19..20
```

The function call:

```
status = DC_READ_FIXED('simple.dat', var1, $
    Format='(I4)', /Column)
```

results in `var1=[1.0, 6.0, 11.0, 16.0]`. Because `var1` was not predefined, `DC_READ_FIXED` creates it as a one-dimensional floating-point array.

On the other hand, the commands:

```
Var1 = INTARR(2)
Var2 = INTARR(2)
status = DC_READ_FIXED('simple.dat', var1, $
    var2, Format='(2(4X, I4))', Nskip=2)
```

skip the first two records in the file and result in `var1=[12, 14]` and `var2=[17, 19]`. Because neither the *Row* or *Column* keyword was supplied, the file is assumed to use row organization.

## Example 4

The data file shown below is a fixed-formatted ASCII file; this file is named `nimrod.dat`. The ‘.’ characters in `nimrod.dat` represent blank spaces. `nimrod.dat` is very much like the data file in [Example 3 on page 212](#), except that it has a missing value where you would expect to see the numeral “8”:

```
...1...2...3...4...5
...6...7.....9..10
..11..12..13..14..15
..16..17..18..19..20
```

When reading this file as column-oriented data, the results vary, depending on whether a C or FORTRAN format string is being used, and whether the *Resize* keyword has been included in the function call to `DC_READ_FIXED`.

For example, the commands:

```
A = INTARR(20) & B = INTARR(20)
C = INTARR(20) & D = INTARR(20)
```

```
E = INTARR(20)
status = DC_READ_FIXED('nimrod.dat', $
    A, B, C, D, E, Format='(2X, I2)', $
    Resize=[1, 2, 3, 4, 5], /Column)

result in A=[1, 6, 11, 16], B=[2, 7, 12, 17], C=[3, 0, 13, 18],
D=[4, 9, 14, 19], and E=[5, 10, 15, 20]. The missing value is
interpreted as a zero (0). All variables are resized to a length of 4.
```

On the other hand, the commands:

```
A = INTARR(20) & B = INTARR(20)
C = INTARR(20) & D = INTARR(20)
E = INTARR(20)
status = DC_READ_FIXED('nimrod.dat', $
    A, B, C, D, E, Format='%d', $
    Resize=[1, 2, 3, 4, 5], /Column)

result in A=[1, 6, 11, 16], B=[2, 7, 12, 17],
C = [3, 9, 13, 18], D = [4, 10, 14, 19], and
E = [5, 15, 20]. The missing value is skipped altogether, and E is resized to
a length of 3 to reflect the number of values that were transferred into the variable.
The other variables are resized to 4.
```

Any variable that is not resizable (because it was omitted from the *Resize* vector), will be padded to the end with extra values. For the latter of the two calls to `DC_READ_FIXED` shown in this example, A, B, C, and D would be padded with an additional 16 zeroes, while E would be padded with an additional 17 zeroes. (Zeroes are used for the padding because *Miss\_Vals* was not specified.)

If the file `nimrod.dat` had used some other character as a delimiter, such as commas or slashes, both the C and FORTRAN format strings would have yielded the same result, namely, `C = [3, 0, 13, 18]`. It is only because of the way a C format skips over blank space that the C format was unable to detect the presence of a missing value.

## Example 5

The data file shown below contains 18 pairs of XY data that could be used to create a scatter plot:

```
5.992E+04,7.121E-01,8.348E+04,7.562E-01,5.672E+04,9.451E-01,
5.459E+04,8.659E-01,7.088E+04,8.659E-01,8.541E+04,3.437E-01,
4.981E+04,4.679E-01,8.438E+04,5.019E-01,6.902E+04,7.340E-01,
6.239E+04,8.023E-01,7.865E+04,6.643E-01,5.870E+04,9.992E-01,
7.439E+04,9.456E-01,4.672E+04,9.801E-01,6.872E+04,4.325E-01,
```

6.362E+04,5.894E-01,8.992E+04,7.509E-01,2.785E+04,4.796E-01,

For data organized like this, you use the *Bytes\_Per\_Rec* keyword to specify the exact length of the record. In this example, all X values are single-precision floating-point numbers with an exponent of E+04, and all Y values are single-precision floating-point numbers with an exponent of E-01. Therefore, each XY pair uses 18 ASCII characters (bytes) apiece. Thus, you would specify 20 bytes per record (9 times 2, plus 2 more bytes for the comma delimiters separating values).

```
status=DC_READ_FIXED(/Column, "xy5.dat", Xa, $  
Ya, Format="(E9.3, 1X)", Bytes_Per_Rec=20)
```

If you omit the *Bytes\_Per\_Rec* keyword, but still read the file as a column-oriented file, only the first pair of data values on each line would actually be transferred into the variables Xa and Ya. Nor can the file be read as row-oriented data, because Xa would be filled completely before any data was transferred to Ya.

---

**TIP** Only include the *Bytes\_Per\_Rec* keyword when you have a logical record that is longer or shorter than one line in the file. For the majority of column-oriented data files, one and only one value from each variable is on a single line, and the *Bytes\_Per\_Rec* keyword is completely unnecessary.

---

## Example 6

Assume that you have a file, *chrono.dat*, that contains some data values and also some chronological information about when those data values were recorded:

```
01/01/92 10:30:35 10.00 04-30-92 32767  
02/01/92 23:22:15 15.89 06-15-91 99999  
05/15/91 03:03:03 14.22 12-25-92 87654
```

The date/time templates that will be used to transfer this data have the following definitions:

Number	Template Description
1	MM*DD*YY (* = any delimiter)
-1	HH*MM*SS (* = any delimiter)

To read the date and time from the first two columns into one date/time variable and read the third column of floating point data into another variable, use the following commands:

```
date1 = REPLICATE({!DT}, 3)
```

```

date2 = REPLICATE({!DT}, 3)
; The system structure definition of date/time is !DT. Date/time
; variables must be defined as !DT structure arrays before being
; used if the date/time data is to be read as such.

status = DC_READ_FIXED("chrono.dat", date1, date1, decibels,
Dt_Template=[1, -1], $
Format="(2(A8, 1X), F5.2)", /Column)
; The variable date1 is listed twice; this way, both the date data
; and the time data can be stored in the same variable, date1.

```

To read all columns, change the call to DC\_READ\_FIXED and define a new variable:

```

calib = INTARR(3)
status = DC_READ_FIXED("chrono.dat", date1, $
date1, decibels, date2, calib, /Column, $
Format="%8s %8s %f %8s %d", Ignore= $
["$BAD_DATE_TIME"], Dt_Template=[1, -1])

```

Notice how the date/time templates are reused. For each new record, Template 1 is used first to read the *date* data into `date1`. Next, Template -1 is used to read the *time* data into `date1`. Finally, since there is another date/time variable to be read (`date2`) and there are no more templates left, the template list is reset and Template 1 is used again. The template list is reset for each record.

---

**NOTE** Because of the internal conversion that DC\_READ\_FIXED performs to convert the date strings to PV-WAVE's date/time internal structure, the date and time data *must* be read with the A8 (FORTRAN) or %8s (C) format string.

---

Normally an error would be reported if the input text to be read as date/time is invalid and cannot be converted. But because the `Ignore=["$BAD_DATE_TIME"]` keyword was provided, any record containing this type of error is ignored and no error is reported.

## Example 7

The data file shown below is a fixed-formatted ASCII file named `wages.wp`. All floating-point data in the file has been decimal-point-aligned by a word-processing application:

```

1070.00   9007.97   1100.00   1250.00   850.50   2010.00
5000.00   3050.00   1044.12   3500.00   6031.00   905.00

```

415.00	5200.00	1300.10	350.00	745.00	3000.00
200.00	3100.00	8100.00	7050.00	6780.00	2310.25
950.00	1050.00	1350.00	410.00	797.00	200.36
2600.00	2000.00	1500.00	2000.00	1000.00	400.00
1000.00	9000.00	1100.00	2091.00	3440.10	2000.37
5000.00	3000.00	1000.01	3500.00	6000.00	900.12

The following commands:

```

Maria = Fltarr(12) & Naomi = Fltarr(12)
Klaus = Fltarr(12) & Carlos = Fltarr(12)
status = DC_READ_FIXED('wages.wp', Maria, $
    Carlos, Klaus, Naomi, Format="(F7.2,5X)", $
    Ignore=["$BLANK_LINES"])

```

read the data from file `wages.wp` and places the data into four variables: `Maria`, `Carlos`, `Klaus`, and `Naomi`. By default, row organization is assumed in the file, with five spaces separating the values in the file.

With row organization, each variable is “filled up” before any data is transferred to the next variable in the variable list. This means that the first two lines of the file are transferred into the variable `Maria`, the new two lines of the file are transferred into the variable `Carlos`, the next two lines of the file are transferred into the variable `Klaus`, and the last two lines of the file are transferred into the variable `Naomi`. The blank lines in the file are skipped entirely, preventing those lines from being interpreted as a series of zeroes.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_READ\\_FREE](#), [DC\\_WRITE\\_FIXED](#)

For more information about fixed format I/O in `PV-WAVE`, see the *PV-WAVE Programmer's Guide*.

---

## ***DC\_READ\_FREE Function***

Reads freely-formatted ASCII files.

### **Usage**

*status* = DC\_READ\_FREE(*filename*, *var\_list*)

### **Input Parameters**

*filename* — A string containing the pathname and filename of the file containing the data.

### **Output Parameters**

*var\_list* — The list of variables into which the data is read. Include as many variables names in *var\_list* as you want to be filled with data, up to a maximum of 2048. Note that variables of type structure are not supported. An exception to this is the !DT, or date/time, structure. It is possible to transfer date/time data using this routine.

---

**NOTE** The variables in the *var\_list* do not need to be predefined unless multiple data types exist in the data file. An example of a file with multiple data types is:

```
08/04/1994 10:00:00 23.00 -94.00 11.00
```

Since the above example contains date/time and float data types, all of the variables holding this data will need to be declared before the DC\_READ\_FIXED function is called.

---

### **Returned Value**

*status* — The value returned by DC\_READ\_FREE; expected values are:

- < 0    Indicates an error, such as an invalid filename or an I/O error.
- 0      Indicates a successful read.

## Keywords

**Column** — A flag that signifies *filename* is a column-organized file.

**Delim** — An array of single-character strings that are the field separators used in the data file. If not provided, a comma- or space- delimited file is assumed.

**Dt\_Template** — An array of integers indicating the date/time templates that are to be used for interpreting date/time data. Positive numbers refer to date templates; negative numbers refer to time templates. For more details, see [Example 5 on page 229](#). To see a complete list of date/time templates, see the *PV-WAVE Programmer's Guide*.

**Filters** — An array of one-character strings that PV-WAVE should check for and filter out as it reads the data. A character found on the keyboard can be typed; a special character not found on the keyboard is specified by ASCII code. For more details about the *Filters* keyword, see [Filtering and Substitution While Reading Data on page 223](#).

**Get\_Columns** — An array of integers indicating column numbers to read in the file. If not provided or if set equal to zero (0), all columns are read. Ignored if the *Row* keyword is supplied.

**Ignore** — An array of strings; if any of these strings are encountered, PV-WAVE skips the entire line and starts reading data from the next line. Any string is allowed, but the following three strings have special meanings:

\$BLANK_LINES	Skip all blank lines; this prevents those lines from being interpreted as a series of zeroes.
\$TEXT_IN_NUMERIC	Skip any line where text is found in a numeric field.
\$BAD_DATE_TIME	Skip any line where invalid date/time data is found.

For an example showing how to use the *Ignore* keyword, see [Example 2 on page 226](#).

**Miss\_Str** — A string array that specifies strings that may be present in the data file to represent missing data. If not present, PV-WAVE does not check for missing data as it reads the file.

**Miss\_Vals** — An array of floating-point values, each of which corresponds to a string in *Miss\_Str*. As PV-WAVE reads the input data file, occurrences of strings

that match those in *Miss\_Str* are replaced by the corresponding element of *Miss\_Vals*.

**Nrecs** — Number of records to read. If not provided or if set equal to zero (0), the entire file is read. For more information about records, see [Physical Records vs. Logical Records on page 222](#).

**Nskip** — Number of physical records in the file to skip before data is read. If not provided or if set equal to zero (0), no records are skipped.

**Resize** — An array of integers indicating the variables in *var\_list* that can be resized based on the number of records detected in the input data file. Values in *Resize* should be in the range:

$$1 \leq \text{Resize}_n \leq \#\_of\_vars\_in\_var\_list$$

For an example showing how to use the *Resize* keyword, see DC\_READ\_FIXED, [Example 4 on page 213](#), or DC\_READ\_FREE, [Example 4 on page 227](#).

**Row** — A flag that signifies *filename* is a row-organized file. If neither *Row* nor *Column* is present, *Row* is the default.

**Vals\_Per\_Rec** — A long integer that specifies how many values comprise a single record in the input data file; use only with column-oriented files. If not provided, each line of data in the file is treated as a new record. For more details about when to use the *Vals\_Per\_Rec* keyword, see [Example 4 on page 227](#).

## Discussion

DC\_READ\_FREE is very adept at reading column-oriented data files. Also, DC\_READ\_FREE handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done reading the data.

DC\_READ\_FREE relieves you of the task of composing a format string that describes the organization of the data in the input file. All you do is tell DC\_READ\_FREE which delimiters to expect in the file; comma and space are the default delimiters expected. In other words, DC\_READ\_FREE easily reads data values separated by any combination of commas and spaces, or any other delimiters that you explicitly define using the *Delim* keyword.

If neither the *Row* or *Column* keywords are provided, the file is assumed to be organized by rows. If both keywords are used, the *Row* keyword is assumed.

---

**NOTE** This function can be used to read data into date/time structures, but not into any other kind of structures.

---

## String Resources Used By This Function

Upon execution, the DC\_READ\_FREE function examines two strings in a string resource file. These strings, described below, allow you to control how the function handles binary files.

The string resource file is:

**(UNIX)** <wavedir>/xres/!Lang/kernel/dc.ads

**(OpenVMS)** <wavedir>:[XRES.!Lang.KERNEL]DC.ADS

**(Windows)** <wavedir>\xres\!Lang\kernel\dc.ads

Where <wavedir> is the main PV-WAVE directory.

The strings that are examined are DC\_binary\_check and DC\_allow\_chars.

**DC\_binary\_check** — This string can be set to the values True or False. If set to True, the data file is checked for the presence of binary characters before the file is read. If binary characters are found, the file is not read. If this string is set to False, no binary character checking is performed. (Default: True)

For example, to turn off binary checking, set the string as follows in the dc.ads file:

```
DC_binary_check: False
```

**DC\_allow\_chars** — This string lets you specify additional characters to allow in the check for binary files. Before a file is read, the first several lines are checked for the presence of non-printable characters. If non-printable characters are found, the file is considered to be a binary file and the file is not read. By default, all printable characters in the system locale are allowed. Characters may be specified either by entering them directly or numerically by three digit decimal values by preceding them with a “\” (backslash).

For example, to allow characters 165 and 220, set the string as follows in the dc.ads file:

```
DC_allow_chars: \165\220
```

## How the Data is Transferred into Variables

As many as 2048 variables can be included in the input argument *var\_list*. You can use the continuation character (\$) to continue the function call onto additional lines, if needed. Any undeclared variables in *var\_list* are assumed to have a data type of float (single-precision floating-point).

As data is being transferred into multi-dimensional variables, those variables are treated as collections of scalar variables, meaning the first subscript of the import variable varies the fastest. For two-dimensional import variables, this implies that the column index varies faster than the row index. In other words, data is transferred into a two-dimensional import variable one row at a time. For more details about reading column-oriented data into multi-dimensional variables, see [Example 4 on page 227](#).

If the current input line is empty or DC\_READ\_FREE has reached the end of the line and there are still unused variables in *var\_list*, the next line is read. When there are no unused variables left in *var\_list*, the remainder of the line is ignored.

When reading into numeric variables, PV-WAVE attempts to convert the input into a value of the expected type. Decimal points are optional and scientific notation is allowed. If a real value is provided for an integer variable, the value is truncated at the decimal point.

---

**NOTE** If the file contains string data, make sure the strings do not contain delimiter characters. Otherwise, the string will be interpreted as more than one string, and the data in the file will not match the variable list.

---

Once all variables in the variable list have been filled with data, DC\_READ\_FREE stops reading data, and returns a status code of zero (0). Even if an error occurs, and *status* is nonzero, the data that has been read successfully (prior to the error) is returned in the *var\_list* variables.

---

**TIP** If an error does occur, use the PRINT command to view the contents of the variables to see where the last successfully read value occurs. This will enable you to isolate the portion of the file in which the error occurred.

---

## Physical Records vs. Logical Records

In an ASCII text file, the end-of-line is signified by the presence of either a CTRL-J or a CTRL-M character, and a *record* extends from one end-of-line character to the next. However, there are actually two kinds of records:

- ✓ physical records
- ✓ logical records

For column-oriented files, the amount of data in a *physical record* is often sufficient to provide exactly one value for each variable in *var\_list*, and then it is a *logical record*, as well. For row-oriented files, the concept of logical records is not relevant, since data is merely read as contiguous values separated by delimiters, and the end-of-line is merely interpreted as another delimiter.

---

**NOTE** The *Nrecs* keyword counts by logical records, if they have been defined. The *Nskip* keyword, on the other hand, counts by physical records, regardless of any logical record size that has been defined.

---

### ***Changing the Logical Record Size***

You can use the *Vals\_Per\_Rec* keyword to explicitly define a different logical record size, if you wish. However, in most cases, you do not need to provide this keyword. For an example of when to use the *Vals\_Per\_Rec* keyword, see [Example 4 on page 227](#).

---

**NOTE** By default, DC\_READ\_FREE considers the physical record to be one line in the file, and the concept of a logical record is not needed. But if you *are* using logical records, the physical records in the file must all contain the same number of values. The *Vals\_Per\_Rec* keyword can be used only with column-oriented data files.

---

### **Filtering and Substitution While Reading Data**

If you want certain characters filtered out of the data as it is read, use the *Filters* keyword to specify these characters. Each character (or sequence of digits that represents the ASCII code for a character) must be enclosed with single quotes. For example, either of the following is a valid specification:

' , ' or ' 44 '

Furthermore, the two specifications shown above are equivalent to one another. For another example of using the *Filters* keyword, see [Example 4 on page 227](#).

---

**TIP** Be sure not to filter characters that were used in the file as delimiters. The delimiters enable `DC_READ_FREE` to discern where one data value ends and another one begins.

---

Characters that match one of the values in *Filters* are treated as if they aren't even there; in other words, these characters are not treated as data and do not contribute to the size of the logical record, if one has been defined using the *Vals\_Per\_Rec* keyword.

---

**NOTE** If you want to supply multi-character strings instead of individual characters, you can do this with the *Ignore* keyword. However, keep in mind that a character that matches *Filters* is simply discarded, and filtering resumes from that point, while a string that matches *Ignore* causes that entire line to be skipped.

---

So if you are reading a data file that contains a value such as `#$*10.00**`, but you don't want the entire line to be skipped, filter the characters individually with *Filters* = ['#', '\$', '\*'] instead of collectively with *Ignore* = ['#\$', '\*\*'].

### **Missing Data Substitution**

PV-WAVE expects to substitute a value from *Miss\_Vals* whenever it encounters a string from *Miss\_Str* in the data. Consequently, if the number of elements in *Miss\_Str* does not match the number of elements in *Miss\_Vals*, a nonzero status is returned and no data is read. The maximum number of values permitted in *Miss\_Str* and *Miss\_Vals* is 10.

If the end of the file is reached before all variables are filled with data, the remainder of each variable is set to *Miss\_Vals*(0) if it was specified, or 0 (zero) if *Miss\_Vals* was not specified. In this case, *status* is returned with a value less than zero to signify an unexpected end-of-file condition.

### **Delimiters in the Input File**

Values in the file can be separated by commas, spaces, and any other delimiter characters specified with the *Delim* keyword. If you use any other delimiter, the delimiter character is treated as data and type conversion is attempted. If type conversion is not possible, *status* is set to less than zero to signify an error condition.

---

**NOTE** Use a different delimiter to separate data values in the file than you use to separate the different fields of dates and times, such as months, days, hours, and

minutes. Otherwise, your date/time data may not be interpreted correctly. The only delimiters that can be used inside date/time data are: slash (/), colon (:), hyphen (-), and comma (,).

---

## Reading Row-Oriented Files

If you include the *Row* keyword, each variable in *var\_list* is completely filled before any data is transferred to the next variable.

When reading row-oriented data, only the dimensionality of the last variable in *var\_list* can be unknown; a variable of length *n* is created, where *n* is the number of values remaining in the file. All other variables in *var\_list* must be pre-dimensioned.

If you include the *Resize* keyword with the call to the function `DC_READ_FREE`, the last variable can be redimensioned to match the actual number of values that were transferred to the variable during the read operation.

If you are interested in an illustration showing what row-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

## Reading Column-Oriented Files

If you include the *Column* keyword, `DC_READ_FREE` views the data files as a series of columns, with a one-to-one correspondence between columns in the file and variables in the variable list. In other words, one value from the first record of the file is transferred into each variable in *var\_list*, then another value from the next record of the file is transferred into each variable in *var\_list*, and so forth, until all the data in the file has been read, or until the variables are completely filled with data.

If a variable in *var\_list* is undefined, a floating-point variable of length *n* is created, where *n* is the number of records read from the file. To get a similar effect in an existing variable, include the *Resize* keyword with the function call.

All variables specified with the *Resize* keyword are redimensioned to the same length — the length of the longest column of data in the file. The variables that correspond to the shortest columns in the file will have one or more values added to the end; either *Miss\_Vals*(0) if it was specified, or 0 (zero) if *Miss\_Vals* was not specified.

If you are interested in an illustration demonstrating what column-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

For more information about how column-oriented data in a file is read into multi-dimensional variables, see [Multi-dimensional Variables on page 211](#).

## Example 1

The data file shown below is a freely-formatted ASCII file named `monotonic.dat`:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

The function call:

```
status = DC_READ_FREE('monotonic.dat', var1, $
    /Column, Get_Columns=[3])
```

results in `var1=[3.0, 8.0, 13.0, 18.0]`. Because `var1` was not predefined, `DC_READ_FREE` creates it as a resizable one-dimensional floating-point array.

On the other hand, the commands:

```
var1 = INTARR(2)
var2 = INTARR(2)
status = DC_READ_FREE('monotonic.dat', var1, $
    var2, /Column, Get_Columns=[2, 4], Nskip=2)
```

result in `var1=[12, 17]` and `var2=[14, 19]`.

## Example 2

The data file shown below is a freely-formatted ASCII file named `measure.dat`:

```
0 5 10 15 20 25 30 35 40 45 50 56 61 66 71
76 81 86 91

96 101 107 112 117 122 127 132 137 142 147 152 158 163 168 173
178 183 188

193 198 203 209 214 219 224 229 234 239 244 249 255 255 255 255
255 255 255

255 255 255 255 255
```

The commands:

```
var1 = INTARR(5)
var2 = INTARR(5)
```

```
status = DC_READ_FREE('measure.dat', $
    var1, var2, Ignore=["$BLANK_LINES"])
```

result in `var1 = [0, 5, 10, 15, 20]` and `var2 = [25, 30, 35, 40, 45]`. Note that the file was interpreted as row-oriented data, since neither the *Row* or *Column* keyword was specified. All totally blank lines are ignored

---

**NOTE** If the *Resize = [2]* keyword had been provided, `var2` would have been resizable and would have ended up having many more elements. Specifically, `var2` would have ended up with 57 elements instead of just 5.

---

### Example 3

The data file shown below is a freely-formatted ASCII file named `intake.dat`:

```
151-182-BADY-214-515
316-197-BADX-199-206
```

The commands:

```
valve = INTARR(30)
status = DC_READ_FREE('intake.dat', $
    valve, Miss_Str=["BADX","BADY"], $
    Miss_Vals=[9999, -9999], Resize=[1], $
    Delim=['-'])
```

results in `valve = [151, 182, -9999, 214, 515, 316, 197, 9999, 199, 206]`. The hyphens in the data are filtered out. Because `valve` is resizable, it ends up with 10 elements instead of 30. The two values from *Miss\_Vals* are substituted for the two strings in the file, "BADX" and "BADY".

### Example 4

The data file shown below is a freely-formatted ASCII file named `level.dat`. This data file uses the semi-colon (;) and the slash (/) as delimiters, and the comma (,) to separate the thousands digit from the hundreds digit. This file has three logical records on every line; at the end of each logical record is a slash:

```
5,992;17,121/8,348;17,562/5,672;19,451/
5,459;18,659/7,088;17,052/8,541;13,437/
6,362;15,894/8,992;17,509/7,785;14,796/
```

The commands:

```
gap = INTARR(20)
bar = INTARR(20)
status = DC_READ_FREE('level.dat', gap, bar, $
```

```

/Column, Delim=[';', '/'], Filter=[','], $
Resize=[1, 2], Vals_Per_Rec=2)

```

result in:

```

gap = [5992, 8348, 5672, 5459, 7088, 8541,
6362, 8992, 7785] and bar = [17121, 17562,
19451, 18659, 17052, 13437, 15894, 17509,
14796].

```

The commas have been filtered out of the data because of the value of the string that was provided with the *Filter* keyword.

Suppose you wanted *gap* and *bar* to be dimensioned as 3-by-3 arrays instead of 1-by-9 vectors. The best way to do this is by reading the data with the commands shown above, and then using the REFORM command to redimension the variables:

```

gaparr = REFORM(gap, 3, 3)
bararr = REFORM(bar, 3, 3)

```

By approaching the data transfer in this way, DC\_READ\_FREE does not expect to transfer two columns of data into the same multi-dimensional variable.

For example, the following commands demonstrate the problem:

```

gap = INTARR(3, 3)
bar = INTARR(3, 3)
status = DC_READ_FREE('level.dat', gap, bar, $
/Column, Delim=[';', '/'], Filter=[','], $
Resize=[1, 2], Vals_Per_Rec=2)

```

results in:

```

gap =
    5992 17121 0
    8348 17562 0
    5672 19451 0
    5459 18659 0
    7088 17052 0
    8541 13437 0
    6362 15894 0
    8992 17509 0
    7785 14796 0

```

and

$$bar = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

The data is transferred into `gap` using the rule, “The first subscript varies fastest.” With `Vals_Per_Rec` set to “2”, no value is available for the third column—hence, every element in the third column is set equal to “0” (zero). Furthermore, notice that `gap` gets all the data (it is resizable) and `bar` gets none of the data.

## Example 5

Assume that you have a file, `events.dat`, that contains some data values and also some chronological information about when those data values were recorded:

```
01/01/92 5:45:12 10 01-01-92 3276
02/01/92 10:10:10 15.89 06-15-91 99
05/15/91 2:02:02 14.2 12-25-92 876
```

The date/time templates that will be used to transfer this data have the following definitions:

Number	Template Description
1	MM*DD*YY (* = any delimiter)
-1	HH*MM*SS (* = any delimiter)

To read the date and time from the first two columns into one date/time variable and read the third column of floating point data into another variable, use the following commands:

```
date1 = REPLICATE({!DT},3)
; The system structure definition of date/time is !DT. Date/time
; variables must be defined as !DT structure arrays before being
; used if the date/time data is to be read as such.

status = DC_READ_FREE("events.dat", date1, $
    date1, float1, /Column, $
    Dt_Template=[1,-1], Delim=[' '])
; The variable date1 is listed twice; this way, both the date data
; and the time data can be stored in the same variable, date1.
```

To see the values of the two variables, you can use the `PRINT` command:

```

FOR I = 0,2 DO BEGIN
    PRINT, date1(I), float1(I)
        ; Print one row at a time.
ENDFOR

```

Executing these statements results in the following output:

```

{ 1992    01    01    05    45    12.00    87402.240    0}
  10.0000    { 1992    02    01    10    10    10.00    87433.424
    0} 15.8900    { 1992    05    15    02    02    02.00
    87537.035    0} 14.2000

```

Because `date1` is a structure, curly braces, “{” and “}”, are placed around the output. When displaying the value of `date1` and `float1`, **PV-WAVE** uses default formats for formatting the values, and attempts to place as many items as possible onto each line.

---

**TIP** Another alternative to view the contents of `date1` and `float1` is to use the `DT_PRINT` command instead of `PRINT`.

---

For more information about the internal organization of the !DT system structure, see the *PV-WAVE Programmer's Guide*.

To read the first, second, fourth, and fifth columns, define an integer array and another date/time variable, and change the call to `DC_READ_FREE` as shown below:

```

calib = INTARR(3)
date2 = REPLICATE({!DT},3)
status = DC_READ_FREE("events.dat", date1, $
    date1, date2, calib, /Column, Delim=[' '], $
    Get_Columns= [1, 2, 4, 5], Dt_Template = $
    [1, -1], Ignore=["$BAD_DATE_TIME"])

```

Notice how the date/time templates are reused. For each new record, Template 1 is used first to read the *date* data into `date1`. Next, Template -1 is used to read the *time* data into `date1`. Finally, since there is another date/time variable to be read (`date2`) and there are no more templates left, the template list is reset and Template 1 is used again. The template list is reset for each record.

---

**NOTE** Because of the internal conversion that `DC_READ_FIXED` performs to convert the date strings to **PV-WAVE**'s date/time internal structure, the date and time data *must* be read with the `A8` (FORTRAN) or `%8s` (C) format string.

---

Normally an error would be reported if the input text to be read as date/time is invalid and cannot be converted. But because the `Ignore=["$BAD_DATE_TIME"]`

keyword was provided, any record containing this type of error is ignored and no error is reported.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_READ\\_FIXED](#), [DC\\_WRITE\\_FREE](#)

See the *PV-WAVE Programmer's Guide* for more information about free format I/O in PV-WAVE.

---

## DC\_READ\_TIFF Function

Reads a Tag Image File Format (TIFF) file.

---

**NOTE** This function was retired with version 6.1, because the new `IMAGE_READ` function provides the same capability. Although `DC_READ_TIFF` is still available for backward compatibility, we strongly recommend that you use `IMAGE_READ` instead.

---

### Usage

*status* = `DC_READ_TIFF(filename, imgarr)`

### Input Parameters

*filename* — A string containing the pathname and filename of the TIFF file.

### Output Parameters

*imgarr* — The variable into which the TIFF image data is read. May be an array of any dimension and type; *imgarr*'s data type is changed to byte and then *imgarr* is re-dimensioned using information in the TIFF file. Variables of type structure are not supported.

### Returned Value

*status* — Value returned by `DC_READ_TIFF`; expected values are:

- < 0    Indicates an error, such as an invalid filename or image number.
- 0      Indicates a successful read.

## Keywords

***BitsPerSample*** — The number of bits that comprise each sample in the TIFF image is returned; a pixel consists of one or more “samples”. *BitsPerSample* is returned as an integer; typical values are 2, 4, and 8.

***Colormap*** — The TIFF image colormap. If present, the colormap associated with the TIFF image is returned. *Colormap* is returned as a 2-dimensional array of long integers.

***Compression*** — The compression style used in the TIFF image. *Compression* is returned as an integer; expected values are:

1	None (no compression)
2	CCITT Group 3
5	LZW
32733	PackBits

***Imagelength*** — The TIFF image length. If present, the TIFF image length is returned. *Imagelength* is returned as a long integer value.

***Imagewidth*** — The TIFF image width. If present, the TIFF image width is returned. *Imagewidth* is returned as a long integer value.

***Imgnum*** — The number of the image to read from the file. If not provided, the first image (image number 0) is read.

***Order*** — If nonzero, *Order* reverses the y-axis direction of the original image. In other words, if the original image is stored from top to bottom, the returned image is stored from bottom to top.

***PhotometricInterpretation*** — The class of the TIFF image. If present, retrieves photometric information from the TIFF image header. *PhotometricInterpretation* is returned as an integer; expected values are:

0	Bilevel/Grayscale
2	Full RGB color
3	Palette color
4	Transparency mask†

† Transparency mask indicates the image is used to define an irregularly shaped region of another image in the same TIFF file. *PhotometricInterpretation=4* is not supported by PV-WAVE.

The first four classes of TIFF images are explained in more detail in the *PV-WAVE Programmer's Guide*.

***PlanarConfig*** — The arrangement of the RGB information. If present, retrieves RGB configuration information from the TIFF image header. *PlanarConfig* is returned as an integer; expected values are:

- 1 RGB triplets (pixel interleaving)
- 2 Separate planes (image interleaving)

The methods for interleaving image data are explained more fully in the *PV-WAVE Programmer's Guide*.

***ResolutionUnit*** — The type of resolution units specified in the TIFF image header. If present, retrieves unit information from the TIFF image. *ResolutionUnit* is returned as an integer; expected values are:

- 1 None (no absolute units)
- 2 Inches
- 3 Centimeters

***SamplesPerPixel*** — The number of samples associated with each pixel in the TIFF image is returned. *SamplesPerSample* is returned as an integer; expected values are:

- 1 Bilevel, Grayscale, Palette color
- 3 RGB images

***XResolution*** — The number of pixels per *ResolutionUnit* in the X direction. If present, retrieves information about the number of X pixels from the TIFF image header. *XResolution* is returned as a floating-point value.

***YResolution*** — The number of pixels per *ResolutionUnit* in the Y direction. If present, retrieves information about the number of Y pixels from the TIFF image header. *YResolution* is returned as a floating-point value.

For more information about the output keywords described in this section, see the Technical Memorandum, *Tag Image File Format Specification, Revision 5.0 (FINAL)*, published jointly by Aldus™ Corporation and Microsoft® Corporation.

## Discussion

DC\_READ\_TIFF enables you to import TIFF images into PV-WAVE. It also handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done reading the data.

DC\_READ\_TIFF sets the dimension and type (byte array) of *imgarr* automatically, depending on the width and height of the image. For 24-bit images, the interleaving method (see description of *PlanarConfig* keyword) is considered, as well. PV-WAVE uses the following guidelines to dimension *imgarr*:

Interleaving Method	Dimensions of Image Variable
Pixel (RGB triplets)	Dimension <i>imgarr</i> as $3 \times w \times h$ , where <i>w</i> and <i>h</i> are the width and length of the image in pixels.
Image (separate planes)	Dimension <i>imgarr</i> as $w \times h \times 3$ , where <i>w</i> and <i>h</i> are the width and length of the image in pixels.

The difference between pixel-interleaved and image-interleaved image data is discussed in the *PV-WAVE Programmer's Guide*.

---

**NOTE** Compressed TIFF images are uncompressed before being transferred to the named variable.

---

### Example 1

The function call:

```
status = DC_READ_TIFF('oxford.tif', oximage)
```

reads the file `oxford.tif` and returns the TIFF image data contained in the first image of that file. The data is transferred to the variable `oximage`.

### Example 2

The function call:

```
status = DC_READ_TIFF('shamu.tif', shamu, $  
    Imagewidth=xsiz, Imagelength=ysiz, $  
    PlanarConfig=planar, Photometric=photo)
```

reads a complete description of the first TIFF image in the file `shamu.tif`. The width and length of the image are returned in `xsiz` and `ysiz`, respectively.

*PlanarConfig* and *PhotometricInterpretation* are returned in `planar` and `photo`, respectively.

The fact that the *PlanarConfig* keyword is being returned with the function call suggests that the image in `shamu.tif` is a full-color RGB (24-bit) image. The *PlanarConfig* keyword is used to return the image interleaving method for 24-bit images.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_WRITE\\_TIFF](#), [IMAGE\\_READ](#)

See the *PV-WAVE Programmer's Guide* for more information about TIFF image I/O.

---

## ***DC\_SCAN\_CONTAINER Function***

Determines the number and location of defined variables in an HP VEE Container by scanning the file.

### **Usage**

*status* = DC\_SCAN\_CONTAINER(*filename*, *num\_variables*, *start\_records*, *end\_records*)

### **Input Parameters**

*filename* — A string containing the path name and filename of the Container file.

### **Output Parameters**

*num\_variables* — The number of variables described within the specified Container file.

*start\_records* — A long array containing the starting position of each variable within the Container file. This information can be used as input to the DC\_READ\_CONTAINER function to extract a given variable.

*end\_records* — A long array containing the ending position of each variable within the Container file. This information can be used as input to DC\_READ\_CONTAINER to extract a given variable.

## Returned Value

*status* — The value returned by DC\_READ\_CONTAINER indicating the validity of the returned output parameter information.

- < 0 Indicates an error, such as an invalid filename or incorrect file format.
- 0 Indicates a successful read.

## Keywords

None.

## Discussion

HP VEE is Hewlett-Packard's *Visual Engineering Environment*, a graphical programming language for creating test systems and solving engineering problems.

The Container file format is a proprietary HP ASCII file format which contains a header description of the enclosed data. PV-WAVE reads this header information and creates a PV-WAVE variable of the appropriate type and dimension to hold the enclosed data.

DC\_SCAN\_CONTAINER enables you to determine the number of variables described within an HP VEE Container file. The returned parameter arrays *start\_records* and *end\_records* can be used as inputs to the DC\_READ\_CONTAINER function to extract individual variables.

An HP VEE Container file is created in HP VEE by using the **Write Container** transaction in the **To File** object. Please refer to your HP VEE documentation for more details.

## Example

This example shows how to scan a Container file for multiple containers. Then, a loop is created that reads each container found in the file.

```
status = DC_SCAN_CONTAINER(!Data_dir+'hpvee_multi.con', $
    num_vars, start_recs, end_recs)
; Scan the container file for multiple containers. Next, loop for every
; container in the file. At this point, you could check num_vars.
; The DC_READ_CONTAINER function can only read 1 container at a time.

FOR I=0, num_vars-1 DO BEGIN
; Make a new name for each variable. There are other ways to do this as well.
var = 'var'+STRCOMPRESS (STRING(I), /Remove_All)
```

```

rc= EXECUTE("status = $
DC_READ_CONTAINER(!Data_dir+'hpvee_multi.con', " + $
var+ ",Start_Record=start_recs(I), End_Record=end_recs(I))" )
ENDFOR
; At the WAVE> prompt, use INFO to show the new variables.

INFO
VAR0          FLOAT      = 18.0000
VAR1          COMPLEX    = Array(4)
VAR2          FLOAT      = Array(10)

```

## See Also

[DC\\_READ\\_CONTAINER](#)

## ***DC\_WRITE\_8\_BIT* Function**

Writes 8-bit image data to a file.

### Usage

*status* = DC\_WRITE\_8\_BIT(*filename*, *imgarr*)

### Input Parameters

*filename* — A string containing the pathname and filename of the file where the 8-bit image data is to be stored.

*imgarr* — The 2D byte array variable from which the 8-bit image data is transferred.

### Returned Value

*status* — The value returned by DC\_WRITE\_8\_BIT; expected values are:

- < 0    Indicates an error, such as an invalid filename.
- 0     Indicates a successful write.

### Keywords

None.

## Discussion

DC\_WRITE\_8\_BIT handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done writing the data.

---

**NOTE** Only one 8-bit image can be stored at a time when using the DC\_WRITE\_8\_BIT function.

---

If *imgarr* is not a 2D byte array, DC\_WRITE\_8\_BIT returns an error status and no data is written to the output file.

## Example

If *fft\_flow* is a 600-by-800 byte array containing image data, the function call:

```
status = DC_WRITE_8_BIT('fft_flow1.img', fft_flow)
```

creates the file *fft\_flow1.img* and uses it to store the image data contained in the variable *fft\_flow*. The file that is created contains raw binary data, and is easily read with DC\_READ\_8\_BIT.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_READ\\_8\\_BIT](#), [DC\\_WRITE\\_24\\_BIT](#)

See the *PV-WAVE Programmer's Guide* for more information about 8-bit (binary) data.

---

## DC\_WRITE\_24\_BIT Function

Writes 24-bit image data to a file.

### Usage

```
status = DC_WRITE_24_BIT(filename, imgarr)
```

### Input Parameters

*filename* — A string containing the pathname and filename of the file where the 24-bit image data is to be stored.

*imgarr* — The 3D byte array from which the 24-bit image data is transferred. Either the first or last dimension of *imgarr* must be 3; see the *Discussion* section for more details.

## Returned Value

*status* — The value returned by DC\_WRITE\_24\_BIT; expected values are:

- < 0 Indicates an error, such as an invalid filename.
- 0 Indicates a successful write.

## Keywords

*Org* — Organization of the 24-bit image data. Allowed values are:

- 0 Pixel interleaving (RGB triplets).
- 1 Image interleaving (separate planes).

If not provided, 0 (pixel interleaving) is assumed.

## Discussion

DC\_WRITE\_24\_BIT handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done writing the data.

---

**NOTE** Only one 24-bit image can be stored at a time when using the DC\_WRITE\_24\_BIT function.

---

If *imgarr* is not a 3D byte array, DC\_WRITE\_24\_BIT returns an error status and no data is written to the output file. Either the first or last dimension of *imgarr* must be equal to 3, as shown in the following table:

Interleaving Method	Dimensions of Image Variable
Pixel (RGB triplets)	Dimension <i>imgarr</i> as $3 \times w \times h$ , where <i>w</i> and <i>h</i> are the width and length of the image in pixels.
Image (separate planes)	Dimension <i>imgarr</i> as $w \times h \times 3$ , where <i>w</i> and <i>h</i> are the width and length of the image in pixels.

The difference between pixel-interleaved and image-interleaved data is discussed in the *PV-WAVE Programmer's Guide*.

## Example

If `hi_glow` is a 400-by-400-by-3 byte array containing 24-bit image data, the function call:

```
status = DC_WRITE_24_BIT('hi_glow.img', hi_glow, Org=1)
```

creates the file `hi_glow.img` and uses it to store the image data contained in the variable `hi_glow`, using image interleaving. The file that is created contains raw binary data, and is easily read with the function `DC_READ_24_BIT`.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_READ\\_24\\_BIT](#), [DC\\_WRITE\\_8\\_BIT](#)

See the *PV-WAVE Programmer's Guide* for or more information about 24-bit (binary) data.

---

**Windows USERS** For an example showing how to use `DC_WRITE_FREE` to export data from PV-WAVE into a Microsoft® Excel spreadsheet, see the *PV-WAVE Programmer's Guide*.

---

---

## ***DC\_WRITE\_DIB Function (Windows)***

Writes image data from a variable to a Device Independent Bitmap (DIB) format file.

### Usage

```
status = DC_WRITE_DIB(filename, imgarr)
```

### Input Parameters

*filename* — A string containing the pathname and filename of the DIB file.

*imgarr* — The variable containing the image data to be saved as a DIB file.

## Returned Value

*status* — Value returned by DC\_WRITE\_DIB; expected values are:

- < 0 Indicates an error, such as an invalid filename.
- 0 Indicates a successful write.

## Keywords

**ColorClass** — An integer specifying the DIB color class level. If not provided, ColorClass=2 is assumed. Valid values are 2, 16, and 256.

**ColorsUsed** — The number of colors that the bitmap image uses. If this keyword is not provided, it is set to the number of elements in *SystemPalette* or *Palette* if one of them is provided; otherwise, *ColorsUsed* defaults to 2.

**Compression** — A string that specifies the kind of image data compression to use as the data is written to the file. This keyword is valid only if *ColorClass* is specified as 16 or 256. If this keyword is not provided, no compression is performed. Valid values are:

'None' — None (no compression)

'RLE4' — Run-length encoded format for bitmaps with 4 bits per pixel

'RLE8' — Run-length encoded format for bitmaps with 8 bits per pixel

**ImportantColors** — The number of important colors that the bitmap image needs to display. If not provided, *ImportantColors* defaults to 0.

**Palette** — Specifies a color table to be saved with the DIB file. *Palette* must be a 3-by-*n* array of integers, where *n* is either 1, 16, or 256. *n* specifies the number of colors associated with the bitmap. If the *ColorClass* keyword is set to 2, then the *Palette* keyword is ignored and a monochrome color table is saved.

**SystemPalette** — If set to a nonzero value, this keyword causes the system color table to be saved with the DIB file. *SystemPalette* always takes precedence over the *Palette* keyword.

## Discussion

Device Independent Bitmap (DIB) is a bitmap format that is useful for transporting graphics and color table information between different devices and applications in the Windows environment. DIB files can be produced by graphics applications such as Microsoft Image Editor, Microsoft Paintbrush, and PV-WAVE.

You must specify a color table to be saved with the DIB file, or an error is returned. The *Palette* and *SystemPalette* keywords let you specify a color table.

DC\_WRITE\_DIB exports DIB images from PV-WAVE. It handles: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done writing the data.

If *imgarr* is not a 2- or 3-dimensional byte array, the function DC\_WRITE\_DIB returns an error status and no data is written to the output file.

## Example

If the variable *maverick* is a 512-by-512 byte array, the function call:

```
status = DC_WRITE_DIB('mav.bmp', maverick, $
    ColorClass = 256, Compression = None', $
    /SystemPalette)
```

creates the file *mav.bmp* and uses it to store the system color table and the image data contained in the variable *maverick*. The created DIB file is not compressed and has a color class of 256.

## See Also

[DC\\_READ\\_DIB](#), [WREAD\\_DIB](#), [WWRITE\\_DIB](#)

For more information, see the *PV-WAVE Programmer's Guide*.

---

## DC\_WRITE\_FIXED Function

Writes the contents of one or more variables (in ASCII fixed format) to a file using a format that you specify.

### Usage

```
status = DC_WRITE_FIXED(filename, var_list)
```

### Input Parameters

*filename* — A string containing the pathname and filename of the file where the data will be stored.

*var\_list* — The list of variables containing the values to be written. Note that variables of type structure are not supported. An exception to this is the !DT, or date/time, structure. It is possible to transfer date/time data using this routine.

## Returned Value

*status* — The value returned by DC\_WRITE\_FIXED; expected values are:

- < 0 Indicates an error, such as an invalid filename or an I/O error.
- 0 Indicates a successful write.

## Keywords

*Column* — A flag that signifies *filename* is to be written as a column-organized file.

*Dt\_Template* — An array of integers indicating the data/time templates that are to be used for interpreting date/time data. Positive numbers refer to date templates; negative numbers refer to time templates. For more details, see DC\_WRITE\_FREE, [Example 4 on page 254](#). To see a complete list of date/time templates, see the *PV-WAVE Programmer's Guide*.

*Format* — A string containing the C- or FORTRAN-like format statement that will be used to write the data. The format string must contain at least one format code that transfers data; FORTRAN formats must be enclosed in parentheses. If not provided, C format(s) that match the data type(s) of the variables in *var\_list* are assumed; for example %lf for float, %i for integer, and %s for string.

*Miss\_Str* — An array of strings that specifies strings that are substituted in the output file (to represent missing data) for each value in *Miss\_Vals*. If not provided, no strings are substituted for missing data.

*Miss\_Vals* — An array of integer or floating-point values, each of which corresponds to a string in *Miss\_Str*. As PV-WAVE writes the data, it checks for values that match *Miss\_Vals*; whenever it encounters one, it substitutes the corresponding value from *Miss\_Str*.

*Row* — A flag that signifies *filename* is to be written as a row-organized file. If neither *Row* nor *Column* is present, *Row* is the default.

## Discussion

DC\_WRITE\_FIXED is capable of interpreting either FORTRAN-or C-style formats, and is very adept at storing data in a column-oriented manner. Also, DC\_WRITE\_FIXED handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done writing the data.

If neither the *Row* or *Column* keywords are provided, the data is stored in rows. If both keywords are used, the *Row* keyword is assumed.

---

**NOTE** This function can be used to write data from date/time structures, but not from any other kind of structures.

---

## How the Data is Written to the File

As many as 2048 variables can be included in the output argument *var\_list*. You can use the continuation character (\$) to continue the function call onto additional lines, if needed. The entire contents of each variable in *var\_list* is written to the specified file. If an error occurs, a nonzero status is returned.

---

**NOTE** Any variable you include in *var\_list* must have been previously created; otherwise, an error occurs.

---

As data is being transferred from multi-dimensional variables, those variables are treated as collections of scalar variables, meaning the first subscript of the export variable varies the fastest. For two-dimensional export variables, this implies that the column index varies faster than the row index. In other words, data transfer is *row major*; it occurs one row at a time. For more details about storing multi-dimensional variables in a column-oriented manner, see [Writing Column-Oriented Data on page 245](#).

The format string is processed from left to right. Record terminators and format codes are processed until no variables are left in *var\_list* or until an error occurs. In a FORTRAN format string, when a slash record terminator ( / ) is encountered, a new output record is started.

Format codes that transfer data are matched with the next available variable (or element of a multi-dimensional variable) in the variable list *var\_list*. Data is written to the file and formatted according to the format code. If the data in the variable does not agree with the format code, or the format code does not agree with the type of the variable, a type conversion is performed. If no type conversion is possible, an error results and a nonzero *status* is returned.

Once all variables in the variable list have been stored in the file, DC\_WRITE\_FIXED stops writing data, and returns a status code of zero (0). This is true even if there are format codes in *Format* that did not get used. Even if an error occurs, and *status* is nonzero, the data that was written successfully (prior to the error) is left intact in the file.

---

**TIP** If an error does occur, view the contents of the file (using an operating system command) to see how much data was transferred. This will enable you to isolate the portion of the variable list in which the error occurred.

---

If the format string does not contain any format codes that transfer data, an error occurs and a nonzero status is returned. The format codes that PV-WAVE recognizes are listed in the *PV-WAVE Programmer's Guide*. If a format code that does not transfer data is encountered, it is processed as discussed in that appendix.

### **Format Reversion when Writing Data**

If the last closing parenthesis of the format string is reached and there are still variables in *var\_list* whose contents have not been written to the file, *format reversion* occurs. In format reversion, the current output record is terminated, a new one is started, and format string processing reverts to the first group repeat specification that does not have an explicit repeat count. If the format does not contain a group repeat specification, format processing reverts to the initial opening parenthesis of the format string.

For more information about format reversion and group repeat specifications, see the *PV-WAVE Programmer's Guide*.

### **Missing Data String Substitution while Writing Data**

PV-WAVE expects to substitute a string from *Miss\_Str* whenever it encounters a value from *Miss\_Vals* in the data. Consequently, if the number of elements in *Miss\_Str* does not match the number of elements in *Miss\_Vals*, a nonzero status is returned and no data is written to the file. The maximum number of values permitted in *Miss\_Str* and *Miss\_Vals* is 10.

### **Writing Row-Oriented Data**

If the *Row* keyword has been provided, each variable in *var\_list* is written to the file in its entirety before any data is transferred from the next variable.

If you are interested in an illustration showing what row-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

### **Writing Column-Oriented Data**

The following table shows how variables of any dimensions are stored in a columnar format:

Dimensions of Variable	Organization of Saved File
One-dimensional ( $1 \times n$ )	One value from each variable written to each record (repeated $n$ times)
Two-dimensional ( $m$ columns by $n$ rows)	$m$ values from each variable written to each record (repeated $n$ times)
Three-dimensional ( $m \times n \times p$ )	$m$ values from each variable written to each of $n$ records (entire process repeated $p$ times)
$q$ -dimensional ( $m \times n \times p \times q$ )	$m$ values from each variable written to each of $n$ records (above process repeated $p$ times) (entire process repeated $q$ times)

If you are interested in an illustration demonstrating what column-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

## Example 1

If variable `sara` is a floating-point array with 10 elements all equal to 1.0, `tana` is a floating-point array with 5 elements all equal to 2.0, and `cora` is a floating-point array with 8 elements all equal to 3.0, the function call:

```
status = DC_WRITE_FIXED('outfile.dat', sara, $
    tana, cora, Format="(5(1X, F7.4))")
```

creates `outfile.dat` containing the following values:

```
..1.0000..1.0000..1.0000..1.0000..1.0000
..1.0000..1.0000..1.0000..1.0000..1.0000
..2.0000..2.0000..2.0000..2.0000..2.0000
..3.0000..3.0000..3.0000..3.0000..3.0000
..3.0000..3.0000..3.0000
```

The periods are used to represent blank spaces in the file.

## Example 2

If variable `bogus` is a 2-by-4 integer array with values 1 through 4 in the first column and values 5 through 8 in the second column, the following function call:

```
status=DC_WRITE_FIXED('intfile.dat',/Column, bogus, Format='(I5)')
```

replicates that structure in the created file `intfile.dat`, as shown below:

```
....1....5
....2....6
....3....7
....4....8
```

The periods are used to represent blank spaces in the file.

On the other hand, the following function call:

```
status = DC_WRITE_FIXED('intfile.dat', $
    bogus(1,*), Format='(4I5)')
```

with a slightly different format string, results in four values all being written in the same record, using a row orientation:

```
....5....6....7....8
```

Because of the array subscripting notation used in the function call, only the second column of data values is written to the file. Without the “4” inside the parentheses of the format string, each value would have been written on a separate line in the file.

### Example 3

If variable `foo` is a floating-point array with 6 elements all equal to 1.0, `hoo` is a floating-point array with 6 elements all equal to 2.0, `doe` is a floating-point array with 6 elements all equal to 3.0, and `boo` is a floating-point array with 6 elements all equal to 4.0, the function call:

```
status = DC_WRITE_FIXED('omni.dat', foo, $
    hoo, doe, boo, Format="%f, %f, %f, %f", $
    /Column)
```

creates an output file `omni.dat` that is organized as shown below:

```
1.0000, 2.0000, 3.0000, 4.0000
1.0000, 2.0000, 3.0000, 4.0000
1.0000, 2.0000, 3.0000, 4.0000
1.0000, 2.0000, 3.0000, 4.0000
1.0000, 2.0000, 3.0000, 4.0000
1.0000, 2.0000, 3.0000, 4.0000
```

The data is arranged in columns. The C format code `"%f, "` causes a comma followed by a space to be inserted after every value written to the file.

---

**TIP** An even easier way to write this data is to use another “DC” function, `DC_WRITE_FREE`. The `DC_WRITE_FREE` function writes CSV (Comma Sep-

arated Values) data by default, or you can use the *Delim* keyword to specify some other delimiter besides the comma.

---

## Example 4

Assume that you have two variables, `float` and `date`, that contain some data values and also some chronological information about when those data values were recorded:

```
date = STR_TO_DT(['10/10/92', '11/11/92', '12/12/92'])
float = [1.2, 3.4, 5.6]
```

The `STR_TO_DT` function creates a date/time variable `date`. For more information on the internal structure of date/time structure variables, see the *PV-WAVE Programmer's Guide*.

In this example, date/time Template 1 (MM/DD/YY) is used to transfer this data, which means the month, day, and year will be written as adjacent values separated by a slash (/).

If you enter the following command:

```
status = DC_WRITE_FIXED("thymus.dat", $
    date, float, /Column, Dt_Template=[1], $
    Format="(A10, 1X, F4.2)")
```

you create a file that looks like this:

```
10/10/1992 1.20
11/11/1992 3.40
12/12/1992 5.60
```

Notice that date data is written into the file `thymus.dat` as a series of strings. In each new output record, Template 1 is used to write the data from `date`, using the A10 character format, and a value from `float` is written, using the F4.2 format.

---

**NOTE** If you have date and time data stored in the same variable, the variable must be listed twice in the variable list in order to extract both the date and time data. For more details, see `DC_READ_FREE`, [Example 4 on page 254](#).

---

## Example 5

Suppose you have a number of variables that contain data about recent phone activity. The names of these variables are `date`, `time`, `mins`, `type`, `ext`, `cost`,

and `num_called`. The following command writes this information to a file and organizes the values by columns:

```
status = DC_WRITE_FIXED('phonedata.dat', $
    date, time, mins, type, ext, cost, $
    num_called, /Column, $
    Format="%s %s %5.2f %i %i %5.2f %s")
```

In this example, `date` and `time` are variables with a data type of string. Because they are not defined as a date/time structure, such as the variable `date` that was part of the previous example, `date` and `time` are not stored using any of the date or time templates. Thus, there is no need to include the *Dt\_Template* keyword as part of the function call.

The result is a file `phonedata.dat` that is organized as shown below:

```
901002 093200 21.40 1 311 5.78 2158597430
901002 093600 51.56 1 379 13.92 2149583711
901002 093700 61.39 2 435 16.58 9137485920
```

The following function call could be used instead of the one shown above if you prefer to use a FORTRAN-style format string:

```
status = DC_WRITE_FIXED('phonedata.dat', $
    date, time, mins, type, ext, cost, $
    num_called, /Column, Format="(A6,1X,A6,"+ $
    "2X,F5.2,4X,I2,4X,I3,2X,F5.2,1X,A12)")
```

---

**NOTE** If you wish to enter a format string similar to the FORTRAN one shown above, try to get the entire format string on the same line. Otherwise, use the string concatenation operator (+), as shown in the above example, to split the format string into two shorter strings.

---

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_READ\\_FIXED](#), [DC\\_WRITE\\_FREE](#)

See the *PV-WAVE Programmer's Guide* for more information about fixed format I/O in PV-WAVE.

---

## ***DC\_WRITE\_FREE* Function**

Writes the contents of one or more variables to a file in ASCII free format.

### **Usage**

*status* = DC\_WRITE\_FREE(*filename*, *var\_list*)

### **Input Parameters**

*filename* — A string containing the pathname and filename of the file where the data will be stored.

*var\_list* — The list of variables containing the values to be written. Note that variables of type structure are not supported. An exception to this is the !DT, or date/time, structure. It is possible to transfer date/time data using this routine.

### **Returned Value**

*status* — The value returned by DC\_WRITE\_FREE; expected values are:

- < 0    Indicates an error, such as an invalid filename or an I/O error.
- 0      Indicates a successful write.

### **Keywords**

*Column* — A flag that signifies *filename* is to be written as a column-organized file.

*Delim* — A single-character string that will be placed between values in the output data file. If you provide an array of strings, only the first string in the array will be used. If not provided, commas are used as delimiters in the file.

*Dt\_Template* — An array of integers indicating the date/time templates that are to be used for interpreting date/time data. Positive numbers refer to date templates; negative numbers refer to time templates. For more details, see [Example 4 on page 254](#). To see a complete list of date/time templates, see the *PV-WAVE Programmer's Guide*.

*Miss\_Str* — An array of strings that are substituted in the output file (to represent missing data) for each value in *Miss\_Vals*. If not provided, no strings are substituted for missing data.

*Miss\_Vals* — An array of integer or floating-point values, each of which corresponds to a string in *Miss\_Str*. As PV-WAVE writes the data, it checks for

values that match *Miss\_Vals*; whenever it encounters one, it substitutes the corresponding value from *Miss\_Str*.

**Row** — A flag that signifies *filename* is to be written as a row-organized file. If neither *Row* nor *Column* is present, *Row* is the default.

## Discussion

DC\_WRITE\_FREE is very adept at storing data in a column-oriented manner. Also, DC\_WRITE\_FREE handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done writing the data.

DC\_WRITE\_FREE relieves you of the task of composing a format string to describe the organization of the data in the output file. By default, DC\_WRITE\_FREE generates CSV (Comma Separated Values) files. However, you can override this default by using the *Delim* keyword to provide a different delimiter, if you wish.

If neither the *Row* or *Column* keywords are provided, the data is stored in rows. If both keywords are used, the *Row* keyword is assumed.

---

**NOTE** This function can be used to write data from date/time structures, but not from any other kind of structures.

---

### ***How the Data is Written to the File***

As many as 2048 variables can be included in the output argument *var\_list*. You can use the continuation character (\$) to continue the function call onto additional lines, if needed. The entire contents of each variable in *var\_list* is written to the specified file. If an error occurs, a nonzero status is returned.

---

**NOTE** Any variable you include in *var\_list* must have been previously created; otherwise, an error occurs.

---

The values in the output file are separated with the character specified with the *Delim* keyword. If no *Delim* keyword is provided, a comma delimiter is used by default.

As data is being transferred from multi-dimensional variables, those variables are treated as collections of scalar variables, meaning the first subscript of the export variable varies the fastest. For two-dimensional export variables, this implies that

the column index varies faster than the row index. In other words, data transfer is *row major*; it occurs one row at a time. For more details about storing multi-dimensional variables in a column-oriented manner, see [Writing Column-Oriented Data on page 245](#).

Once all variables in the variable list have been stored in the file, DC\_WRITE\_FREE stops writing data, and returns a status code of zero (0). Even if an error occurs, and *status* is nonzero, the data that has been written successfully (prior to the error) is left intact in the file.

---

**TIP** If an error does occur, view the contents of the file (using an operating system command) to see how much data was transferred. This will enable you to isolate the portion of the variable list in which the error occurred.

---

### ***Formatting in the Output File***

When writing row-organized files, output lines are formatted to be no more than 80 characters. When writing column-organized files, the output line length depends on the number, type, and dimensions of the variables in *var\_list*.

The various data types are stored using the default formats shown in the following table:

<b>Data Type</b>	<b>Output Formats used by DC_WRITE_FREE</b>
Byte	I4
Integer	I8
Long Integer	I12
Float	G13.6
Double	G16.8
Complex	(', G13.6, ',', G13.6, ')
Double Complex	(', G16.8, ',', G16.8, ')
String	A (character data)

---

**NOTE** When writing data of type string, each string is written to the file, flanked with a delimiter on each side. This implies that the strings should not contain delimiter characters if you intend to read the file later with the DC\_READ\_FREE function.

---

## Writing Row-Oriented Data

If the *Row* keyword has been provided, each variable in *var\_list* is written to the file in its entirety before any data is transferred from the next variable.

If you are interested in an illustration showing what row-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

## Writing Column-Oriented Files

*PV-WAVE Programmer's Guide* For more information about how data from multi-dimensional export variables is stored in a columns in the output file, see [Writing Column-Oriented Data on page 245](#).

If you are interested in an illustration demonstrating what column-oriented data can look like inside a file, see the *PV-WAVE Programmer's Guide*.

## Example 1

If variable *sara* is a floating-point array with 10 elements all equal to 1.0, *tana* is a floating-point array with 5 elements all equal to 2.0, and *cora* is a floating-point array with 8 elements all equal to 3.0, the function call:

```
status = DC_WRITE_FREE('outfile.dat', sara, tana, cora, /Row)
```

creates *outfile.dat* containing the following values:

```
1.0000, 1.0000, 1.0000, 1.0000, 1.0000,  
1.0000, 1.0000, 1.0000, 1.0000, 1.0000,  
2.0000, 2.0000, 2.0000, 2.0000, 2.0000,  
3.0000, 3.0000, 3.0000, 3.0000, 3.0000,  
3.0000, 3.0000, 3.0000,
```

A comma is used by default to separate the values in the output file.

## Example 2

If variable *bogus* is a 4-by-4 integer array with values 1 through 4 in the first column, values 5 through 8 in the second column, values 9 through 12 in the third column, and values 13 through 16 in the fourth column, the following function call:

```
status = DC_WRITE_FREE('infile.dat', bogus, Delim=[','], /Column)
```

creates a file *infile.dat*, as shown below:

```
1,      5,      9,      13  
2,      6,     10,     14  
3,      7,     11,     15  
4,      8,     12,     16
```

Notice that the organization of values in the output file mimics that of the variable, `bogus`.

On the other hand, the function call:

```
status = DC_WRITE_FREE('intfile.dat', $
    bogus(1,*), /Row, Delim='*')
```

results in the following organization in `intfile.dat`, as shown below:

```
5*      6*      7*      8
```

Because of the array subscripting notation used in the function call, only the second column of data values is written to the file.

### Example 3

Suppose you have three variables that contain data taken from an electronic sensor. The names of these variables are `date`, `time`, and `phase_shift`. `date` and `time` are long integer vectors, and `phase_shift` is a vector of complex (floating-point) values. The function call:

```
status = DC_WRITE_FREE('day539.dat', date, $
    time, phase_shift, Delim='/', /Column)
```

results in a file `day539.dat` that is organized as shown below:

```
921002/  091200/(  -0.139528,    0.983407)
921002/  091205/(  -0.149962,    0.407378)
921002/  091210/(   1.002340,   -0.039187)
921002/  091215/(   1.130523,    0.983482)
921002/  091220/(  -0.947966,    0.171492)
921002/  091225/(   1.275390,    0.789446)
```

The complex numbers are stored as two floating-point values, separated with a comma and enclosed in parentheses.

### Example 4

Assume that you have two variables, `float` and `date`, that contain some data values and also some chronological information about when those data values were recorded:

```
date = [10/10/92 05:45:12,
        11/11/92 10:10:51,
        12/12/92 23:03:19]
float = [1.2, 3.4, 5.6]
```

---

**NOTE** The variable `date` is shown above as a series of strings, even though it is actually stored in a date/time structure as a series of integer and floating-point values.

---

The variable `date` is a date/time structure, and holds both date and time data. For more information on the internal structure of date/time structure variables, see the *PV-WAVE Programmer's Guide*.

When you have date and time data stored in the same variable, the variable must be listed twice in the variable list in order to extract both the date and time data. The date/time templates that will be used to transfer this data have the following definitions:

<b>Number</b>	<b>Template Description</b>
1	MM/DD/YY (/ = delimiter)
-1	HH:MM:SS (: = delimiter)

If you enter the following command:

```
status = DC_WRITE_FREE("thymus.dat", date, $
    float, date, /Column, Dt_Template=[1, -1])
```

you create a file that looks like this:

```
10/10/92 1.2 5:45:12
11/11/92 3.4 10:10:51
12/12/92 5.6 23:03:19
```

Notice that data is written from `date` two different times. In each new output record, Template 1 is used first to write the *date* data from `date`. Next, a value from `float` is written, and finally, Template -1 is used to write the *time* data from `date`.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_READ\\_FREE](#), [DC\\_WRITE\\_FIXED](#)

See the *PV-WAVE Programmer's Guide* for more information about free format I/O in PV-WAVE.

---

## ***DC\_WRITE\_TIFF Function***

Writes image data to a file using the Tag Image File Format (TIFF) format.

---

**NOTE** This function was retired with version 6.1, since the new `IMAGE_WRITE` function provides the same capability. Although `DC_READ_TIFF` is still available for backward compatibility, we strongly recommend that you use `IMAGE_WRITE` instead.

---

### **Usage**

*status* = `DC_WRITE_TIFF(filename, imgarr)`

### **Input Parameters**

*filename* — A string containing the pathname and filename of the TIFF file.

*imgarr* — The 2 or 3-D byte array from which the image data is transferred. Note that variables of type structure are not supported.

### **Returned Value**

*status* — The value returned by `DC_WRITE_TIFF`; expected values are:

- < 0    Indicates an error, such as an invalid filename or image number.
- 0      Indicates a successful write.

### **Keywords**

*Class* — TIFF class conformance level; supplied as a string. If not provided, *Class*='Bilevel' is assumed. Valid values are:

'Bilevel'  
'Grayscale'  
'Palette Color'  
'RGB Full Color'

The strings can be abbreviated to one letter, if you wish.

The four classes of TIFF image conformance are explained in more detail in the *PV-WAVE Programmer's Guide*.

**Compress** — A string that signifies the kind of image data compression to use as the data is written to the file (if TIFF class 'B' (Bilevel) is specified). If not provided, no compression is performed. Valid values at this time are:

'None'

'PackBits'

**Negative** — If set, the dithering function is reversed so that all pixels in the image that are less than or equal to the threshold are set to 255. All other pixels are set to 0. This keyword is only valid when the *Class* keyword is set to `Bilevel`. The default threshold is 128.

**Order** — If nonzero, returns the image mirrored in the y-direction. (Default: Do not mirror the image.)

**Palette** — The color table to store with the image data if TIFF class P (Palette Color) is specified. *Palette* must be a 3-by-256 array of integers.

**Threshold** — An integer specifying the threshold value for dithering a grayscale image to a binary image. When the *Class* keyword is set to `Bilevel`, the image is converted to a binary image before being stored to disk. Pixels in the image that are greater than the threshold level are set to 255. All other pixels are set to 0. This keyword is only valid when the *Class* keyword is set to `Bilevel`. If `Bilevel` is specified and no threshold is given, the threshold value defaults to 128.

## Discussion

`DC_WRITE_TIFF` facilitates the exporting of TIFF images from PV-WAVE. It also handles many steps that you have to do yourself when using other PV-WAVE functions and procedures. These steps include: 1) opening the file, 2) assigning it a logical unit number (LUN), and 3) closing the file when you are done writing the data.

If *imgarr* is not a 2 or 3-D byte array, `DC_WRITE_TIFF` returns an error status and no data is written to the output file.

### **Requirements of the Various TIFF Classes**

If TIFF class 'B' (Bilevel) is specified, you can use the *Compress* keyword to create compressed TIFF image files.

If TIFF class 'P' (Palette Color) is specified, you must use the *Palette* keyword to specify a palette array.

If TIFF class 'RGB' (RGB Full Color) is specified, *imgarr* must be a 3-D byte array with the last dimension equal to 3. In other words, *imgarr* must be an image-interleaved image; pixel-interleaved images cannot be stored in a TIFF file when using the `DC_WRITE_TIFF` function. The difference between pixel-interleaved and image-interleaved data is discussed in the *PV-WAVE Programmer's Guide*.

## Example 1

If the variable `maverick` is a 512-by-512 byte array, the function call:

```
status = DC_WRITE_TIFF('mav.tif', maverick, $
                    Class='Bi', Compress='Pack')
```

creates the file `mav.tif` and uses it to store the image data contained in the variable `maverick`. The created TIFF file is compressed and conforms to the TIFF 'Bilevel' classification.

## Example 2

If the variable `true` is a 400-by-400-by-3 true-color 24-bit image (byte array), the function call:

```
status = DC_WRITE_TIFF('true_c.tif', true, $
                    Class='RGB')
```

creates the file `true_c.tif` and uses it to store the RGB color image data contained in the variable `true`; image interleaving is used because the variable is 400-by-400-by-3. The created TIFF file conforms to the TIFF RGB Full Color classification.

## See Also

[DC\\_ERROR\\_MSG](#), [DC\\_READ\\_TIFF](#), [IMAGE\\_WRITE](#)

See the *PV-WAVE Programmer's Guide* for more information about TIFF image I/O.

---

## DEFINE\_KEY Procedure

Programs a function key with a string value or with an action. Also programs a control key with an action (UNIX only).

### Usage

DEFINE\_KEY, *key* [, *value*]

### Input Parameters

*key* — The name of a function key to be programmed. Must be a scalar string. PV-WAVE maintains an internal list of function key names and the escape sequences they send.

---

**UNIX USERS** Under UNIX, if *key* is not already on PV-WAVE's internal list, you must use the *Escape* keyword to specify the escape sequence; otherwise, *key* alone will suffice. The section [Standard Function Keys Under UNIX on page 262](#) describes the standard key definitions; however, available function keys and the escape sequences they send vary from keyboard to keyboard.

---

**OpenVMS USERS** Under OpenVMS, key names are defined by the Screen Management utility (SMG). The section [Standard OpenVMS Function Keys on page 263](#) describes some of these keys. For a complete description, see the *OpenVMS RTL Screen Management (SMG\$) Manual*.

---

*value* — (optional) The scalar string that *key* will be programmed with. Afterwards, pressing the programmed key results in *value* being entered as if it had been typed manually at the keyboard. If *value* is not present, and no function is specified for the key with one of the keywords, the key is cleared, and nothing will happen when it is pressed.

### Keywords

---

**NOTE** Most of the following keywords work under UNIX only.

---

**Back\_Character** — (UNIX only) Programs *key* to move the current cursor position left one character.

**Back\_Word** — (UNIX only) Programs *key* to move the current cursor position left one word.

**Delete\_Character** — (UNIX only) Programs *key* to delete the character to the left of the current cursor.

**Delete\_Forward\_Char** — (UNIX only) — Programs *key* to delete the character to the right of the cursor.

**Delete\_Line** — (UNIX only) Programs *key* to delete all characters to the left of the current cursor.

**Delete\_To\_EOL** — (UNIX only) Programs *key* to delete all characters to the right of the cursor.

**Delete\_Word** — (UNIX only) Programs *key* to delete the word to the left of the current cursor.

**End\_of\_Line** — (UNIX only) Programs *key* to move the current cursor to the end of the line.

**Enter\_Line** — (UNIX only) Programs *key* to enter the current line. This is the action normally performed by the <Return> key.

**Escape** — (UNIX only) Specifies the escape sequence that corresponds to *key*. *Escape* must be a scalar string. See [Defining New UNIX Function Keys on page 262](#) for further details.

**Forward\_Character** — (UNIX only) Programs *key* to move the current cursor position right one character.

**Forward\_Word** — (UNIX only) Programs *key* to move the current cursor position right one word.

**Insert\_Overstrike\_Toggle** — (UNIX only) Programs *key* to toggle between insert and overstrike mode. When characters are typed into the middle of a line, insert mode causes the trailing characters to be moved to the right to make room for the new ones, while overstrike mode causes the new characters to overwrite the existing ones.

**Match\_Previous** — Programs *key* to prompt the user for a string, and then search the saved command buffer for the most recently issued command that contains that string. If a match is found, the matching command becomes the current command; otherwise the last command entered is used.

---

**UNIX USERS** Under UNIX, the default match key is the up caret “^” key when pressed in column 1.

---

---

**OpenVMS USERS** Under OpenVMS, the default match key is <PF1>.

---

**Next\_Line** — (UNIX only) Programs *key* to move forward one command in the saved command buffer and make it the current command.

**Noecho** — If nonzero, and *value* is present, *Noecho* specifies that when *key* is pressed, its value should be entered without being echoed. This is useful for defining keys that perform actions such as erasing the screen. If *Noecho* is specified, the *Terminate* keyword is assumed to be present and nonzero also.

**Previous\_Line** — (UNIX only) Programs *key* to move back one command in the saved command buffer and make it the current command.

**Redraw** — (UNIX only) Programs *key* to redraw the current line.

**Start\_of\_Line** — (UNIX only) Programs *key* to move the current cursor to the start of the line.

**Terminate** — If nonzero, and *value* is present, *Terminate* specifies that pressing *key* terminates the current input operation after its value is entered. It acts as an implicit <Return> added to the end of *value*.

## Discussion

The SETUP\_KEYS procedure should be used once at the beginning of the session to enter the keys for the current keyboard.

It is convenient to include commonly used key definitions in a startup file so that they will always be available.

---

**NOTE** For a discussion of startup files, see the *PV-WAVE User's Guide*.

---

To see information on the currently defined keys, enter:

```
INFO, /Keys
```

## Defining Control Keys

To define a control key, use the circumflex character (^) before any character A through Z, either upper or lowercase. For example:

```
DEFINE_KEY, '^F', /Forward_key
```

This command defines <Control>-F to move the cursor one character to the right.

You cannot bind a control key to a string, and some control keys are used for process management. For example, <Control>-C is usually used to interrupt a UNIX process and <Control>-Z is used to suspend a UNIX process. These special characters are listed in Chapter 2, *Getting Started*, in the *PV-WAVE User's Guide*.

---

**UNIX USERS** The UNIX `stty` command can be used to rebind `tty` control characters or to eliminate them altogether. Refer to the `stty` man page for more information.

---

## Defining New UNIX Function Keys

To add a definition for a function key that is not built into PV-WAVE's default list of recognized keys, use the *Escape* keyword to specify the escape sequence it sends. For example, to add a function key named <HELP> which sends the escape sequence <Escape> [28~, use the command:

```
DEFINE_KEY, 'HELP', Escape = '\033[28~'
```

This command adds the <HELP> key to the list of keys understood by PV-WAVE. Since only the key name and escape sequence were specified, pressing the <HELP> key will do nothing. The *value* parameter, or one of the keywords provided to specify command line editing functions, could have been included in the above statement to program it with an action.

Once a key is defined using the *Escape* keyword, it is contained in the internal list of function keys. It can then be subsequently re-defined without specifying the escape sequence.

However, if the `SETUP_KEYS` procedure is used to define the function keys found on the keyboard, it is not necessary to specify the *Escape* keyword. For example, the following statements program the <F2> key on a Sun keyboard to redraw the current line:

```
SETUP_KEYS  
DEFINE_KEY, 'F2', /Redraw
```

## Standard Function Keys Under UNIX

Under UNIX, PV-WAVE can handle arbitrary function keys. Standard UNIX key definitions are listed in the following table.

---

**NOTE** SunOS users, the function keys R8, R10, R12, and R14 (the arrow buttons) are reserved and cannot be set with `DEFINE_KEY`. Also, the L1—L10 keys are reserved for use by the window manager and cannot be set with `DEFINE_KEY`.

---

## UNIX Line Editing Keys

<b>Editing Key</b>	<b>Function</b>
<Control> <A>	Move cursor to start of line.
<Control> <B>	Move cursor left one word.
<Control> <D>	EOF if current line is empty, EOL otherwise.
<Control> <E>	Move to end of line.
<Control> <F>	Move cursor right one word.
<Control> <N>	Move back one line in recall buffer.
<Control> <R>	Redraw current line.
<Control> <U>	Delete from current position to start of line.
<Control> <W>	Delete previous word.
<Backspace>, <Delete>	Delete previous character.
<Escape> <I>	Overstrike/Insert mode toggle.
<Escape> <Delete>	Delete previous word.
Up Arrow	Move back one line in recall buffer.
Down Arrow	Move forward one line in recall buffer.
Left Arrow	Move left one character.
Right Arrow	Move right one character.
<R13>	Move cursor left one word (Sun keyboard).
<R15>	Move cursor right one word (Sun keyboard).
<i>text</i>	If first character, recall first line containing text; if text is blank, recall previous line.
<i>other characters</i>	Insert the character at the current cursor position.

## Standard OpenVMS Function Keys

Under OpenVMS, PV-WAVE uses the SMG screen-management package, which ensures that PV-WAVE command editing will behave in the standard OpenVMS way. Therefore, it is not possible to use a key SMG does not understand. Some of the most commonly used SMG-defined keys are listed in the following table:

## OpenVMS Line Editing Keys

<b>Key Name</b>	<b>Comment</b>
<DELETE>	
<PF1>	Recall most recent command that matches supplied string.
<PF2> — <PF4>	Top row of keypad.
<KP0> — <KP9>	Keypad 0-9 keys.
<ENTER>	Keypad ENTER key.
<MINUS>	Keypad “-” key.
<COMMA>	Keypad “,” key.
<PERIOD>	Keypad “.” key.
<FIND>	Editing keypad FIND key.
<INSERT_HERE>	Editing keypad INSERT_HERE key.
<REMOVE>	Editing keypad REMOVE key.
<SELECT>	Editing keypad SELECT key.
<PREV_SCREEN>	Editing keypad PREV_SCREEN key.
<NEXT_SCREEN>	Editing keypad NEXT_SCREEN key.

## Standard Function Keys Under Windows

Standard key definitions for PV-WAVE running under Windows are listed in the following table:

### Windows Function Keys

<b>Editing Key</b>	<b>Function</b>
<Control> <A>	Move cursor to start of line.
<Control> <B>	Move cursor left one word.
<Control> <D>	EOF if current line is empty, EOL otherwise.
<Control> <E>	Move to end of line.
<Control> <F>	Move cursor right one word.

## Windows Function Keys (Continued)

<b>Editing Key</b>	<b>Function</b>
<Control> <N>	Move back one line.
<Control> <R>	Redraw current line.
<Control> <U>	Delete from current position to start of line.
<Control> <W>	Delete previous word.
<Backspace>, <Delete>	Delete previous character.
Escape <Delete>	Delete previous word.
<Control> <i>text</i>	If the line is empty, find the last command matching <i>text</i> .
Up Arrow	Move back one line.
Down Arrow	Move forward one line.
Left Arrow	Move left one character.
Right Arrow	Move right one character.
Insert/Overstrike	Toggle between insert and overstrike mode.
Page Down	Move forward one line.
Page Up	Move back one line.
End	Move to the end of the current line.
Home	Move to the start of the current line.
Insert	Toggle between insert and overstrike mode.
<F3>	Execute the INFO command.
<F2>	Run the PV-WAVE Gallery.
<F1>	Run the online help system.

### Example

You can define the <F12> key to execute INFO, /Keys with the statement:

```
DEFINE_KEY, /Terminate, 'F12', 'INFO, /Keys'
```

the INFO, /Keys command produces output that includes the line:

```
F12 <\03[P> = INFO, /Keys <Terminate>
```

showing the new key definition.

## See Also

[SETDEMO](#), [SETUP\\_KEYS](#)

---

## **DEFROI Function**

Standard Library function that defines an irregular region of interest within an image by using the image display system and the mouse.

### **Usage**

*result* = DEFROI(*size<sub>x</sub>*, *size<sub>y</sub>* [, *xverts*, *yverts*])

### **Input Parameters**

*size<sub>x</sub>* — The size of the image, in pixels, in the *x* direction.

*size<sub>y</sub>* — The size of the image, in pixels, in the *y* direction.

### **Output Parameters**

*xverts* — (optional) The *x*-coordinates of the vertices enclosing the region.

*yverts* — (optional) The *y*-coordinates of the vertices enclosing the region.

### **Returned Value**

*result* — A vector containing the subscripts of the pixels inside the region.

### **Keywords**

*Noregion* — If nonzero, inhibits the return of the pixel subscripts.

*Xo* — The *x*-coordinate of the lower-left corner of the image in the window. Screen device coordinates are used.

*Yo* — The *y*-coordinate of the lower-left corner of the image in the window. Screen device coordinates are used.

*Zoom* — The zoom factor to be used for displaying the image. If omitted, 1 is assumed.

## Discussion

DEFROI lets you interactively select a portion of an image for further processing—simply point with the mouse to the vertices of an irregular polygon containing the region of interest inside the image.

The write mask for the display is set so that only bit 0 may be written. Bit 0 is erased for all pixels and is used to draw the outline of the region. (This may have to be changed to fit the capabilities and procedures of your device.) The common block COLORS is used to obtain the current color table, which is modified and then restored. The color tables are loaded with odd values complemented and even values unchanged.

A message is printed to assist you in selecting the region with the mouse. The POLYFILLV function is used to compute the subscripts within the region.

### Example

This example uses DEFROI to define a region of interest within an image. The subscripts of pixels within the region, which are returned by DEFROI, are used to invert the colors within the region of interest. The pixels outside the region are not altered.

```
OPENR, unit, FILEPATH('whirlpool.img', $
    Subdir = 'data'), /Get_Lun
    ; Open the whirlpool.img file.

a = ASSOC(unit, BYTARR(512, 512))
    ; Associate a 512-by-512 byte array with the file unit number
    ; of whirlpool.img.

g = a(0)
    ; Read the galaxy image into the variable g.

FREE_LUN, unit
    ; Free the file unit number in unit.

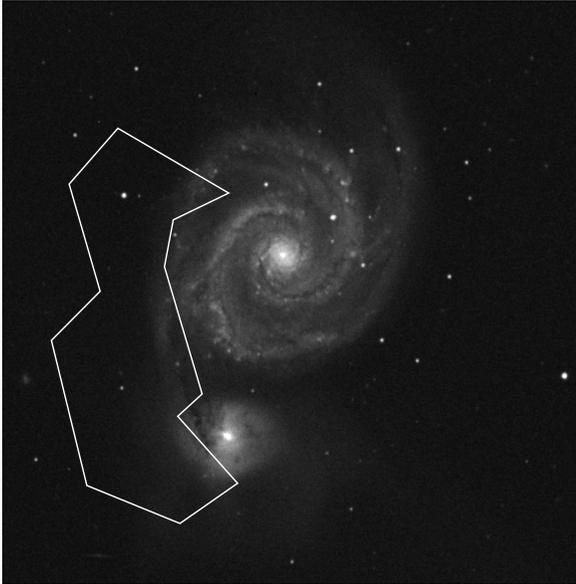
!Order = 1

LOADCT, 3
    ; Load the red temperature color table. Scale the array containing
    ; the image so that the maximum color used is !D.N_Colors.

g = BYTSCL(g, Top = !D.N_Colors)
    ; Display the image.

WINDOW, 0, Xsize = 512, Ysize = 512
TV, g
```

```
subs = DEFROI(512, 512)  
; Define a region of interest using DEFROI.
```



**Figure 2-16** Galaxy image with region of interest defined by DEFROI.

```
g(subs) = !D.N_Colors - g(subs)  
; Invert the colors of pixels that lie within the specified region.  
TV, g  
; Display the resulting image.
```



**Figure 2-17** Galaxy image with region of interest inverted.

## See Also

[POLYFILLV](#)

---

## ***DEFSYSV Procedure***

Creates a new system variable initialized to the specified value.

### **Usage**

`DEFSYSV, name, value [, read_only]`

### **Input Parameters**

***name*** — A scalar string containing the name of the system variable to be created. All system variables must begin with the ! character.

***value*** — An expression from which the type, structure, and initial value of the new system variable is taken. May be a scalar, array, or structure.

*read\_only* — (optional) If present and nonzero, causes the resulting system variable to be read-only. Otherwise, the value for *name* may be modified.

## Keywords

None.

## Discussion

System variables can be defined at any program level (in functions and procedures, and at the main program level).

## Example

This example uses procedure DEFSYSV to create read-only system variables to hold the constant  $e$ , which is the base of the natural logarithm, in both single-precision and double-precision forms.

```
DEFSYSV, '!e', 2.71828, 1
    ; Create the single-precision system variable containing e.
DEFSYSV, '!de', 2.718218D, 1
    ; Create the double-precision system variable containing e.
INFO, /System_Variables
    ; Use the INFO procedure to display all system variables.
.
.
.
!DE = 2.7182180
.
.
.
!E = 2.71828
.
.
.
```

---

## ***DELETE\_SYMBOL Procedure (OpenVMS)***

Deletes a DCL (Digital Command Language) interpreter symbol from the current process.

### **Usage**

`DELETE_SYMBOL, name`

### **Input Parameters**

*name* — A scalar string containing the name of the symbol to be deleted.

### **Keywords**

*Type* — Indicates the OpenVMS table from which *name* will be deleted:

- 1 Specifies the local symbol table (the default).
- 2 Specifies the global symbol table.

### **Example**

```
DCL COMMAND LINE> my_sym := $dev$:[mydir]my.exe
DCL COMMAND LINE> wave
. . .
WAVE> DELETE_SYMBOL, 'my_sym', Type=2
```

### **See Also**

[DELLOG](#), [GET\\_SYMBOL](#), [SETLOG](#), [SET\\_SYMBOL](#), [TRNLOG](#)

---

## ***DEL\_FILE Procedure***

Deletes a specified file on your system.

### **Usage**

`DEL_FILE, filename`

### **Input Parameters**

*filename* — A string containing the full pathname of the file to be deleted.

### **Keywords**

None.

### **Discussion**

This function uses SPAWN to execute a platform-specific command to delete the specified file.

---

**OpenVMS USERS** On OpenVMS systems this function deletes all versions of the specified file.

---

### **UNIX Example 1**

```
full_name = FILEPATH('myproc.pro', Subdirectory='lib/user')
PRINT, full_name
      /usr/local/vni/wave/lib/usr/myproc.pro
DEL_FILE, full_name
```

### **UNIX Example 2**

```
DEL_FILE, FILEPATH('scratch10', /Tmp)
```

### **OpenVMS Example 1**

```
full_name = FILEPATH('myproc.pro', Subdirectory='lib.user')
PRINT, full_name
      WAVE_DIR:[lib.user]myproc.pro
DEL_FILE, full_name
```

## OpenVMS Example 2

```
DEL_FILE, FILEPATH('scratch10',/Tmp)
```

## Windows Example 1

```
full_name = FILEPATH('myproc.pro', Subdirectory='lib\user')
PRINT, full_name
      d:\vni\wave\lib\user\myproc.pro
DEL_FILE, full_name
```

## Windows Example 2

```
DEL_FILE, FILEPATH('scratch10',/Tmp)
```

## See Also

[FINDFILE](#), [SPAWN \(UNIX/OpenVMS\)](#), [SPAWN \(Windows\)](#)

---

## ***DELFUNC Procedure***

Deletes one or more compiled functions from memory.

### **Usage**

```
DELFUNC, function1, ..., functionn
```

### **Input Parameters**

*function*<sub>i</sub> — (string) The name of a compiled function to delete.

### **Keywords**

*All* — When present and nonzero, deletes all currently compiled functions. When this keyword is used, all other parameters are ignored.

### **Discussion**

Use this procedure to free the memory taken by compiled functions. You can obtain a list of compiled functions by entering: `INFO, /Routines`.

## Example

```
INFO, /Routines
  Saved Procedures:
  LOADCT table_number "SILENT"
  Saved Functions:
  DIST n
  FILEPATH filename "SUBDIRECTORY"
DELFUNC, "Filepath", "Dist"
  ; Deletes compiled FILEPATH and DIST functions from memory.
```

## See Also

[DELPROC](#), [DELSTRUCT](#), [DELVAR](#)

---

## ***DELLOG Procedure (OpenVMS)***

Deletes a logical name.

### Usage

DELLOG, *logname*

### Input Parameters

*logname* — A scalar string containing the name of the logical to be deleted.

### Keywords

*Table* — A scalar string giving the name of the logical table from which to delete *logname*. If *Table* is not specified, the system LNM\$PROCESS\_TABLE is used.

## See Also

[DELETE\\_SYMBOL](#), [GET\\_SYMBOL](#), [SETLOG](#), [SET\\_SYMBOL](#), [TRNLOG](#)

---

## ***DELPROC Procedure***

Deletes one or more compiled procedures from memory.

### **Usage**

`DELPROC, procedure1, ..., proceduren`

### **Input Parameters**

*procedure<sub>i</sub>*— A string containing the name of a compiled procedure to be deleted.

### **Keywords**

*All*— When present and nonzero, deletes all currently compiled procedures. When this keyword is used, all other parameters are ignored.

### **Discussion**

Use this procedure to free the memory taken by compiled procedures. You can obtain a list of compiled procedures by entering: `INFO, /Routines`.

### **Example**

```
INFO, /Routines
  Saved Procedures:
  LOADCT table_number "SILENT"
  Saved Functions:
  DIST n
  FILEPATH filename "SUBDIRECTORY"
DELPROC, "Loadct"
      ; Deletes the compiled LOADCT procedure from memory.
```

### **See Also**

[DELFUNC](#), [DELSTRUCT](#), [DELVAR](#)

---

## ***DELSTRUCT Procedure***

Deletes one or more named structure definitions from memory.

### **Usage**

DELSTRUCT, {*structure*<sub>1</sub>} ,..., {*structure*<sub>*n*</sub>}

### **Input Parameters**

*structure*<sub>*i*</sub> — The name of the structure definition to be deleted. The name can be specified as {*structure*}, "*structure*", or *x*, where *x* is a variable of type structure.

### **Keywords**

**All** — If nonzero, deletes all named structure definitions not currently being referenced by a variable.

**Rename** — If present and nonzero, this keyword causes the structure definition to be re-named. DELSTRUCT cannot delete a structure definition if it is currently being used (referenced) by a variable, common block, or other structure definition. If a structure is being used and **Rename** is given, then the structure definition will be renamed. If the structure is not being used, it will be deleted.

**Unnull** — If nonzero, deletes all unnamed structure definitions not currently being referenced by a variable.

### **Discussion**

DELSTRUCT is useful for freeing memory that is currently taken by unused structure definitions. In addition, this procedure can be used if you need to correct an existing structure definition. To do this, delete the incorrect definition and then create a new, correct structure definition.

If the structure definition is not currently referenced by any variables, other structure definitions, or common blocks, then the structure is deleted. If any references to *structure* exist, then, by default, you receive an error message and the structure is not deleted. The **Rename** keyword overrides this default, and renames the existing structure so that the structure name can be reused. When the **Rename** keyword is used, the original variables remain valid (continue to reference the renamed structure definition); however, no memory is freed.

Use the STRUCTREF function to determine if a structure is currently referenced by any variables, common blocks, or other structure definitions.

---

**TIP** You cannot delete structure definitions that are system structure definitions, such as !Axis and !Plot, or any structures that begin with an exclamation mark (!). Therefore, if you want to create a new structure that cannot be deleted, begin its name with an exclamation mark (!).

---

## Example

```
x = {struct1, a:float(0)}  
    ; Create a structure.  
DELVAR, x  
    ; Delete the variable.  
DELSTRUCT, {struct1}  
x = {struct1, a:double(0)}  
    ; Now delete and recreate the structure, changing the data  
    ; type of x.a.
```

## See Also

[DELFUNC](#), [DELPROC](#), [DELVAR](#), [STRUCTREF](#)

For more information on structures, see the *PV-WAVE Programmer's Guide*.

---

## ***DELVAR Procedure***

Deletes variables and their symbols from \$MAIN\$, the main program level of PV-WAVE.

### Usage

```
DELVAR, var1, ... ,varn
```

### Input Parameters

var<sub>i</sub> — One or more named variables to be deleted.

### Keywords

All — If specified, deletes all variables and their symbols on the main program level.

## Discussion

DELVAR may be called from any program level to delete a variable from the main program level. If DELVAR is called from the main program level, the variable is directly deleted.

When DELVAR is used to delete a local variable, the variable is also deleted from the main program level. If DELVAR is called from a procedure, one of the two following requirements applies:

- UPVAR must be used to bind the local variable on the procedure level to the variable on the main program level.
- The variable on the main program level must be passed as a parameter to the procedure or function from which DELVAR is called.

## Example

This example creates three variables of differing type and structure, then deletes them using DELVAR.

```
a = FINDGEN(3)
    ; Create a single-precision, floating-point vector, a.
b = {structb, field1:1.0, field2:[5, 6, 7], $
    field3:"pv-wave"}
    ; Create an anonymous structure, b.
c = 6L
    ; Create a longword scalar, c.
INFO, a, b, c
A FLOAT = Array(3)
B STRUCT = -> STRUCTB Array(1)
C LONG = 6
DELVAR, a, c
    ; Delete variables a and c.
INFO, a, b, c
A UNDEFINED = <Undefined>
B STRUCT = -> STRUCTB Array(1)
C UNDEFINED = <Undefined>
DELVAR, b
    ; Delete variable b.
INFO, a, b, c
A UNDEFINED = <Undefined>
```

B UNDEFINED = <Undefined>

C UNDEFINED = <Undefined>

## See Also

[DELFUNC](#), [DELPROC](#), [DELSTRUCT](#)

For more information on releasing memory to the operating system, see the *PV-WAVE Programmer's Guide*.

---

## DERIV Function

Standard Library function that calculates the first derivative of a function in  $x$  and  $y$ .

### Usage

*result* = DERIV([ $x$ ,]  $y$ )

### Input Parameters

$x$  — (optional) The vector of independent  $x$ -coordinates of the data (i.e., the variable with respect to which the function should be differentiated). Must be a one-dimensional array (a vector).

$y$  — The vector of dependent  $y$ -coordinates at which the derivative of function  $f$  is evaluated. Must be a one-dimensional array (a vector).

### Returned Value

*result* — The first derivative of the vector  $y$ , with respect to the independent variable  $x$ . The result has the same size as  $y$ .

### Keywords

None.

### Discussion

---

**NOTE** DERIV does not support complex numbers.

---

The numerical differentiation algorithm for DERIV uses a three-point Lagrangian interpolation.

The vector of  $x$ -coordinates,  $x$ , is optional. The conditions set on this vector are given below:

- If you specify  $x$ , then both  $x$  and  $y$  must be one-dimensional and have the same number of elements. Selecting this option allows you to define the spacing along the  $x$ -axis, for the case where the independent data is not monotonically increasing.
- If you don't specify  $x$ , then it is automatically provided with even spacing, using a unit of one, along the  $x$ -axis. (In other words,  $x(i) = i$ , where  $i = 0, 1, 2, 3, \dots n$ .)

### Example 1

```
x = FINDGEN(10)
xx = x^2
d = DERIV(xx)
PRINT, d
PLOT, xx
OPLOT, d, Linestyle=2
```

### Example 2

```
x = FINDGEN(100) * 10 * !Dtor
; Create an array that contains values 0, 10, 20 ... and multiply these
; by !Dtor (which equals 0.0174533) to convert the values from
; degrees to radians.
sin_x = SIN(x)
d_sin_x = DERIV(x, sin_x)
PLOT, sin_x
OPLOT, d_sin_x, Linestyle=2
```

### See Also

[DERIVN](#)

For an example of the three-point Lagrangian interpolation used in DERIV, see the *Introduction to Numerical Analysis* by F. B. Hildebrand, Dover Publishing, New York, 1987.

---

## ***DERIVN Function***

Standard Library function that differentiates a function represented by an array.

### **Usage**

*result* = DERIVN(*a*, *n*)

### **Input Parameters**

*a* — An array of values of the dependent variable.

*n* — An integer ( $\geq 0$ ) designating which dimension to differentiate.

### **Returned Value**

*result* — An array of the same dimensions as *a*, representing the derivative with respect to the *n*'th independent variable.

### **Keywords**

*x* — A vector defining the independent variable of differentiation. *x* defaults to the indices into dimension *n* of *a*.

### **Examples**

```
pm, derivn( [0,2,1,0,1], 0 )
```

```
pm, derivn( [[0,2,1,0,1],[2,1,0,2,0],[1,0,2,1,2]], 0 )
```

```
pm, derivn( [[0,2,1,0,1],[2,1,0,2,0],[1,0,2,1,2]], 1 )
```

### **See Also**

[DERIV](#), [JACOBIAN](#)

---

## ***DETERM Function***

Standard Library function that calculates the determinant of a square, two-dimensional input variable.

### **Usage**

*result* = DETERM(*array*)

### **Input Parameters**

*array* — An array with two equal dimensions. Can be any data type except string.

### **Returned Value**

*result* — The determinant of the square matrix of *array*. The result is a scalar value, of either single- or double-precision floating-point data type.

### **Keywords**

None.

### **Discussion**

Determinants can be used to evaluate systems of linear equations.

### **Example**

```
a = INTARR(4,4)
FOR i=0,3 DO a(i,*)=[1,i+2,(i+2)^2,(i+2)^3]
PRINT, a
PRINT, DETERM(a)
      12.0000
```

---

## ***DEVICE Procedure***

Provides device-dependent control over the current graphics device (as specified by the SET\_PLOT procedure).

### **Usage**

DEVICE

### **Parameters**

None.

### **Keywords**

Each type of device uses its own unique set of keywords. For a description of these keywords, see [Appendix B, \*Output Devices and Window Systems\*](#).

### **Discussion**

---

**CAUTION** Do not use the DEVICE command when VDA Tools or the Navigator are running. To do so can cause PV-WAVE to crash. VDA Tools and the Navigator expect to be running under the X Window system or Microsoft Windows. Using DEVICE to switch to another device is not supported. VDA Tools set the device internally, e.g., when printing. If you are on a 24-bit display, but are not using the VDA Tools for real 24 bit applications, then set your display to 8-bit *before* starting any VDA Tools. Issuing the command `DEVICE, Pseudo=8` can cause PV-WAVE to crash under these circumstances.

---

The graphics procedures and functions are device-independent. This means that PV-WAVE presents you with a consistent interface to all devices.

However, most devices have extra abilities that are not directly available through this interface. Use DEVICE with the appropriate keywords to control these additional capabilities.

### **See Also**

[SET\\_PLOT](#)

System Variables: [!D](#)

---

## DIAG Function

Makes a diagonal array or extracts the diagonal of an array.

### Usage

$d = \text{DIAG}(a)$

### Input Parameters

$a$  — An array.

### Returned Value

$d$  — If  $a$  is one-dimensional then  $d$  is the  $n$ -dimensional diagonal array with diagonal  $a$ ; otherwise,  $d$  is the diagonal of  $a$ .

### Keywords

$n$  — The dimensionality of  $d$  when  $a$  is one-dimensional. (Default: 2)

### Example

```
a = DIAG( [1,1,1] )    &    PM, a
      1      0      0
      0      1      0
      0      0      1
PM, DIAG( a )
      1
      1
      1
```

---

## DICM\_TAG\_INFO Function

Extracts Digital Imaging and Communications in Medicine (DICOM) tags information from an image associative array.

### Usage

*result* = DICM\_TAG\_INFO (filename, image)

### Input Parameters

*filename* — On input, a string containing the name of the file which contains the descriptions for the DICOM tags.

*image*— An associative array in image format.

### Returned Value

*result* — An associative array containing DICOM tags information.

### Discussion

The DICM\_TAG\_INFO function extracts the tag information from an image associative array that contains a DICOM image. The tag information is returned as an associative array. The following table describes each key of the associative array:

Array Key Name	Variable Type	Description
<i>tag</i>	STRING	A 1-dimensional array containing the DICOM tags
<i>description</i>	STRING	A 1-dimensional array containing the DICOM tag descriptions
<i>value</i>	STRING	A 1-dimensional array containing the DICOMtag values

The DICM\_TAG\_INFO function needs a file containing the tag description as input. This file contains a tag followed by a description for this tag. The tags in this file must be in ascending order. For example:

```
(0002,0000) Group Length UL 1
(0002,0001) File Meta Information Version OB 1
```

## Example

This example uses `IMAGE_READ` to read a DICOM image file. Then it extracts the DICOM tags and displays the information of the result variable.

```
image = IMAGE_READ('test.dcm', File_type='dicm')
tags = DICM_TAG_INFO('dict.txt',image)
INFO, tags, /Full
```

```
TAGS          AS. ARR  = Associative Array(3)
  tag          STRING  = Array(36)
  description  STRING  = Array(36)
  value        STRING  = Array(36)
```

## See Also

[IMAGE\\_READ](#), [IMAGE\\_WRITE](#)

---

## ***DIGITAL\_FILTER Function***

Standard Library function that constructs finite impulse response digital filters for signal processing.

### Usage

```
result = DIGITAL_FILTER(flow, fhigh, gibbs, nterm)
```

### Input Parameters

***flow*** — The value of the lower frequency of the filter, expressed as a fraction of the Nyquist frequency. Must be between 0 and 1.

***fhigh*** — The value of the upper frequency of the filter, expressed as a fraction of the Nyquist frequency. Must be between 0 and 1.

***gibbs*** — The size of the Gibbs Phenomenon variations. Expressed in units of  $-db$  (decibels).

***nterm*** — The number of terms in the filter formula used. Determines the order of the filter.

## Returned Value

**result** — The coefficients of a convolution mask to be used in the filtering of digital signals.

## Keywords

None.

## Discussion

The coefficients returned by `DIGITAL_FILTER` form the convolution mask or kernel that can be used with the `CONVOL` function to apply the filter to a signal. The size of this vector is equal to:

$$(2 * nterm) - 1$$

Highpass, lowpass, bandpass, and bandstop filters can be constructed with `DIGITAL_FILTER`. Use the following values for *fhigh* and *flow* to specify the type of filter you want to obtain:

Desired Effect	Value
No filtering	flow = 0, fhigh = 1
Lowpass filter	flow = 0, 0 < fhigh < 1
Highpass filter	0 < flow < 1, fhigh = 1
Bandpass filter	0 < flow < fhigh < 1
Bandstop filter	0 < fhigh < flow < 1

These non-recursive filters require evenly spaced data points. Frequencies are expressed in terms of the Nyquist frequency,  $1/2T$ , where  $T$  is the time elapsed between data samples.

The Gibbs Phenomenon variations are oscillations which result from the abrupt truncation of the infinite FFT series. Setting the *gibbs* parameter either too high or too low may yield unacceptable results.

---

**TIP** A value of 50 for *gibbs* is a good choice for most filters.

---

## Sample Usage

`DIGITAL_FILTER` is used extensively in image and signal processing applications to build image or signal filters. It provides a convenient way of creating convolution

kernels (containing the filter coefficients)—all you need do is specify the desired filter with respect to the high and low cutoff frequencies, the Gibb's variations, and the number of terms. You can then use the constructed kernels with the CONVOL function to perform the actual filtering operation upon a signal or image.

To evaluate the coefficients of a digital filter and then apply them to a signal, use the following sequence of equations:

$$\text{Coeff} = \text{DIGITAL\_FILTER}(\text{flow}, \text{fhigh}, \text{gibbs}, \text{nterm})$$

$$\text{Filtered\_Signal} = \text{CONVOL}(\text{input\_signal}, \text{coeff})$$

The end result is an image or signal that has certain frequencies or bands of frequencies filtered out of it. For example, an electrical engineer may want to filter out high frequency harmonics or low frequency flutter from a signal. This can easily be achieved by using the high and low pass filters constructed with DIGITAL\_FILTER as the coefficients in the CONVOL function.

---

**NOTE** Two or more filters created by DIGITAL\_FILTER can be combined by addition, subtraction, or averaging to create multiple filtering effects with one filter.

---

## Example 1

A digital signal processing example follows:

```
av_temp = FLTARR(140)
OPENR, unit, !Data_dir + 'example_air_q.dat', /Get_lun
READF, unit, av_temp, Format='(5X,F9.4)'
    ; Read the average temperature field from the air quality test dataset.

FREE_LUN, unit
PLOT, av_temp
    ; Display the original data.

filter = DIGITAL_FILTER(0.0, 0.1, 50, 10)
    ; Create the convolution kernel for a lowpass filter.

filt_temp = CONVOL(av_temp, filter)
    ; Filter the data by convolving it with the kernel.

OPLOT, filt_temp, Linestyle=2
    ; Display the filtered data using a dashed line.
```

## Example 2

An image processing example follows:

```

mandril = BYTARR(512,512)
OPENR, unit, !Data_dir + 'mandril.img', /Get_lun
READU, unit, mandril
    ; Read the mandril demo image.

FREE_LUN, unit

WINDOW, XSize=512, YSize=512
TV, mandril
    ; Display the original image.

mandril = FLOAT(mandril)
    ; Convert the byte data to floating-point for filtering.

filter = DIGITAL_FILTER(0.0, 0.1, 50, 10)
    ; Create the convolution kernel for a lowpass filter.

filt_image = CONVOL(mandril, filter)
    ; Filter the image by convolving it with the kernel.

TV, filt_image
    ; Display the filtered image.

```

## See Also

### [CONVOL](#)

DIGITAL\_FILTER is adapted from the article “Digital Filters,” by Robert Walraven, in *Proceedings of the Digital Equipment User’s Society*, Fall 1984, Department of Applied Science, University of California, Davis, CA 95616.

## ***DILATE Function***

Implements the morphologic dilation operator for shape processing.

### **Usage**

*result* = DILATE(*image*, *structure* [, *x<sub>0</sub>*, *y<sub>0</sub>*])

### **Input Parameters**

*image* — The array to be dilated.

*structure* — The structuring element. May be a one- or two-dimensional array. Elements are interpreted as binary (values are either zero or nonzero), unless the *Gray* keyword is used.

$x_0$  — (optional) The  $x$ -coordinates of *structure*'s origin.

$y_0$  — (optional) The  $y$ -coordinates of *structure*'s origin.

## Returned Value

*result* — The dilated image.

## Keywords

*Gray* — A flag which, if present, indicates that gray scale, rather than binary dilation, is to be used.

*Values* — An array providing the values of the structuring element. Must have the same dimensions and number of elements as *structure*.

## Discussion

If *image* is not of the byte type, PV-WAVE makes a temporary byte copy of *image* before using it for the processing.

The optional parameters  $x_0$  and  $y_0$  specify the row and column coordinates of the structuring element's origin. If omitted, the origin is set to the center, ( $\lfloor N_x / 2 \rfloor$ ,  $\lfloor N_y / 2 \rfloor$ ), where  $N_x$  and  $N_y$  are the dimensions of the structuring element array. However, the origin need not be within the structuring element.

Nonzero elements of the *structure* parameter determine the shape of the structuring element (neighborhood).

If the *Values* keyword is not used, all elements of the structuring element are 0, yielding the neighborhood maximum operator.

You can choose whether you want to use gray scale or binary dilation:

- If you select binary dilation type, the image is considered to be a binary image with all nonzero pixels considered as 1. (You will automatically select binary dilation if you don't use either the *Gray* or *Values* keyword.)
- If you select gray scale dilation type, each pixel of the result is the maximum of the sum of the corresponding elements of *values* overlaid with *image*. (You will automatically select gray scale dilation if you use either the *Gray* or *Values* keyword.)

## Background Information

The DILATE function implements the morphologic dilation operator on both binary and gray scale images. Mathematical morphology provides an approach to

the processing of digital images on the basis of shape. This approach is summarized below.

DILATE returns the dilation of *image* by the structuring element, *structure*. This operation is also commonly known as filling, expanding, or growing. It can be used to fill holes that are equal in size or smaller than the structuring element, or to grow features contained within an image. The result is an image that contains items that may touch each other and become one. Sharp-edged items and harsh angles typically become dull as they expand and grow.

---

**NOTE** Dilation can be used to change the morphological structure of objects or features in an image to see what would happen if they were to actually expand over time.

---

Used with gray scale images, which are always converted to a byte type, the DILATE function is accomplished by taking the maximum of a set of sums. It may be conveniently used to implement the neighborhood maximum operator, with the shape of the neighborhood given by the structuring element.

Used with binary images, where each pixel is either 1 or 0, dilation is similar to convolution. On each pixel of the image, the origin of the structuring element is overlaid. If the image pixel is nonzero, each pixel of the structuring element is added to the result using the logical OR operator.

Letting  $A \oplus B$  represent the dilation of an image  $A$  by structuring element  $B$ , dilation may be defined as:

$$C = A \oplus B = \bigcup_{b \in B} (A)_b$$

where  $(A)_b$  represents the translation of  $A$  by  $b$ . Intuitively, for each nonzero element  $b_{ij}$  of  $B$ ,  $A$  is translated by  $i,j$  and summed into  $C$  using the OR operator.

### ***Openings and Closings***

The *opening* of image  $B$  by structuring element  $K$  is defined as:

$$(B \ominus K) \oplus K$$

The *closing* of image  $B$  by  $K$  is defined as:

$$(B \oplus K) \ominus K$$

where the erosion operator is denoted by  $\ominus$  and is implemented by the ERODE function.

As stated by Haralick *et al*:

“The result of iteratively applied dilations and erosions is an elimination of specific image detail smaller than the structuring element without the global geometric distortion of unsuppressed features. For example, opening an image with a disk structuring element smooths the contour, breaks narrow isthmuses, and eliminates small islands and sharp peaks or capes.

Closing an image with a disk structuring element smooths the contours, fuses narrow breaks and long thin gulfs, eliminates small holes, and fills gaps on the contours.”

## Example 1

In the example below, the origin of the structuring element is at (0,0):

0100	0110
0100	0110
0110	$\oplus 11 = 0111$
1000	1100
0000	0000

## Example 2

Here is what an aerial image looks like before and after applying the DILATE function three different times. For this example, the following parameters were used each time:

```
img = DILATE(aerial_img, struct, /Gray)
```

where `struct` has a value of `[1 0 1]`.

Because the DILATE function was applied to the image three times, the “blurring” is more pronounced than it would have been with only one dilation.



**Figure 2-18** The DILATE function has been used to fuse the visual elements of this 512-by-512 aerial image.

## See Also

### [ERODE](#)

For details on the approach used in the DILATE function, refer to the source document: Haralick, Sternberg, and Zhuang, “Image Analysis Using Mathematical Morphology,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, No. 4, pp. 532-550, July 1987.

---

## ***DINDGEN Function***

Returns a double-precision floating-point array with the specified dimensions.

### **Usage**

*result* = DINDGEN(*dim*<sub>1</sub>, ..., *dim*<sub>*n*</sub>)

### **Input Parameters**

*dim*<sub>*i*</sub> — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### **Returned Value**

*result* — An initialized double-precision, floating-point array. If the resulting array is treated as a one-dimensional array, then its initialization is given by the following:

$$\text{array}(i) = \text{DOUBLE}(i), \text{ for } i = 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right)$$

## Keywords

None.

## Example

This example creates a 4-by-2 double-precision, floating-point array.

```
a = DINDGEN(4, 2)
; Create double-precision, floating-point array.

INFO, a
A          DOUBLE          = array(4, 2)

PRINT, a
0.0000000  1.0000000  2.0000000  3.0000000
4.0000000  5.0000000  6.0000000  7.0000000
```

## See Also

[BINDGEN](#), [CINDGEN](#), [DBLARR](#), [FINDGEN](#), [INDGEN](#),  
[LINDGEN](#), [SINDGEN](#)

## ***DIST Function***

Standard Library function that generates a square array in which each element equals the euclidean distance from the nearest corner.

### Usage

*result* = DIST(*n*, [*m*])

### Input Parameters

*n* — The size of the resulting array.

*m* — If this parameter is supplied, the function generates a rectangular Euclidean distance array.

### Returned Value

*result* — The resulting floating-point array.

## Keywords

None.

## Discussion

DIST generates a square array in which each element is proportional to its frequency. A three-dimensional plot of this function displays a surface where each quadrant is a curved quadrilateral forming a common cusp at the center.

The result of the DIST function is an  $n$ -by- $n$  single-precision floating-point array, as defined by:

$$result(i, j) = \sqrt{F(i)^2 + F(j)^2}$$

where

$$F(x) = x \text{ if } 0 \leq x < n/2$$

or

$$F(x) = n - 1 - x \text{ if } x \geq n/2$$

The DIST function is particularly useful for creating arrays that can be used for frequency domain filtering in image and signal processing applications.

---

**TIP** DIST is an excellent choice when you need a two-dimensional array of any size for a fast test display.

---

If the optional parameter  $m$  is supplied, the result is an  $n$ -by- $m$  rectangular Euclidean distance array.

### Example 1

```
mandril = BYTARR(512,512)
OPENR, unit, !Data_dir + 'mandril.img', /Get_lun
READU, unit, mandril
FREE_LUN, unit
    ; Read the demo image.

WINDOW, XSize=512, YSize=512
TV, mandril
    ; Display the original image.

d = DIST(512)
```

```

; Use the DIST function to create a frequency image of the same
; size as the demo image.

n = 1.0
d0 = 10.0
; Set n, the order (steepness) of the Butterworth filter to use, and
; d0, the cutoff frequency.

filter = 1.0 / (1.0 + (d/d0)^(2.0 * n))
; Create a Butterworth low-pass filter to be applied to the image.
; (For other common filters that could be substituted here, see the
; reference listed in the See Also section.)

filt_image = FFT(FFT(mandril, -1) * filter, 1)
; Filter the image by transforming it to the frequency domain,
; multiplying by the filter, and then transforming back to the
; spatial domain. (Note that this operation may take a while.)

TVSCL, filt_image
; Display the resulting image.

```

## Example 2

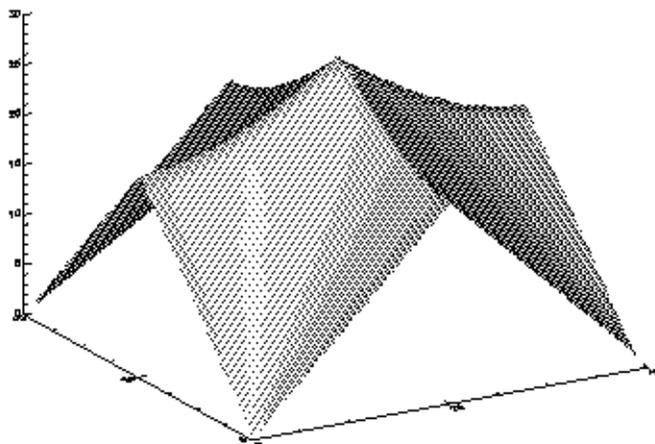
Use these commands:

```

testarr = DIST(40)
CONTOUR, testarr
SURFACE, testarr
LOADCT, 7
SHADE_SURF, testarr
testing = DIST(200)
TVSCL, testing

```

to create the a surface of an array:



**Figure 2-19** Surface view of an array.

## See Also

For more information, see on frequency domain techniques, see the *PV-WAVE User's Guide*.

---

## ***DOC\_LIBRARY Procedure (UNIX/OpenVMS)***

Standard Library procedure that extracts header documentation for user-written procedures and functions.

### **Usage**

`DOC_LIBRARY [, name]`

### **Input Parameters**

***name*** — A string containing the name of the user-written module for which documentation is desired. The search for the file follows the current path in the system variable `!Path`.

## Keywords

**Directory** — (UNIX only) The name of the directory to search. If this keyword is omitted, the current directory and !Path are used.

**File** — (OpenVMS only) If present and nonzero, sends the output to the file `userlib.doc` in the current directory.

**Multi** — (UNIX only) A flag that allows for the printing of more than one file. To do this, *Multi* must be nonzero and the named file must exist in more than one directory in the path.

**Path** — (OpenVMS only) The directory/library search path. It has the same format and semantics as the system variable !Path. If this keyword is omitted, !Path is used.

**Print** — A flag to direct the output:

- A value of 1 specifies the output from the procedure is to be sent to the default printer.
- A string value specifies a command to redirect the standard output.
- If the *Print* keyword is not used, documentation is sent to the standard output.

## Discussion

The first line of the header documentation must begin with the characters ; + and the last line of the header documentation must begin with the characters ; - . `DOC_LIBRARY` extracts all the information between the + and - characters. (Each line of the header must begin with the semicolon character to denote a comment line.)

`DOC_LIBRARY` is a useful tool for finding out what is available in the undocumented Users' Library. This procedure can be used to search each routine in the Users' Library and extract all the text that is bracketed by the + and the - characters. This includes the routine's name, purpose, category, calling sequence, inputs, outputs, and modification history.

`DOC_LIBRARY` checks to see what operating system you are using, and then calls the appropriate version `DOC_LIB_UNIX` or `DOC_LIB_VMS`.

Keywords allow you to have the output sent to a printer or displayed on the screen. If the procedure is called without keywords, you are prompted for specific information about the search.

When creating your own PV-WAVE routines, it is helpful to include a ; + as the second line in the file and a ; - as the last informational line so that

DOC\_LIBRARY can then be used to create documentation for the routine. An example of a file set up to use DOC\_LIBRARY in this way is shown below. (All the information shown in bold will be extracted by DOC\_LIBRARY.)

```

FUNCTION COSINES, x, m
;+
; NAME:
; COSINES
; PURPOSE:
; Example of a function to be used by SVDFIT. ;
; Returns COS(i*COS(x(j))).
; CATEGORY:
; Curve fitting.
; CALLING SEQUENCE:
; r = COSINES(x, m)
; INPUTS:
; x = vector of data values. n elements.
; m = order, or number of terms.
; OUTPUTS:
; Function result = (n,m) array,
; where n is the number of points in x,
; and m is the order. r(i,j) = COS(j * x(i))
; MODIFICATION HISTORY:
; DMS, Nov, 1987.
;-
ON_ERROR, 2
;Return to caller if an error occurs.
RETURN, COS(x # FINDGEN(m))
;Couldn't be much simpler.
END

```

## UNIX Examples

```

DOC_LIBRARY, 'gamma'
; On the screen, display the header for the gamma.pro procedure
; using the default search path.

DOC_LIBRARY, 'gamma', Print='cat > gamma_header'
; Print the header for gamma.pro to the file gamma_header.

DOC_LIBRARY, '*', directory= $
' $VNI_DIR/wave/lib/std', $
Print='cat > user_lib_headers'
; Print the headers for all the files located in the Users' Library.

```

## OpenVMS Examples

```
DOC_LIBRARY, 'gamma'  
    ; On the screen, display the header for the gamma.pro procedure  
    ; using the default search path.  
  
DOC_LIBRARY, 'gamma', /File  
    ; Print the header for gamma.pro to the file userlib.doc.
```

## See Also

[INFO](#)

System Variables: [!Path](#)

---

## DOUBLE Function

Converts an expression to double-precision floating-point data type.

Extracts data from an expression and places it in a double-precision floating-point scalar or array.

### Usage

*result* = DOUBLE(*expr*)

This form is used to convert data.

*result* = DOUBLE(*expr*, *offset*, [*dim*<sub>1</sub>, ..., *dim*<sub>*n*</sub> ])

This form is used to extract data.

### Input Parameters

To convert data:

*expr* — The expression to be converted.

To extract data:

*expr* — The expression from which to extract data.

*offset* — The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim*<sub>*i*</sub> — (optional) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

## Returned Value

For data conversion:

**result** — A copy of *expr* converted to double-precision floating-point data type.

For data extraction:

**result** — If *offset* is used, DOUBLE does not convert *result*, but allows fields of data extracted from *expr* to be treated as double-precision floating-point data. If no dimensions are specified, the result is scalar.

## Keywords

None.

## Example

In this example, DOUBLE is used in two ways. First, DOUBLE is used to convert an integer array to double precision, floating point. Next, DOUBLE is used to extract a subarray from the double precision array created in the first step.

```
a = INDGEN(6)
PRINT, a
    0  1  2  3  4  5
; Create an integer vector of length 6 that is initialized to the
; a = INDGEN(6) value of its one-dimensional subscript.

b = DOUBLE(a)
; Convert a to double precision, floating point.

INFO, b
    B                DOUBLE                = Array(6)

PRINT, b
    0.0000000  1.0000000  2.0000000  3.0000000
    4.0000000  5.0000000

c = DOUBLE(b, 16, 2, 2)
; Extract the last four elements of b, and place them in a 2-by-2
; double-precision, floating-point array.

INFO, c
    C                DOUBLE                = Array(2, 2)

PRINT, c
    2.0000000  3.0000000
    4.0000000  5.0000000
```

---

**NOTE** If you want to place the double-precision value of a constant into a variable, it is more efficient to use the `d` or `D` constant notation rather than the double function. For example:

```
x = .0705230784D
```

---

## See Also

[BYTE](#), [COMPLEX](#), [DBLARR](#), [DCOMPLEX](#), [FIX](#), [FLOAT](#), [LONG](#)

For more information on using this function to extract data, see the *PV-WAVE Programmer's Guide*.

---

## ***DROP\_EXEC\_ON\_SELECT Procedure (UNIX)***

Drops a single item from the EXEC\_ON\_SELECT list.

### ***Usage***

DROP\_EXEC\_ON\_SELECT, *lun*

### ***Input Parameters***

*lun* — Logical unit number.

### ***Keywords***

None.

### ***Description***

A logical unit number and associated command is dropped from the EXEC\_ON\_SELECT list. This procedure is designed to be called from an EXEC\_ON\_SELECT callback procedure. When the logical unit number and associated command are dropped from the EXEC\_ON\_SELECT list, the EXEC\_ON\_SELECT procedure returns to the calling routine.

## See Also

[ADD\\_EXEC\\_ON\\_SELECT](#), [EXEC\\_ON\\_SELECT](#), [SELECT\\_READ\\_LUN](#)

---

## ***DT\_ADD Function***

Increments the values in a date/time variable by a specified amount.

### **Usage**

*result* = DT\_ADD(*dt\_var*)

### **Input Parameters**

*dt\_var* — The original date/time variable or array of variables.

### **Returned Value**

*result* — A date/time variable incremented by the specified amount.

### **Keywords**

***Compress*** — If present and nonzero, excludes predefined weekends and holidays from the result. The default is no compression (0).

***Day*** — Specifies an offset value in days.

***Hour*** — Specifies an offset value in hours.

***Minute*** — Specifies an offset value in minutes.

***Month*** — Specifies an offset value in months.

***Second*** — Specifies an offset value in seconds.

***Year*** — Specifies an offset value in years.

---

**NOTE** Only one keyword can be specified at a time. You cannot, for example, specify both years and months in a single DT\_ADD call. But if you need to add, for example, one day and one hour, you can simply add 25 hours.

---

### **Discussion**

The DT\_ADD function returns a date/time variable containing one or more dates/times that have been offset a specified amount.

The keywords specify how the dates and/or times are incremented (added to). If no keyword is specified, the default increment is one day.

Only positive whole numbers (including zero) can be used with the keywords to specify an increment. Therefore, the smallest unit that can be added to *dt\_var* is one second.

## Example

This example shows how to add one day to a date/time variable containing two date/time values.

```
dtarray = STR_TO_DT(['17-03-92', $
    '8-04-93'], Date_Fmt=2)
    ; Convert two date strings to a date/time variable.

DT_PRINT, dtarray
    03/17/1992
    04/18/1993
    ; The date/time variable dtarray contains two dates.

dtarray1 = DT_ADD(dtarray, /Day)
    ; Create a new date/time variable dtarray1 that contains two
    ; dates with one day added to each date.

DT_PRINT, dtarray1
    03/18/1992
    04/19/1993
```

## See Also

[DT\\_SUBTRACT](#), [DT\\_DURATION](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

---

## ***DT\_COMPRESS Function***

Removes previously defined holidays and weekends from the Julian day portion of !DT structures in a date/time variable.

### Usage

```
result = DT_COMPRESS(dt_array)
```

### Input Parameters

*dt\_array* — A date/time variable containing an array of date/time structures.

## Returned Value

*result* — An array of double-precision values containing the compressed Julian days; that is, all days representing holidays and weekends are removed from each value of the array. In addition, the fractional time component of each Julian value is removed.

## Keywords

None.

## Discussion

This function is primarily used to generate compressed date/time data for specialized, user-written plotting applications, such as bar charts. If the *XType* keyword is set to 2, the compressed data can be used with the PLOT and OPLOT procedures.

---

**NOTE** Avoid using DT\_COMPRESS for normal XY plotting with the PLOT and OPLOT commands. Use the *Compress* keyword with PLOT and OPLOT to create compressed date/time results.

---

The value of the system variables !PDT.Exclude\_Holiday and/or !PDT.Exclude\_Weekend must be set to one (the default) before DT\_COMPRESS is called. In addition, the functions CREATE\_WEEKENDS and CREATE\_HOLIDAYS must be run before you use DT\_COMPRESS.

Note that the result of DT\_COMPRESS is a double array of Julian days, not another array of !DT structures.

## Example 1

This example demonstrates how DT\_COMPRESS can be used to compress the weekend days from a date/time variable containing the days in the month of March, 1992. The resulting array of compressed Julian numbers is then processed so that it can be used to create a date/time plot in a specialized plotting application.

```
march1 = VAR_TO_DT(1992, 3, 1, 11, 30, 0)
PRINT, march1
```

```
  { 1992 3 1 11 30 0.00000 87462.479, 0}
```

```
  ; Creates and prints out a variable march1 which is a date/time
  ; variable. Note the Julian Day carefully, 87462. After
  ; compression, this value will be smaller. This is because all the
```

```

        ; weekends from Julian day 1 (September 14, 1752) are
        ; compressed.
marray = DTGEN(march1,31, /Day)
        ; Generates a date/time array containing the 31 days of the month of
        ; March, 1992.
PRINT, marray
    { 1992 3 1 0 0 0.00000 87462.479 0}
    .
    .
    { 1992 3 31 0 0 0.00000 87492.479 0}
CREATE_WEEKENDS, ['Saturday', 'Sunday']
        ; Defines Saturday and Sunday as weekend days.
cmarray = DT_COMPRESS(marray)
PRINT, cmarray
    62472.5 62473.0 62474.0 62475.0
    62476.0 62477.0 62477.5 62477.5
    62478.0 62479.0 62480.0 62481.0
    62482.0 62482.5 62482.5 62483.0
    62484.0 62485.0 62486.0 62487.0
    62487.5 62487.5 62488.0 62489.0
    62490.0 62491.0 62492.0 62492.5
    62492.5 62493.0 62494.0
        ; Creates and prints out a compressed array for the month of
        ; March, 1992. Weekend (compressed) days can be identified
        ; by fraction .5. Note that the values of the weekend days fall
        ; between the end and beginning of the week. Also note that the
        ; Julian numbers are smaller than in the original array. This is
        ; because all of the weekends from Julian day 1 are
        ; compressed.

```

The following block of code must be run before you can use this array of Julian numbers to generate a date/time axis. The DT\_COMPRESS function removed the fractional Time portion of each Julian day, leaving date values with .0 or .5 appended to them. A .0 value indicates that the day is a weekday. A .5 value indicates a compressed day (a weekend). To generate a meaningful plot with this data, two things must be done.

First, the compressed days (the ones ending in .5) must be incremented to the value of the next whole day. Second, the Time portion of the Julian numbers representing weekdays must be restored.

The following code accomplishes both of these objectives:

```

FOR i=0, 30 DO BEGIN $
    whole_day = DOUBLE(FIX(cmarray(i))) $

```

```

delta_day = carray(i) - whole_day $
IF (delta_day GE 0.4) AND $
(delta_day LE 0.6) THEN BEGIN $
    ; Determine if a date value is a weekend day. If it is, then
    ; increment its value to the value of the next whole day.
marray(i) = whole_day + 1.0d $
ENDIF $
ELSE BEGIN $
    fract_day = marray(i).julian - $
    DOUBLE(FIX(marray(i).julian)) $
    carray(i) = whole_day + fract_day $
    ; Restore the fractional portion of weekday Julian values
    ; from the original date/time array variable.
ENDELSE $
ENDFOR

```

After this code is run, `carray` can be used to generate a date/time plot for a specialized plotting application—one where the regular `PLOT` routine is not sufficient. For example, this data could be used to generate a bar chart.

Before using this date data, however, you must first call `PLOT` or `OPLOT` with the `XType` keyword set to 2. This establishes the plot axis and coordinate system, and allows the date/time axis to be generated from an array of Julian numbers.

## Example 2

This example defines some holidays for the year 1992 with the `CREATE_HOLIDAYS` procedure, creates an array with all the days of the year, and then excludes these holidays using the `DT_COMPRESS` function.

```

christmas = VAR_TO_DT(1992, 12, 25)
PRINT, christmas
    { 1992 12 31 0 0 0.00000 87761.000 0 }
    ; Create and print out a date/time variable for Christmas.
    ; The purpose is to show the Julian day before using
    ; DT_COMPRESS. Note the Julian Day is 87761.

day1 = VAR_TO_DT(1992, 1, 1)
yarray = DTGEN(day1, 366)
    ; Create a variable day1 which is used to generate an array that
    ; contains all the days of the year (where 366 is used because
    ; 1992 is a leap year).

x = ['1-1-92', '5-31-92', '7-4-92', $
    '-1-92', '11-24-92', '12-25-92']
    ; Create an array containing date information for the following

```

```

; holidays: New Years, Memorial day, Fourth of July, Labor Day,
; Thanksgiving, and Christmas.
holidays = STR_TO_DT(x, DateFmt=1)
; Create a date/time variable for the holidays.
CREATE_HOLIDAYS, holidays
; Define the holidays by setting the !Holiday_List system variable.
cyarray = DT_COMPRESS(yarray)
; Creates an array that excludes the holidays for 1992. The
; compressed array cyarray appends the .5 decimal to all of the
; holidays. Non-holidays end in .0. When you print out cyarray,
; note the Julian day for Christmas is 87755.5. This is six days
; less than for the yarray, because six holidays were defined
; and compressed out of the result.

```

## See Also

[CREATE\\_HOLIDAYS](#), [CREATE\\_WEEKENDS](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

## ***DT\_DURATION Function***

Standard Library function that determines the elapsed time between the values in two date/time variables.

### Usage

```
result = DT_DURATION(dt_var_1, dt_var_2)
```

### Input Parameters

*dt\_var\_1* — The date/time variable to be subtracted from. Can be a scalar or array variable.

*dt\_var\_2* — The date/time variable to subtract. Can be a scalar or array variable.

### Returned Value

*result* — A double-precision array containing the difference between *dt\_var\_1* and *dt\_var\_2* in days and fractions of days.

## Keywords

**Compress** — If present and nonzero, excludes predefined weekends and holidays from the calculation of duration. The default is no compression (0).

## Discussion

If the input arrays are not of the same dimension, the output will be the size of the smallest input array.

## Example

```
DT1 = str_to_dt('01-02-92', Date_Fmt=2)
DT2 = str_to_dt('01-03-92', Date_Fmt=2)
      ; Create two date/time variables containing February 1, 1992
      ; and March 1, 1992.

diff = DT_DURATION(DT2, DT1)
PRINT, diff
      29.000000
      ; The difference between these dates is 29 days.
```

## See Also

[DT\\_ADD](#), [DT\\_SUBTRACT](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

---

## DTGEN Function

Returns a date/time array variable beginning with a specified date and incremented by a specified amount.

### Usage

*result* = DTGEN(*dt\_start*, *dimension*)

### Input Parameters

**dt\_start** — A date/time variable containing a value representing the first date and time in the new data set.

**dimension** — Specifies the number of date/time values to generate.

### Returned Value

**result** — A date/time array variable containing the specified number of date/time values.

## Keywords

**Compress** — If present and nonzero, excludes predefined weekends and holidays from the result. The default is no compression (0).

**Day** — Specifies an offset value in days.

**Hour** — Specifies an offset value in hours.

**Minute** — Specifies an offset value in minutes.

**Month** — Specifies an offset value in months.

**Second** — Specifies an offset value in seconds.

**Year** — Specifies an offset value in years.

---

**NOTE** Only one keyword can be specified at a time. You cannot, for example, specify both years and months in a single DTGEN call. But if you need to add, for example, one day and one hour, you can simply add 25 hours.

---

## Discussion

Each value in the result is offset from the previous value by the amount specified with a keyword.

DTGEN lets you generate date and time data that match a particular dataset. For example, if you have gathered data at regular intervals, but do not have time stamps in your dataset, you can use DTGEN to generate date and time data that corresponds to your data-gathering intervals.

Only whole numbers (including zero) can be used with the keywords to specify the offset between dates and times. Therefore, the smallest unit by which generated dates can be offset is one second. If no keyword is specified, the default offset is one day.

## Example 1

This example shows how to generate an array of date/time structures for consecutive years.

```
date1 = TODAY()
      ; Create a date/time variable containing the current date.
date2 = DTGEN(date1, 4, /Year)
      ; Use DTGEN to create a new date/time variable containing four
      ; date/time values. The four date/time values represent four
      ; consecutive years with identical months, days, and times.
PRINT, date2
      { 1992 3 26 6 28 50.0000 87487.270 0 }
      { 1993 3 26 6 28 50.0000 87852.270 0 }
```

```
{ 1994 3 26 6 28 50.0000 88217.270 0}  
{ 1995 3 26 6 28 50.0000 88582.270 0}
```

## Example 2

The second example shows how to create an array containing date/time structures for every other month of a year.

```
date = VAR_TO_DT(1992, 1, 1)  
      ; Create a date/time variable for January, 1992.  
date1 = DTGEN(date, 6, Month=2)  
       ; Create an array variable containing date/time data for every other  
       ; month of the year 1992.  
PRINT, date1  
      { 1992 1 1 0 0 0.00000 87402.000 0}  
      { 1992 3 1 0 0 0.00000 87462.000 0}  
      { 1992 5 1 0 0 0.00000 87523.000 0}  
      { 1992 7 1 0 0 0.00000 87584.000 0}  
      { 1992 9 1 0 0 0.00000 87646.000 0}  
      { 1992 11 1 0 0 0.00000 87707.000 0}
```

## See Also

[DT\\_ADD](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

---

## ***DT\_PRINT Procedure***

Standard Library procedure that prints the value in date/time variables in a readable format.

### **Usage**

DT\_PRINT, *dt\_var*

### **Input Parameters**

*dt\_var* — A date/time variable containing one or more date/time structures.

### **Keywords**

None.

## Discussion

The system variables `!Date_Separator` and `!Time_Separator` determine which characters are used to separate the date and time elements in the output. The default delimiter for dates is a slash (/), and the default delimiter for printing times is a colon (:). For example:

```
4/2/1992 7:7:51
```

You can change these separators by changing the values of `!Date_Separator` and `!Time_Separator`.

## Examples

```
x = TODAY()
DT_PRINT, x
    05/06/1992 14:34:54
PRINT, x
    { 1992 5 6 14 34 54.0000 87528.608 0}
dtarray = DTGEN(x,4)
DT_PRINT, dtarray
    4/2/1992 7:7:51.000
    4/3/1992 7:7:51.000
    4/4/1992 7:7:51.000
    4/5/1992 7:7:51.000
```

## See Also

System Variables: [!Date\\_Separator](#), [!Time\\_Separator](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

---

## ***DT\_SUBTRACT Function***

Decrements the values in a date/time variable by a specified amount.

### Usage

```
result = DT_SUBTRACT(dt_var)
```

### Input Parameters

*dt\_var* — The original date/time variable or array of variables.

### Returned Value

*result* — A date/time variable decremented by the specified amount.

## Keywords

**Compress** — If present and nonzero, excludes predefined weekends and holidays from the result. The default is no compression (0).

**Day** — Specifies an offset value in days.

**Hour** — Specifies an offset value in hours.

**Minute** — Specifies an offset value in minutes.

**Month** — Specifies an offset value in months.

**Second** — Specifies an offset value in seconds.

**Year** — Specifies an offset value in years.

---

**NOTE** Only one keyword can be specified at a time. You cannot, for example, specify both years and months in a single DT\_SUBTRACT call. But if you need to subtract, for example, one day and one hour, you can simply subtract 25 hours.

---

## Discussion

The DT\_SUBTRACT function returns a date/time variable containing one or more dates/times that have been offset by the specified amount.

The keywords specify how the dates and/or times are decremented (subtracted from). If no keyword is specified, the default decrement is one day.

Only positive whole numbers (including zero) can be used with the keywords to specify a decrement. Therefore, the smallest unit that can be subtracted from *dt\_var* is one second.

## Example 1

```
dtvar = VAR_TO_DT(1992, 03, 17, 09, 30, 54)
      ; Create a date/time variable containing a date/time.
dtvar1= DT_SUBTRACT(dtvar, Year=4)
      ; Create a new date/time variable by subtracting 4 years from dtvar.
PRINT, dtvar1
      { 1988 3 17 9 30 54.0000 86017.396 0}
      ; Display the new date/time variable.
```

## Example 2

This example shows how to add one day to a date/time variable containing two date/time values.

```
dtarray = STR_TO_DT(['17-03-92', $
                   '8-04-93'], Date_Fmt=2)
      ; Convert two date strings to a date/time variable.
```

```

DT_PRINT, dtarray
    03/17/1992
    04/18/1993
    ; The date/time variable dtarray contains two dates.
dtarray1 = DT_SUBTRACT(dtarray, /Day)
    ; Create a new date/time variable dtarray1 that contains two
    ; dates with one day subtracted from each date.
DT_PRINT, dtarray1
    03/16/1992
    04/17/1993

```

### Example 3

This example shows the effect of using the *Compress* keyword with DT\_SUBTRACT. Assume that you have defined Christmas (December 25, 1992) to be a holiday with the procedure CREATE\_HOLIDAYS.

```

x = VAR_TO_DT(1992, 12, 26)
    ; Begin with a date variable containing December 26, 1992.
y = DT_SUBTRACT(x, /Day, /Compress)
    ; Subtract one day from the variable.
DT_PRINT, y
    12/24/1992
    ; The result is December 24. Normally, the result would be
    ; 12/25/92, but because December 25 is defined as a holiday,
    ; the Compress keyword causes the 25th to be skipped.

```

### See Also

[DT\\_ADD](#), [DT\\_DURATION](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

## DT\_TO\_SEC Function

Standard Library function that converts a date/time variable to a double-precision variable containing the number of seconds elapsed from a base date.

### Usage

```
result = DT_TO_SEC(dt_var)
```

### Input Parameters

*dt\_var* — A date/time variable.

## Returned Value

**result** — A double-precision variable containing the number of seconds elapsed between the base date and the date(s) contained in *dt\_var*. The value of the base date is maintained in the system variable !DT\_Base.

## Keywords

**Base** — A string containing a date, such as “3-27-92”. This is the base date from which the number of elapsed seconds is calculated. *Base* can be used to override the default value in the system variable !DT\_Base.

**Date\_Fmt** — Specifies the format of the base date, if passed into the function. Possible values are 1, 2, 3, 4, or 5, as summarized in the following table:

Value	Format Description	Examples for May 1, 1992
1	MM*DD*[YY]YY	05/01/92
2	DD*MM*[YY]YY	01-05-92
3	ddd*[YY]YY	122,1992
4	DD*mmm[mmmmmm]*[YY]YY	01/May/92
5	[YY]YY*MM*DD	1992-05-01

where the asterisk (\*) represents one of the following separators: dash (–), slash (/), comma (,), period (.), or colon (:).

For a detailed description of these formats, see the *PV-WAVE User’s Guide*.

## Discussion

This function is useful for converting date/time values to relative time. The default base date is September 14, 1752.

## Example1

Assume that you have created the array `date1` that contains the following date/time data:

```
date1=[{!dt, $ 1992,3,27,7,18,57.0000,87488.305,0},$  
{!dt, 1993,3,27,7,18,57.0000,87853.305,0}, $  
{!dt, 1994,3,27,7,18,57.0000,87218.305,0}]
```

To find out the number of seconds for each date from the default base, September 14, 1752, use:

```
seconds = DT_TO_SEC(date1)
PRINT, seconds
      7.5589031e+09 7.5904391e+09 7.5355751e+09
```

## Example 2

Assume that you have created the array `date1` that contains the following date/time data:

```
date1=[{!dt, $ 1992,4,15,7,29,19.0000,87507.312,0},$
{!dt, 1993,4,15,7,29,19.0000,87872.312,0}, $
{!dt, 1994,4,15,7,29,19.0000,88237.312,0}]
```

To find out the number of seconds for each date from January 1, 1970, use:

```
seconds = DT_TO_SEC(date1, $
      Base='1-1-70', Date_Fmt=1)
PRINT, seconds
      7.0332296e+08 7.3485896e+08 7.6639496e+08
```

## See Also

[DT\\_TO\\_STR](#), [DT\\_TO\\_VAR](#), [SEC\\_TO\\_DT](#)

System Variables: [!DT\\_Base](#)

For more information, see the *PV-WAVE User's Guide*.

---

## ***DT\_TO\_STR Procedure***

Converts date/time variables to string data.

### **Usage**

```
DT_TO_STR, dt_var, [, dates] [, times]
```

### **Input Parameters**

*dt\_var* — A date/time variable containing one or more date/time structures.

## Output Parameters

*dates* — (optional) A variable containing the date strings extracted from the date/time variable.

*times* — (optional) A variable containing the time strings extracted from date/time variable.

## Keywords

*Date\_Fmt* — Specifies the format of the date data in the input variable. Possible values are 1, 2, 3, 4, or 5, as summarized in the following table:

Value	Format Description	Examples for May 1, 1992
1	MM*DD*YYYY	05/01/1992
2	DD*MM*YYYY	01-05-1992
3	ddd*YYYY	122,1992
4	DD*mmm[mmmmmm]*YYYY	01/May/1992
5	YYYY*MM*DD	1992-05-01

where the asterisk (\*) represents one of the following separators: dash (–), slash (/), comma (,), period (.), or colon (:).

*Time\_Fmt* — Specifies the format of the time portion of the data in the input variable. Possible values are –1 or –2, as summarized in the following table:

Value	Format Description	Examples for 1:30 p.m.
–1	HH*Mn*SS.sss	13:30:35.25
–2	HHMn	1330

where the asterisk (\*) represents one of the following separators: dash (–), slash (/), comma (,), or colon (:). No separators are allowed between hours and minutes for the –2 format. Both hours and minutes must occupy two spaces.

Date and time separators are specified with the !Date\_Separator and !Time\_Separator system variables. It is possible to use any character or string as a separator with the DT\_TO\_STR function; however, if you use a non-standard separator (one other than dash (–), slash (/), comma (,), period (.), or colon (:)), you will be unable to convert the data back to a date/time variable with STR\_TO\_DT. If Either of these system variables is set to an empty string, then you receive a default separator.

You must specify a date and/or time format if the *dates* and/or *times* parameters are specified.

## Examples

Assume you have a date/time variable named `date1` that contains the following date/time structures:

```
date1=[{!dt, $
        1992,3,13,1,10,34.0000,87474.049,0}, $
        {!dt, 1983,4,20,16,18,30.0000,84224.680,0}, $
        {!dt, 1964,4,24,5,7,25.0000,77289.213,0}]
```

To convert to string data, use the `DT_TO_STR` procedure:

```
DT_TO_STR, date1, d, t, Date_Fmt=1, $
        Time_Fmt=-1
        ; Convert date/time data. Store the date data in d and the time
        ; data in t.

PRINT, d
        3/13/1992 4/20/1983 4/24/1964
PRINT, t
        01:10:34 16:18:30 05:07:25
```

## See Also

[DT\\_TO\\_SEC](#), [DT\\_TO\\_VAR](#), [STR\\_TO\\_DT](#)

System Variables: [!Date\\_Separator](#), [!Time\\_Separator](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

---

## ***DT\_TO\_VAR Procedure***

Standard Library procedure that converts a date/time variable to regular numerical data.

### **Usage**

*DT\_TO\_VAR*, *dt\_var*

### **Input Parameters**

*dt\_var* — A date/time variable.

### **Keywords**

*Year* — Specifies an integer variable to contain the years.

*Month* — Specifies a byte variable to contain the months.

*Day* — Specifies a byte variable to contain the days of the month.

*Hour* — Specifies a byte variable to contain the hours.

*Minute* — Specifies a byte variable contain the minutes.

*Second* — Specifies a floating-point variable to contain the seconds and fractional seconds.

### **Discussion**

Use one or more keywords to specify the kind of output produced by this procedure. For example, to create a new variable containing the years in the date/time variable *mydtvar*, use:

```
DT_TO_VAR, mydtvar, year=myyear
```

The result is a new variable called *myyear* that contains integer values.

### **Example**

Assume that you have created a date/time variable named *date1* that contains the following date/time data:

```
date1=[{!dt, $
        1992,3,13,10,34,15.000,87474.440,0}, $
       {!dt, 1983,4,20,12,30,19.000,84224.521,0}, $
```

```
{!dt, 1964,4,24,16,25,14.000,77350.684,0}]
```

To extract each date/time element into a separate variable:

```
DT_TO_VAR, date1, Year=years, Month=months, Day=days  
; This procedure creates several variables containing the date time data.
```

```
PRINT, "Years = ", years  
Years = 1992 1983 1964  
PRINT, "Months = ", months  
Months = 3 4 6  
PRINT, "Days = ", days  
Days = 13 20 24
```

## See Also

[DT\\_TO\\_SEC](#), [DT\\_TO\\_STR](#), [VAR\\_TO\\_DT](#)

For more information on date/time, see the *PV-WAVE User's Guide*.

---

## ***EMPTY Procedure***

Causes all buffered output for the current graphics device to be written.

### **Usage**

EMPTY

### **Parameters**

None.

### **Keywords**

None.

### **Discussion**

PV-WAVE uses buffered output on many image devices for reasons of efficiency. This leads to rare occasions where a program needs to be certain that data are not waiting in a buffer, but have actually been output. This procedure is handy for such occasions.

EMPTY is a low-level graphics routine. PV-WAVE graphics routines generally handle the flushing of buffered data transparently to you, so the need for EMPTY is extremely rare.

### **See Also**

[FLUSH](#)

---

## ***ENVIRONMENT Function (UNIX/Windows)***

Returns a string array containing all the environment strings for the PV-WAVE process.

### **Usage**

```
result = ENVIRONMENT()
```

### **Parameters**

None.

### **Returned Value**

*result* — A string array containing all the environment strings for the PV-WAVE process. Each element of the result contains one environment string.

### **Keywords**

None.

### **Discussion**

PV-WAVE inherits its environment from its parent process, which is usually the shell (UNIX) or Command Window (Windows) from which it was started.

### **Example**

```
p = ENVIRONMENT()  
PRINT, p  
; This statement prints a list of the environment variables defined for  
; the shell in which PV-WAVE was started.
```

### **See Also**

[GETENV](#), [SETENV](#)

---

## EOF Function

Tests the specified file unit for the end-of-file condition.

### Usage

*result* = EOF(*unit*)

### Input Parameters

*unit* — The logical unit number (LUN) of the file to be tested.

### Returned Value

*result* — Returns 1 if the file is positioned at the end of the file. Otherwise, returns 0.

### Keywords

None.

### Discussion

---

**OpenVMS USERS** Under OpenVMS, the EOF function has the following limitations:

---

- It does not work with files accessed via DECNET.
- It is meaningless when used with files having an indexed organization structure.

In such cases, we recommend using the ON\_IOERROR procedure to handle end-of-file.

### Example

In this example, a file of test data is created. That file is then read and printed until the end-of-file condition is detected by EOF.

```
OPENW, unit, 'eoffile.dat', /Get_Lun
      ; Open the file eoffile.dat for writing.
```

```

PRINTF, unit, 'This is'
PRINTF, unit, 'some sample'
PRINTF, unit, 'data.'
    ; Write some text to the file.

FREE_LUN, unit
    ; Close the file and free the associated unit number.

OPENR, unit, 'eoffile.dat', /Get_Lun
    ; Open the file eoffile.dat for reading.

a = ' '
    ; Define a string variable.

WHILE NOT eof(unit) DO BEGIN &$
    READF, unit, a &$
    PRINT, a &$
    ; Read data and print it until end-of-file.

ENDWHILE

This is
some sample
data.

FREE_LUN, unit

```

## See Also

[ON\\_IOERROR](#), [RETALL](#), [RETURN](#)

For information on opening files and choosing LUNs, see .

## ***ERASE Procedure***

Erases the display surface of the currently active window.

### **Usage**

ERASE [, *background\_color*]

### **Input Parameters**

*background\_color* — (optional) The background color index.

---

**NOTE** Not all devices support this parameter. Workstations and display terminals, such as X workstations and Tektronix terminals, generally do, while some hard-copy devices, such as HPGL plotters, do not.

---

## Keywords

**Channel**— The destination channel index or mask for the operation. Use only with devices with multiple display channels. If *Channel* is omitted, the system variable !P.Channel is used.

**Color**— The background color index. If specified (and the parameter *background\_color* is not specified), *Color* overrides the value of the system variable !P.Background.

## Discussion

ERASE is a low-level graphics routine. It resets the display surface to the default background color (normally 0), which is indexed from the current color translation tables by the system variable !P.Background. You can override the default by specifying *background\_color*.

ERASE affects the current window only; to switch windows, use the WINDOW command.

A side effect of ERASE is that the device is reset to alphanumeric mode if it has such a mode (e.g., Tektronix terminals).

## Example 1

```
ERASE
    ; Erase the display surface for the current window and use the value
    ; in !P.Background to set the background color.
```

## Example 2

```
COLOR_PALETTE
    ; Create a color palette so the color table can be easily viewed.

LOADCT, 2
    ; Load in color table 2, GRN-RED-BLU-WHT, as it has distinctive colors.

WINDOW, 1
    ; Create window 1.

ERASE
```

```

        ; Erase it using !P.Background.
WINDOW, 2
        ; Create window 2.
ERASE, 22
        ; Erase it, setting the background color to 22 (lime green).
WINDOW, 1
        ; Switch back to window 1.
ERASE, 64
        ; Reset the background color to 64 (bright red).
WINDOW, 2
        ; Switch back to window 2.
!P.Background=180
        ; Explicitly set the background color to 180 (lavender).
ERASE
        ; Set the background color based on !P.Background.

```

## See Also

[!P.Background](#), [WDELETE](#), [WINDOW](#)

---

## ***ERODE*** Function

Implements the morphologic erosion operator for shape processing.

### Usage

*result* = ERODE(*image*, *structure* [, *x*<sub>0</sub>, *y*<sub>0</sub>])

### Input Parameters

*image* — The array to be eroded.

*structure* — The structuring element. May be a one- or two-dimensional array. Elements are interpreted as binary (values are either zero or nonzero), unless the *Gray* keyword is used.

*x*<sub>0</sub> — (optional) The *x*-coordinates of *structure*'s origin.

*y*<sub>0</sub> — (optional) The *y*-coordinates of *structure*'s origin.

## Returned Value

*result* — The eroded image.

## Keywords

*Gray* — A flag which, if present, indicates that gray scale, rather than binary erosion, is to be used.

*Values* — An array providing the values of the structuring element. Must have the same dimensions and number of elements as *structure*.

## Discussion

If *image* is not of the byte type, PV-WAVE makes a temporary byte copy of *image* before using it for the processing.

The optional parameters  $x_0$  and  $y_0$  specify the row and column coordinates of the structuring element's origin. If omitted, the origin is set to the center,  $(\lfloor N_x / 2 \rfloor, \lfloor N_y / 2 \rfloor)$ , where  $N_x$  and  $N_y$  are the dimensions of the structuring element array. However, the origin need not be within the structuring element.

Nonzero elements of the *structure* parameter determine the shape of the structuring element (neighborhood).

If the *Values* keyword is not used, all elements of the structuring element are 0, yielding the neighborhood minimum operator.

You can choose whether you want to use gray scale or binary erosion:

- If you select binary erosion type, the image is considered to be a binary image with all nonzero pixels considered as 1. (You will automatically select binary erosion if you don't use either the *Gray* or *Values* keyword.)
- If you select gray scale erosion type, each pixel of the result is the minimum of the difference of the corresponding elements of *Values* overlaid with *image*. (You will automatically select gray scale erosion if you use either the *Gray* or *Values* keyword.)

## Background Information

The ERODE function implements the morphologic erosion operator on binary and gray scale images and vectors. Mathematical morphology provides an approach to the processing of digital images on the basis of shape. This approach is summarized in the description of the DILATE function. Erosion is the complement (dual) of dilation; it does to the background what dilation does to the foreground.

Briefly, ERODE returns the erosion of *image* by the structuring element, *structure*. This operation is also commonly known as contracting or reducing. It can be used to remove islands smaller than the structuring element.

The result is an image that contains items that now contract away from each other. Features that either slightly touch or are connected by narrow areas may disconnect, becoming separate, smaller objects. Any holes or gaps in or between features become larger as the features in the image shrink away from each other. Sharp-edged items and harsh angles typically become dull as they are worn away; however, in some cases areas that were dull may become somewhat sharper as a feature erodes away.

---

**TIP** Erosion can be used to change the morphological structure of objects or features in an image to see what would happen if they were to actually shrink over time.

---

Used with gray scale images, which are always converted to a byte type, the ERODE function is accomplished by taking the minimum of a set of differences. It may be conveniently used to implement the neighborhood minimum operator, with the shape of the neighborhood given by the structuring element.

Used with binary images, the origin of the structuring element is moved to each pixel of the image. If each nonzero element of the structuring element is contained in the image, the output pixel is set to one.

Letting  $A \ominus B$  represent the erosion of an image  $A$  by structuring element  $B$ , erosion may be defined as:

$$C = A \ominus B = \bigcap_{b \in B} (A)_{-b}$$

where  $(A)_{-b}$  represents the translation of  $A$  by  $b$ . The structuring element  $B$  may be visualized as a probe which slides across the image  $A$ , testing the spatial nature of  $A$  at each point. Where  $B$  translated by  $i, j$  can be contained in  $A$  (by placing the origin of  $B$  at  $i, j$ ), then  $A_{i,j}$  belongs to the erosion of  $A$  by  $B$ .

### Example 1

In this example, the origin of the structuring element is at (0, 0):

0100	0000
0100	0000

```
1110 0 11 =1100
1000    0000
0000    0000
```

## Example 2

This example demonstrates what an aerial image looks like before and after applying the ERODE function three different times. For this example, the following parameters were used each time:

```
img = ERODE(aerial_img, struct, /Gray)
```

where `struct` has a value of `[1 0 1]`.

Because the ERODE function was applied to the image three times, the “blurring” is more pronounced than it would have been with only one erosion.



**Figure 2-20** The ERODE function has been used to “wear away” the visual elements of this 512-by-512 aerial image.

## See Also

[DILATE](#)

---

## **ERRORF Function**

Calculates the standard error function of the input variable.

### **Usage**

*result* = ERRORF(*x*)

### **Input Parameters**

*x* — The expression for which the error function will be evaluated.

### **Returned Value**

*result* — The standard error function of *x*. It is of floating-point data type, and has the same dimensions as *x*.

### **Keywords**

None.

### **Discussion**

The standard error function is central to many calculations in statistics. The ERRORF function can be used in a variety of applications; one example is to solve diffusion equations in heat transfer problems.

The error function is a special case of the incomplete gamma function. ERRORF is defined as:

$$\left(\frac{2}{\sqrt{\pi}}\right) \int_0^x e^{-t^2} dt$$

ERRORF has the following limiting values and symmetries:

$$\text{erf}(0) = 0$$

$$\text{erf}(\infty) = 1$$

$$\text{erf}(-x) = -\text{erf}(x)$$

It is related to the incomplete gamma function by:

$$\operatorname{erf}(x) = \Gamma(1/2, x^2)$$

where  $x \geq 0$ .

## See Also

[GAMMA](#), [GAUSSINT](#)

The method used to determine the error function of complex operands is taken from: W. Gautschi, "Efficient computation of the complex error function," *Siam Journal of Numerical Analysis*, Volume 7, page 187, 1970.

---

## ERRPLOT Procedure

Standard Library procedure that overplots error bars over a previously-drawn plot.

### Usage

ERRPLOT [, *points*], *low*, *high*

### Input Parameters

*points* — (optional) A vector containing the independent or abscissae values of the function. If *points* is omitted, the abscissae values are taken to be unit distances along the  $x$ -axis, beginning with 0.

*low* — A vector containing the lower bounds of the error bars. The value of *low*( $i$ ) is equal to the data value at  $i$  minus the lower error bound.

*high* — A vector containing the upper bounds of the error bars. The value of *high*( $i$ ) is equal to the data value at  $i$  plus the upper error bound.

### Keywords

*Width* — The width of the error bars. If omitted, the width is set to one percent of the plot width.

### Discussion

Error bars are drawn for each element, extending from *low* to *high*.

## Example

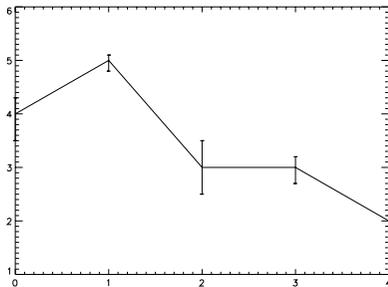
Assume the vector `y` contains the data values to be plotted, and that `err` is the symmetrical error estimate. The commands to plot the data and overplot the error bars are:

```
y = [4.0, 5.0, 3.0, 3.0, 2.0]
err = 0.2
PLOT, y, YRange=[1, 6]
ERRPLOT, y-err, y+err
```

If the error estimates are asymmetrical, they should be placed in the vectors *low* and *high*. For example:

```
low = [3.5, 4.8, 2.5, 2.7, 1.9]
high = [4.3, 5.1, 3.5, 3.2, 2.1]
PLOT, y, YRange=[1, 6]
ERRPLOT, low, high
```

This produces the following plot:

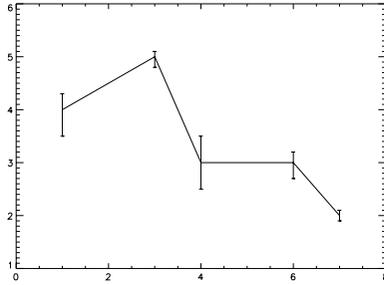


**Figure 2-21** In this example, asymmetrical error estimates have been constrained by using ERRPLOT's *low* and *high* parameters.

To plot error bars versus a vector containing specific points along the X axis, use the following commands:

```
points = [1.0, 3.0, 4.0, 6.0, 7.0]
PLOT, points, y, YRange=[1, 6]
ERRPLOT, points, low, high
```

This produces the plot shown below:



**Figure 2-22** In this example, error bars have been plotted over a vector containing specific points along the *x*-axis.

## See Also

[OPLOT](#), [OPLOTERR](#), [PLOT](#), [PLOTERR](#)

## ***EUCLIDEAN Function***

Standard Library function that transforms the Euclidean metric for a Jacobian  $j =$  Jacobian ( $f$ )

### Usage

$e = \text{euclidean}(j)$

### Input Parameters

$j$  — A Jacobian defined by an  $n$ -element list of  $m$ -element lists of  $m$ -dimensional arrays of dimensions  $d$ .

### Returned Value

$e$  — The Euclidean metric under a transformation with Jacobian  $j$ : an  $m$ -element list of  $m$ -element lists of  $m$ -dimensional arrays.  $(e(p))(q)$  is the  $m$ -dimensional array (of dimensions  $d$ ) that represents the  $(p, q)$  component of the metric.

### Keywords

None.

## Example

See `wave/lib/user/examples/euclidean_ex.pro`.

## See Also

[CURVATURES](#), [JACOBIAN](#), [NORMALS](#)

---

## **EXEC\_ON\_SELECT Procedure (UNIX)**

Registers callback procedures on input for a vector of logical unit numbers (LUNs).

### **Usage**

`EXEC_ON_SELECT`, *luns*, *commands*

### **Input Parameters**

*luns* — Vector of logical unit numbers.

*commands* — Vector of procedure names. It must have the same number of entries as the vector *luns*.

### **Keywords**

*Widget* — If present and nonzero, registers LUNs and commands with the WAVE Widgets or Widget Toolbox event loop (WwLoop or WtLoop).

*Just\_reg* — If present and nonzero, registers the unit numbers and callback procedures. Do not wait for any input. This is useful when this procedure is used with widgets.

### **Description**

This procedure checks for input on all the logical unit numbers specified in the *luns* vector. When there is input available on *luns* (*k*), the *commands* (*k*) procedure is called with *luns* (*k*) as its (only) argument. This procedure never returns; it just keeps handling callbacks when input is available.

When used with the WAVE Widgets applications (i.e., when the *Widget* and *Just\_reg* keywords are used), the callback procedures must have the following parameters:

top, data, nparams, id, lun, source

Refer to Example 2 for more information.

### **Example 1**

In this example, three servers are started and their output handled using EXEC\_ON\_SELECT. For simplicity, the servers are all the same program, EX1, with a different command line argument. The servers occasionally output a four-byte integer. This input is handled by the callback procedures SERVER1, SERVER2, SERVER3. The server is the following C program:

```
#include <stdio.h>
#include <string.h>
main(int argc, char *argv[])
{
    int tag = atoi(argv[1]);
    for (; ;) {
        sleep(5);
        write(1, &tag, sizeof(tag));
    }
}
```

The following are the PV-WAVE procedures that use the above server:

```
PRO SERVER1, lun
    code = 0L
    READU, lun, code
    PRINT, 'SERVER1', code
END

PRO SERVER2, lun
    code = 0L
    READU, lun, code
    PRINT, 'SERVER2', code
END

PRO SERVER3, lun
    code = 0L
    READU, lun, code
    PRINT, 'SERVER3', code
END

PRO EX1
    ; Start servers.
    SPAWN, 'EX1 1', Unit = lun1, /Sh
```

```

    SPAWN, 'EX1 2', Unit = lun2, /Sh
    SPAWN, 'EX1 3', Unit = lun3, /Sh
        ; Handle servers.
    EXEC_ON_SELECT, [lun1, lun2, lun3], $
        ['SERVER1', 'SERVER2', 'SERVER3']
END

```

## **Example 2**

This example is an extension of the previous example. Again, three servers are started and their output handled using EXEC\_ON\_SELECT. The input is handled by the callback procedures SERVER1, SERVER2, SERVER3. Now, a widget menu also is displayed and serviced. The EXEC\_ON\_SELECT procedure registers the LUNs and callback procedures with the WAVE Widgets event loop (WwLoop) and returns values using the *Widget* and *Just\_reg* keywords.

When there is input, WwLoop calls the appropriate callback routine to handle the input and returns to waiting for more input.

```

PRO SERVER1, top, data, nparams, id, lun, $
    source
    code = 0L
    READU, lun, code
    PRINT, 'SERVER1', code, lun
END

PRO SERVER2, top, data, nparams, id, lun, $
    source
    code = 0L
    READU, lun, code
    PRINT, 'SERVER2', code, lun
END

PRO SERVER3, top, data, nparams, id, lun, source
    code = 0L
    READU, lun, code
    PRINT, 'SERVER3', code, lun
END

PRO MenuCB, wid, index
    ; Create a menu.
    PRINT, 'Menu Item', index, ' selected.'
    value = WwGetValue(wid)
    PRINT, value
END

```

```

PRO Ex2
    SPAWN, 'EX1 1', unit = lun1, /sh
    SPAWN, 'EX1 2', unit = lun2, /sh
    SPAWN, 'EX1 3', unit = lun3, /sh

top = WwInit('ex2','Test',layout,/Vertical)

button = WwButtonBox(layout, ['Fonts', $
    'Size', 'Icons'], 'MenuCB')

status = WwSetValue(top, /Display)
EXEC_ON_SELECT, [lun1, lun2, lun3], $
    ['SERVER1', 'SERVER2', 'SERVER3'], /Widget, $
    /Just_reg

WwLoop

CLOSE, lun1, lun2, lun3

END

```

## See Also

[ADD\\_EXEC\\_ON\\_SELECT](#), [DROP\\_EXEC\\_ON\\_SELECT](#),  
[SELECT\\_READ\\_LUN](#)

---

## **EXECUTE Function**

Compiles and executes one or more PV-WAVE statements contained in a string at run-time.

### **Usage**

*result* = EXECUTE(*string*)

### **Input Parameters**

*string* — A string containing the PV-WAVE command(s) to be compiled and executed. Cannot contain a command that starts with either a dollar (\$), period (.), or at (@) character; such commands must be entered at the PV-WAVE prompt.

## Returned Value

**result** — Returns 1 if the string was successfully compiled and executed; returns 0 if an error occurs during either phase.

## Keywords

None.

## Discussion

When the EXECUTE function is used inside a procedure or function, the compiler directive `..LOCALS` can be used to allocate memory for local variables created at compile time (see *Using the ..LOCALS Compiler Directive* in Chapter 9 of the *PV-WAVE Programmer's Guide*).

## Example

This example creates a procedure, `TABLE`, that prints a table giving the results of evaluating a user-defined function of two variables at the values in two vectors. A user-defined printing procedure is used to actually display the table of values.

Function EXECUTE is used to invoke both the user-defined function of two variables and the user-defined printing procedure. The name of the function is passed to `TABLE` using keyword `Func`, and the name of the printing procedure is passed using keyword `Prt_Pro`. The following is a listing of `TABLE`:

```
PRO TABLE, x, y, Func = func, Prt_Pro = pp
tab = FLTARR(3, n_elements(x))
tab(0, *) = x
tab(1, *) = y

val = EXECUTE('tab(2, *) = ' $
             + func + '(tab(0, *), tab(1, *))')
             ; Use EXECUTE to invoke the function in the string
             ; variable func. Assign the result to column 2 of tab.

IF val EQ 1 THEN BEGIN
    val = EXECUTE(pp + ', tab')
             ; Use EXECUTE to invoke the procedure in the string variable
             ; pp. This procedure prints the table.

    IF val EQ 0 THEN BEGIN
        PRINT, "***Error in execution of " + $
              "printing procedure! ***"
    ENDIF
ENDIF
```

```

ENDIF ELSE BEGIN
    PRINT, "*** Error in execution of " + "function! ***"
ENDELSE
END

```

If this procedure is placed in the file `table.pro` in your working directory, it will be compiled automatically when it is invoked. Note that the string concatenation operator, along with several string literals, are used to construct the statements to execute using EXECUTE.

The user-defined function requires two arguments, which are the values of the independent variables of the function. The function should return the result of the function evaluation. The user-defined printing procedure requires one argument, which is the two-dimensional table to be printed. The following commands can be entered at the interactive prompt to create and compile a function of two variables:

```

.RUN
- FUNCTION fcn, x, y
- RETURN, x^2 - y^2
- END

```

The following procedure prints the table:

```

PRO prt, arr
PRINT, Format = $
    '(4x, "x", 13x, "y", 10x, "func(x, y)">'
PRINT, Format = '(39("-"))'
PRINT, Format = '(2(f9.4, 5x), f10.4)', arr
END

```

If this procedure is placed in the file `prt.pro` in your working directory, it will be compiled automatically when it is invoked. The following commands can be used to create the vectors of values at which to evaluate `fcn` and to invoke TABLE:

```

x = [1, 2, 3, 4, 5]
y = REVERSE(x)
TABLE, x, y, Func = 'fcn', Prt_pro = 'prt'

```

x	y	func(x, y)
1.0000	5.0000	-24.0000
2.0000	4.0000	-12.0000
3.0000	3.0000	0.0000
4.0000	2.0000	12.0000
5.0000	1.0000	24.0000

## See Also

For more information, see [in](#)

---

## **EXIT Procedure**

Exits PV-WAVE and returns you to the operating system.

### **Usage**

EXIT

### **Parameters**

None.

### **Keywords**

None.

### **Discussion**

All buffers are flushed and open files are closed. The values of all variables that were not saved are lost.

## See Also

[QUIT](#)

---

## **EXP Function**

Raises  $e$  to the power of the value of the input variable.

### **Usage**

*result* = EXP(*x*)

### **Input Parameters**

*x* — The value to be evaluated.

### **Returned Value**

*result* — The natural exponential function of *x*.

### **Keywords**

None.

### **Discussion**

EXP is defined as:

$$y = e^x$$

If *x* is of double-precision floating-point or complex data type, EXP yields results of the same type. All other types yield a single-precision floating-point result.

EXP handles complex values in the following way:

$$\exp(x) = \text{complex}(e^r \cos(i), e^r \sin(i))$$

where *r* is the real part of *x*, and *i* is the imaginary part of *x*. If *x* is an array, the result has the same dimensions as *x*, with each element containing the result for the corresponding element of *x*.

### **Example**

```
exp_of_1 = EXP(1)
PRINT, exp_of_1
      2.71828
exp_of_0 = EXP(0)
```

```
PRINT, exp_of_0
      1.00000
exp_of_10 = EXP(10)
PRINT, exp_of_10
      22026.5
```

## See Also

For a list of other transcendental functions, see [Transcendental Mathematical Functions](#) on page 32.

---

## EXPAND Function

Standard Library function that expands an array into higher dimensions.

### Usage

```
result = EXPAND(a, d, i)
```

### Input Parameters

*a* — An array of *n* dimensions.

*d* — A vector specifying the dimensions for the new array.

*i* — A monotonically increasing vector of *n* indices into *d* specifying which of the new dimensions correspond to old dimensions:

*d*(*i*) must equal `SIZE(a, /Dimensions)`.

### Returned Value

*result* — An array of dimensions *d*, the expansion of the input array.

### Keywords

None.

### Examples

```
pm, EXPAND( [0,1], [2,3], [0] )
```

```
pm, EXPAND( [0,1], [3,2], [1] )
```

```
pm, EXPAND( [[0,1,2],[3,4,5]], [5,3,2], [1,2] )
pm, EXPAND( [[0,1,2],[3,4,5]], [3,5,2], [0,2] )
pm, EXPAND( [[0,1,2],[3,4,5]], [3,2,5], [0,1] )
```

## See Also

[REBIN](#), [REPLV](#)

---

## ***EXPON Function***

Standard Library function that performs general exponentiation.

### **Usage**

*result* = EXPON(*a*, *b*)

### **Input Parameters**

*a* — An array (scalar) of any numerical data type.

*b* — An array (scalar) of any numerical data type.

### **Returned Value**

*result* — A double complex array (scalar) containing the values  $a^b$ .

### **Keywords**

None.

### **Example**

```
pm, EXPON( [complex(0,1), -1], [complex(2,3), 0.5] )
```

---

## **EXTREMA Function**

Standard Library function that finds the local extrema in an array.

### **Usage**

```
result = EXTREMA(array)
```

### **Input Parameters**

*array* — The array for which the local extrema will be found.

### **Returned Value**

*result* — A list containing two vectors of indices into *array*. *result*(0) contains the local minima and *result*(1) contains the local maxima

### **Keywords**

None.

### **Examples**

```
e = EXTREMA( [0,1,2,2,2,3,2,1,3] ) & pm, e(0), ' ' & pm, e(1)
a = bytscl( randomu(s,5,5), top=9 ) & pm, a
e = EXTREMA( a ) & pm, e(0), ' ' & pm, e(1)
```

### **See Also**

[MAX](#), [MIN](#)

---

## ***FACTOR Function***

Standard Library function that returns the prime factorization of an integer greater than 1.

### **Usage**

*result* = FACTOR(*i*)

### **Input Parameters**

*i* — An integer greater than 1.

### **Returned Value**

*result* — Sorted vector of longs containing the prime factorization of *i*.

### **Keywords**

*a* — If set, *result* contains all factors instead of just prime factors.

### **Examples**

```
pm, FACTOR(12, /a)
```

```
pm, FACTOR(12)
```

### **See Also**

[GCD](#), [LCM](#), [PRIME](#)

---

## **FAST\_GRID2 Function**

Returns a gridded, 1D array containing  $y$  values, given random  $x$ ,  $y$  coordinates (this function works best with dense data points).

### **Usage**

*result* = FAST\_GRID2(*points*, *grid\_x*)

### **Input Parameters**

*points* — A  $(2, n)$  array containing the random  $x$ ,  $y$  points to be gridded.

*grid\_x* — The  $x$  dimension of the grid. The  $x$  values are scaled to fit this dimension.

### **Returned Value**

*result* — A gridded 1D array containing  $y$  values.

### **Keywords**

*Iter* — The number of iterations on the smooth function. The execution time increases linearly with the number of iterations. The default is 3, but any non-negative integer (including zero) may be specified.

*Nghbr* — The size of the neighborhood to smooth. If not supplied, the neighborhood size is calculated from the distribution of the points. The amount of memory required increases by the square of the neighborhood size.

*No\_Avg* — Normally, if multiple data points fall in the same cell in the gridded array, then the value of that cell is the average value of all the data points that fall in that cell.

If the *No\_Avg* keyword is present and nonzero, however, the value of the cell in the gridded array is the total of all the points that fall in that cell.

*XMax* — The  $x$ -coordinate of the right edge of the grid. If omitted, maps the maximum  $x$  value found in the *points*(0, \*) array to the right edge of the grid.

*XMin* — The  $x$ -coordinate of the left edge of the grid. If omitted, maps the minimum  $x$  value found in the *points*(0, \*) array to the left edge of the grid.

## Discussion

FAST\_GRID2 uses a neighborhood smoothing technique to interpolate missing data values for 2D gridding. The gridded array returned by FAST\_GRID2 is suitable for use with the PLOT function.

FAST\_GRID2 is similar to GRID\_2D. FAST\_GRID2, however, works best with dense data points (more than 1000 points to be gridded) and is considerably faster, but slightly less accurate, than GRID\_2D. (GRID\_2D works best with sparse data points and is stable when extrapolating into large void areas.)

---

**TIP** For best results, use a small neighborhood (such as 3) and a large number of iterations (more than 16).

---

## Examples

```
PRO f_gridemo2
    ; This program shows 2D gridding with dense data points.

points = INTARR(2, 10)
points(*, 0) = [1,2]
points(*, 1) = [2,3]
points(*, 2) = [5,5]
points(*, 3) = [8,0]
points(*, 4) = [9,6]
points(*, 5) = [4,9]
points(*, 6) = [7,15]
points(*, 7) = [6,-5]
points(*, 8) = [0,3]
points(*, 9) = [0,-1]
    ; Set up the data points.
WINDOW, 0, Colors=128
LOADCT, 4
T3D, /Reset
    ; Set up the viewing window and load the color table.

!Y.Range = [MIN(points), MAX(points)]
    ; Set the y-axis range for plotting.

yval = FAST_GRID2(points, 256, Iter=0)
PLOT, yval, Color=60
yval = FAST_GRID2(points, 256, Iter=150, Nghbr=3)
OPLOT, yval, Color=80
yval = FAST_GRID2(points, 256, Nghbr=77)
```

```

OPLOT, yval, Color=100
yval = FAST_GRID2(points, 256)
OPLOT, yval, Color=120
      ; Grid and plot the points using different values for the
      ; neighborhood and number of iterations.

!Y.Range = [0.0, 0.0]
      ; Reset the y-axis range to the default value.

END

```

## See Also

[FAST\\_GRID3](#), [FAST\\_GRID4](#), [GRID\\_2D](#), [GRID\\_3D](#), [GRID\\_4D](#),  
[GRIDN](#), [GRID\\_SPHERE](#)

---

**UNIX and OpenVMS USERS** For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---



---

## FAST\_GRID3 Function

Returns a gridded, 2D array containing  $z$  values, given random  $x$ -,  $y$ -,  $z$ -coordinates (this function works best with dense data points).

### Usage

*result* = FAST\_GRID3(*points*, *grid\_x*, *grid\_y*)

### Input Parameters

*points* — A  $(3, n)$  array containing the random  $x$ ,  $y$ ,  $z$  points to be gridded.

*grid\_x* — The  $x$  dimension of the grid. The  $x$  values are scaled to fit this dimension.

*grid\_y* — The  $y$  dimension of the grid. The  $y$  values are scaled to fit this dimension.

### Returned Value

*result* — A gridded, 2D array containing  $z$  values.

## Keywords

***Iter*** — The number of iterations on the smooth function. The execution time increases linearly with the number of iterations. The default is 3, but any non-negative integer (including zero) may be specified.

***Nghbr*** — The size of the neighborhood to smooth. If not supplied, the neighborhood size is calculated from the distribution of the points. The amount of memory required increases by the square of the neighborhood size.

***No\_Avg*** — Normally, if multiple data points fall in the same cell in the gridded array, then the value of that cell is the average value of all the data points that fall in that cell.

If the *No\_Avg* keyword is present and nonzero, however, the value of the cell in the gridded array is the total of all the points that fall in that cell.

***XMin*** — The *x*-coordinate of the left edge of the grid. If omitted, maps the minimum *x* value found in the *points*(0, \*) array to the left edge of the grid.

***XMax*** — The *x*-coordinate of the right edge of the grid. If omitted, maps the maximum *x* value found in the *points*(0, \*) array to the right edge of the grid.

***YMin*** — The *y*-coordinate of the bottom edge of the grid. If omitted, maps the minimum *y* value found in the *points*(1, \*) array to the bottom edge of the grid.

***YMax*** — The *y*-coordinate of the top edge of the grid. If omitted, maps the maximum *y* value found in the *points*(1, \*) array to the top edge of the grid.

## Discussion

FAST\_GRID3 uses a neighborhood smoothing technique to interpolate missing data values for 3D gridding. The gridded array returned by FAST\_GRID3 is suitable for use with the SURFACE, TV, AND CONTOUR procedures.

FAST\_GRID3 is similar to GRID\_3D. FAST\_GRID3, however, works best with dense data points (more than 1000 points to be gridded) and is considerably faster, but slightly less accurate, than GRID\_3D. (GRID\_3D works best with sparse data points and is stable when extrapolating into large void areas.)

---

**TIP** For best results, use a small neighborhood (such as 3) and a large number of iterations (more than 16).

---

## Examples

```
PRO f_gridemo3
```

```
    ; This program shows 3D gridding with dense data points.
```

```
points = RANDOMU(s, 3, 1000)
points(0, *) = points(0, *) * 10.0
points(1, *) = points(1, *) * 10.0
points(*, 0) = [1.7, 1.6, 2.9]
points(*, 1) = [1.4, 1.2, 3.7]
points(*, 2) = [9.8, 9.2, 5.5]
points(*, 3) = [9.8, 8.4, 0.1]
points(*, 4) = [4.8, 9.9, 6.3]
points(*, 5) = [0.2, 9.0, 9.0]
points(*, 6) = [3.1, 7.2, 15.2]
points(*, 7) = [5.6, 6.0, -5.9]
points(*, 8) = [0.3, 0.5, 3.3]
points(*, 9) = [9.7, 0.7, -1.6]
```

```
    ; Generate random data points.
```

```
zval = FAST_GRID3(points, 48, 32)
```

```
    ; Grid the resulting data points.
```

```
WINDOW, 0, Colors=128
```

```
SURFR
```

```
SURFACE, zval, Bottom=90, Ax=30.0, Az=30.0, /T3d
```

```
    ; Display the gridded data as a surface in the specified window.
```

```
END
```

## See Also

[FAST\\_GRID2](#), [FAST\\_GRID4](#), [GRID\\_2D](#), [GRID\\_3D](#),  
[GRID\\_4D](#), [GRIDN](#), [GRID\\_SPHERE](#)

---

**UNIX and OpenVMS USERS** For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---

---

## **FAST\_GRID4 Function**

Returns a gridded, 3D array containing intensity values, given random 4D coordinates (this function works best with dense data points).

### **Usage**

```
result = FAST_GRID4(points, grid_x, grid_y, grid_z)
```

### **Input Parameters**

*points* — A (4, *n*) array containing the random 4D points to be gridded. Typically, *points*(0, \*) contains the *x* values, *points*(1, \*) contains the *y* values, *points*(2, \*) contains the *z* values, and *points*(3, \*) contains the intensity values. (You may, however, choose to put other variables in these four vectors.)

*grid\_x* — The *x* dimension of the grid. The *x* values are scaled to fit this dimension.

*grid\_y* — The *y* dimension of the grid. The *y* values are scaled to fit this dimension.

*grid\_z* — The *z* dimension of the grid. The *z* values are scaled to fit this dimension.

### **Returned Value**

*result* — A gridded, 3D array containing intensity values.

### **Keywords**

*Iter* — The number of iterations on the smooth function. The execution time increases linearly with the number of iterations. The default is 3, but any non-negative integer (including zero) may be specified.

*Nghbr* — The size of the neighborhood to smooth. If not supplied, the neighborhood size is calculated from the distribution of the points. The amount of memory required increases by the square of the neighborhood size.

*No\_Avg* — Normally, if multiple data points fall in the same cell in the gridded array, then the value of that cell is the average value of all the data points that fall in that cell.

If the *No\_Avg* keyword is present and nonzero, however, the value of the cell in the gridded array is the total of all the points that fall in that cell.

***XMax*** — The  $x$ -coordinate of the right edge of the grid. If omitted, maps the maximum  $x$  value found in the *points*(0, \*) array to the right edge of the grid.

***XMin*** — The  $x$ -coordinate of the left edge of the grid. If omitted, maps the minimum  $x$  value found in the *points*(0, \*) array to the left edge of the grid.

***YMax*** — The  $y$ -coordinate of the top edge of the grid. If omitted, maps the maximum  $y$  value found in the *points*(1, \*) array to the top edge of the grid.

***YMin*** — The  $y$ -coordinate of the bottom edge of the grid. If omitted, maps the minimum  $y$  value found in the *points*(1, \*) array to the bottom edge of the grid.

***ZMax*** — The  $z$ -coordinate of the front edge of the grid. If omitted, maps the maximum  $z$  value found in the *points*(2, \*) array to the front edge of the grid.

***ZMin*** — The  $z$ -coordinate of the back edge of the grid. If omitted, maps the minimum  $z$  value found in the *points*(2, \*) array to the back edge of the grid.

## Discussion

FAST\_GRID4 uses a neighborhood smoothing technique to interpolate missing data values for 4D gridding. The gridded array returned by FAST\_GRID4 is suitable for use with the SHADE\_VOLUME and VOL\_REND functions.

FAST\_GRID4 is similar to GRID\_4D. FAST\_GRID4, however, works best with dense data points (more than 1000 points to be gridded) and is considerably faster, but slightly less accurate, than GRID\_4D. (GRID\_4D works best with sparse data points and is stable when extrapolating into large void areas.)

---

**TIP** For best results, use a small neighborhood (such as 3) and a large number of iterations (more than 16).

---

## Examples

See the Examples section in the description of the [CENTER\\_VIEW](#) routine.

## See Also

[FAST\\_GRID3](#), [GRID\\_2D](#), [GRID\\_3D](#), [GRID\\_4D](#), [GRIDN](#), [GRID\\_SPHERE](#)

---

**UNIX and OpenVMS USERS** For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---

---

## FFT Function

Returns the fast Fourier transform (FFT) for the input variable.

### Usage

*result* = FFT(*array*, *direction*)

### Input Parameters

***array*** — The array for which the FFT or the reverse FFT is computed. The size of each dimension may be any positive integer value.

***direction*** — A signed scalar value that determines the direction of the transform, between the time (or spatial) domain and the frequency domain.

### Returned Value

***result*** — The fast Fourier transform of *array*. The Cooley-Tukey fast Fourier transform algorithm is used for calculating the FFT.

### Keywords

***Intleave*** — A scalar string indicating the type of interleaving of 2D input signals containing signal-interleaved signals; and 3D input arrays containing image-interleaved images, or a volume. Valid strings and the corresponding interleaving methods are:

'signal' — The 2D input image array arrangement is  $(x, p)$  for  $p$  signal-interleaved signals of length  $x$ .

'image' — The 3D image array arrangement is  $(x, y, p)$  for  $p$  image-interleaved images of  $x$ -by- $y$ .

'volume' — The input image array is treated as a single entity.

### Discussion

The FFT function supports input arrays composed of multiple images (multi-layer band interleaved images) as well as input arrays composed of multiple signals. The *Intleave* keyword is used to specify whether the input array is a multi-signal or multi-image array. When the *Intleave* keyword is used to indicate multiple signals or images in this way, each signal or image in the array is operated on separately and an array of the individual results is returned.

The Fourier transform of a scaled-time function is defined by:

$$F(w) = F(f(t)) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t} dt$$

where  $w$  relates to the frequency domain, and  $t$  relates to the time (space) domain.

The data type of *array* is converted to complex, with the real part described by *array* and the imaginary part set to 0, unless it is already complex. The output array will have the same number and size of dimensions as *array*.

---

**TIP** For more efficient transforms, choose dimensions for *array* that are a power of 2.

---

The *direction* parameter controls the direction of the transform:

- Set *direction* to a negative value to transform from space to frequency.
- Set *direction* to a positive (or zero) value to go from frequency to space.

A normalization factor of  $1/n$ , where  $n$  is the number of points in *array*, is applied to the transformation when going from space to frequency.

---

**CAUTION** Take care to avoid wrap-around artifacts when filtering and convolving in the frequency domain. In particular, make sure your images are properly windowed and sampled before applying the Fast Fourier Transform, or false and misleading values will result.

---

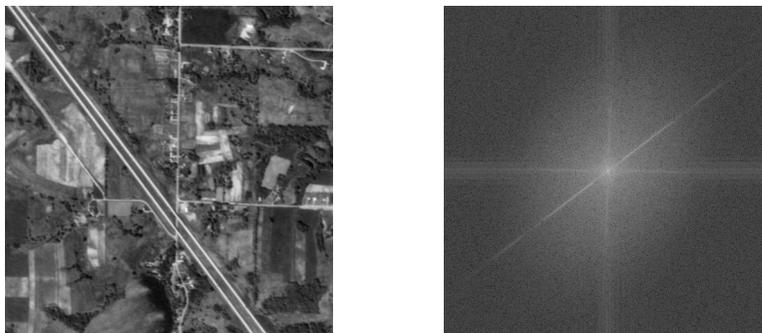
## Example 1

This example shows what an aerial image looks like before and after applying the FFT function, in conjunction with other functions.

The FFT function is used to transform the image into the frequency domain. For the example shown in , the following parameters are used:

```
fft_aerial = FFT(aerial_img, -1)
```

FFT places the frequency component into the first element of the image, which appears in the lower-left corner. However, it is customary to display Fourier spectra of images with the frequency component in the center of the image. This can be done using the SHIFT function to move the origin to the center, and the ABS and ALOG functions to convert the data back into a format that can be displayed:



**Figure 2-23** PV-WAVE makes it easy to generate the Fourier spectrum for any image. Note that the diagonal, vertical, and horizontal lines in the Fourier spectrum correspond to the roads in the original 512-by-512 image, but are perpendicular to them; this is because of the 90-degree phase shift that occurs when moving from the space domain to the frequency domain.

- Use the `SHIFT` function to shift the image so the point with a subscript of (0,0) is in the center (assuming the image is a 512-by-512 image).
- Use the `ALOG` function to return the natural logarithm of each pixel.
- Use the `ABS` function to calculate the magnitude of each complex-valued pixel.

The result of the initial FFT operation (`fft_aerial`) can be run through these other three functions as follows:

```
display = SHIFT(ALOG(ABS(fft_aerial)), 256, 256)
```

The resulting variable, `display`, is the image displayed on the right in .

## Example 2

For an example of an FFT used in windowing, see the description of the [HANNING](#) function.

## See Also

[HANNING](#), [HILBERT](#)

For background information, see the section *Frequency Domain Techniques* in Chapter 6 of the *PV-WAVE User's Guide*.

For details on the Cooley-Tukey Fast Fourier Transform algorithm, see the Special Issue on FFT in *IEEE Audio Transactions*, June 1967.

---

## **FILEPATH Function**

Standard Library function that returns the file path to use to open a file, when given a file name within the PV-WAVE distribution.

Optionally, can also return the file name of the user's terminal and the default location for temporary files for the current operating system.

### **Usage**

*result* = FILEPATH(*filename*)

### **Input Parameters**

*filename* — A string containing the name of a file. Must be in all lowercase. Do not enter any device or directory information.

### **Returned Value**

*result* — The fully qualified file path for *filename*.

### **Keywords**

*Subdirectory* — The name of the subdirectory in the PV-WAVE distribution area in which *filename* is located.

*Terminal* — The file name of the user's terminal.

*Tmp* — The path to the default location for temporary files for the current operating system (*filename* is a temporary or "scratch" file).

### **Discussion**

FILEPATH is used to get path information for a file. It is not a search facility, but simply builds the file path by padding information based on the operating system and keyword information passed into the function in the system variable !Dir.

FILEPATH does not check for the existence of *filename*, but rather only contracts a fully qualified pathname. It does account for operating system dependencies.

This routine is useful when you are writing a procedure that will be used on different platforms that support PV-WAVE and will open files in the PV-WAVE distribution.

## UNIX Examples

```
PRINT, FILEPATH('wvstartup')
    /usr/local/vni/wave/wvstartup
full_name = FILEPATH('errplot', $
    Subdirectory='lib/std')
PRINT, full_name
    /usr/local/vni/wave/lib/std/errplot
PRINT, FILEPATH('dummy',/Terminal)
    /dev/tty
PRINT, FILEPATH('scratch10',/Tmp)
    /tmp/scratch10
```

## VMS Examples

```
PRINT, FILEPATH('wvstartup')
    WAVE_DIR:[000000]wvstartup
full_name = FILEPATH('errplot', $
    Subdirectory='lib.std')
PRINT, full_name
    WAVE_DIR:[lib.std]errplot
PRINT, FILEPATH('dummy',/Terminal)
    SYS$OUTPUT:
PRINT, FILEPATH('scratch10',/Tmp)
    SYS$LOGIN:scratch10
```

## Windows Examples

```
full_name = FILEPATH('errplot', $
    Subdirectory='lib\std')
PRINT, full_name
    d:\vni\wave\lib\std\errplot
PRINT, FILEPATH('scratch10',/Tmp)
    \tmp\scratch10
```

## See Also

[FINDFILE](#)

System Variables: [!Dir](#), [!Path](#)

---

## ***FINDFILE Function***

Returns a string array containing the names of all files matching a specified file description.

### **Usage**

*result* = FINDFILE(*file\_specification*)

### **Input Parameters**

*file\_specification* — A scalar string used to find files. May contain any valid shell wildcard characters. If omitted, all files in the current directory are supplied.

---

**Windows USERS** If a filename or directory name used in the file specification string contains a space, the entire string must be enclosed in quotes (either single or double quotes). For example, to find files in the directory:

'Visual Numerics\wave\xres', you must use the command:

```
files=FINDFILE(' "\Visual Numerics\wave\xres" ' )
```

---

### **Returned Value**

*result* — A string array containing the names of all files matching *file\_specification*. If no files with matching names exist, returns a null scalar string.

### **Keywords**

*Count* — A named variable into which the number of files found is placed. A value of 0 indicates that no files were found.

### **Discussion**

FINDFILE returns all matched filenames in a string array, one file name per array element.

---

**UNIX USERS** Under UNIX, FINDFILE uses the shell specified by the SHELL environment variable (or /bin/sh if SHELL is not defined) to search for any files matching *file\_specification*.

---

---

**OpenVMS USERS** Under OpenVMS, FINDFILE uses the command language interpreter.

---

## Example

This example assumes you have two files, `test.c` and `test_2.c`, in your current directory.

```
x=FINDFILE('*.c', Count=cntr)
PRINT, x
    test.c  test_2.c
PRINT, cntr
    2
```

## See Also

[FILEPATH](#)

---

## ***FINDGEN Function***

Returns a single-precision floating-point array with the specified dimensions.

### Usage

*result* = FINDGEN(*dim*<sub>1</sub>, ..., *dim*<sub>*n*</sub>)

### Input Parameters

*dim*<sub>*i*</sub>— The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### Returned Value

*result* — An initialized single-precision, floating-point array. If the resulting array is treated as a one-dimensional array, then its initialization is given by the following:

$$array(i) = \text{FLOAT}(i), \text{ for } i = 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right)$$

## Keywords

None.

## Discussion

Each element of the array is set to the value of its one-dimensional subscript.

## Example

This example creates a 4-by-2 single-precision, floating-point array.

```
a = FINDGEN(4, 2)
    ; Create single-precision, floating-point array.
INFO, a
  A          FLOAT          = Array(4, 2)
PRINT, a
  0.00000    1.00000    2.00000    3.00000
  4.00000    5.00000    6.00000    7.00000
```

## See Also

[BINDGEN](#), [CINDGEN](#), [DINDGEN](#), [INDGEN](#), [LINDGEN](#), [SINDGEN](#)

---

## FINITE Function

Returns a value indicating if the input variable is finite or not.

## Usage

*result* = FINITE(*x*)

## Input Parameters

*x* — A scalar or array expression of single-precision complex, double-precision complex, single-precision floating point, or double-precision floating point data type.

## Returned Value

*result* — Returns 1 if *x* is finite. Returns 0 if *x* is infinite or not a defined number (NaN). Undefined numbers result from ill-defined operations, such as dividing zero by zero, or taking the logarithm of zero or a negative number.

## Keywords

None.

## Example

```
fmach = MACHINE(/Float)
; Get the single-precision, floating-point machine constants.
a = [fmach.nan, 3.0, fmach.pos_inf, 5.2, $
    fmach.neg_inf]
; Create a five-element vector containing single-precision,
; floating-point NaN, positive infinity, negative infinity, and
; finite values.

b = FINITE(a)
; View result of FINITE.

INFO, b
      B          BYTE          = Array(5)
FOR i = 0, 4 DO PRINT, a(i), b(i)
      NaN          0
      3.00000      1
      Inf          0
      5.20000      1
      -Inf         0
; Print vectors a and b. Note that vector b contains a 0 when
; NaN or infinity occurs in a. Vector b contains a 1 at the indices
; where vector a contains finite values.
```

## See Also

[CHECK\\_MATH](#), [ON\\_ERROR](#), [ON\\_IOERROR](#)

For more details, see Chapter 10, *Programming with PV-WAVE*, in the *PV-WAVE Programmer's Guide*.

---

## **FIX Function**

Converts an expression to integer data type.

Extracts data from an expression and places it in a integer scalar or array.

### **Usage**

*result* = FIX(*expr*)

This form is used to convert data.

*result* = FIX(*expr*, *offset*, [*dim*<sub>1</sub>, ..., *dim*<sub>*n*</sub>])

This form is used to extract data.

### **Input Parameters**

To convert data:

*expr* — The expression to be converted.

To extract data:

*expr* — The expression from which to extract data.

*offset* — The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim*<sub>*i*</sub> — (optional) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### **Returned Value**

For data conversion:

*result* — A copy of *expr* converted to integer data type.

For data extraction:

*result* — If *offset* is used, FIX does not convert *result*, but allows fields of data extracted from *expr* to be treated as integer data. If no dimensions are specified, the result is scalar.

### **Keywords**

None.

## Discussion

---

**CAUTION** If the values of *expr* are within the range of a long integer, but outside the range of the integer data type (−32,768 to +32,767), a misleading result occurs, without an accompanying message. For example, `FIX(66000)` erroneously results in 464.

---

In addition, **PV-WAVE** does not check for overflow during conversion to integer data type. The values in *expr* are simply converted to long integers and the low 16 bits are extracted.

## Examples

`FIX` is used in two ways here. First, `FIX` is used to convert a single-precision, floating-point array to integer. Next, `FIX` is used to extract a subarray from the integer array created in the first step.

```
a = FINDGEN(6) + 0.6
    ; Create a single precision, floating point vector of length 6. Each
    ; element has a value equal to its one-dimensional subscript plus 0.6.
```

```
PRINT, a
0.600000 1.60000 2.60000 3.60000 4.60000
5.60000
```

```
b = FIX(a)
    ; Convert a to type integer.
```

```
INFO, b
      B          INT          = Array(6)
```

```
PRINT, b
      0    1    2    3    4    5
      ; Notice that the floating-point numbers in a were truncated by
      ; FIX.
```

```
c = FIX(b, 4, 2, 2)
    ; Extract the last four elements of b, and place them in a 2-by-2
    ; integer array.
```

```
INFO, c
      C          INT          = Array(2, 2)
```

```
PRINT, c
      2          3
```

## See Also

[BYTE](#), [COMPLEX](#), [DOUBLE](#), [FLOAT](#), [LONG](#), [SMALL\\_INT](#)

For more information on using this function to extract data, see the *PV-WAVE Programmer's Guide*.

## **FLOAT Function**

Converts an expression to single-precision floating-point data type.

Extracts data from an expression and places it in a single-precision floating-point scalar or array.

### Usage

*result* = FLOAT(*expr*)

This form is used to convert data.

*result* = FLOAT(*expr*, *offset*, [*dim*<sub>1</sub>, ..., *dim*<sub>*n*</sub> ])

This form is used to extract data.

### Input Parameters

To convert data:

*expr* — The expression to be converted, or from which to extract data.

To extract data:

*expr* — The expression to be converted, or from which to extract data.

*offset* — The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim*<sub>*i*</sub> — (optional) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### Returned Value

For data conversion:

**result** — A copy of *expr* converted to single-precision floating-point data type.

For data extraction:

**result** — If *offset* is used, `FLOAT` does not convert *result*, but allows fields of data extracted from *expr* to be treated as single-precision floating-point data. If no dimensions are specified, the result is scalar.

## Keywords

None.

## Example

In this example, `FLOAT` is used in two ways. First, `FLOAT` is used to convert an integer array to single precision, floating point. Next, `FLOAT` is used to extract a subarray from the single-precision array created in the first step.

```
a = INDGEN(6)
      ; Create an integer vector of length 6. Each element has a
      ; value equal to its one-dimensional subscript.

PRINT, a
      0      1      2      3      4      5

b = FLOAT(a)
      ; Convert a to single precision, floating point.

INFO, b
      B              FLOAT              = Array(6)

PRINT, b
      0.00000  1.00000  2.00000  3.00000  4.00000
      5.00000

c = FLOAT(b, 8, 2, 2)
      ; Extract the last four elements of b, and place them in a 2-by-2
      ; single-precision, floating-point array.

INFO, c
      C              FLOAT              = Array(2, 2)

PRINT, c
      2.00000      3.00000
      4.00000      5.00000
```

## See Also

[BYTE](#), [COMPLEX](#), [DOUBLE](#), [FIX](#), [LONG](#)

For more information on using this function to extract data, see the *PV-WAVE Programmer's Guide*.

---

## FLTARR Function

Returns a single-precision floating-point vector or array.

### Usage

```
result = FLTARR(dim1, ..., dimn)
```

### Input Parameters

*dim<sub>i</sub>* — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### Returned Value

*result* — A single-precision floating-point vector or array.

### Keywords

*Nozero* — If *Nozero* is nonzero, this zeroing is not performed, thereby causing FLTARR to execute faster.

### Discussion

Normally, FLTARR sets every element of *result* to zero.

### Example

```
PRINT, FLTARR(4)
      0.00000  0.00000  0.00000  0.00000
PRINT, FLTARR(4, /Nozero)
      5.60519e-45  1.98225e-39  2.35149e-38  5.60519e-45
```

## See Also

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [FINDGEN](#), [INTARR](#), [LONARR](#),  
[MAKE\\_ARRAY](#), [REPLICATE](#), [STRARR](#)

---

## ***FLUSH Procedure***

Causes all buffered output on the specified file units to be written.

### **Usage**

FLUSH, *unit*<sub>1</sub>, ..., *unit*<sub>*n*</sub>

### **Input Parameters**

*unit*<sub>*i*</sub> — The file units (logical unit numbers) to flush.

### **Keywords**

None.

### **Discussion**

PV-WAVE uses buffered output for reasons of efficiency. This leads to rare occasions where a program needs to be certain that output data are not waiting in a buffer, but have actually been output. This procedure is handy for such occasions.

## See Also

[CLOSE](#), [EMPTY](#)

For background information, see Chapter 8, *Working with Data Files*, in the *PV-WAVE Programmer's Guide*.

---

## ***FREE\_LUN Procedure***

Deallocates file units previously allocated with GET\_LUN.

### **Usage**

FREE\_LUN, *unit*<sub>1</sub>, ..., *unit*<sub>*n*</sub>

### **Input Parameters**

*unit*<sub>*i*</sub> — The file units (logical unit numbers) to deallocate.

### **Keywords**

None.

### **Discussion**

If the specified file units are open, they are closed prior to the deallocation process.

### **Example**

Suppose that the first available logical unit number is 100.

```
GET_LUN, log_unit
    ; Returns the logical unit number to allocate (100).
OPENR, log_unit, 'test.dat'
    ; Open test.dat file for reading.
READF, log_unit, my_var
    ; Read the file.
FREE_LUN, log_unit
    ; Closes the file and frees the logical unit 100.
```

### **See Also**

[CLOSE](#), [GET\\_LUN](#), [POINT\\_LUN](#)

For background information, see the section *Logical Unit Numbers (LUNs)* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

---

## ***FSTAT Function***

Returns an expression containing status information about a specified file unit.

### **Usage**

*result* = FSTAT(*unit*)

### **Input Parameters**

*unit* — The file unit (logical unit number) about which information is required.

### **Returned Value**

*result* — A structure expression of type FSTAT containing status information about *unit*.

### **Keywords**

None.

### **Discussion**

FSTAT can be used to get more detailed information, as well as information that can be used from within a PV-WAVE program.

### **Example 1**

To get detailed information about the standard input, enter the command:

```
INFO, /Structures, FSTAT(0)
```

This causes the following to be displayed on the screen:

```
** Structure FSTAT, 10 tags, 32 length:
```

UNIT	LONG	0
NAME	STRING	'<stdin>'
OPEN	BYTE	1
ISATTY	BYTE	1
READ	BYTE	1
WRITE	BYTE	0
TRANSFER_COUNT	LONG	0
CUR_PTR	LONG	8112
SIZE	LONG	0
REC_LEN	LONG	0

---

The fields of the FSTAT structure provide the following information:

**UNIT** — The file unit number.

**NAME** — The name of the file.

**OPEN** — Nonzero if the file unit is open. If OPEN is 0, the remaining fields in FSTAT contain no useful information.

**ISATTY** — Nonzero if the file is actually a terminal instead of a normal file.

**READ** — Nonzero if the file is open for read access.

**WRITE** — Nonzero if the file is open for write access.

**TRANSFER\_COUNT** — The number of scalar data items transferred in the last I/O operation on the unit. This is set by the following routines: READ, READF, READU, PRINT, PRINTF, and WRITEU.

**TRANSFER\_COUNT** is useful when you are attempting to recover from input/output errors.

**CUR\_PTR** — The current position of the file pointer, given in bytes from the start of the file. If the device is a terminal (ISATTY is nonzero), the value of CUR\_PTR will not contain useful information.

**SIZE** — The current length of the file, in bytes. If the device is a terminal (ISATTY is nonzero), the value of SIZE will not contain useful information.

**REC\_LEN** — OpenVMS-specific record length, in bytes. This field is always zero under UNIX and Windows.

## Example 2

The following function can be used to read single-precision floating point data from a file into a vector when the number of elements in the file is not known. This function uses FSTAT to get the size of the file in bytes and then divides by 4 (the size of a single-precision floating-point value) to determine the number of values:

```
FUNCTION read_data, file
    ; Read_data reads all the floating-point values from file and returns
    ; the result as a floating-point vector.

OPENR, /Get_Lun, unit, file
    ; Get a unique file unit and open the data file.
```

```

status = FSTAT(unit)
    ; Retrieve the file status.
data = FLTARR(status.size / 4.0)
    ; Make an array to hold the input data. The size tag of status gives the number
    ; of bytes in the file and single-precision floating-point values are four bytes each.
READU, unit, data
    ; Read the data.
FREE_LUN, unit
    ; Deallocate the file unit and close the file.
RETURN, data
    ; Return the data.
END
    ; This is the end of the read_data function.

```

Assuming that a file named `herc.dat` exists and contains 10 floating-point values, the following statements:

```

a = read_data('herc.dat')
    ; Read floating-point values from herc.dat.
INFO, a
    ; Show the result.

```

will produce the following output:

```

A                                FLOAT      = Array(10)

```

## See Also

[CLOSE](#), [FREE\\_LUN](#), [GET\\_LUN](#), [OPEN \(UNIX/OpenVMS\)](#),  
[OPEN \(Windows\)](#), [POINT\\_LUN](#)

For more information, see the section *Getting Information About Files* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

For background information, see the section *Logical Unit Numbers (LUNs)* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

---

## ***FUNCT Procedure***

Standard Library procedure that evaluates a function that is a sum of a Gaussian and a second order polynomial.

### **Usage**

`FUNCT, x, parms, funcval [, pder]`

### **Input Parameters**

*x* — The values of the independent variable.

*parms* — The parameters of the equation described in the *Discussion* section.

*funcval* — The value of the function, described in the *Discussion* section, at each *x(i)*.

### **Output Parameters**

*pder* — (optional) An N\_ELEMENT(*x*)-by-6 array containing the partial derivatives of the function. The parameter *pder(i, j)* is equal to the derivative at the *i*<sup>th</sup> point with respect to the *j*<sup>th</sup> parameter.

### **Keywords**

None.

### **Discussion**

The FUNCT procedure is used primarily by the CURVEFIT function to fit the sum of a line and a varying background to actual data. The function to be evaluated is:

$$F(x) = A_0 e^{(-z^2/2)} + A_3 + A_4 x + A_5 x^2 ,$$

where  $z = (x - A_1) / (a_2)$

### **See Also**

[CURVEFIT](#)



---

## **GAMMA Function**

Calculates the gamma function of the input variable.

### **Usage**

*result* = GAMMA(*x*)

### **Input Parameters**

*x* — The expression for which the gamma function will be evaluated. *x* must evaluate to < 34.5; otherwise, a floating-point over-flow will result.

### **Returned Value**

*result* — The gamma function of *x*. The result is floating-point, regardless of the data type for *x*.

### **Keywords**

None.

### **Discussion**

The gamma function can be used in a variety of applications; one example is to solve nonlinear flow problems, such as the creep of metals.

GAMMA is defined as:

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt \quad x > 0$$

The gamma function has the following properties:

$$\Gamma(x + 1) = x\Gamma(x)$$

A special value of the gamma function occurs when  $x = 1/2$ :

$$\Gamma\left(\frac{1}{2}\right) = \sqrt{\pi}$$

### **See Also**

[ERRORF](#), [GAUSSINT](#)

---

## GAUSSFIT Function

Standard Library function that fits a Gaussian curve through a data set.

### Usage

*result* = GAUSSFIT(*x*, *y* [, *coefficients*])

### Input Parameters

*x* — A real vector containing the values of the independent variable.

*y* — A real vector containing the values of the dependent variable. Should be the same length as *x*.

### Output Parameters

*coefficients* — (optional) A six-element vector with the coefficients  $A_0$  through  $A_5$  of the equation described in the *Discussion* section.

### Returned Value

*result* — A real vector containing the dependent *y* values of the fitted function.

### Keywords

None.

### Discussion

The GAUSSFIT function fits  $y = F(x)$ , where:

$$F(x) = A_0 * \text{EXP}(-z^2/2) + A_3 + A_4x + A_5x^2$$

and

$$z = z(x - A_1)/(A_2)$$

GAUSSFIT calls the POLY\_FIT function to fit a straight line through the data for the purpose of determining estimates of the height, center, orientation, and width (approximately  $1/e$ ) of the Gaussian function to be fitted to the data.

These estimated parameters—along with the constant, linear, and quadratic coefficients of the straight line polynomial—are sent to the CURVEFIT function as trial coefficients of the Gaussian function. CURVEFIT uses a nonlinear least-squares method to fit a function with an arbitrary number of parameters. Any

nonlinear function can be fitted as long as the partial derivatives of the function are known or can be approximated.

The peak or minimum of the Gaussian function returned will be located at the index of the largest or smallest value, respectively, in the *y* vector.

## See Also

[CURVEFIT](#), [GAUSSINT](#), [POLY\\_FIT](#)

---

## **GAUSSINT Function**

Evaluates the integral of the Gaussian probability function.

### Usage

*result* = GAUSSINT(*x*)

### Input Parameters

*x* — The expression for which the Gaussian function is computed. Can be a scalar or array expression of any type except string.

### Returned Value

*result* — The integral of the Gaussian probability function. Yields floating-point results, regardless of the data type of *x*. Scalar inputs yield scalar results, and array inputs yield array results.

### Keywords

None.

### Discussion

The Gaussian probability function provides a good mathematical model for many different physically observed random phenomena. It can easily be extended to handle an arbitrarily large number of random variables. It is most commonly associated with the standard bell-shaped curve.

GAUSSINT is defined by:

$$\text{Gaussint}(x) \equiv \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{(-t^2)/2} dt$$

## See Also

[ERRORF](#), [GAMMA](#), [GAUSSFIT](#)

---

## ***GCD Function***

Standard Library function that returns the greatest common divisor of some integers greater than 0.

### **Usage**

*result* = GCD(*i*)

### **Input Parameters**

*i* — An array of integers greater than 0.

### **Returned Value**

*result* — An integer: the greatest common divisor of the integers *i*.

### **Keywords**

None.

### **Examples**

```
pm, GCD( [12,20,32] )
```

## See Also

[FACTOR](#), [LCM](#), [PRIME](#)

---

## GETENV Function

Returns the specified equivalence string from the environment of the PV-WAVE process.

### Usage

*result* = GETENV(*name*)

### Input Parameters

*name* — The scalar string specifying which equivalence string to return from the environment.

### Returned Value

*result* — The equivalence string for *name*. If *name* does not exist in the environment, returns a null string.

### Keywords

None.

### Discussion

---

**OpenVMS USERS** OpenVMS does not directly support the concept of environment variables. Instead, it is emulated in the manner described below, which allows you to use GETENV portably between UNIX and OpenVMS:

- If *name* is one of HOME, TERM, PATH, or USER, an appropriate response is generated. This mimics the most common UNIX environment variables.
  - An attempt is made to translate *name* as a logical name. All four logical name tables are searched in the standard order.
  - An attempt is made to translate *name* as a command-language interpreter symbol.
- 

### UNIX and OpenVMS Examples

This command prints information about the environment:

```
PRINT, 'Current shell is: ', GETENV('SHELL')
```

This example shows how to read data from a file in a manner that will work on either UNIX or OpenVMS systems:

```
IF !Version.platform EQ 'VMS' THEN $
    OPENR, u, GETENV('WAVE_DIR')+ $
        '[data]heartbeat.dat', /Get_Lun $
ELSE $
OPENR, u, '$WAVE_DIR/data/heartbeat.dat', /Get_Lun
```

## Windows Example

This command prints information about the environment:

```
PRINT, 'Home Drive is: ', GETENV('HOMEDRIVE')
```

This example shows how to read data using GETENV to obtain part of a file's pathname:

```
OPENR, u, GETENV('WAVE_DIR')+ $
    '\data\heartbeat.dat', /Get_lun
```

## See Also

[ENVIRONMENT](#), [SETENV](#)

---

## GET\_KBRD Function

Returns the next character available from standard input (file unit 0).

### Usage

```
result = GET_KBRD(wait)
```

### Input Parameters

*wait* — If *wait* is zero, GET\_KBRD returns the null string if there are no characters in the terminal typeahead buffer. If it is nonzero, GET\_KBRD waits for a character to be typed before returning.

### Returned Value

*result* — The next character available from standard input, as a one-character string.

## Keywords

None.

### Example

In this example, a character is read from the keyboard, and the character and its ASCII code are echoed to the screen. The loop is terminated when “q” or “Q” is typed.

```
REPEAT BEGIN
    ; Retrieve keyboard input, placing result in the variable a.
a = GET_KBRD(1)
PRINT, a, ' = ', BYTE(a)
ENDREP UNTIL STRLOWCASE(a) EQ 'q'
    ; Display the character entered and its associated ASCII code.
    ; Terminate loop when “q” or “Q” is entered.
```

---

## GET\_LUN Procedure

Allocates a file unit from a pool of free units.

### Usage

GET\_LUN, *unit*

### Input Parameters

*unit* — A named variable.

### Output Parameters

*unit* — On output, *unit* is converted into an integer containing the file unit number.

### Keywords

None.

### Discussion

GET\_LUN sets *unit* to the first available logical unit number. This number can then be used to open a file.

User-written PV-WAVE functions and procedures should use GET\_LUN to reserve unit numbers to avoid conflicts with other routines. (Similarly, they should use FREE\_LUN to free them when finished).

---

**NOTE** The *Get\_Lun* keyword, used with the OPENR, OPENU, and OPENW procedures, calls GET\_LUN to allocate a file unit number.

---

## Example

Suppose that the first available logical unit number is 100.

```
GET_LUN, log_unit
    ; Returns the logical unit number to allocate (100).
OPENR, log_unit, 'test.dat'
    ; Open test.dat file for reading.
READF, log_unit, my_var
    ; Read the file.
FREE_LUN, log_unit
    ; Closes the file and frees the logical unit 100.
```

## See Also

[CLOSE](#), [FREE\\_LUN](#), [ON\\_IOERROR](#), [OPEN \(UNIX/OpenVMS\)](#), [OPEN \(Windows\)](#), [POINT\\_LUN](#), [READ](#), [WRITEU](#)

For background information, see the section *Logical Unit Numbers (LUNs)* in Chapter 8 of the *PV-WAVE Programmer's Guide*

---

## GETNCERR Function

Retrieves the current value of the “ncerr” variable as discussed in the error section of the *NetCDF User’s Guide*.

### Usage

```
ncerr = GETNCERR([errstr])
```

### Input Parameters

*errstr* — (optional) A variable to hold the corresponding error string to the *ncerr* variable.

### Keywords

*Help* — List the usage for this function.

*Usage* — List the usage for this function. (Same as the *Help* keyword.)

### Return Value

*ncerr* — The current value of the *ncerr* variable.

### Discussion

GETNCERR retrieves the current value of the “ncerr” variable as discussed in the Error Handling section of the *NetCDF User’s Guide*. This variable gets set to a non-zero value when an error occurs in a call to the NetCDF functions. A string describing the error will be returned in the optional “errstr” parameter.

---

**NOTE** GETNCERR is only valid for the NetCDF functionality.

---

The value of “ncerr” does not change when a valid NetCDF function call is made.

### Example

```
ncid = NCOPEN("foo.nc", NC_NOWRITE)
status = NCCLOSE(ncid)
status = NCREDEF(ncid)
    ncredef: 0 is not a valid cdfid
```

```
    % NCREDEF: error in HDF return status.
ncerr = GETNCERR(errstr)
INFO, ncerr, errstr
    NCERR LONG = 1
    ERRSTR STRING = 'Not a netcdf id'
```

## See Also

[GETNCOPTS](#), [SETNCOPTS](#)

Also refer to the *NetCDF User's Guide*.

For more information on using the PV-WAVE HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## GETNCOPTS Function

Retrieves the current value of the *ncopts* variable as discussed in the error section of the *NetCDF User's Guide*.

### Usage

```
ncopts = GETNCOPTS( )
```

### Input Parameters

None.

### Keywords

*Help* — List the usage for this function.

*Usage* — List the usage for this function. (Same as the *Help* keyword.)

### Return Value

*ncopts* — The current value of the “ncopts” variable.

## Discussion

GETNCOPTS retrieves the current value of the “ncopts” variable as discussed in the Error Handling section of the *NetCDF User’s Guide*. This variable defines the level of error reporting by the netCDF functions.

---

**NOTE** GETNCOPTS is only valid for the netCDF functionality.

---

## Example

```
ncopts = GETNCOPTS()
INFO, ncopts
    NCOPTS LONG = 2
        ; The default value of "ncopts" is set to NC_VERBOSE which is
        ; equal to 2.

ncid = NCOOPEN("foo.nc", NC_NOWRITE)
status = NCCLOSE(ncid)
status = NCREDEF(ncid)
    ncredef: 0 is not a valid cdfid
    % NCREDEF: error in HDF return status.
```

## See Also

[GETNCERR](#), [SETNCOPTS](#)

Also refer to the *NetCDF User’s Guide*.

For more information on using the PV-WAVE HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## ***GET\_SYMBOL Function (OpenVMS)***

Returns the value of an OpenVMS DCL interpreter symbol as a scalar string.

### **Usage**

*result* = GET\_SYMBOL(*name*)

### **Input Parameters**

*name* — A scalar string containing the name of the symbol to be translated.

### **Returned Value**

*result* — A scalar string containing the value of an OpenVMS Digital Command Language interpreter symbol. If the symbol is undefined, the null string is returned.

### **Keywords**

*Type* — Indicates in which OpenVMS table *name* is found:

- 1 Specifies the local symbol table (the default).
- 2 Specifies the global symbol table.

### **Example**

This example assumes that on your system `kermit` is a symbol that points to the KERMIT communications software.

```
my_kermit=GET_SYMBOL('kermit')  
; This converts my_kermit into the string SYS$SYSTEM:KERMIT.
```

### **See Also**

[DELETE\\_SYMBOL](#), [DELLOG](#), [SETLOG](#), [SET\\_SYMBOL](#), [TRNLOG](#)

---

## ***GREAT\_INT Function***

Greatest Integer Function. Standard Library function that returns the greatest integer less than or equal to the passed value. Also known as the Floor Function.

### **Usage**

*result* = GREAT\_INT(*values*)

### **Input Parameters**

*values* — An array (scalar).

### **Returned Value**

*result* — A long array (scalar) of the same dimensions as *values*: *result*(i) is the greatest integer less than or equal to *values*(i).

### **Keywords**

None.

### **Example**

```
PM, GREAT_INT( [-0.5,0,0.5] )
```

### **See Also**

[SMALL\\_INT](#)

---

## GRID Function

Standard Library function that generates a uniform grid from irregularly-spaced data.

### Usage

*result* = GRID(*xtmp*, *ytmp*, *ztmp*)

### Input Parameters

*xtmp* — The vector containing the *x*-coordinates of the irregularly-spaced input data.

*ytmp* — The vector containing the *y*-coordinates of the irregularly-spaced input data.

*ztmp* — The vector containing the *z*-coordinates of the irregularly-spaced input data.

### Returned Value

*result* — An array containing the gridded *z* values applied to a uniform XY grid.

### Keywords

*Nghbr* — The number of neighboring data points to be used in the gridding algorithm. Must be in the range of {3 ... 25}. (Default: 3)

*Nx* — The number of columns in the resulting array. Must be  $\leq 200$ .

*Ny* — The number of rows in the resulting array. Must be  $\leq 200$ .

### Discussion

For PV-WAVE Version 6.0, the previously used GRID procedure algorithm has been discontinued. GRID now calls the FAST\_GRID3 procedure directly. If you prefer to use the previously supported GRID algorithm, please contact Visual Numerics Technical Support.

---

**UNIX and OpenVMS USERS** PV-WAVE:GTGRID is an optional software package for advanced gridding. It gives you additional interpolation and extrapolation power by providing access to a library of gridding routines provided by Geophysical Techniques, Inc. For information on PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---

## See Also

[FAST\\_GRID3](#), [GRID\\_3D](#)

---

## GRIDN Function

Standard Library function that grids n dimensional data.

### Usage

*result* = GRIDN(*d*, *i*)

### Input Parameters

*d* — An (m,n+1) array of m datapoints in n independent variables and one dependent variable; *d*(\*,*n*) is the dependent variable.

*i* — A vector of n integers specifying the dimensions of the grid.

### Returned Value

*result* — An n dimensional array of values of the dependent variable on a regular grid over the independent variables.

### Keywords

*r* — A scalar specifying the order of the weighting function. The dependent variable at a grid point is computed as a weighted average of the variable over all neighborhood datapoints. The weighting function is  $1/e^r$  where e is the Euclidean distance between the grid point and the datapoint. *r* defaults to 2

*t* — A scalar between 0 and 1 specifying neighborhood size. *t*=1 gives a maximal neighborhood which includes all datapoints, while lower *t* values yield smaller neighborhoods. *t* defaults to 1

*b* — A 2 x n array fixing the boundary of the grid. *b*(0,\*) is the minimum corner and *b*(1,\*) is the maximum corner. The default extent of the grid is the same as that of the data.

*f* — The name of a user-supplied procedure describing voids in the independent variable space (datapoints and gridpoints within these regions are ignored in computation). Input to *f* is a (p,n) array of p points in the independent variable space. *f* outputs two items where the first item is a vector of indices indicating which of the

$p$  input points are within bounds, and where the second item is a scalar that will appear as a place holder for the dependent variable at out-of-bounds gridpoints.

$c$  — (output) A list of  $n$  vectors defining the grid coordinates.

## Examples

See `wave/lib/user/examples/gridnex1.pro`

`wave/lib/user/examples/gridnex2.pro`

## See Also

[FAST\\_GRID2](#), [FAST\\_GRID3](#), [FAST\\_GRID4](#), [INTERPOLATE](#)

---

## GRID\_2D Function

Returns a gridded, 1D array containing  $y$  values, given random  $x,y$  coordinates (this function works best with sparse data points).

### Usage

`result = GRID_2D(points, grid_x)`

### Input Parameters

*points* — A  $(2, n)$  array containing the random  $x,y$  points to be gridded.

*grid\_x* — The size of the vector to return.

### Returned Value

*result* — A gridded, 1D array containing  $y$  values.

### Keywords

*Order* — The order of the weighting function to use for neighborhood averaging. Points are weighted by the function:

$w = 1.0 / (\text{dist} \wedge \text{Order})$

where `dist` is the distance to the point. (Default: 2)

*XMax* — The  $x$ -coordinate of the right edge of the grid. If omitted, maps the maximum  $x$  value found in the `points(0, *)` array to the right edge of the grid.

*XMin* — The *x*-coordinate of the left edge of the grid. If omitted, maps the minimum *x* value found in the *points(0, \*)* array to the left edge of the grid.

## Discussion

GRID\_2D uses an inverse distance averaging technique to interpolate missing data values for 2D gridding. The gridded array returned by GRID\_2D is suitable for use with the PLOT function.

GRID\_2D is similar to FAST\_GRID2. GRID\_2D, however, works best with sparse data points (say, less than 1000 points to be gridded) and is stable when extrapolating into large void areas. (FAST\_GRID2 works best with dense data points; it is considerably faster, but slightly less accurate, than GRID\_2D.)

## Examples

```
PRO grid_demo2
    ; This program shows 2D gridding with sparse data points.
points = INTARR(2, 10)
points(*, 0) = [1,2]
points(*, 1) = [1,3]
points(*, 2) = [9,5]
points(*, 3) = [8,0]
points(*, 4) = [9,6]
points(*, 5) = [9,9]
points(*, 6) = [7,15]
points(*, 7) = [6,-5]
points(*, 8) = [0,3]
points(*, 9) = [0,-1]
    ; Generate the data.
WINDOW, 0, Colors=128
LOADCT, 4
T3D, /Reset
    ; Reset the viewing window and load the color table.
!Y.Range = [MIN(points), MAX(points)]
    ; Set the y-axis range for plotting.
yval = GRID_2D(points, 256, Order=0.5)
PLOT, yval, Color=60
yval = GRID_2D(points, 256, Order=1.0)
OPLOT, yval, Color=80
yval = GRID_2D(points, 256, Order=2.0)
OPLOT, yval, Color=100
yval = GRID_2D(points, 256, Order=3.0)
```

```
OPlot, yval, Color=120
    ; Grid and plot the resulting data.
!Y.Range = [0.0, 0.0]
    ; Reset the y-axis range to the default value.
END
```

## See Also

[FAST\\_GRID2](#), [FAST\\_GRID3](#), [FAST\\_GRID4](#), [GRID\\_3D](#),  
[GRID\\_4D](#), [GRID\\_SPHERE](#)

---

**UNIX and OpenVMS USERS** For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---

---

## GRID\_3D Function

Returns a gridded, 2D array containing  $z$  values, given random  $x$ -,  $y$ -,  $z$ -coordinates (this function works best with sparse data points).

### Usage

*result* = GRID\_3D(*points*, *grid\_x*, *grid\_y*)

### Input Parameters

*points* — A (3,  $n$ ) array containing the random  $x$ ,  $y$ ,  $z$  points to be gridded.

*grid\_x* — The  $x$  dimension of the grid. The  $x$  values are scaled to fit this dimension.

*grid\_y* — The  $y$  dimension of the grid. The  $y$  values are scaled to fit this dimension.

### Returned Value

*result* — A gridded, 2D array containing  $z$  values.

### Keywords

*Order* — The order of the weighting function to use for neighborhood averaging. Points are weighted by the function:

$w = 1.0 / (\text{dist} \wedge \text{Order})$

where `dist` is the distance to the point. (Default: 2)

***XMax*** — The  $x$ -coordinate of the right edge of the grid. If omitted, maps the maximum  $x$  value found in the `points(0, *)` array to the right edge of the grid.

***XMin*** — The  $x$ -coordinate of the left edge of the grid. If omitted, maps the minimum  $x$  value found in the `points(0, *)` array to the left edge of the grid.

***YMax*** — The  $y$ -coordinate of the top edge of the grid. If omitted, maps the maximum  $y$  value found in the `points(1, *)` array to the top edge of the grid.

***YMin*** — The  $y$ -coordinate of the bottom edge of the grid. If omitted, maps the minimum  $y$  value found in the `points(1, *)` array to the bottom edge of the grid.

## Discussion

GRID\_3D uses an inverse distance averaging technique to interpolate missing data values for 3D gridding. The gridded array returned by GRID\_3D is suitable for use with the SURFACE, TV, and CONTOUR procedures.

GRID\_3D is similar to FAST\_GRID3. GRID\_3D, however, works best with sparse data points (say, less than 1000 points to be gridded) and is stable when extrapolating into large void areas. (FAST\_GRID3 works best with dense data points; it is considerably faster, but slightly less accurate, than GRID\_3D.)

## Examples

```
PRO grid_demo3
    ; This program shows 3D gridding with sparse data points.

points = INTARR(3, 10)
points(*, 0) = [1,1,2]
points(*, 1) = [1,1,3]
points(*, 2) = [9,9,5]
points(*, 3) = [9,8,0]
points(*, 4) = [4,9,6]
points(*, 5) = [0,9,9]
points(*, 6) = [3,7,15]
points(*, 7) = [5,6,-5]
points(*, 8) = [0,0,3]
points(*, 9) = [9,0,-1]
    ; Generate the data points.

zval = GRID_3D(points, 48, 32, Order=2.0)
    ; Grid the data points.

WINDOW, 0, Colors=128
```

```
SURFR
SURFACE, zval, Bottom=90, Ax=30.0, Az=30.0, /T3d
      ; Display the gridded data as a surface.
END
```

## See Also

[FAST\\_GRID2](#), [FAST\\_GRID3](#), [FAST\\_GRID4](#), [GRID\\_2D](#),  
[GRID\\_4D](#), [GRID\\_SPHERE](#)

---

**UNIX and OpenVMS USERS** For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---

---

## GRID\_4D Function

Grids a 3D array containing intensity values, given random 4D coordinates (this function works best with sparse data points).

### Usage

*result* = GRID\_4D(*points*, *grid\_x*, *grid\_y*, *grid\_z*)

### Input Parameters

*points* — A (4, *n*) array containing the random 4D points to be gridded. Typically, *points*(0, \*) contains the *x* values, *points*(1, \*) contains the *y* values, *points*(2, \*) contains the *z* values, and *points*(3, \*) contains the intensity values. (You may, however, choose to put other variables in these four vectors.)

*grid\_x* — The *x* dimension of the grid. The *x* values are scaled to fit this dimension.

*grid\_y* — The *y* dimension of the grid. The *y* values are scaled to fit this dimension.

*grid\_z* — The *z* dimension of the grid. The *z* values are scaled to fit this dimension.

### Returned Value

*result* — A gridded, 3D array containing intensity values.

## Keywords

**Order** — The order of the weighting function to use for neighborhood averaging. Points are weighted by the function:

```
w = 1.0 / (dist ^ Order)
```

where `dist` is the distance to the point. (Default: 2)

**XMax** — The  $x$ -coordinate of the right edge of the grid. If omitted, maps the maximum  $x$  value found in the `points(0, *)` array to the right edge of the grid.

**XMin** — The  $x$ -coordinate of the left edge of the grid. If omitted, maps the minimum  $x$  value found in the `points(0, *)` array to the left edge of the grid.

**YMax** — The  $y$ -coordinate of the top edge of the grid. If omitted, maps the maximum  $y$  value found in the `points(1, *)` array to the top edge of the grid.

**YMin** — The  $y$ -coordinate of the bottom edge of the grid. If omitted, maps the minimum  $y$  value found in the `points(1, *)` array to the bottom edge of the grid.

**ZMax** — The  $z$ -coordinate of the front edge of the grid. If omitted, maps the maximum  $z$  value found in the `points(2, *)` array to the front edge of the grid.

**ZMin** — The  $z$ -coordinate of the back edge of the grid. If omitted, maps the minimum  $z$  value found in the `points(2, *)` array to the back edge of the grid.

## Discussion

GRID\_4D uses an inverse distance averaging technique to interpolate missing data values for 4D gridding. The gridded array returned by GRID\_4D is suitable for use with the SHADE\_VOLUME and VOL\_REND functions.

GRID\_4D is similar to FAST\_GRID4. GRID\_4D, however, works best with sparse data points (say, less than 1000 points to be gridded) and is stable when extrapolating into large void areas. (FAST\_GRID4 works best with dense data points; it is considerably faster, but slightly less accurate, than GRID\_4D.)

## Examples

```
PRO grid_demo4
```

```
    ; This program shows 4D gridding with sparse data points and a cut-away.
```

```
points = INTARR(4, 10)
points(*, 0) = [1, 1, 2, 86]
points(*, 1) = [1, 1, 3, 44]
points(*, 2) = [9, 9, 5, 37]
points(*, 3) = [5, 4, 7, 99]
```

```

points(*, 4) = [4, 0, 6, 9]
points(*, 5) = [0, 9, 9, 32]
points(*, 6) = [3, 5, 5, 2]
points(*, 7) = [6, 6, 5, 55]
points(*, 8) = [0, 0, 5, 66]
points(*, 9) = [9, 0, 0, 44]
    ; Generate the data to be used for shading.

ival = GRID_4D(points, 32, 32, 32, Order=4.0)
ival = BYTSCL(ival)
    ; Grid the generated data.

block = BYTARR(30, 30, 30)
block(*, *, *) = 255
block = VOL_PAD(block, 1)
    ; Pad the data with zeroes.

block(0:16, 0:16, 16:31) = 0
    ; Cut away a section of the block by setting the desired elements to zero.

WINDOW, 0, Colors=128
LOADCT, 3
CENTER_VIEW, Xr=[0.0, 31.0], Yr=[0.0, 31.0], Zr=[0.0, 31.0], $
    Ax=(-60.0), Az=45.0, Zoom=0.6
    ; Set up the viewing window and load the color table.

SET_SHADING, Light=[-1.0, 1.0, 0.2]
    ; Change the direction of the light source for shading.

SHADE_VOLUME, block, 1, vertex_list, $
    polygon_list, Shades=ival, /Low
    ; Compute the 3D contour surface.

img1 = POLYSHADE(vertex_list, polygon_list, /T3d)
    ; Render the cut-away block with light source shading.

img2 = POLYSHADE(vertex_list, polygon_list, Shades=ival, /T3d)
    ; Render the cut-away block shaded by the gridded data.

TVSCL, (FIX(img1) + FIX(img2))
    ; Display the resulting composite image of the light
    ; source-shaded block and the data-shaded block.

END

```

For another example, see the `vol_demo4` demonstration program in:

```

(UNIX)      <wavedir>/demo/ar1
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows) <wavedir>\demo\ar1

```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[FAST\\_GRID2](#), [FAST\\_GRID3](#), [FAST\\_GRID4](#), [GRID\\_2D](#),  
[GRID\\_3D](#), [GRID\\_SPHERE](#)

---

**UNIX and OpenVMS USERS** For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---

---

## ***GRID\_SPHERE* Function**

Returns a gridded, 2D array containing radii, given random longitude, latitude, and radius values.

### **Usage**

*result* = GRID\_SPHERE(*points*, *grid\_x*, *grid\_y*)

### **Input Parameters**

*points* — A (3, *n*) array containing the random longitude, latitude, and radius coordinates to be gridded.

*grid\_x* — The *x* dimension of the grid. The longitude values are scaled to fit this dimension (unless the *XMin* or *XMax* keywords are set).

*grid\_y* — The *y* dimension of the grid. The latitude values are scaled to fit this dimension (unless the *YMin* or *YMax* keywords are set).

### **Returned Value**

*result* — A gridded, 2D array containing radius values.

### **Keywords**

*Degrees* — If present and nonzero, reads the input coordinates in degrees instead of in radians.

**Order** — The order of the weighting function to use for neighborhood averaging. Points are weighted by the function:

$$w = 1.0 / (\text{dist} \wedge \text{Order})$$

where `dist` is the distance to the point. (Default: 2)

**Radius** — The radius of the sphere to grid on. The minimum allowable radius is 0.5. (Default: 1.0)

With a smaller radius, the data points are closer together and more smoothing occurs. A larger radius causes less smoothing.

**XMax** — The longitude of the right edge of the grid. Should be in the range  $-\pi$  to  $+\pi$  radians ( $-180$  to  $+180$  degrees). The *XMax* value is assumed to be in radians unless the *Degrees* keyword is set.

If *XMax* is omitted, then a longitude of  $\pi$  is mapped to the right edge of the grid.

**XMin** — The longitude of the left edge of the grid. Should be in the range  $-\pi$  to  $+\pi$  radians ( $-180$  to  $+180$  degrees). The value is assumed to be in radians unless the *Degrees* keyword is set.

If *XMin* is omitted, then a longitude of  $-\pi$  is mapped to the left edge of the grid.

**YMax** — The latitude of the top edge of the grid. Should be in the range  $-\pi/2$  to  $+\pi/2$  radians ( $-90$  to  $+90$  degrees). The *YMax* value is assumed to be in radians unless the *Degrees* keyword is set.

If *YMax* is omitted, then a latitude of  $\pi/2$  is mapped to the top edge of the grid.

**YMin** — The latitude of the bottom edge of the grid. Should be in the range  $-\pi/2$  to  $+\pi/2$  radians ( $-90$  to  $+90$  degrees). The *YMin* value is assumed to be in radians unless the *Degrees* keyword is set.

If *YMin* is omitted, then a latitude of  $-\pi/2$  is mapped to the bottom edge of the grid.

## Discussion

GRID\_SPHERE uses an inverse distance averaging technique to interpolate missing data values. The gridded array returned by GRID\_SPHERE is suitable for use with the POLY\_SPHERE function.

The longitude values are assumed to be in the range  $-\pi$  to  $+\pi$  ( $-180$  to  $+180$  if the *Degrees* keyword is set). The latitude values are assumed to be in the range  $-\pi/2$  to  $+\pi/2$  ( $-90$  to  $+90$  if the *Degrees* keyword is set).

To grid on a portion of a sphere rather than on an entire sphere, use the *XMin*, *XMax*, *YMin*, and *YMax* keywords.

## Examples

```
PRO grid_demo5
    ; This program shows spherical gridding.

sphere = FLTARR(3, 6)
sphere(*, 0) = [ 33.0, -64.0, 0.2]
sphere(*, 1) = [280.0,   5.0, 1.8]
sphere(*, 2) = [350.0,  41.0, 1.9]
sphere(*, 3) = [310.0,  83.0, 0.3]
sphere(*, 4) = [ 67.0, -16.0, 1.6]
sphere(*, 5) = [133.0, -75.0, 0.2]
sphere(0, *) = sphere(0, *) - 180.0
    ; Generate the data — random longitude, latitude, and radius points.

sphere = GRID_SPHERE(sphere, 32, 32, $
    /Degrees, Order=4.0, Radius=20.0)
    ; Grid the resulting sphere.

POLY_SPHERE, sphere, 32, 32, vertex_list, polygon_list
    ; Generate the polygons representing the spherical surface.

WINDOW, 0, Colors=128, XSize=800, YSize=600
LOADCT, 3
CENTER_VIEW, Xr=[-2.0, 2.0], Yr=[-2.0, 2.0], $
    Zr=[-2.0, 2.0], Ax=(-40.0), Az=0.0, $
    Zoom=1.0, Winx=800, Winy=600
    ; Set up the viewing window and load the color table.

SET_SHADING, Light=[-0.5, 0.5, 1.0]
TVSCL, POLYSHADE(vertex_list, polygon_list, /T3d)
    ; Specify the shading and display the resulting spherical surface.

END
```

For another example, see the `sphere_demo3` demonstration program in:

```
(UNIX)      <wavedir>/demo/ar1
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows)  <wavedir>\demo\ar1
```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[FAST\\_GRID2](#), [FAST\\_GRID3](#), [FAST\\_GRID4](#), [GRID\\_2D](#),  
[GRID\\_3D](#), [GRID\\_4D](#), [POLY\\_SPHERE](#)

---

**UNIX and OpenVMS USERS** For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

---

---

## ***GROUP\_BY* Function**

Performs summary (aggregate) functions to groups of rows in a PV-WAVE table variable.

### **Usage**

```
result = GROUP_BY(in_table, 'sum_column [alias] [ASC | DESC']')
```

---

**NOTE** The entire second parameter is a string and must be enclosed in quotes. Also, note that the vertical bar (|) means “or” in this usage. For instance, use either *ASC* or *DESC*, but not both.

---

### **Input Parameters**

*in\_table* — An input PV-WAVE table variable on which to perform the summary functions.

*sum\_column* — A single column in the input table that determines how to perform the summarization. For each distinct value of *sum\_column* in the original table, all rows that contain this value are grouped together to produce one row in the resulting table. A column with the same name and value as *sum\_column* is created in the resulting table.

*alias* — Specifies a new name for *sum\_column* in the resulting table.

*ASC* — Requires that the rows of the result are sorted in ascending order by the value in *sum\_column*. If no sort order is specified, *ASC* is the default.

*DESC* — Requires that the rows of the result are sorted in descending order by the value in *sum\_column*.

## Returned Value

**result** — A PV-WAVE table variable, containing one column specified by *sum\_column*, and one for each column specified by the keywords. If the query result is empty, and no syntax or other errors occurred, the result returned is  $-1$ .

## Input Keywords

---

**NOTE** For each of the following keywords, the value is a string in the format '*col1* [*alias*], [*col2* [*alias*],] ... [*coln* [*alias*]]', which represents a list of column names from *in\_table* and associated aliases for *result*.

---

**Avg** — Calculates the group average for each column listed in the string.

**Count** — Counts the number of occurrences of each data value within the group for each column listed in the string.

**Max** — Calculates the group maximum for each column listed in the string.

**Min** — Calculates the group minimum for each column listed in the string.

**Sum** — Calculates the group total value for each column listed in the string.

## Discussion

The GROUP\_BY function produces similar output to the *Group By* option of the QUERY\_TABLE function, but GROUP\_BY has a more compact and convenient syntax. For each unique value in *sum\_column* of *in\_table*, GROUP\_BY forms a sub-table from all of the rows in *in\_table* that have *sum\_column* equal to that value. For each keyword specified, GROUP\_BY performs the indicated function (*Avg*, *Count*, *Max*, *Min*, *Sum*) on each column given in the list, over all rows in the current sub-table. For each sub-table, GROUP\_BY returns one row in a PV-WAVE table variable. The row contains the current value of *sum\_column*, and an additional column for each column in the list for each keyword.

## Example

Consider the following PV-WAVE table variable, already defined during the current session:

```
INFO, prop_trx
PROP_TRX          STRUCT      = -> TABLE_3745584016934985140252399 Array(10000)
```

```
INFO, prop_trx, /Structure
```

```
** Structure TABLE_3745584016934985140252399, 8 tags, 72 length:
```

```
TRX_ID      LONG          0
PROP_TYPE   STRING      'OTHER          '
PROP_ADDRESS STRING     ''
PROP_POST_CD STRING     ''
PROP_XGRID  DOUBLE      0.0075200000
PROP_YGRID  DOUBLE      1.6357100
TRX_AMT     DOUBLE      116383.00
TRX_DATE    STRUCT      -> !DT Array(1)
```

Suppose that we would like to find the total amount, average amount, count, average x grid value, and average y grid value for each property type. We could accomplish this with the following call to `GROUP_BY`:

```
trx_sum = GROUP_BY( prop_trx, $
  'prop_type my_prop_type ', $
  AVG='trx_amt my_avg_amt, ' + $
  'prop_xgrid my_avg_x, ' + $
  'prop_ygrid my_avg_y ', $
  SUM='trx_amt my_total_amt', $
  COUNT='prop_type my_type_cnt')
```

We would get a new `PV-WAVE` table, `trx_sum`, which has 9 rows (one for each unique value of `prop_type`).

```
INFO, trx_sum
```

```
TRX_SUM      STRUCT      = -> TABLE_2654125490145392573020051 Array(9)
```

The columns in `trx_sum` are as follows:

```
INFO, trx_sum, /Structure
```

```
** Structure TABLE_2654125490145392573020051, 6 tags, 48 length:
```

```
MY_PROP_TYPE STRING     'STUDIO          '
MY_AVG_AMT    DOUBLE     54541.422
MY_AVG_X      DOUBLE     2.5688594
```

```

MY_AVG_Y      DOUBLE          1.5601237
MY_TYPE_CNT   LONG           1075
MY_TOTAL_AMT DOUBLE          58632029.

```

Note that this could also be accomplished with the following call to `QUERY_TABLE`:

```

trx_sum2 = QUERY_TABLE( prop_trx, $
'prop_type my_prop_type, ' + $
'AVG(trx_amt) my_avg_amt, ' + $
'AVG(prop_xgrid) my_avg_x, ' + $
'AVG(prop_ygrid) my_avg_y, ' + $
'SUM(trx_amt) my_total_amt, ' + $
'COUNT(prop_type) my_type_cnt ' + $
'GROUP BY prop_type ')

```

which produces the following results:

```

INFO, trx_sum2
TRX_SUM2      STRUCT      = -> TABLE_7241171353020130317830120 Array(9)

```

```

INFO, trx_sum2, /Structure
** Structure TABLE_7241171353020130317830120, 6 tags, 48 length:
MY_PROP_TYPE STRING      'STUDIO
MY_AVG_AMT    DOUBLE      54541.422
MY_AVG_X      DOUBLE      2.5688594
MY_AVG_Y      DOUBLE      1.5601237
MY_TYPE_CNT   LONG        1075
MY_TOTAL_AMT DOUBLE      58632029.

```

## See Also

[ORDER\\_BY](#), [QUERY\\_TABLE](#), [UNIQUE](#)

For more information on `BUILD_TABLE`, see .

For information on reading data into variables, see



---

## **HAK Procedure**

Standard Library procedure that lets you implement a “hit any key to continue” function.

### **Usage**

HAK

### **Parameters**

None.

### **Keywords**

*Mesg* — Alerts the user that the procedure is momentarily stopped. If set to 1 (the default), the following message is printed on the display screen:

```
Hit any key to continue...
```

*Mesg* may also be set to an arbitrary string.

### **Discussion**

HAK waits for keyboard input, clears the typeahead buffer, and allows the application to continue. It is used primarily to stop a procedure momentarily — for example, to allow the user to read an explanation screen or view a temporary plot.

### **Example 1**

```
a = INDGEN(200)
PRINT, a
HAK, Mesg='Press a key to go on.'
```

### **Example 2**

```
t = 'When ready, press any key.'
PRINT, t
HAK, Mesg=t
```

### **See Also**

[WAIT](#)

---

## **HANNING Function**

Standard Library function that implements a window function for Fast Fourier Transform signal or image filtering.

### **Usage**

*result* = HANNING(*col* [, *row*])

### **Input Parameters**

*col* — The number of columns in the result.

*row* — (optional) The number of rows in the result.

### **Returned Value**

*result* — The processed result.

### **Keywords**

None.

### **Discussion**

HANNING is a window function for signal or image filtering using a fast fourier transform. By processing data through HANNING before applying FFT, more realistic results can be obtained.

The window calculated by HANNING is basically the first half of a cosine — in other words, only the positive cosine values. When used with only the *col* parameter, HANNING returns a vector of the same length as *col*. The vector starts and ends with zeros and rises to a peak in the center (just as a cosine goes up and then comes back down to zero).

For one dimension, the result of HANNING is determined by the following equation:

$$result(i) = \frac{1}{2} \left( 1 - \cos \left( \frac{2\pi i}{n-1} \right) \right)$$

where *n* is the total number of elements described by *row* and *col*.

When used with both the *col* and *row* parameters, HANNING returns an array whose dimensions are the same as these two parameters. The resultant array has zeros around the sides and rises to a peak in the center.

For two dimensions, the result  $(i, j) = \text{result}(i) * \text{result}(j)$ .

## Example

```
OPENR, unit, !Data_dir + 'mandril.img', /Get_lun
    ; Open the mandrill image file.

aa = ASSOC(unit, BYTARR(512, 512))
image = aa(0)
    ; Store the image.

CLOSE, unit
    ; Close the logical unit number.

han = HANNING(512, 512)
    ; Create the HANNING window.

a = FFT(image, -1)
    ; Create an FFT of the image without the HANNING window.

WINDOW, 0, XSize=512, YSize=512, $
    Title='FFT without HANNING Window Applied'

TVSCL, SHIFT(ALOG(ABS(a)), 256, 256)
    ; Shifting makes it easier to see.

b = FFT(image * han, -1)
    ; Create an FFT of the image with the HANNING window. This
    ; diminishes the effect of the outside edges of the image on the
    ; FFT result.

WINDOW, 2, Xsize=512, Ysize=512, $
    Title='FFT with HANNING Window Applied'

TVSCL, SHIFT(ALOG(ABS(b)), 256, 256)
    ; Take a look at the result for comparison, and notice that the vertical
    ; and horizontal streaking is gone.
```

## See Also

[FFT, HILBERT](#)

For details on how processing data through HANNING before applying FFT improves results, see Chapter 5 in *Digital Signal Processing*, by Oppenheim and Schaffer, Prentice-Hall, Englewood Cliffs, NJ, 1975.

---

## **HDFGET24 Function**

Obtains an HDF Raster 24 image.

### **Usage**

*status* = HDFGET24 (*filename*, *image*)

### **Input Parameters**

*filename* — The name of the HDF file.

### **Output Parameters**

*image* — A byte array into which the HDF Raster 24 image is placed.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

### **Keywords**

**Help** — If present and nonzero, lists the usage for this routine.

**Interlace** — Set the HDF interlace scheme to use in reading the image. Possible values include:

DFIL\_PIXEL  
DFIL\_LINE  
DFIL\_PLANE

The default will be the interlace scheme used when the image was written to the file.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

## Discussion

HDFGET24 obtains an HDF Raster 24 image from the named HDF file. HDFGET24 reads the current (or first) Raster 24 image in the file and positions the HDF active pointer to the next Raster 24 image. You can read all Raster 24 images in an HDF file using successive calls to HDFGET24. The return status is set to FAIL when the function reads beyond the last Raster 24 image.

## Example

```
testfile = 'raster24.hdf'
status = DF24RESTART()
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed at DF24restart for'+HDFGET24.'
status = HDFGET24(testfile, image, interlace=DFIL_PIXEL)
IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFGET24 failed.'
TV, image, True=1
```

## See Also

[HDFGETR8](#), [HDFPUT24](#)

Also refer to the following routines in the *HDF Reference Manual*:

DF24GETDIMS, DF24GETIMAGE, DF24READREF, DF24RESTART,  
HISHDF

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## **HDFGETANN Function**

Obtains HDF object (e.g., an SDS, Raster 8 image, etc.) annotations, either a label or a description.

### **Usage**

*status* = HDFGETANN (*filename*, *tag*, *ref*)

### **Input Parameters**

*filename* — The name of the HDF file.

*tag* — The HDF tag number associated with the annotation.

*ref* — The HDF reference number of the target object.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)                      Indicates success.

FAIL (-1)                         Indicates failure.

### **Keywords**

**Description** — A byte array that contains the description for the specified filename/tag/ref object. This byte array may require further processing for display of the description.

**Help** — If present and nonzero, lists the usage for this routine.

**Label** — A string variable that contains the label for the specified filename/tag/ref object.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

### **Discussion**

HDFGETANN is used to get a label or description for an arbitrary HDF object, as specified by a filename/tag/ref triplet.

## Example

```
newdesc = BYTE ( ' ' )
tag = DFTAG_RI8
ref = 2
status = HDFGETANN (testfile, tag, ref, Description=newdesc)
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed HDFGETANN with'+Description.'
PRINT, STRING (newdesc)
```

## See Also

Also refer to the following routines in the *HDF Reference Manual*:

DFANGETDESC, DFANGETDESCLEN, DFANGETLABLEN,  
DFANGETLABEL, DFANPUTDESC, DFANPUTLAB

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## HDFGETFILEANN Function

Obtains an HDF file annotation, either label or description.

### Usage

```
status = HDFGETFILEANN (filename)
```

### Input Parameters

*filename* — The name of the HDF file.

### Return Value

*status* — The status of the function call, where:

SUCCEED (0)                      Indicates success.

FAIL (-1)                      Indicates failure.

## Keywords

**Description** — A byte array that contains the description for the specified filename/tag/ref object. This byte array may require further processing for display of the description.

**Help** — If present and nonzero, lists the usage for this routine.

**Isfirst** — Specifies whether to select the first or next label or description. Values for *Isfirst* are:

1 for the first label/descriptor.

0 for the next label/descriptor.

**Label** — A string variable that contains the label for the specified filename/tag/ref object.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

## Discussion

HDFGETFILEANN is used to get an HDF file annotation, either a file label or a file description. File labels and file descriptions differ from HDF object labels/descriptions in that you can access multiple labels or descriptions using the *Isfirst* keyword.

## Example

```
status = HDFGETFILEANN (testfile, Description=newdesc)
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed HDFGETFILEANN with'+ 'Description.'
PRINT, STRING (newdesc)
```

## See Also

[HDFPUTFILEANN](#)

Also refer to the following routines in the *HDF Reference Manual*:

DFANGETFDS, DFANGETFDSLEN, DFANGETFID, DFANGETFIDLEN,  
HCLOSE, HOPEN

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, \*The PV-WAVE HDF Interface\*](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, \*Functional Summary of Routines\*](#).

---

## **HDFGETNT Function**

Obtains the HDF number type (i.e., data type) and descriptive number type string for the current HDF Scientific Data Set.

### **Usage**

*status* = HDFGETNT (*type*)

### **Output Parameter**

*type* — HDF Scientific Data Set numeric data type. The range of possible values are defined in HDF\_COMMON:

DFNT\_FLOAT32  
DFNT\_FLOAT64  
DFNT\_INT8  
DFNT\_UINT8  
DFNT\_INT16  
DFNT\_UINT16  
DFNT\_INT32  
DFNT\_UINT32

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

### **Keywords**

*Help* — If present and nonzero, lists the usage for this routine.

**Name** — Returns HDF SDS numeric data type as a string, e.g., “DFNT\_FLOAT32”. This keyword is more for human use than programmatic use.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

**Wavecast** — Returns a string containing the PV-WAVE data type cast that is equivalent to an HDF SDS numeric data type. You can use the string in a PV-WAVE EXECUTE statement to dynamically declare a PV-WAVE variable for use with HDF. Example value: “FLTARR”.

**Wavetype** — Returns a string containing the PV-WAVE data type that is equivalent to a HDF SDS numeric data type. This string can be used to programmatically change the data type of a given PV-WAVE variable to the type required for HDF. Example value: “FLOAT”.

## Discussion

HDFGETNT gets the current HDF Scientific Data Set numeric (data) type in a number of different formats. The primary format, where only *type* is returned, is a simple numeric value that corresponds to a specific numeric type. Keywords facilitate mapping of this value to strings that are of greater use to the PV-WAVE programmer. These strings can be used with the PV-WAVE command EXECUTE to dynamically declare an array for use with an HDF Scientific Data set.

## Example

```
wavetype = ''
status = HDFGETNT (type, Name=typename, $
                  Wavetype=wavetype)
IF (status EQ FAIL) THEN $
    MESSAGE, /Continue, 'HDFGETNT failed.'
IF (!Hdf_debug GE 1) THEN $
    MESSAGE, /Continue, 'Data type is:' + $
            typename + '(' + wavetype + ')'
arg_size = SIZE (maxvalue)
arg_type = arg_size (arg_size(0) + 1)
IF (arg_type EQ 0) THEN BEGIN
    decl = 'maxvalue = ' + wavetype + '(0)'
    status = EXECUTE (decl)
ENDIF
```

## See Also

[EXECUTE](#), [HDFGETRANGE](#), [HDFSETNT](#)

Also refer to the following routine in the *HDF Reference Manual*:

DFSDGETNT

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## HDFGETR8 Function

Obtains an HDF Raster 8 image and associated palette.

### Usage

*status* = HDFGETR8 (*filename*, *image*, *palette*)

### Input Parameters

*filename* — The name of the HDF file.

### Output Parameters

*image* — A byte array which contains the obtained HDF Raster 8 image.

*palette* — A byte array (3-by-256) which contains the color palette associated with the HDF image. Use the HDNFLCT procedure to load the palette.

### Return Value

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

## Keywords

**Help** — If present and nonzero, lists the usage for this routine.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

## Discussion

HDFGETR8 obtains an HDF Raster 8 image from the named HDF file. HDFGETR8 will read the current (or first) Raster 8 image in the file and position the HDF active pointer to the next Raster 8 image. You can read all Raster 8 images in an HDF file using successive calls to HDFGETR8. The return status is set to FAIL when the function reads beyond the last Raster 8 image.

You can load the HDF palette associated with the image into PV-WAVE directly using the HDFLCT procedure.

Use DFR8READREF or DFR8RESTART to position the active pointer within the HDF file.

## Example

```
testfile = 'raster8.hdf'
status = DFR8RESTART()

IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed at DFR8restart for'+ ' HDFGETR8.'
status = HDFGETR8 (testfile, image, palette)

IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFGETR8 failed.'

TV, image
HDFLCT, palette
```

## See Also

[HDFGET24](#), [HDFLCT](#), [HDFPUTR8](#)

Also refer to the following routines in the *HDF Reference Manual*:

DFR8GETDIMS, DFR8GETIMAGE, DFR8NIMAGES,  
DFR8READREF, DFR8RESTART, HISHDF

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## **HDFGETRANGE Function**

Gets the maximum and minimum range for the current HDF Scientific Data Set.

### **Usage**

*status* = HDFGETRANGE (*maxvalue*, *minvalue*)

### **Output Parameters**

*maxvalue* — The maximum data value for the HDF Scientific Data Set.

*minvalue* — The minimum data value for the HDF Scientific Data Set.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

### **Keywords**

*Help* — If present and nonzero, lists the usage for this routine.

*Usage* — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

### **Discussion**

The maximum and minimum data values for the current Scientific Data Set are obtained.

The data type of *maxvalue* and *minvalue* is determined by the numeric type returned by HDFGETNT.

## Example

```
status = HDFGETRANGE(min, max)
IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFGETRANGE failed.'
```

## See Also

[HDFGETNT](#), [HDFGETSDS](#)

Also refer to the following routine in the *HDF Reference Manual*:

DFSDGETRANGE

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## ***HDFGETSDS Function***

Gets an HDF Scientific Data Set.

### **Usage**

*status* = HDFGETSDS (*filename*, *data*)

### **Input Parameters**

*filename* — The name of the HDF file.

### **Output Parameters**

*data* — An array containing values from the Scientific Data Set. The data type and dimensions for data are automatically set using calls to DFSDGETNT and DFSDGETDIMS.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

## Keywords

**Help** — If present and nonzero, lists the usage for this routine.

**Maxrank** — Maximum number of dimensions expected with this dataset. The default value is 10.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

## Discussion

HDFGETSDS obtains the current Scientific Data Set in an HDF file and positions the active pointer to the next SDS. This function is preferred for getting Scientific Data Sets, as it automatically dimensions and casts a PV=WAVE array prior to obtaining the actual data.

## Example

```
data = 0
status = DFSDrestart ()
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed DFSDrestart for'+ ' HDFGETSDS.'
status = HDFGETSDS (testfile, data, Maxrank=100)
IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFGETSDS failed.'
INFO, data
```

## See Also

### [HDFGETNT](#)

Also refer to the following routines in the *HDF Reference Manual*:

DFSDGETDATA, DFSDGETDIMS, DFSDLASTREF

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, \*Functional Summary of Routines\*](#).

---

## **HDFLCT Procedure**

Loads an HDF palette as a PV-WAVE color table.

### **Usage**

HDFLCT, *palette*

### **Input Parameters**

*palette* — A byte array containing the HDF palette to load into PV-WAVE.

### **Keywords**

*Help* — If present and nonzero, lists the usage for this routine.

*Usage* — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

### **Discussion**

HDFLCT allows direct loading of an HDF palette into PV-WAVE when displaying HDF Raster 8 images.

### **Example**

```
status = HDFGETR8 (testfile, image, palette)
IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFGETR8 failed.'
TV, image
HDFLCT, palette
```

### **See Also**

[HDFGETR8](#), [TVLCT](#)

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, \*The PV-WAVE HDF Interface\*](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, \*Functional Summary of Routines\*](#).

---

## **HDFPUT24 Function**

Puts an HDF Raster 24 image into an HDF file.

### **Usage**

*status* = HDFPUT24 (*filename*, *image*)

### **Input Parameters**

*filename* — A string containing the name of the HDF file.

*image* — A byte array containing the 24 bit image to write to the file.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

### **Keywords**

***Append*** — If nonzero, append the image to the end of the HDF file. Otherwise, the HDF file is rewritten to contain only the image provided. The default is 0, do not append the image to the file, overwrite file contents.

***Help*** — If present and nonzero, lists the usage for this routine.

***Interlace*** — Set the HDF interlace scheme to use in writing the image. Possible values include:

DFIL\_PIXEL  
DFIL\_LINE  
DFIL\_PLANE

The default interlace scheme used will be DFIL\_PIXEL.

---

**NOTE** The interlace should match the dimensions of the image provided. Refer to “Writing 24-Bit Raster Images to a File” in the *NCSA HDF Calling Interfaces and Utilities* manual for details.

---

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the Help keyword.)

## Discussion

HDFPUT24 writes 24 bit images to an HDF file. The *Append* keyword lets you write additional 24 bit images the same file.

## Example

```
testmonkey = 'testmandril24.hdf'
status = HDFPUT24 (testmonkey, smallimg)
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed to write the small'+ $
    ' monkey.'
status = HDFGET24 (testmonkey, bigimg, $
    /Append)
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed to write the big monkey.'
```

## See Also

[HDFGET24](#), [HDFPUTR8](#), [HDFPUTSDS](#)

Also refer to the following routines in the *HDF Reference Manual*:

DF24ADDIMAGE, DF24PUTIMAGE, DF24SETIL

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## **HDFPUTFILEANN Function**

Inserts HDF file labels and file descriptions (annotations) into a file.

### **Usage**

*status* = HDFPUTFILEANN (*filename*)

### **Input Parameters**

*filename* — A string containing the name of the HDF file.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

### **Keywords**

**Description** — A byte array that contains the description for the specified file. This byte array may require further processing later for display of the description.

**Help** — If present and nonzero, lists the usage for this routine.

**Label** — A string variable that contains the label for the specified file.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

### **Discussion**

Multiple calls to HDFPUTFILEANN will cause additional labels/descriptions to be added in the file. There is no way known to overwrite an existing file label/description, other than to start writing a new file.

### **Example 1**

```
label = 'Put File Label (ID) Test'  
status = HDFPUTFILEANN (testfile, Label=label)  
IF (status EQ FAIL) THEN $  
    MESSAGE, 'Failed HDFPUTFILEANN with Label.'
```

## Example 2

```
desc = BYTE ('Put File Description Test')
status = HDFPUTFILEANN (testfile, $
    Description=desc)
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed HDFPUTFILEANN with'+ $
    ' Description.'
```

## See Also

### [HDFGETFILEANN](#)

Refer to the following routines in the *HDF Reference Manual*:

DFANADDFID, DFANADDFDS, HCLOSE, HOPEN

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## **HDFPUTR8 Function**

Writes an 8 bit image to an HDF file.

### **Usage**

*status* = HDFPUTR8 (*filename*, *image*)

### **Input Parameters**

*filename* — A string containing the name of the HDF file.

*image* — A byte array containing the 8 bit image to write to the file.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)                      Indicates success.

FAIL (-1)                      Indicates failure.

## Keywords

**Append** — If present and nonzero, append the image to the end of the HDF file. By default, the image is not appended and the contents of the file are overwritten.

**Compression** — Defines the compression scheme to use when writing the image. Possible values are:

0 — No compression

DFTAG\_RLE — Run length encoding (RLE)

DFTAG\_IMCOMP — IMCOMP compression

The default value is 0 (no compression is used).

**Help** — If present and nonzero, lists the usage for this routine.

**Palette** — A byte array containing an HDF palette to write with the image. This palette is pixel interlaced (r, g, b, r, g, b, ...) and cannot be a standard PV-WAVE color table.

**Usage** — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

## Discussion

HDFPUTR8 writes an 8 bit image and associated palette to an HDF file. The *Append* keyword lets you write additional 8 bit images to the same file.

## Example

```
status = HDFPUTR8 (testfile, smallimage)
IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFPUTR8 failed.'
status = HDFPUTR8 (testfile, image, /Append)
If (status EQ FAIL) THEN $
    MESSAGE, 'HDFPUTR8 failed.'
```

## See Also

[HDFGETR8](#), [HDFPUT24](#), [HDFPUTSDS](#)

Also refer to the following routines in the *HDF Reference Manual*:

DFR8ADDIMAGE, DFR8PUTIMAGE, DFR8SETPALETTE

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## HDFPUTSDS Function

Writes a Scientific Data Set to an HDF file.

### Usage

*status* = HDFPUTSDS (*filename*, *data*)

### Input Parameters

*filename* — A string containing the name of the HDF file.

*data* — An array containing values for the Scientific Data Set. The keyword *data* may be of any PV-WAVE numeric data type or dimension.

### Return Value

*status* — The status of the function call, where:

SUCCEED (0)                      Indicates success.

FAIL (-1)                         Indicates failure.

### Keywords

*Append* — If present and nonzero, append the image to the end of the HDF file. By default, the image is not appended and the contents of the file are overwritten.

*Help* — If present and nonzero, lists the usage for this routine.

*Usage* — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

## Discussion

HDFPUTSDS is used to write an arbitrary array of data to an HDF file as a Scientific Data Set. The numeric data type and dimensions of the Scientific Data Set are based on the numeric data type and dimensions of the array passed. The *Append* keyword writes additional Scientific Data Sets to the HDF file.

## Example 1

```
testfile = 'testsdsgen.hdf'
data = BINDGEN (11, 7, 5, 3)
status = HDFPUTSDS (testfile, data)
IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFPUTSDS failed for BYTE data.'
```

## Example 2

```
data = INDGEN (6, 8, 7)
status = HDFPUTSDS (testfile, data, /Append)
IF (status EQ FAIL) THEN $
    MESSAGE, 'HDFPUTSDS failed for INTEGER'+ ' data.'
```

## See Also

[HDFGETSDS](#), [HDFPUT24](#), [HDFPUTR8](#), [HDFSETNT](#)

Also refer to the following routines in the *HDF Reference Manual*:

[DFSDADDDATA](#), [DFSDPUTDATA](#), [DFSDSETDIMS](#)

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## HDFSCAN Procedure

Scans an HDF file and prints a simple list of file contents by HDF object type.

### Usage

HDFSCAN, *filename*

### Input Parameters

*filename* — A string containing the name of the HDF file.

### Keywords

*Help* — If present and nonzero, lists the usage for this routine.

*Usage* — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

### Discussion

HDFSCAN is a simple HDF file reader that examines an HDF file for Raster 8 images, HDF Palettes, Raster 24 images, and Scientific Data Sets.

---

**TIP** For cleaner output, run HDFSCAN with !HDF\_debug = -1.

---

### Example

```
HDFSCAN, 'testfile.hdf'
```

### See Also

[HDFGETNT](#)

Also refer to the following routines in the *HDF Reference Manual*:

DFPGETPAL, DFPLASTREF, DFPNPALS, DFPRESTART, DF24GETDIMS, DF24LASTREF, DF24RESTART, DFR8GETDIMS, DFR8LASTREF, DFR8NIMAGES, DFR8RESTART, DFSGETDIMS, DFSDLASTREF, DFSDRESTART, HCLOSE, HGETLIBVERSION, HISHDF, HOPEN

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## **HDFSETNT Function**

Computes and sets the HDF number type (i.e., data type) and descriptive number type string for the specified data array.

### **Usage**

*status* = HDFSETNT (*data*)

### **Input Parameters**

*data* — An array containing data for a Scientific Data Set. Must be a numeric PV-WAVE data type.

### **Return Value**

*status* — The status of the function call, where:

SUCCEED (0)	Indicates success.
FAIL (-1)	Indicates failure.

### **Keywords**

*Help* — If present and nonzero, lists the usage for this routine.

*Name* — HDF SDS numeric data type as a string, e.g., “DFNT\_FLOAT32”. This keyword is more for human use than programmatic use.

*Type* — HDF Scientific Data Set numeric data type. The range of possible values are defined in HDF\_COMMON:

DFNT\_FLOAT32  
DFNT\_FLOAT64  
DFNT\_INT8  
DFNT\_UINT8  
DFNT\_INT16  
DFNT\_UINT16  
DFNT\_INT32  
DFNT\_UINT32

*Usage* — If present and nonzero, lists the usage for this routine. (Same as the *Help* keyword.)

## Discussion

HDFSETNT gets the current HDF Scientific Data Set numeric type (data type) for the specified data array (*data*). Additionally, the SDS numeric type is set using DFSDSETNT. Keywords are optionally set if you want to explicitly know the HDF SDS numeric type set as an INTEGER (*type*) or a string (*name*).

## Example

```
testfile = 'testsdset.hdf'
data = INDGEN (6, 8, 7)
status = HDFSETNT (data)
IF (status EQ FAIL) THEN $
    MESSAGE, 'Failed HDFSETNT for SDS '+ $
    ' set annotation test.'
```

## See Also

### [HDFGETNT](#)

Also refer to the following routine in the *HDF Reference Manual*:

### DFSDSETNT

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## ***HDF\_TEST Procedure***

Runs the PV-WAVE HDF test suite.

### **Usage**

HDF\_TEST

### **Input Parameters**

None.

## Keywords

*No\_display* — Prevents visual output from being displayed.

## Discussion

HDF\_TEST runs the PV-WAVE HDF test suite after first changing to the directory containing the tests before starting PV-WAVE. The procedure determines the path to the test directory, changes to that directory, runs the tests, and changes back to the original working directory.

## Example

```
@hdf_startup
    PV-WAVE:HDF 3.30 Module Initialized

hdf_test
    ; Test output not shown.
```

## See Also

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, \*The PV-WAVE HDF Interface\*](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, \*Functional Summary of Routines\*](#).

---

## HELP Procedure

Starts the online help system or the online documentation system.

## Usage

HELP [, *topic*]

## Parameters

*topic* — (optional) A string containing the name of a PV-WAVE command.

## Keywords

**Contents** — If present and nonzero, displays the Contents topic of the help file.

**Documentation** — If present and nonzero, starts the online documentation system (Manuals Online) instead of online help. This system contains the complete text and graphics of the entire PV-WAVE documentation set.

**Filename** — Specifies a string containing the name of a help file to load, if other than the default help file. See the *Discussion* section for information on the default help file.

**Help** — If present and nonzero, displays the topic **How to Use Help** in the help viewer.

**Index** — If present and nonzero, displays the Contents topic of the help file (same as the *Contents* keyword).

**Keyword** — Specifies a string containing the name of a help keyword. The first help topic matching the keyword is displayed.

**PartialKey** — Specifies a string containing a partial keyword. The first help topic containing the partial keyword is displayed.

**Quit** — Exits the online help viewer.

## Discussion

The PV-WAVE online help system uses the Bristol Hyperhelp™ viewer to display detailed information on PV-WAVE commands. Use the HELP command to start the help viewer and display a topic. For information on how to use the online help system, select **How to Use Help** from the **Help** menu of the viewer.

The HELP procedure checks to see if the help viewer is already running. If it is running, the specified topic is displayed in the viewer. If it is not running, the viewer is started and the topic is displayed.

---

**NOTE** On Microsoft Windows systems, the Windows help viewer (WINHELP) is used.

---

The main PV-WAVE online help file (the file that is loaded into the Hyperhelp™ viewer by default) is:

(UNIX) <vndir>/hyperhelp/wave.hlp

(OpenVMS) <vndir>:[HYPERHELP] WAVE.HLP

Where <vndir> is the main product installation directory.

(Windows) <wavedir>\help\wavewin.hlp

Where <wavedir> is the main PV-WAVE directory.

## Examples

The following commands entered at the WAVE> prompt demonstrate some of the features of the HELP command.

```
HELP, 'REBIN'
```

; Displays documentation for the REBIN command in the online help viewer.

```
HELP, Filename='vdatools.hlp', /Contents
```

; Loads the help file vdatools.hlp and displays the Contents topic.

```
HELP, /Documentation
```

; Starts the online documentation system (Manuals Online).

## See Also

[INFO](#)

---

## HILBERT Function

Standard Library function that constructs a Hilbert transformation matrix.

### Usage

```
result = HILBERT(x [, d])
```

### Input Parameters

**x** — The vector to be transformed. Can be of either floating-point or complex data type, and can contain any number of elements.

**d** — (optional) A flag to indicate the direction of rotation:

+1 Shifts the vector +90 degrees.

-1 Shifts the vector -90 degrees.

## Returned Value

*result* — The value of the Hilbert transform of  $x$ . Result is of a complex data type, with the same dimensions as  $x$ .

## Keywords

None.

## Discussion

A Hilbert transform is a series of numbers in which all periodic components have been phase-shifted by 90 degrees. Angle shifting is accomplished by multiplying or dividing by the complex number  $i = (0.000, 1.000)$ .

A Hilbert series has the interesting property that the correlation between it and its own Hilbert transform is mathematically zero.

The HILBERT function generates a Hilbert matrix by generating the Fast Fourier Transform of the data with the FFT function and shifting the first half of the transform products by +90 degrees and the second half by -90 degrees. The constant elements of the transform are not changed.

The shifted vector is then submitted to the FFT function for the transformation back to the “time” domain. Before it is returned, the output is divided by the number of elements in the vector to correct for the multiplication effect characteristic of the FFT algorithm.

## Example

```
a = FINDGEN(1000)
sine_wave = SIN(a/(MAX(a)/(2 * !pi)))
; Create a sine wave.

PLOT, sine_wave
; Plot the sine wave.

OPLOT, HILBERT(sine_wave, -1)
; Plot the sine wave phase-shifted to the right by 90 degrees.

rand = RANDOMN(seed, 1000) * 0.05
; Create an array of random numbers to mimic a noisy signal.

PLOT, rand
; Plot the random numbers.

sandwich = [sine_wave, rand, sine_wave]
; Sandwich the random data between two sine waves.
```

```
PLOT, sandwich, XStyle=1
; Plot the two sine waves with the random noise in the middle, thereby
; turning them into a single signal.

OPLOT, HILBERT(sandwich, -1)
; Plot the sandwiched wave forms. Note that the sine waves are
; phase-shifted to the right by 90 degrees, while the noise data has
; not shifted at all, but rather has been distorted vertically (its
; amplitude) by the effect of the two adjacent phase-shifted sine
; waves. This is because the sine waves and the noise data were set
; up to be a single signal.
```

## See Also

[FFT](#), [HANNING](#)

---

## HIST\_EQUAL Function

Standard Library function that returns a histogram-equalized image or vector.

### Usage

```
result = HIST_EQUAL(image)
```

### Input Parameters

*image* — The image to be equalized.

### Returned Value

*result* — An array that has been histogram equalized.

### Keywords

**Binsize** — The size of the bin, i.e., the number of elements to consider as having a single value. If not specified, a value of 1 is used.

**Maxv** — The maximum value to be used. If not specified, the largest value of the elements in *image* is used. Input elements greater than *max* are output as 255.

**Minv** — The minimum value to be used. Should be greater than 0. All input elements in *image* less than or equal to *min* will be output as 0. If not specified, 0 is used.

*Top* — If specified, scales the result from 0 to *Top* before it is returned.

## Discussion

In many images, most pixels reside in a few small subranges of the possible values. By spreading the distribution so that each range of pixel values contains an approximately equal number of members, the information content of the display is maximized.

To equalize the histogram of display values, the count-intensity histogram of the image is required. This is a vector in which the *i*th element contains the number of pixels with an intensity equal to the minimum pixel value of the image plus *i*. The vector is of long integer type and has one more element than the difference between the maximum and minimum values in the *image*. (This assumes a *Binsize* of 1 and an *image* that is not of byte type.) The sum of all the elements in the vector is equal to the number of pixels in the image.

HIST\_EQUAL uses the HISTOGRAM function to obtain the density distribution of the image. This distribution is integrated to obtain the cumulative density probability function. Finally, the distribution is normalized so that its maximum element has a value of 255.

If *image* is of floating-point data type, its range of values should be at least 255, unless the *Binsize* keyword is used. If *image* is of byte data type, any *Binsize* keyword is ignored.

## Sample Usage

Histogram equalization is commonly used in medical photography and X-rays. It causes the image gray levels that have the most pixels to be allocated the most display levels, thereby maximizing the transfer of information from an image.

Unfortunately, it is this very effect that can sometimes cause unsatisfactory results—histogram equalization chooses a display map-ping based on the area covered by the various features in the image, rather than their importance. This can cause the contrast enhancement of reconstruction artifacts in the large background area, while small features of medical interest are sacrificed.

## Example

This example uses the HIST\_EQUAL function to manipulate the whirlpool image found in:

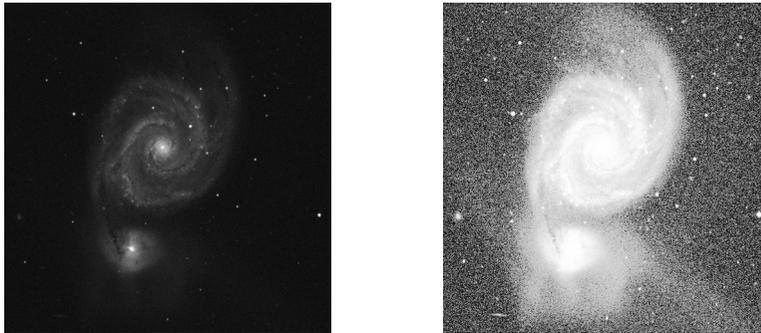
```
(UNIX)      <wavedir>/data
```

(OpenVMS) <wavedir>:[DATA]

(Windows) <wavedir>\data

Where <wavedir> is the main PV-WAVE directory.

The commands shown in this example produce the image on the right in :



**Figure 2-24** The HIST\_EQUAL function has been used to make the visual elements of this 512-by-512 galaxy image more pronounced.

```
whirlpool = BYTARR(512,512)
GET_LUN, unit
filename = 'whirlpool.img'
OPENR, unit, filename
READU, unit, whirlpool
CLOSE, unit
FREE_LUN, unit
    ; Create a 512-by-512 byte array, get the next free logical unit
    ; (LUN), open the file, read the image into the array, and free the LUN.

!Order = 1
    ; Transfer the image from top to bottom.

WINDOW, 2, XSize=500, YSize=500
TVSCL, whirlpool
    ; Open the window and display the image.

WINDOW, 0, XSize=500, YSize=500
image = HIST_EQUAL(whirlpool)
TVSCL, HIST_EQUAL(image)
    ; Open another window and display the histogram-equalized image.
```

## See Also

[HIST\\_EQUAL\\_CT](#), [HISTOGRAM](#)

For more information, see the section *Histogram Equalization* in Chapter 6 of the *PV-WAVE User's Guide*.

---

## ***HIST\_EQUAL\_CT Procedure***

Standard Library procedure that uses an input image parameter, or the region of the display you mark, to obtain a pixel distribution histogram. The cumulative integral is taken and scaled, and the result is applied to the current color table.

### Usage

`HIST_EQUAL_CT [, image]`

### Input Parameters

*image* — (optional) The image whose histogram is to be used in determining the new color tables:

- If *image* is supplied, it is assumed to be the image that was last loaded to the display.
- If *image* is omitted, you are prompted to mark the diagonal corners of a region of the display with the mouse. The *image* must be a byte image, scaled the same way as the image loaded to the display.

### Keywords

None.

### Example

This is an example of how to obtain a pixel distribution histogram of a displayed image. It uses the aerial view of Boulder, Colorado in:

**(UNIX)**      `<wavedir>/data`

**(OpenVMS)** `<wavedir>:[DATA]`

**(Windows)** `<wavedir>\data`

Where `<wavedir>` is the main PV-WAVE directory.

The result is applied to the current color table using the *image* parameter. Note that this example will only work if the original image contains values in the range of 0 to 255.

```
aerial_view = BYTARR(512,512)
GET_LUN, unit

OPENR, unit, 'aerial_demo.img'
    ; Create a 512-by-512 byte array, get the next free logical unit number
    ; (LUN), and open the file.

READU, unit, aerial_view
CLOSE, unit
FREE_LUN, unit
    ; Read the data into the array aerial_view, close the file, and free the LUN.

WINDOW, 1, XSize=512, YSize=512, $
    title='Aerial View of Boulder, CO'
TVSCL, aerial_view
    ; Open the window and display the image.

HIST_EQUAL_CT, aerial_view
    ; Load the color table with a histogram-equalized distribution.
```

## See Also

[HIST\\_EQUAL](#), [HISTOGRAM](#)

For more information, see the section *Histogram Equalization* in Chapter 6 of the *PV-WAVE User's Guide*.

---

## HISTN Function

Standard Library function that computes an n dimensional histogram.

### Usage

```
result = HISTN(d [, axes])
```

### Input Parameters

*d* — An (m,n) array of m data points in n-space.

## Returned Value

**result** — An n dimensional array of size *binnum*.

**axes** = (optional - use with */scale*) an (n, *binnum*) array containing properly scaled axes with which to plot the results. For example:

```
CONTOUR, result, transpose(axes(0,*)), transpose(axes(1,*))
```

## Keywords

**Binnum** — The number of bins for the histogram.

**Binsize** — The size of bins for the histogram. The default = 1.

---

**NOTE** Only 1 of *binnum* or *binsize* can be set.

---

**/Scale** — If set, the result is scaled so to have unit volume under the curve/surface when plotted against *x.i/stdev(x.i)*

**/Compatible** — If set, the result will align with HISTOGRAM. The default behavior of HISTN is “binnum-central” logic, while the default behavior of HISTOGRAM is “binsize-central” logic. Setting */compatible* will force HISTN to be “binsize-central.”

---

**NOTE** When using */compatible* with 2D arrays and setting *binsize* manually, you may see poor results if the *binsize* is not appropriate for all variables. In this case, you should either set *binnum* or not use */compatible*.

---

## Examples

Interpreting an n-dimensional histogram, a 2D example.

Consider two sets of 10 random numbers. If one computes 1D histograms with 3 bins, you may find these results (number of items in each bin):

```
x = RANDOMN(s,10)    &  y = RANDOMN(s,10)
xy = FLTARR(10,2)    &  xy(*,0) = x        &  xy(*,1) = y
PRINT, HISTN(x, binnum=3)
           4           5           1
PRINT, HISTN(y, binnum=3)
           3           4           4
```

The result for the 2D histogram may give:

```

PRINT, HISTN(xy, binnum=3)
      1      1      1
      2      2      0
      1      2      0

```

Summing this result vertically yields the 1D result for  $x$ . Summing this result horizontally yields the 1D result for  $y$ . To plot a probability distribution function with unit volume under the surface, one would call HISTN with the /Scaled keyword and plot the results against  $x/\text{std}(x)$  and  $y/\text{STD}(y)$ . Using the *axes* output parameter with HISTN returns these vectors in a 2D array.

```

h2 = HISTN(xy, axes, binnum=3, /scaled)
shade_surf, h2, axes(0,*), axes(1,*)

```

## See Also

[HISTOGRAM](#)

## HISTOGRAM Function

Returns the density function of an array.

### Usage

```
result = HISTOGRAM(array)
```

### Input Parameters

*array* — The array for which the density function will be computed. The size of each dimension of *array* may be any integer value.

### Returned Value

*result* — A longword vector equal to the density function of the input *array*.

### Keywords

*Armax* — An array of the maximum values to consider for each image or signal in *array*. If *Armax* is a single element array, *Armax*(0) is used for every image or signal. For a multi-image or multi-signal input array, *Armax* must contain the same number of elements as there are images or signals in *array*.

---

**NOTE** When both *Armin* and *Armax* are supplied, they must contain the same number of elements.

---

**Armin** — An array of the minimum values to consider for each image or signal in *array*. If *Armin* is a single element array, *Armin*(0) is used for every image or signal. For a multi-image or multi-signal input array, *Armin* must contain the same number of elements as there are images or signals in the input array.

**Binsize** — The range of values to consider as having a single value. If no value is specified, the *Binsize* range defaults to a value of 1.

**Intleave** — (If used, requires the *Armin* and *Armax* keywords.) A scalar string indicating the type of interleaving of 2D input signals containing signal-interleaved signals; and 3D input arrays containing image-interleaved images, or a volume. Valid strings and the corresponding interleaving methods are:

'signal' — The 2D input *image* array arrangement is  $(x, p)$  for  $p$  signal-interleaved signals of length  $x$ .

'image' — The 3D *image* array arrangement is  $(x, y, p)$  for  $p$  image-interleaved images of  $x$ -by- $y$ .

'volume' — The input image array is treated as a single entity.

**Max** — The maximum value to consider. If set but no value is specified, *array* is searched for its largest value. If the *Max* keyword is specified, its value is used for every signal or image in the array.

**Min** — The minimum value to consider. If set but no value is specified, *array* is searched for its smallest value. If the *Min* keyword is specified, its value is used for every signal or image in the array.

**Omax** — Specifies a variable used to hold the maximum value considered in the array. (Equal to *max* when the *Max* keyword is given.) If the input array is composed of multiple images or signals, *Omax* is an array of the maximum values, one for each signal or image. (Equal to *max* or *armax* when *Max* or *Armax* is given.)

**Omin** — Specifies a variable to hold the minimum value considered in the array. (Equal to *min* when *Min* is given.) If the input array is composed of multiple images or signals, *Omin* will be an array of the minimum values, one for each signal or image. (Equal to *min* or *armin* when *Min* or *Armin* is given.)

## Discussion

The HISTOGRAM function supports input arrays composed of multiple images (multi-layer band interleaved images) as well as input arrays composed of multiple signals. The *Intleave* keyword is used to specify whether that the input array is a multi-signal or multi-image array. When the *Intleave* keyword is used to indicate multiple signals or images in this way, each signal or image in the array is operated on separately and an array of the individual results is returned.

In the simplest case (in which *array* ranges in value from 0 to some maximum value), the value of the density function at subscript *i* is equal to the number of array elements with a value of *i*.

For example, let  $F_i$  equal the value of element *i*, for *i* in the range {0 ... *n*-1}. Then  $H_v$  (the result of the HISTOGRAM function) is given by:

$$H_v = \sum_{i=0}^{n-1} P(F_i, v)$$

where

$$v = 0, 1, 2, \dots, \left[ \frac{Max - Min}{Binsize} \right]$$

and  $P(F_i, v) = 1$  when

$$v \leq (F_i - Min) / (Binsize < v + 1)$$

and  $P(F_i, v) = 0$  otherwise.

---

**NOTE** There may not always be enough virtual memory available to create density functions of arrays that contain a large number of bins. For information on virtual memory and PV-WAVE, see the section *Tips for Efficient Programming* in Chapter 11 of the *PV-WAVE Programmer's Guide*.

---

## Sample Usage

Histograms are useful in a variety of applications, and can often provide signs as to what type of image processing should be performed. For example, photos sent back via satellite from outer space are usually accompanied by histograms. If the histogram for a photo contains a large spike, and the rest of the histogram is generally flat, this typically indicates that a histogram equalizing operation (such as the HIST\_EQUAL function) is needed. Such an operation would spread out the pixels more evenly, thereby improving contrast and bringing out greater detail in the image.

Histograms can also be used to provide clues about images. For example, running a histogram on a series of identical photos taken at different times of the day may show the histogram peak shifting to the right—an indication that the average brightness is higher in that photo, and therefore more likely to have been taken at a sunnier part of the day.

Similarly, you can use histograms to compare two images of the same scene more fairly. By shifting the histogram of one scene so that it is aligned with that of the other scene, you can equalize the level of brightness in both images.

### Example 1

The data for this example is from Hinkley (1977) and Velleman and Houglin (1981). Data includes measurements (in inches) of precipitation in Minneapolis/St. Paul during the month of March for 30 consecutive years.

```
x = [0.77, 1.74, 0.81, 1.20, 1.95, 1.20, $
      0.47, 1.43, 3.37, 2.20, 3.00, 3.09, 1.51, $
      2.10, 0.52, 1.62, 1.31, 0.32, 0.59, 0.81, $
      2.81, 1.87, 1.18, 1.35, 4.75, 2.48, 0.96, $
      1.89, 0.90, 2.05]
```

```
table = HISTOGRAM(x, Binsize = 0.444)
; Call HISTOGRAM.
```

```
PRINT, '   Bin Number   Count' &$
PRINT, '   - - - - -   - - - -' &$
FOR i = 1, 10 DO PRINT, i, table(i-1)
```

```
Bin Number   Count
- - - - -   - - - -
      1         4
      2         8
      3         5
      4         5
```

5	3
6	1
7	3
8	0
9	0
10	1

## Example 2

This example exhibits how histogram equalization of an image better distributes pixel values. An image of a galaxy is read and displayed, then the density function of that image is plotted. After the image is histogram equalized, it is displayed, along with a plot of the density function of the equalized image.

```
OPENR, unit, FILEPATH('whirlpool.img', $
    Subdir = 'data'), /Get_Lun
    ; Open the file containing the galaxy image.

g = BYTARR(512, 512)
    ; Create an array large enough to hold the image.

READU, unit, g
    ; Read the image.

FREE_LUN, unit
    ; Close the file and free the file unit number.

!Order = 1

WINDOW, 0, Xsize = 512, Ysize = 512
    ; Create a window with the same dimensions as the image.

TVSCL, g
    ; Display the original image.
```

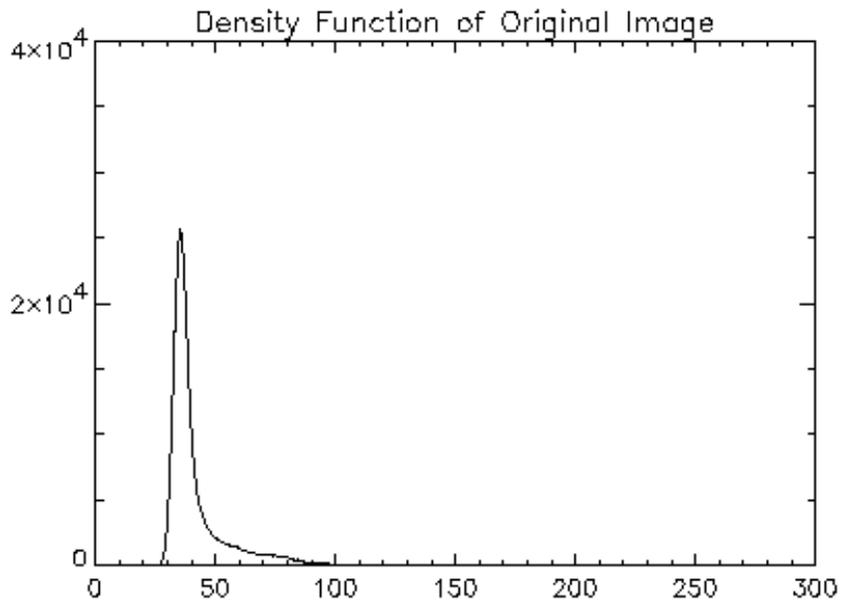


**Figure 2-25** Original galaxy image.

```
hist_g = HISTOGRAM(g)
        ; Compute the density function of the image.

WINDOW, 1, Xsize = 512, Ysize = 512
        ; Create a window to display a plot of the result of the density function
        ; of the image.

PLOT, hist_g, Xrange = [20, 100], $
      Yrange = [0, 30000], Title = $
      'Density Function of Original Image'
        ; Plot the result of the density function of the image.
```



**Figure 2-26** Plot of density function of original galaxy image.

```
g2 = HIST_EQUAL(g)
      ; Histogram equalize the galaxy image.

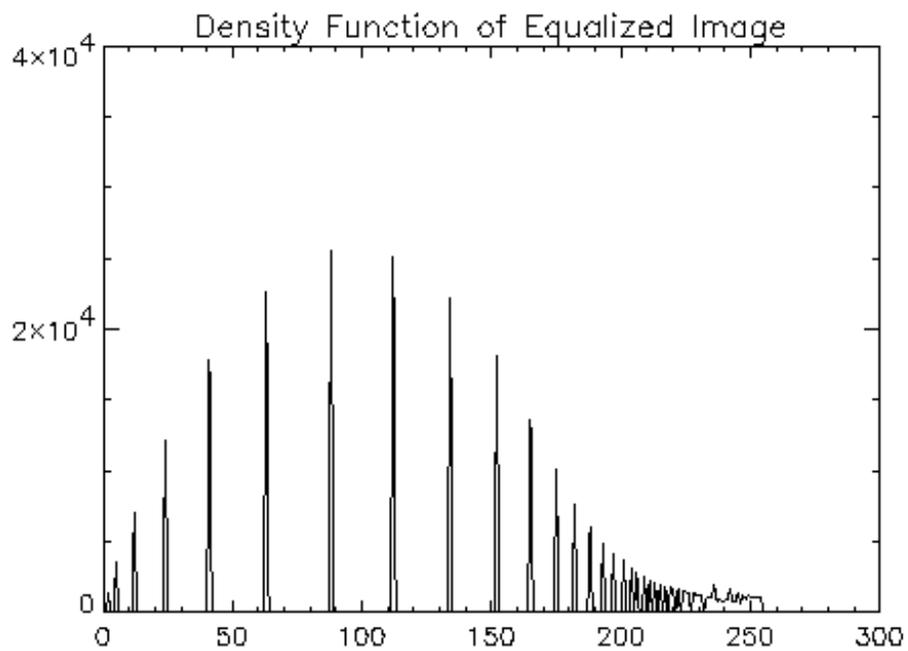
WINDOW, 2, Xsize = 512, Ysize = 512
      ; Create a window with the same dimensions as the histogram equalized image.

TVSCL, g2
      ; Display the equalized image.
```



**Figure 2-27** Histogram equalized galaxy image.

```
hist_g2 = HISTOGRAM(g2)
    ; Compute the density function of the equalized image.
WINDOW, 3, Xsize = 512, Ysize = 512
    ; Create a window to display a plot of the result of the density
    ; function of the equalized image.
PLOT, hist_g2, Xrange = [20, 100], $
    Yrange = [0, 30000], Title = $
    'Density Function of Equalized Image'
    ; Plot the result of the density function of the equalized image.
```



**Figure 2-28** Plot of density function of equalized galaxy image.

**See Also**

[HIST\\_EQUAL](#), [HIST\\_EQUAL\\_CT](#), [HIST\\_EQUAL\\_CT](#)

---

## HLS Procedure

Standard Library procedure that generates and loads color tables into an image display device based on the HLS color system. The resulting color table is loaded into the display system.

### Usage

HLS, *ltlo*, *lthi*, *stlo*, *sthi*, *hue*, *lp* [, *rgb*]

### Input Parameters

*ltlo* — The starting color lightness or intensity, expressed as 0 to 100 percent. Full lightness (the brightest color) is 100 percent.

*lthi* — The ending color lightness or intensity, expressed as 0 to 100 percent.

*stlo* — The starting color saturation, expressed as 0 to 100 percent. Full saturation (undiluted or pure color) is expressed as 100 percent.

*sthi* — The ending color saturation, expressed as 0 to 100 percent.

*hue* — The starting hue, expressed as 0 to 360 degrees.

*lp* — The number of loops around the color cone. The value may be floating-point. A positive value will traverse the color cone in a clockwise direction; a negative value will traverse the color cone in a counterclockwise direction.

### Output Parameters

*rgb* — (optional) A 256-by-3 integer output array containing the red, green, and blue vector values that were translated from the HSV system and loaded into the color tables. The following example shows the ordering of the RGB values in the output array:

```
Red_Vec(i) = RGB(i, 0)
Green_Vec(i) = RGB(i, 1)
Blue_Vec(i) = RGB(i, 2)
```

### Keywords

None.

## Discussion

The HLS procedure traces a spiral through the HLS color cone. Points along the spiral are converted from HLS values to RGB values and then loaded into the color tables with the TVLCT procedure. The color representation of pixel values between 0 and 255 is linearly interpolated from the hue, saturation, and lightness of the end points.

## Example

The statement:

```
HLS, 0, 100, 50, 100, 0, -2.5
```

loads a color table that ranges from 0 to 100 percent in lightness or intensity and from 50 to 100 percent in saturation. This color table begins with a color of red, and makes two and a half full loops around the color solid in the direction of red to blue.

## See Also

[COLOR\\_CONVERT](#), [COLOR\\_EDIT](#), [HSV](#), [LOADCT](#),  
[MODIFYCT](#), [RGB\\_TO\\_HSV](#), [TVLCT](#), [WgCeditTool](#), [WgCfTool](#)

For more information, see the section *The HSV and HLS Color Systems* in Chapter 11 of the *PV-WAVE User's Guide*.

The HLS procedure is adapted from a program in *Fundamentals of Interactive Computer Graphics* by Foley and Van Dam, Addison Wesley Publishing Company, Reading, MA, 1982.

---

## HSV Procedure

Standard Library procedure that generates and loads color tables into an image display device based on the HSV color system. The final color table is loaded into the display device.

### Usage

HSV, *vlo*, *vhi*, *stlo*, *sthi*, *hue*, *lp* [, *rgb*]

### Input Parameters

*vlo* — The starting color value or intensity, expressed as 0 to 100 percent. Full intensity is 100 percent.

*vhi* — The ending color value or intensity, expressed as 0 to 100 percent.

*stlo* — The starting color saturation, expressed as 0 to 100 percent. Full saturation (undiluted or pure color) is expressed as 100 percent.

*sthi* — The ending color saturation, expressed as 0 to 100 percent.

*hue* — The starting hue, expressed as 0 to 360 degrees.

*lp* — The number of loops around the color cone. The value may be floating-point. A positive value will traverse the color cone in a clockwise direction; a negative value will traverse the color cone in a counterclockwise direction.

### Output Parameters

*rgb* — (optional) A 256-by-3 integer output array containing the red, green, and blue vector values that were translated from the HSV system and loaded into the color tables. The following example shows the ordering of the RGB values in the output array:

```
Red_Vec(i) = RGB(i, 0)
```

```
Green_Vec(i) = RGB(i, 1)
```

```
Blue_Vec(i) = RGB(i, 2)
```

### Keywords

None.

## Discussion

The HSV procedure traces a spiral through the HSV color cone. Points along the spiral are converted from HSV values to RGB values and then loaded into the color tables with the TVLCT procedure. The color representation of pixel values between 0 and 255 is linearly interpolated from the hue, saturation, and value of the end points.

## Example

The statement:

```
HSV, 0, 100, 50, 100, 0, -2.5
```

loads a color table that ranges from 0 to 100 percent in intensity or brightness and from 50 to 100 percent in saturation. This color table begins with a color of red, and makes two and a half full loops around the color solid in the direction of red to blue.

## See Also

[COLOR\\_CONVERT](#), [COLOR\\_EDIT](#), [HLS](#), [LOADCT](#),  
[MODIFYCT](#), [RGB\\_TO\\_HSV](#), [TVLCT](#), [WgCeditTool](#), [WgCfTool](#)

For more information, see the section *The HSV and HLS Color Systems* in Chapter 11 of the *PV-WAVE User's Guide*.

The HSV procedure is adapted from a program in *Fundamentals of Interactive Computer Graphics* by Foley and Van Dam, Addison Wesley Publishing Company, Reading, MA, 1982.

---

## HSV\_TO\_RGB Procedure

Standard Library procedure that converts colors from the HSV color system to the RGB color system.

### Usage

```
HSV_TO_RGB, h, s, v, red, green, blue
```

### Input Parameters

*h* — The hue variable. May be either vector or scalar, in the range of 0 to 360.

*s* — The saturation variable. Must be the same dimension as *h*, in the range of 0 to 1.

*v* — The value variable. Must be the same dimension as *h*, in the range 0 to 1.

## Output Parameters

**red** — Red color output value(s). Will be short integer(s), the same dimension as *h*, and in the range of 0 to 255.

**green** — Green color output value(s). Will be short integer(s), the same dimension as *h*, and in the range of 0 to 255.

**blue** — Blue color output value(s). Will be short integer(s), the same dimension as *h*, and in the range of 0 to 255.

## Keywords

None.

## Discussion

HSV\_TO\_RGB provides a convenient way to convert from the HSV (hue, saturation, value) color system to the RGB (red, green, blue) color system. Most output devices capable of displaying color use the RGB color system.

## Example

The statement:

```
HSV_TO_RGB, 0, 1, 1, red, green, blue
```

returns red=255, green=0, blue=0. You can use the red, green, and blue values to set color table values with the TVLCT procedure.

## See Also

[COLOR\\_CONVERT](#), [COLOR\\_EDIT](#), [HLS](#), [HSV](#), [LOADCT](#),  
[MODIFYCT](#), [RGB\\_TO\\_HSV](#), [TVLCT](#), [WgCeditTool](#), [WgCtTool](#)

For more information, see the section *The HSV and HLS Color Systems* in Chapter 11 of the *PV-WAVE User's Guide*.

---

## HTML\_BLOCK Procedure

Writes out a specifically formatted block of HTML text.

### Usage

HTML\_BLOCK, *text*

### Input Parameters

*text* — An array of strings.

### Keywords

*BlockQuote* — Creates a <BLOCKQUOTE> text block (indented on both sides).

*Pre* — Creates a <PRE> block (pre-formatted).

*Safe* — Handles HTML special characters (see HTML\_SAFE).

*Tag* — Accepts a string indicating which HTML tag to use, allowing you to use any formatting tags (e.g., `tag="address"`). If *Tag* is set, it overrides the predefined keywords *Pre* and *BlockQuote*.

### Discussion

This procedure allows you to format blocks of text with specific HTML opening and closing tags. You may use the preset tag keywords (*BlockQuote* and *Pre*) or identify others using the *Tag* keyword. HTML\_BLOCK is particularly useful if your target browser uses nonstandard extensions.

### Example

This example creates a paragraph that will be indented on both sides when viewed.

```
HTML_OPEN
```

```
HTML_BLOCK, 'This paragraph will be ' + $
            'indented on both sides of the ' + $
            'viewer window. Use HTML_BLOCK ' + $
            'whenever you want to control ' + $
            'the appearance of a paragraph.', $
            /Blockquote
```

```
HTML_CLOSE
```

## See Also

[HTML\\_HEADING](#), [HTML\\_HIGHLIGHT](#), [HTML\\_LIST](#), [HTML\\_OPEN](#),  
[HTML\\_PARAGRAPH](#), [HTML\\_SAFE](#), [HTML\\_TABLE](#)

For a complete listing of HTML tag elements, see *HTML Sourcebook*, Second Edition, by Ian S. Graham, John Wiley & Sons, Inc., 1996, New York.

---

## ***HTML\_CLOSE Procedure***

Closes an HTML file, after end-tagging major elements.

### **Usage**

HTML\_CLOSE

### **Input Parameters**

None.

### **Keywords**

None.

### **Discussion**

You must always use HTML\_CLOSE when you finish creating an HTML file.

## See Also

[HTML\\_OPEN](#)

---

## **HTML\_HEADING Procedure**

Creates a heading with a level specification.

### **Usage**

HTML\_HEADING, *text*

### **Input Parameters**

*text* — The heading text.

### **Keywords**

*Center* — Centers the heading.

*Justify* — Fully justifies the heading to the left and right.

*Left* — Left-justifies the heading. (Default: set)

*Level* — Defines the heading level. (Default: 1)

*Right* — Right-justifies the heading.

*Safe* — Handles HTML special characters.

### **Discussion**

HTML allows six levels of headings, from 1 to 6. There is no forced hierarchy; in other words, you may use any heading at any time.

### **Example**

This example creates a centered, level-2 heading.

```
HTML_OPEN
HTML_HEADING,$
    'This <H2> heading will be ' + $
    'centered', /Center, /Safe, Level = 2
HTML_CLOSE
```

### **See Also**

[HTML\\_BLOCK](#), [HTML\\_HIGHLIGHT](#), [HTML\\_LIST](#),  
[HTML\\_OPEN](#), [HTML\\_PARAGRAPH](#), [HTML\\_RULE](#),  
[HTML\\_SAFE](#), [HTML\\_TABLE](#)

---

## HTML\_HIGHLIGHT Function

Allows the use of all the basic textual highlighting elements in HTML.

### Usage

```
html_text = HTML_HIGHLIGHT(str, tag)
```

### Input Parameters

*str* — The string or array of strings to be highlighted.

*tag* — The highlighting tag or array of tags. If *tag* is scalar but *str* is a list of strings, then *tag* is applied around each of the items in *str*. If *tag* is an array, it must be the same length as *str*. This parameter is the string inside the HTML angle brackets: e.g., B elicits <B>...</B> (for boldfacing).

### Keywords

*Safe* — Handles HTML special characters (see HTML\_SAFE).

### Returned Value

*html\_text* — A string (array) of HTML text tagged with highlighting tags in place.

### Discussion

HTML\_HIGHLIGHT supports both physical (e.g., ‘U’, ‘B’, etc.) and logical (e.g., ‘CITE’, ‘CODE’, etc.) text highlighting. The flexibility of the function also allows you to take advantage of any extension to standard formatting used by a particular browser.

### Example

```
HTML_OPEN
HTML_PARAGRAPH, $
    HTML_HIGHLIGHT('Example citation text.', $
        'CITE', /Safe)
HTML_CLOSE
```

## See Also

[HTML\\_BLOCK](#), [HTML\\_HEADING](#),  
[HTML\\_LIST](#), [HTML\\_OPEN](#), [HTML\\_PARAGRAPH](#),  
[HTML\\_SAFE](#), [HTML\\_TABLE](#)

For a complete listing of HTML text highlighting elements, see *HTML Sourcebook*, Second Edition, by Ian S. Graham, John Wiley & Sons, Inc., 1996, New York, p. 251.

---

## **HTML\_IMAGE Function**

Allows for the creation of an HTML image reference.

### **Usage**

*html\_img* = HTML\_IMAGE(*url*)

### **Input Parameters**

*url* — A string or array of strings containing URL(s) for images.

### **Keywords**

*Alt* — A string which is an alternate attribute used to replace the image in browsers unable to display graphics.

*Bottom* — Aligns the image at the bottom of its containing element.

*Left* — Aligns the image at the left of the browser window.

*Middle* — Aligns the image vertically in the middle of its containing element.

*Right* — Aligns the image at the right of the browser window.

*Top* — Aligns the image at the top of its containing element.

### **Returned Value**

*html\_img* — A string or an array of strings containing the HTML image code.

## Discussion

Coupled with PV-WAVE imaging, rendering, and visualization capabilities, the HTML\_IMAGE function allows you the flexibility of visualizing your data in an HTML document over the Internet.

For an example of how to create a GIF file from PV-WAVE for inclusion in an HTML document, see the following demonstration:

```
(UNIX)      <wavedir>/demo/web/html/html_demo.pro
(OpenVMS)  <wavedir>: [DEMO.WEB.HTML] HTML_DEMO.PRO
(Windows)  <wavedir>\demo\web\html\html_demo.pro
```

where <wavedir> is the main PV-WAVE directory.

---

**TIP** Although HTML\_IMAGE has many keywords enabling you to position the image within the containing element or browser window, you can also use the HTML\_HIGHLIGHT function to specify any HTML image alignment attribute that is not included as a keyword for HTML\_IMAGE. For example, the following line of code can be used to center an image:

```
HTML_HIGHLIGHT(HTML_IMAGE(url), 'CENTER')
```

---

## Example

This example centers the PV-WAVE logo in a browser window.

```
HTML_OPEN
HTML_BLOCK, HTML_HIGHLIGHT(HTML_IMAGE( $
    'wave_logo.gif'), 'CENTER')
HTML_CLOSE
```

## See Also

[HTML\\_BLOCK](#), [HTML\\_HIGHLIGHT](#),  
[HTML\\_LINK](#), [HTML\\_OPEN](#), [IMAGE\\_CREATE](#),  
[IMAGE\\_READ](#), [IMAGE\\_WRITE](#), [READ\\_XBM](#),  
[WRITE\\_XBM](#)

---

## HTML\_LINK Function

Sets up hypertext links to Uniform Resource Locations (URLs).

### Usage

*link\_text* = HTML\_LINK(*url*, *text*)

### Input Parameters

*url* — A string or string array specifying the URL of the hypertext link.

*text* — A string or string array specifying the text displayed for the link.

---

**NOTE** The two input parameters, *url* and *text*, must have the same number of elements.

---

### Keywords

*Safe* — Handles HTML special characters (see HTML\_SAFE).

### Returned Value

*link\_text* — A string or a string array of HTML hypertext links to URLs.

### Discussion

The HTML\_LINK function creates the hypertext links that most HTML browsers commonly identify using an underline and/or a specific text color.

---

**TIP** To make an image that activates a hypertext link, use the HTML\_IMAGE function as the “text” in HTML\_LINK as follows:

```
link = HTML_LINK(url, HTML_IMAGE('imagefile.gif'))
```

---

### Example

This example creates a link to the VNI home page.

```
HTML_OPEN
```

```
HTML_HEADING, 'A Hypertext Link from HTML_LINK', $  
    Level = 2, /Center
```

```
HTML_BLOCK, HTML_LINK('http://www.vni.com', $
    'Visual Numerics Home Page')

HTML_CLOSE
```

## See Also

[HTML\\_BLOCK](#), [HTML\\_IMAGE](#), [HTML\\_LIST](#),  
[HTML\\_OPEN](#), [HTML\\_SAFE](#)

For complete information of HTML URLs and hyperlinks, see *HTML Sourcebook*, Second Edition, by Ian S. Graham, John Wiley & Sons, Inc., 1996, New York.

---

## HTML\_LIST Procedure

Generates HTML code for lists of all types.

### Usage

HTML\_LIST, *list\_item*

### Input Parameters

*list\_item* — A 1D string array of items for the list.

### Keywords

*Add* — Adds more elements to the current list level.

*AllClose* — Closes off all list levels.

*CloseCurrent* — Closes the current list level to further entries.

*Compact* — Renders the list in a compact format.

---

**NOTE** The HTML compact attribute for lists is not interpreted by some browsers.

---

*Dir* — Creates a directory list.

*DL* — Creates a glossary list of paired items. The glossary list is a special case list interpreted as follows: list items with even-numbered indices (0, 2, 4, ...) are treated as glossary terms (<DT>); list items with odd-numbered indices (1, 3, 5, ...) are treated as definitions (<DD>).

---

**NOTE** Glossary lists require that the *list\_item* array have an even number of elements.

---

*Menu* — Creates a menu list.

---

**NOTE** The *Compact* keyword cannot be used with the *Menu* keyword; menu lists don't accept the compact attribute.

---

*NoClose* — Keeps the list open, so more lists can be added using the *Add* keyword in subsequent HTML\_LIST commands.

*OL* — Creates an ordered list, where each item is ordered numerically or by ascending letters.

*Safe* — Handles HTML special characters (see HTML\_SAFE).

*UL* — (The default list type.) Creates an unordered list, where each item is indicated by a special symbol. (Default: usually a bullet; however this is browser-dependent)

## Discussion

The HTML\_LIST procedure supports all standard HTML list elements. There are two main types of lists in HTML: “glossary” (DL) lists containing paired items; and “regular” (DIR, MENU, OL, and UL) lists containing individual list items (LI). Unless otherwise specified, the default list type is unordered (UL).

If you want to include a lower-level (sub) list, you must use the *NoClose* keyword in the top list level. When *NoClose* is used, one or more new lists can be nested under the current list level. Open lists are closed in a separate HTML\_LIST procedure call, using either the *CloseCurrent* or the *AllClose* keywords.

## Example

This example creates a nested list — one bulleted list of function types (Statistics and Mathematics), with glossary lists of function names and descriptions sublisted under each bullet.

```
HTML_OPEN, Title = 'Math Functions'
      ; Open an HTML file.

HTML_LIST, ['Statistics'], /NoClose
      ; Create a bullet item for Statistics functions, and
      ; leave the list open.

HTML_LIST, ['AVG', $
      'The mean of the variable', 'STDEV', $
      'The standard deviation'], /DL
```

```

        ; Add a DL (glossary) list under the "Statistics"
        ; bullet to include the AVG and STDEV functions
        ; and their descriptive phrases as its items.

HTML_LIST, ['Mathematics'], $
    /Add, /NoClose
        ; Add a "Mathematics" item to the bullet-level list.

HTML_LIST, ['FFT', $
    'Fast Fourier Transform', $
    'CONVOL', $
    'Convolution of an array '+$
    'with a kernel', 'CROSSP', $
    'The Cross Product of two '+$
    'vectors'], /DL
        ; Add a DL (glossary) list under the "Mathematics"
        ; bullet to include the FFT, CONVOL, and
        ; CROSSP functions as its items.

HTML_LIST, /AllClose
        ; Close all of the lists.

HTML_CLOSE
        ; Close the HTML file.

```

The resulting HTML nested list output looks like this:

- Statistics
  - AVG
    - The mean of the variable
  - STDEV
    - The standard deviation
- Mathematics
  - FFT
    - Fast Fourier Transform
  - CONVOL
    - Convolution of an array with a kernel
  - CROSSP
    - The Cross Product of two vectors

## See Also

[HTML\\_BLOCK](#), [HTML\\_HEADING](#), [HTML\\_HIGHLIGHT](#),  
[HTML\\_OPEN](#), [HTML\\_PARAGRAPH](#), [HTML\\_SAFE](#), [HTML\\_TABLE](#)

For a complete discussion of HTML list elements, see *HTML Sourcebook*, Second Edition, by Ian S. Graham, John Wiley & Sons, Inc., 1996, New York, pp. 172-183.

---

## HTML\_OPEN Procedure

Opens the output HTML file, writes out the basic HTML information and sets an HTML output file information variable.

### Usage

HTML\_OPEN [, *filename*]

### Input Parameters

*filename* — (optional) The output HTML file name.  
(Default on UNIX and OpenVMS: `wave.html`)  
(Default on Windows: `wave.htm`)

### Keywords

---

**NOTE** Some of the attributes that you can specify with these keywords are not used by certain browsers.

---

***ALinkColor*** — Defines the color of active links. An active link is a selected link which is being processed.

***BgColor*** — A string or long integer specifying the background color name or number (24-bit color). Examples of this keyword usage are:

```
BgColor = 'red'  
BgColor = '#ff0000'  
BgColor = 'ff0000'XL  
BgColor = 16711680
```

***BgImage*** — Accepts a URL string to an image for the background.

***CGI*** — Writes a “Content-type” header — useful if you are outputting from a CGI script.

***LinkColor*** — Defines the text color of new links. A new link is one which hasn't been previously selected.

***Stdout*** — If set, writes to standard output instead of to a file.

***Title*** — A string scalar specifying the HTML <TITLE>.

*TextColor* — Defines the text color.

*VLinkColor* — Defines the color of visited links.

## Discussion

The HTML\_OPEN procedure provides the basic formatting information required at the beginning of every HTML file. Optional format features such as background, foreground, and text colors can be added by using the keywords.

A file opened using the HTML\_OPEN procedure must always be closed with an HTML\_CLOSE procedure call at the end. This closes many of the initial HTML formatting elements opened when the file was initiated.

## See Also

[HTML\\_CLOSE](#)

For a complete discussion of HTML, see *HTML Sourcebook*, Second Edition, by Ian S. Graham, John Wiley & Sons, Inc., 1996, New York.

---

## HTML\_PARAGRAPH Procedure

Outputs an HTML paragraph.

### Usage

HTML\_PARAGRAPH, *text*

### Input Parameters

*text* — A scalar or string array containing the text of the paragraph. In the case of a string array, each element is interpreted as a separate paragraph.

### Keywords

*Center* — Centers each line in the paragraph.

*Justify* — Fully justifies the paragraph to the left and right.

*Left* — Left-justifies the paragraph. (Default: set)

*Safe* — Handles HTML special characters (see HTML\_SAFE).

**Right** — Right-justifies the paragraph.

## Discussion

The `HTML_PARAGRAPH` procedure creates HTML paragraphs by inserting the elements `<P> . . . </P>` around the text. The created paragraph is left-justified unless otherwise specified using keywords.

## Example

This example illustrates how to put a standard paragraph into HTML format in `PV-WAVE`.

```
HTML_OPEN
```

```
HTML_PARAGRAPH, 'PV-WAVE provides an ' + $  
    'array-oriented fourth-generation ' + $  
    'language that is compact and ' + $  
    'efficient. Its interactive structure ' + $  
    'reduces coding by up to 80% and ' + $  
    'eliminates compiling and linking. ' + $  
    'It supports variables, collections ' + $  
    'of variables and all the same ' + $  
    'language constructs of FORTRAN ' + $  
    'and C.', /Justify
```

```
HTML_CLOSE
```

## See Also

[HTML\\_BLOCK](#), [HTML\\_HEADING](#), [HTML\\_HIGHLIGHT](#),  
[HTML\\_LIST](#), [HTML\\_OPEN](#), [HTML\\_SAFE](#), [HTML\\_TABLE](#)

---

## **HTML\_RULE Procedure**

Inserts a horizontal-line separator in HTML.

### **Usage**

HTML\_RULE

### **Input Parameters**

None.

### **Keywords**

None.

### **Discussion**

The HTML\_RULE procedure inserts the standard horizontal rule (<HR>) into the HTML flow. To insert a separator other than the standard, use HTML\_IMAGE with the separator image of your choice.

### **See Also**

[HTML\\_BLOCK](#), [HTML\\_HEADING](#), [HTML\\_OPEN](#), [HTML\\_PARAGRAPH](#)

---

## **HTML\_SAFE Function**

Allows special characters defined in HTML to be displayed as text, rather than using them for format tagging.

### **Usage**

*html\_str* = HTML\_SAFE(*str*)

### **Input Parameters**

*str* — A text string that may contain special HTML characters.

## Keywords

None.

## Returned Value

*html\_str* — A string containing the HTML escape sequences for the special characters included in *str*.

## Discussion

The HTML\_SAFE function easily allows the special HTML formatting characters to be interpreted as text. The special characters covered by HTML\_SAFE are the left and right angle brackets (< and >); the ampersand (&); and quotation marks (“ ”). HTML\_SAFE assures that these characters will be translated to their HTML escape sequence, so that the browser will display them as intended.

---

**NOTE** All text-related HTML routines in PV-WAVE have a *Safe* keyword, which calls the HTML\_SAFE function to escape any special characters contained in the text.

---

## Example

For this example, you could achieve the same result by using the *Safe* keyword with HTML\_HEADING procedure.

```
HTML_OPEN
HTML_HEADING, HTML_SAFE('For An <OL> list')
HTML_CLOSE
```

## See Also

[HTML\\_BLOCK](#), [HTML\\_HEADING](#), [HTML\\_HIGHLIGHT](#), [HTML\\_LIST](#), [HTML\\_OPEN](#), [HTML\\_PARAGRAPH](#), [HTML\\_TABLE](#)

---

## **HTML\_ TABLE Procedure**

Creates an HTML table.

### **Usage**

HTML\_TABLE, *table\_text*

### **Input Parameters**

*table\_text* — An ( $m, n$ ) string array of text to put in a table with  $m$  columns and  $n$  rows. A 1D array builds an  $m$ -column, 1-row table.

### **Keywords**

---

**NOTE** Whenever a specified attribute is not supported by a particular browser, the attribute is simply ignored by that browser.

---

***Border*** — The size of the border around cells in the table.

***Bottom*** — Places the cell content at the bottom of each cell.

***Caption*** — The table caption.

***CBottom*** — Table caption displayed beneath the table.

***CellPadding*** — Specifies the space between the borders and the content of the cell.

***CellSpacing*** — Specifies the space between each individual cell.

***Center*** — Centers the cell content.

***ColLabels*** — The column labels.

***EqualWidth*** — Defines all cells as having the same width as the largest one used.

***Left*** — Left-justifies the cell contents in the cell.

***Middle*** — Places the content in the middle of each cell.

***NoWrap*** — When set, the cell contents don't wrap onto multiple lines within the cell.

***Right*** — Right-justifies the cell contents in the cell.

***RowLabels*** — The row labels.

*Safe* — Handles HTML special characters (see HTML\_SAFE).

*TCenter* — Centers the table on the page (left-right centering).

*TLeft* — Left-justifies the table on the page. (Default: set)

*Top* — Places the cell content at the top of each cell.

*TRight* — Right-justifies the table on the page.

## Discussion

This procedure creates an HTML table.

## Example

This example creates a captioned table with headings, which contains random data.

```
HTML_OPEN, Title = 'PV-WAVE ' + 'HTML Output Example'
HTML_HEADING, 'Example HTML ' + 'output from PV-WAVE'
a = RANDOMN(seed, 100, 100)
b = RANDOMU(seed, 100, 100)
    ; Get some data.

table_info = FLTARR(2, 5)

table_info(0, *) = [MIN(a), $
    MAX(a), AVG(a), MEDIAN(a), $
    STDEV(a)]

table_info(1, *) = [MIN(b), $
    MAX(b), AVG(b), MEDIAN(b), $
    STDEV(b)]
    ; Build a variable containing the numbers.

table_text = STRTRIM $
    (STRING(table_info), 2)
    ; The table contents needs to be text.

col_titles = ['Normal Distribution', $
    'Uniform Distribution']

row_titles = ['Minimum', 'Maximum', $
    'Mean', 'Median', $
    'Standard Deviation']
    ; Build arrays of row and column headings.
```

```
HTML_TABLE, table_text, $
    RowLabels = row_titles, $
    ColLabels = col_titles, $
    Caption = 'Two Random ' + $
    'Number Sets', Border = 1
    ; Make the table with a caption and labels.

HTML_CLOSE
```

## See Also

[HTML\\_BLOCK](#), [HTML\\_HEADING](#), [HTML\\_HIGHLIGHT](#), [HTML\\_LIST](#),  
[HTML\\_OPEN](#), [HTML\\_PARAGRAPH](#), [HTML\\_SAFE](#)

For complete information on HTML tables, refer to *HTML Sourcebook*, Second Edition, by Ian S. Graham, John Wiley & Sons, Inc., 1996, New York.

---

## HTML\_TEXT Procedure

Outputs text to the open HTML file.

### Usage

HTML\_TEXT, *text*

### Input Parameters

*text* — A string containing the text to output.

### Keywords

*Safe* — If nonzero, the procedure handles HTML special characters (see HTML\_SAFE).

### Example

```
HTML_OPEN
HTML_TEXT, 'Hello, world!'
HTML_CLOSE
```

## See Also

[HTML\\_PARAGRAPH](#), [HTML\\_SAFE](#)



---

## **IMAGE\_CHECK Function**

Checks if the input variable is a properly defined associative array in image format and ensures that all keys in the array are of the correct data type.

### **Usage**

*status* = IMAGE\_CHECK(*image*)

### **Input Parameters**

*image* — An associative array in image format.

### **Keywords**

**Quiet** — Suppresses successive levels of error messages, depending on the value set. This keyword accepts the same integer values used with the system variable !Quiet.

**Valid** — If set, checks to make sure that the image associative array's status key does not contain an error code and that the data values are within supported limits.

### **Returned Value**

*status* — A value indicating the success or failure of the function.

- < 0    Indicates that the input array is not a valid image associative array.
- 0      Indicates that the input array is a valid image associative array.

### **Discussion**

Call this function after modifying image data to be sure that the resulting image data is valid.

Use the *Valid* keyword to check the input image more thoroughly. If an error is detected, an informative message is printed to the screen.

---

**NOTE** You can use IMAGE\_CREATE or ' to create an image associative array. Complete image information is stored in this type of array. Refer to the IMAGE\_CREATE function description for detailed information on the structure of an image associative array.

---

## Example

In this example, a test image is created using `IMAGE_CREATE`. Then, the test image is modified and the result checked with `IMAGE_CHECK`. In this case, the modification made is invalid — the *width* key in the associative array has been assigned a short integer and it expects a long integer.

```
pix = BYTARR(400, 200)
test_image = IMAGE_CREATE(pix)
test_image('width') = 500
PRINT, IMAGE_CHECK(test_image)
      %IMAGE_CHECK: Wrong datatype of array element width
      -8
; The error message is displayed because the width key in the
; associative array has been assigned a short integer and it expects
; a long integer.
```

## See Also

[IMAGE\\_CREATE](#), [IMAGE\\_QUERY\\_FILE](#),  
[IMAGE\\_READ](#)

System Variables: [!Quiet](#)

---

## ***IMAGE\_COLOR\_QUANT* Function**

Quantizes a 24-bit image to 8-bit pseudo-color.

### Usage

```
result = IMAGE_COLOR_QUANT(image [, n_colors])
```

### Input Parameters

***image*** — Either an image associative array with a 24-bit color image or a 3D byte array.

***n\_colors*** — (optional) An integer specifying the number of colors desired in the output pseudo-color image. This value must be greater than 0 and less than or equal to 256. (Default: `!D.Table_Size`)

## Returned Value

**result** — The returned value depends upon whether the input was a 24-bit color image or a 3D byte array:

Type of Input	Description of Output
Image associative array	Returns an associative array in image format containing the color quantized (8-bit) pseudo-color image. On error, returns an image associative array with the <i>status</i> key set to $< 0$ .
3D byte array	Returns a 2D byte array containing the color quantized (8-bit) pseudo-color image. On error, returns $< 0$ .

## Keywords

**Colormap** — Specifies a variable to hold the colormap of the quantized image. *Colormap* is a 3-by-*n\_colors* byte array.

**Dither** — If set, Floyd-Steinberg dithering is used to quantize the 24-bit image.

**Intleave** — A scalar string indicating the type of interleaving of 3D image arrays. This keyword can only be used when the input *image* is a 3D byte array. If *Intleave* is not specified, the default interleaving method corresponds to the minimum dimension of the array, where *p* is the minimum dimension.

Valid strings and the corresponding interleaving methods are:

'image' — The 3D *image* array arrangement is  $(x, y, p)$  for *p* image-interleaved images of *x*-by-*y*.

'row' — The 3D *image* array arrangement is  $(x, p, y)$  for *p* row-interleaved images of *x*-by-*y*.

'pixel' — The input array arrangement is  $(p, x, y)$  for *p* pixel-interleaved images of *x*-by-*y*.

---

**NOTE** If the input *image* is an associative array, the interleaving method is found in the *interleave* field of the array.

---

**Loadcmap** — If set, the generated colormap is automatically loaded using TVLCT.

**Quiet** — Suppresses successive levels of error messages, depending on the integer value specified. This keyword accepts the same integer values used with the system variable !Quiet.

## Discussion

The `IMAGE_COLOR_QUANT` function is useful for converting 24-bit images for display on 8-bit devices. It can also be used to compress the amount of information in the image to reduce the amount of storage space needed when the image is saved to a file.

---

**NOTE** The original 24-bit image cannot be reconstructed from the 8-bit quantized result.

---

The function quantizes the image using the median-cut algorithm. For information on the algorithm used in this function, refer to:

Paul Heckbert. “Color Image Quantization for Frame Buffer Display”, *Siggraph '82 Proceedings*, pp. 297-307.

## Example 1

The first example uses an associative array in image format as input.

---

**NOTE** The filename used in this example is a UNIX specific filename, which must be modified for use on other platforms.

---

```
chautauqua = IMAGE_READ(GETENV('VNI_DIR') + $
    '/image-1_0/data/chautauqua24.tiff')
    ; Read in a 24-bit image-interleaved image.

chautauqua_8bit = IMAGE_COLOR_QUANT(chautauqua, 256,)
    ; Convert the image to 8-bit pseudo-color.

IMAGE_DISPLAY, chautauqua_8bit
    ; Display the 8-bit image.
```

## Example 2

This example reads 24-bit image data that has been stored in three separate image files — one red, one green, and one blue. Each file is read separately and then combined in one 3D array before being quantized and displayed.

The data used in the example comes from the red, green, and blue images of Boulder in the `PV-WAVE` data directory.

```
imgx = 477
imgy = 512
red_img = BYTARR(imgx, imgy, /Nozero)
```

```

grn_img = BYTARR(imgx, imgy, /Nozero)
blu_img = BYTARR(imgx, imgy, /Nozero)
OPENR, 1, !Data_Dir + 'boulder_red.img'
READU, 1, red_img
CLOSE, 1
OPENR, 1, !Data_Dir + 'boulder_grn.img'
READU, 1, grn_img
CLOSE, 1
OPENR, 1, !Data_Dir + 'boulder_blu.img'
READU, 1, blu_img
CLOSE, 1
    ; Read the separate red, green, and blue image files.
boulder = BYTARR(imgx, imgy, 3, /Nozero)
boulder(*, *, 0) = red_img
boulder(*, *, 1) = grn_img
boulder(*, *, 2) = blu_img
    ; Combine the image files into a single 3D array.
boulder_8bit = IMAGE_COLOR_QUANT(boulder, 256, /Loadcmap)
    ; Convert the image to 8-bit pseudo-color.
TV, boulder_8bit
    ; Display the 8-bit image.

```

## See Also

[IMAGE\\_CREATE](#), [IMAGE\\_DISPLAY](#), [IMAGE\\_READ](#), [IMG\\_TRUE8](#)

System Variables: [!Quiet](#)

---

## ***IMAGE\_CONT Procedure***

Standard Library procedure that overlays a contour plot onto an image display of the same array.

## Usage

IMAGE\_CONT, *array*

## Input Parameters

*array* — The two-dimensional array to display.

## Keywords

*Aspect* — Set to 1 to change the image's aspect ratio. It assumes square pixels. If *Aspect* is not set, the aspect ratio is retained.

*Interp* — Set to 1 to interpolate the image with the bilinear method, when and if the image is resampled. Otherwise, the nearest neighbor method is used.

*Window\_Scale* — Set to 1 to scale the window size to the image size. Otherwise, the image size is scaled to the window size. *Window\_Scale* is ignored when outputting to devices with scalable pixels (e.g., PostScript devices).

## Discussion

If the device you are using has scalable pixels, then the image is written over the plot window.

### *Example*

This example uses IMAGE\_CONT to display the image and overlaid contour plot depicting different elevations of the surface defined by

$$f(x,y) = x\sin(y) + y\cos(x) - 10\sin(xy/4)$$

where

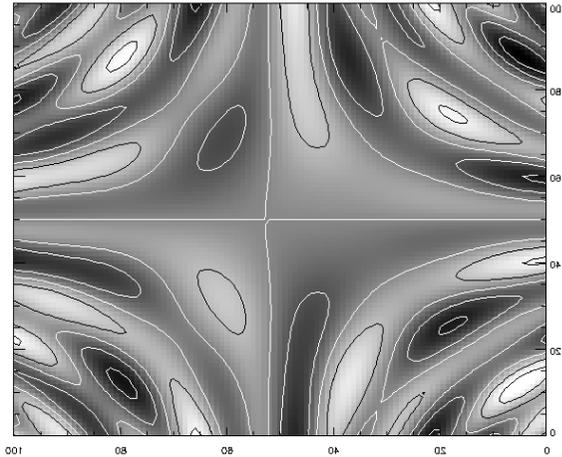
$$(x,y) \in \{\mathbb{R}^2 \mid x, y \in [-10, 10]\}$$

```
.RUN
FUNCTION f, x, y
RETURN, x * SIN(y) + y * COS(x) - 10 * SIN(0.25 * x * y)
END
; Define the function.
x = FINDGEN(101)/5 - 10
; Create vector of x-coordinates.
y = x
; Create vector of y-coordinates.
```

```

z = FLTARR(101, 101)
    ; Create an array to hold the function values.
FOR i = 0, 100 DO FOR j = 0, 100 DO $
    z(i, j) = f(x(i), y(j))
        ; Evaluate the function at the given x- and y-coordinates and
        ; place the result in z.
IMAGE_CONT, z
    ; Display image and contour plot.

```



**Figure 2-29** Image and contour plot of  $z(x,y) = x \sin(y) + y \cos(x) - 10 \sin(xy/4)$ .

### See Also

[CONTOUR](#), [SHOW3](#), [TV](#), [TVSCL](#)

For details on methods of interpolation, see the *PV-WAVE User's Guide*.

---

## ***IMAGE\_CREATE Function***

Creates an associative array in image format.

## Usage

*image* = IMAGE\_CREATE(*pixel\_array*)

## Input Parameters

*pixel\_array* — An array containing image data.

The input array must be in one of the following forms, where *w* is the image width, *h* is the height, and *img\_count*, an optional parameter, is the number of images in the array:

( <i>w</i> , <i>h</i> [, <i>img_count</i> ])	8-bit (pseudo-color) images
(3, <i>w</i> , <i>h</i> [, <i>img_count</i> ])	24-bit (direct color) images, pixel interleaved
( <i>w</i> , 3, <i>h</i> [, <i>img_count</i> ])	24-bit (direct color) images, row interleaved
( <i>w</i> , <i>h</i> , 3 [, <i>img_count</i> ])	24-bit (direct color) images, image interleaved

## Returned Value

*image* — An associative array in image format.

## Keywords

*Colormap* — A 3-by-*n* colormap array.

---

**NOTE** If the input *pixel\_array* is 2D and the *Colormap* keyword is not specified, the returned *image* is assumed to be grayscale.

---

*Colormodel* — A long integer specifying the color model:

0	Monochrome or gray scale
1	RGB
2	CMY *
3	HSV *

\* Currently not supported

*Comments* — A string containing additional information about the image. This feature is supported for GIF, JPEG, MIFF, PNG, TGA, and TIFF formats.

*Depth* — A long integer specifying the number of bits per channel. (Default: 8)

*File\_Name* — A string containing a default filename for the image.

**File\_Type** — A string specifying the type of image file to create. For example, 'sun' denotes a Sun Rasterfile. (Default: TIFF)

---

**NOTE** See the *Discussion* section of the IMAGE\_READ function for a list of valid file types.

---

**Img\_Count** — A long integer specifying the number of images in the output array *image*. (Default: 1)

**Intleave** — A scalar string indicating the desired type of interleaving of 3D image arrays. (Default: image)

Valid strings and the corresponding interleaving methods are:

'image' — The resulting 3D *image* array arrangement is  $(x, y, p)$  for  $p$  image-interleaved images of  $x$ -by- $y$ . (Default)

'row' — The resulting 3D *image* array arrangement is  $(x, p, y)$  for  $p$  row-interleaved images of  $x$ -by- $y$ .

'pixel' — The resulting 3D *image* array arrangement is  $(p, x, y)$  for  $p$  pixel-interleaved images of  $x$ -by- $y$ .

**Quiet** — Suppresses successive levels of error messages, depending on the value set. This keyword accepts the same integer values used with the system variable !Quiet.

**Units** — Units of pixel size. (1 = inches; 2 = millimeters)

**X\_resolution** — Pixel size in *Units*.

**Y\_resolution** — Pixel size in *Units*.

## Discussion

The IMAGE\_CREATE function creates an image associative array. Complete image information is stored in this type of array, which is used by all of the IMAGE\_\* routines in PV-WAVE. (Refer to the *PV-WAVE Programmer's Guide* for additional information on associative arrays.)

---

**NOTE** When creating an image that contains 3D pixel data, you must specify the interleaving method if either the width or height has a value of 3. For example, if the dimensions of your data are 100 x 3 x 3 and the data is plane interleaved, it will be interpreted as line interleaved unless you explicitly set the *Interleave* keyword to the correct value.

---

The image format associative array allows the following operations:

- File I/O of images to and from different file formats
- Displaying images
- Processing images

### ***Image Associative Array Structure***

The following table describes each key of the image associative array.

<b>Array Key Name</b>	<b>Variable Type</b>	<b>Description</b>
<i>file_name</i>	STRING	Absolute or relative pathname of an image file
<i>file_type</i>	STRING	The type of image file. (See IMAGE_READ for a list of supported file types.)
<i>width</i>	LONG	Image width
<i>height</i>	LONG	Image height
<i>img_num</i>	LONG	Index number in the range {0... <i>n</i> } of a subimage in the file (if the image was created with IMAGE_READ)
<i>img_count</i>	LONG	Number of subimages in the pixel array
<b>PIXELS:</b>		
<i>nr_cc</i>	LONG	Number of color channels. This item specifies how many storage units are allocated per pixel. (Supported values: 1 and 3)
<i>depth</i>	LONG	Bits per color channel
<i>pixel_dtype</i>	LONG	Data type of the pixel array: 1 BYTE 2 SHORT ** 3 LONG **
<i>interleave</i>	LONG	Interleave method used: 1 = Pixel interleaving 2 = Row interleaving 3 = Image interleaving
<i>color_model</i>	LONG	Color model: 0 Monochrome or gray scale 1 RGB 2 CMY (currently not supported) 3 HSV (currently not supported)

<b>Array Key Name</b>	<b>Variable Type</b>	<b>Description</b>
<i>pixels</i>	BYTE SHORT ** LONG **	A multidimensional array describing an 8-bit or 24-bit image. Refer to the <i>Pixel Arrays</i> section later for a description of supported pixel arrays.
<i>x_resolution</i>	FLOAT	Pixel size in <i>units</i>
<i>y_resolution</i>	FLOAT	Pixel size in <i>units</i>
<i>units</i>	LONG	Units of pixel size: 1 (inches) 2 (mm)
<b>COLORMAP:</b>		
<i>cmap_type</i>	LONG	Type of colormap: 0 No colormap 1 1 colormap vector (not supported) 2 Multiple colormap vectors
<i>cmap_dtype</i>	LONG	Data type of colormap (only BYTE (1) is supported)
<i>n_colors</i>	LONG	Number of entries in the colormap
<i>colormap</i>	BYTE SHORT ** LONG **	Array whose dimensions depend on <i>cmap_type</i> and <i>n_colors</i> . <i>cmap_type</i> : 1 : not supported 2 : (3 , <i>n_colors</i> )
<b>SUPPLEMENTARY and CONTROL:</b>		
<i>label</i>	STRING	Optional annotation (NOTE: None of the supported graphics file types currently support labels.)
<i>comments</i>	STRING	Optional information about source, etc.
<i>status</i>	LONG	0 = success; < 0 = error
** This data type is currently not supported for IMAGE_READ and IMAGE_WRITE. IMAGE_CREATE accepts an array in this data type, but before writing the image to a file the pixel array must be converted to BYTE.		

### ***Pixel Arrays***

This table describes valid pixel arrays.

Color Model	Number of color channels ( <i>nr_cc</i> )	Bits per pixel	Colormap	Number of color channels in colormap	Dimensions of pixel array
True Color RGB	3	24	No	N/A	3D **
GrayScale	1	8	Yes	3	2D **
Pseudo Color	1	8	Yes	3	2D **
Direct Color	3	24	Yes	3	3D **

\*\* Refer to the *Pixel Array Dimensions* table for more information.

### ***Pixel Array Dimensions***

This table describes the valid pixel array dimensions. The dimensions of the pixel array (*width*, *height*, *nr\_images*, etc.) refer to keys in the image associative array.

Type of Array	Description
<b>2D Pixel Arrays:</b>	
( <i>width</i> , <i>height</i> , [ <i>nr_images</i> ])	8-bit (pseudo-color) image
<b>3D Pixel Arrays:</b>	
( <i>nr_cc</i> , <i>width</i> , <i>height</i> [, <i>img_count</i> ])	24-bit (direct color) image, pixel interleaved
( <i>width</i> , <i>nr_cc</i> , <i>height</i> [, <i>img_count</i> ])	24-bit (direct color) image, row interleaved
( <i>width</i> , <i>height</i> , <i>nr_cc</i> [, <i>img_count</i> ])	24-bit (direct color) image, image interleaved

### **Example**

This example demonstrates how `IMAGE_CREATE` creates an image associative array. The associative array is listed using the `INFO` command.

```
pixels = BYTARR(100, 200)
        ; Create some pixel data (all zeros).

cmap = BYTARR(3, 50)
       ; Create a colormap (all zeros).

image = IMAGE_CREATE(pixels, File_name='test.tif', $
                    Colormap = cmap, File_type = 'tif')
```

```

; Create the image -- an associative array of image-related
; data. Next, use INFO to display the contents of the array.

```

```

INFO, image, /Full

```

```

IMAGE          AS. ARR      =      Associative Array(22)
label          STRING      =      ''
file_type      STRING      =      'tif'
height         LONG        =      200
units          LONG        =      2
img_num        LONG        =      0
y_resolution   FLOAT      =      72.0000
pixels         BYTE       =      Array(100, 200)
color_model    LONG        =      1
n_colors       LONG        =      50
file_name      STRING      =      'test.tif'
width          LONG        =      100
cmap_type      LONG        =      2
x_resolution   FLOAT      =      72.0000
status         LONG        =      0
img_count      LONG        =      1
nr_cc          LONG        =      1
depth          LONG        =      8
colormap       BYTE       =      = Array(3, 50)
cmap_dtype     LONG        =      1
comments       STRING      =      ''
pixel_dtype    LONG        =      1
interleave     LONG        =      3

```

---

## See Also

[IMAGE\\_DISPLAY](#), [IMAGE\\_READ](#), [IMAGE\\_WRITE](#)

System Variables: [!Quiet](#)

---

## ***IMAGE\_DISPLAY Procedure***

Displays an image associative array.

### **Usage**

`IMAGE_DISPLAY, image [, x, y]`

`IMAGE_DISPLAY, image [, position]`

### **Input Parameters**

*image* — An associative array in image format.

*x, y* — (optional) The lower-left *x*- and *y*-coordinates of the displayed image in the window. (See the TV procedure for more information.)

*position* — (optional) A number specifying the position of the image. (See the TV procedure for more information.)

### **Keywords**

*Animate* — If nonzero, multiple images are displayed sequentially. The *Animate* keyword only applies if the image associative array contains multiple images.

*Delay* — A floating-point value determining the frame-to-frame delay in seconds: applies only to animations. (Default: 0.01)

*Quiet* — Suppresses successive levels of error messages, depending on the set value. This keyword accepts the same integer values used with the system variable !Quiet.

*Sub\_Img* — An integer specifying the index of the subimage in the image associative array to display. (Default: 0)

*Window* — An integer specifying the window number. (Default: first free window number)

### ***Keywords Relating to the Image Display (TV)***

*Data* — Specifies that the data coordinate system be used by *x* and *y* (image position).

*Device* — Specifies that the device coordinate system be used by *x* and *y* (image position). This is the default if no other coordinate-system keyword is specified.

*Normal* — Specifies that the normal coordinate system be used by *x* and *y* (image position).

### **Keywords Relating to the Window**

**Wset** — If set and the *Window* keyword is also specified, the image draws into the specified window. If set and the *Window* keyword is not specified, the image draws into the current active window.

The following keywords are only used if a new window is created; they are ignored if you use *Wset* and the window exists.

**Bitmap** — (Windows Only) Specifies that the window being created is actually an invisible portion of the display memory called a bitmap.

---

**TIP** For cross-platform portability, use the *Pixmap* keyword instead of the *Bitmap* keyword.

---

**Colors** — The maximum number of color-table indices to be used.

---

**NOTE** The *Colors* keyword has an effect only if it is supplied when the first window is created; otherwise, **PV-WAVE** uses all of the available color indices.

---

**NoMeta** — (Windows Only) Turns metafiles off for the window. This is the default if *Animate* is specified.

**Pixmap** — Specifies that the window being created is actually an invisible portion of the display memory called a pixmap.

**Retain** — Specifies how backing store for the window should be handled. (See the *Retain* keyword in the **WINDOW** procedure for more information.)

**Title** — If specified with a string value, then the string becomes the title of the window. If set, but no string is specified (for example, */Title*), then the window title is the **COMMENTS** field of the image.

**XPos** — The *x* position of the lower-left corner of the new window, specified in pixels relative to the lower-left corner of the display.

**XSize** — The width of the window, in pixels. (Default: width of the image)

**YPos** — The *y* position of the lower-left corner of the new window, specified in pixels relative to the lower-left corner of the display.

**YSize** — The height of the window, in pixels. (Default: height of the image)

### **Discussion**

The input associative array for the **IMAGE\_DISPLAY** procedure, *image*, must be created by the routines **IMAGE\_READ** or **IMAGE\_CREATE**.

On systems where the display depth is eight or less (for example, when `!D.Display_Depth ≤ 8`), 24-bit images are converted to 8-bit pseudo-color before being displayed. This conversion is performed using the `IMAGE_COLOR_QUANT` function and does not affect the original data. However, 8-bit images may be quantized to fewer colors if there are not enough colors available to display the image. To ensure that enough colors are available, use `Colors=256` when creating the first window.

---

**NOTE** The `IMAGE_DISPLAY` procedure generates errors, and/or will display some or all of an image outside of the window, if the window size and image position are improperly selected.

---

---

**NOTE** Color tables associated with the image are loaded when the image is displayed. This behavior may cause the colors in other windows to appear altered.

---

## Example

This example shows how an image associative array created with `IMAGE_CREATE` can be displayed with `IMAGE_DISPLAY`.

```
pixels = BYTARR(100, 200)
        ; Assign some pixel data (all zeros).

cmap = BYTARR(3, 50)
       ; Assign a colormap (all zeros).

image = IMAGE_CREATE(pixels, File_name = $
                    'test.tif', Colormap = cmap, $
                    File_type = 'tif')

IMAGE_DISPLAY, image
        ; The image is displayed as a black image.
```

## See Also

[IMAGE\\_COLOR\\_QUANT](#), [IMAGE\\_CREATE](#), [IMAGE\\_READ](#), [TV](#), [WINDOW](#)

System Variables: [!Quiet](#)

---

## ***IMAGE\_QUERY\_FILE Function***

Returns the image type in a specified image file.

## Usage

*status* = IMAGE\_QUERY\_FILE(*filename* [, *filetype*])

## Input Parameters

***filename*** — On input, a string containing the name of the file to query. System variables/logicals and special characters recognized by the operating system shell can be used in the path.

***filetype*** — (optional) On output, a string containing the type of image in the file.

## Keywords

***Default\_Filetype*** — A string specifying a supported file type. This keyword is interpreted according to the rules listed in the *Discussion* section.

***Quiet*** — Suppresses successive levels of error messages, depending on the set value. This keyword accepts the same integer values used with the system variable !Quiet.

***Readable*** — If set, checks the file type of the image against the list of supported readable file types.

## Returned Value

*status* — A value indicating the success or failure of the function.

- < 0    Indicates that the file does not contain a supported image type.
- 0      Indicates that the file does contain a supported image type.

## Discussion

---

**NOTE** For a list of supported image types, refer to the *Discussion* section of the IMAGE\_READ function.

---

This function returns an image file type according to the following rules of precedence:

1. If an encoded ID number can be extracted from the file header, and the number matches that of one of the supported image types, return that image type.
2. If no encoded ID number is detected, return the image type corresponding to the type specified by the *Default\_Filetype* keyword.

3. If no encoded ID number is detected and the *Default\_Filetype* keyword is not present, try to determine the image type from the filename suffix. The function tries to match the suffix to a list of standard suffixes for supported image file types.
4. If the image type cannot be determined by the above rules, the function returns an error status.

## Example 1

In these two code examples, the correct image type is obtained from the encoded ID number in the file header.

```
status = IMAGE_QUERY_FILE('photo_pcx.sun', filetype)
PRINT, status, filetype
```

```
0, PCX
; The image file is a PCX file. The correct image type is obtained from
; the encoded ID number, which takes precedence over the specified
; filename extension.
```

```
status = IMAGE_QUERY_FILE('photo_pcx.sun', $
    filetype, Default_Filetype = 'tif')
PRINT, status, filetype
```

```
0, PCX
; The image file is a PCX file. The correct image type is obtained from
; the encoded ID number, which takes precedence over both the filename
; suffix and the Default_Filetype keyword.
```

## Example 2

In this example, the correct filetype for a TGA (Truevision Targa) file is obtained using the *Default\_Filetype* keyword. If the *Default\_Filetype* keyword were not specified, an incorrect image type would have been returned. This is because TGA graphics files don't contain an encoded ID number (see also Example 3).

```
status = IMAGE_QUERY_FILE('photo_tga.sun', $
    filetype, Default_Filetype = 'tga')
PRINT, status, filetype
```

```
0, TGA
```

### Example 3

The following three code examples show how an incorrect file type or an error can be returned for TGA (Truevision Targa) files because they don't contain an encoded ID number. The lack of an encoded ID number results in ambiguity when identifying the file type using the `IMAGE_QUERY_FILE` function. (Example 2 shows how the correct file type can be returned for a TGA file.)

```
status = IMAGE_QUERY_FILE('photo_tga.sun', $
    filetype, Default_Filetype = 'tif')

PRINT, status, filetype

0, TIFF
    ; Note that the wrong image type, TIFF, was returned. Without an
    ; encoded ID number in the graphics file, IMAGE_QUERY_FILE
    ; looks at the value of the Default_Filetype keyword, which in this
    ; case was incorrect.

status = IMAGE_QUERY_FILE('photo_tga.sun', filetype)

PRINT, status, filetype

0, SUN
    ; Here, once again the incorrect image type is returned for
    ; a TGA file; since no encoded ID number was found, and
    ; no Default_Filetype keyword is specified, the function "guesses"
    ; the image type by looking at the filename suffix.

status = IMAGE_QUERY_FILE('photo_tga', filetype)

PRINT, status

-3
    ; This example code also returns an error status. The error results
    ; because the function can't make a reasonable guess at the image type
    ; without the following: an encoded ID number, the Default_Filetype
    ; keyword, and/or a filename suffix.
```

### See Also

[IMAGE\\_CHECK](#), [IMAGE\\_DISPLAY](#), [IMAGE\\_READ](#),  
[DICM\\_TAG\\_INFO](#)

System Variables: [!Quiet](#)

---

## **IMAGE\_READ Function**

Reads an image file and returns an associative array in image format.

### **Usage**

*image* = IMAGE\_READ(*filename*)

### **Input Parameters**

*filename* — A string containing the pathname and filename of an image file. System variables/logicals and special characters recognized by the operating system shell can be used in the path.

### **Keywords**

---

**NOTE** If *File\_Type* is TIFF16, all other keywords are ignored.

---

*All\_Subimages* — If set, all subimages in the file are read and the *Img\_Count* keyword is ignored. The first subimage in the file is read first, unless the *Sub\_Img* keyword is also specified.

---

**NOTE** Multiple images being read with the IMAGE\_READ function must all be of the same height, width, and class. If a file contains images of different heights, widths, and/or classes, only the first contiguous sequence with equal height, width, and class are read.

---

*Cmap\_Compress* — If set, and the colormap in the image file has unused entries, IMAGE\_READ compresses the colormap to the minimum size and re-maps the pixel array.

*File\_Type* — A string specifying the default file type. (See the *Discussion*.)

*Intleave* — A scalar string indicating the desired type of interleaving of 3D image arrays. (Default: `pixel`)

Valid strings and the corresponding interleaving methods are:

'*image*' — The resulting 3D *image* array arrangement is (*x*, *y*, *p*) for *p* image-interleaved images of *x*-by-*y*.

'*row*' — The resulting 3D *image* array arrangement is (*x*, *p*, *y*) for *p* row-interleaved images of *x*-by-*y*.

'*pixel*' — The resulting 3D *image* array arrangement is (*p*, *x*, *y*) for

$p$  pixel-interleaved images of  $x$ -by- $y$ . (Default)

**Img\_Count** — An integer specifying the number of images to read from an array of images.

---

**NOTE** The *Img\_Count* keyword is ignored if the *All\_Subimages* keyword is set.

---

**Order** — If nonzero, returns the image mirrored in the  $y$ -direction. (Default: Do not mirror the image.)

**Quiet** — Suppresses successive levels of error messages, depending on the value set. This keyword accepts the same integer values used with the system variable !Quiet.

**Sub\_Img** — The index number (integer) of the first image to read from an array of images. (Default: 0, the first image)

---

**NOTE** If *Sub\_Img* is used to request a subimage that is not in the image file, the *status* key of the returned *image* associative array is set to a negative number.

---

**Unmap** — When you read in an image containing a 2D pixel array and no colormap, a colormap is created and your pixel data is mapped into this colormap. When nonzero, this keyword restores your original pixel values and places a standard grayscale colormap in the colormap field of the image associative array.

**Verbose** — If nonzero, any available information about the file is printed to the screen.

## Returned Value

*image* — An associative array in image format.

## Discussion

---

**NOTE** Refer to the `IMAGE_CREATE` function for detailed information on the structure of the image associative array.

---

The following table lists the file formats that you can read and write using `IMAGE_READ` and `IMAGE_WRITE` functions, respectively. The table also specifies the conversions that are performed on particular file types.

File type	Pseudo-color (8-bit with colormap)		Pseudo-color (16-bit with colormap)		Direct color (24- bit)		Support subimages (multiple images)	
	Read	Write	Read	Write	Read	Write	Read	Write
BMP	y	y	n	n	y	y	n	n
DICM	y	y	y	y	y	y	y(+)	y(+)
GIF	y	y	n	n	y	y	y	n
JPEG	y	y	n	n	y	y	n	n
MIFF	y	y	n	n	y	y	y	n
PCD	y	n	n	n	y	n	y	n
PCX	y	y	n	n	y	y	y	n
PNG	y	y	n	n	y	y	n	n
SUN	y	y	n	n	y	y	y	n
TGA	y	y(c)	n	n(c)	y	y	y	n
TIFF	y	y	y	y	y	y	y	n
XWD	y(*)	y(*)	n(*)	n(*)	y(*)	y(*)	n	n
XPM	y(c, *)	y(*)	n(c, *)	n(*)	n/a	n/a	n	n
XBM	y	y(bw)	n	n(bw)	n/a	n/a	n	n

c — Converted to 24-bit direct color on read/write. (NOTE: In order to read an XPM file, the DISPLAY environment variable must be set.)

bw — Reduces the image to black and white on write.

n — Not supported.

n/a — File format does not support this class.

\* — Not supported for Windows 95 and Windows NT.

+ — Uses *Sub\_Img* option to read a sub-image one at a time. *All\_Subimages* option is not supported.

---

**NOTE** The TIFF 6.0 Specification (“the Standard”) is used. There are eleven required fields for grayscale and these are the only fields used in processing. Type checking for fields is performed, but only Types 1-12 are supported. A required field must have a type conforming to the Standard. Only positive integer data is supported. The ‘pixel’ field is stored in a long array.

---

Most image files have an encoded ID number to identify the file format. When IMAGE\_READ reads an image file, it determines the type of file according to the rules of precedence used by the function IMAGE\_QUERY\_FILE.

---

**NOTE** Filetype TGA does not have an encoded ID number. To read this type of file you must specify the *File\_type* keyword or *filename* must have a *.tga* suffix.

---

If at least the first requested subimage in the image file is available, but the number of requested images exceeds the highest subimage number in the file, a warning message is displayed and the number of images in the file is returned in the *img\_count* key of the returned image associative array.

### Example 1

```
flowers = IMAGE_READ('flowers.tif')
; Reads the file flowers.tif in the current directory and creates
; the associative array flowers in image format.
```

### Example 2

```
cells = IMAGE_READ('tiff16.tif', file_type='TIFF16')
; Reads the file tiff16.tif in the current directory and creates
; the associative array cells in image format.
```

### See Also

[IMAGE\\_CREATE](#), [IMAGE\\_DISPLAY](#), [IMAGE\\_WRITE](#), [DICM\\_TAG\\_INFO](#)

System Variables: [!Quiet](#)

---

## **IMAGE\_WRITE Function**

Writes an image variable to a file.

### Usage

```
status = IMAGE_WRITE(filename, image)
```

### Input Parameters

*filename* — A string specifying the pathname and filename of the image file to write. This parameter can also be a null string (see *Discussion*).

*image* — An associative array in image format.

### Returned Value

*status* — A value indicating the success or failure of the function.

- < 0 Indicates an error, such as an invalid input array. An informational message describing the error is also output to the screen.

0 Indicates a successful write.

---

**NOTE** IMAGE\_WRITE will process the file as a 16-bit TIFF file if the depth field is 16 and the file type is *tif*. All other keywords are ignored. This assumes *image* was imported with IMAGE\_READ. Only the eleven required fields for grayscale images are processed.

---

## Keywords

**Compress** — If nonzero, and if the requested image file type allows compression, the image file is written in compressed form. (See *Discussion*.)

**File\_type** — A string specifying an image file type. See the *Discussion* section for a list of valid image file types.

**Order** — If nonzero, the image is mirrored in the *y*-axis direction before being written to the file. (Default: The bottom of the image is stored at array index 0, per the PV-WAVE convention.)

**Overwrite** — If the specified file already exists, overwrite it.

**Quality** — Sets the quality level of JPEG compression. (Default: 100)

**Quiet** — Suppresses successive levels of error messages, depending on the value set. This keyword accepts the same integer values used with the system variable !Quiet.

**Verbose** — If nonzero, information about the written image is displayed.

## Discussion

Before writing a file, the IMAGE\_WRITE function checks to make sure that the input array is a valid image associative array, and that the keys of the array contain data consistent with the expected values. Refer to the IMAGE\_CREATE function for detailed information on the structure of the associative array.

---

**NOTE** When bilevel or grayscale TIFF images with no colormap data are written to files, a color map is created and the pixel values are mapped into the colormap. You can recover the original values when the TIFF file is read back into PV-WAVE by using either IMAGE\_READ with the *Unmap* keyword or using DC\_READ\_TIFF.

---

The following table lists file formats that you can write and read using the IMAGE\_WRITE and IMAGE\_READ functions, respectively. The table also shows the conversions that are performed on particular file types.

File type	Pseudo-color (8-bit with colormap)		Pseudo-color (16-bit with colormap)		Direct color (24- bit)		Support subimages (multiple images)	
	Read	Write	Read	Write	Read	Write	Read	Write
BMP	y	y	n	n	y	y	n	n
DICM	y	y	y	y	y	y	y(+)	y(+)
GIF	y	y	n	n	y	y	y	n
JPEG	y	y	n	n	y	y	n	n
MIFF	y	y	n	n	y	y	y	n
PCD	y	n	n	n	y	n	y	n
PCX	y	y	n	n	y	y	y	n
PNG	y	y	n	n	y	y	n	n
SUN	y	y	n	n	y	y	y	n
TGA	y	y(c)	n	n(c)	y	y	y	n
TIFF	y	y	y	y	y	y	y	n
XWD	y(*)	y(*)	n(*)	n(*)	y(*)	y(*)	n	n
XPM	y(c, *)	y(*)	n(c, *)	n(*)	n/a	n/a	n	n
XBM	y	y(bw)	n	n(bw)	n/a	n/a	n	n

c — Converted to 24-bit direct color on read/write. (NOTE: In order to read an XPM file, the DISPLAY environment variable must be set.)

bw — Reduces the image to black and white on write.

n — Not supported.

n/a — File format does not support this class.

\* — Not supported for Windows 95 and Windows NT.

+ — Uses *Sub\_Img* option to read a sub-image one at a time. *All\_Subimages* option is not supported.

---

**NOTE** The input parameter, *image*, must be created with either the IMAGE\_READ or IMAGE\_CREATE routine.

---

The rules of precedence for determining the file type are:

1. Use the value of the *File\_Type* keyword.

2. Use the suffix given as part of the filename given in the *filename* parameter. The suffix will be recognized if it is for one of the supported file types.
3. Use the file type specified in the *file\_type* key of the image associative array.
4. Use the suffix given in the *file\_name* key of the image associative array.

If the *filename* parameter is a null string, the filename is taken from the *file\_name* key of the image associative array. In this case, the filename suffix will be changed to match the file type as determined by the rules of precedence above.

---

**TIP** You can use the *File\_Type* keyword to convert image files from one format to another. For example, if you read a JPEG file, you can convert it to a Sun Rasterfile by writing it with the following command:

---

```
status = IMAGE_WRITE('', image, File_Type = 'SUN')
```

If the image has been created from the file `picture.jpeg`, this command will write a file `picture.sun`.

Available compression algorithms include:

Supported Image File Types	Compression Algorithm
BMP, PCX	RLE
JPEG	JPEG
TIFF (bilevel only)	Packbits

---

**NOTE** LZW compression GIF and TIFF files is a proprietary format and is not available.

---

## Example

```
stat = IMAGE_WRITE('flowers.sun', flowers)
      ; Writes an image file flowers.sun in Sun Rasterfile format, in the
      ; current directory.
```

## See Also

[IMAGE\\_CREATE](#), [IMAGE\\_DISPLAY](#), [IMAGE\\_READ](#), [DICM\\_TAG\\_INFO](#)

---

## **IMAGINARY Function**

Returns the imaginary part of a complex number.

### **Usage**

*result* = IMAGINARY(*complex\_expr*)

### **Input Parameters**

*complex\_expr* — A single or double-precision complex number (one with both a real and imaginary part). Can be of any dimension.

### **Returned Value**

*result* — The imaginary part as a single or double-precision floating-point value. It is of the same dimension as *complex\_expr*.

### **Keywords**

None.

### **Discussion**

If the input parameter, *complex\_expr*, is single-precision complex, the result is single-precision. If *complex\_expr* is double-precision complex, the result is double-precision.

IMAGINARY can be used for a variety of applications — one example is using it to find the phase angle of a complex result, such as a filter, by dividing the arctangent of the imaginary part by the real part.

### **Example**

```
x = COMPLEX(0, 2)
PRINT, IMAGINARY(x)
      2.00000
```

## See Also

[CINDGEN](#), [COMPLEX](#), [DCOMPLEX](#)

---

## **IMG\_TRUE8 Procedure**

Generates a pseudo true-color image suitable for display on devices capable of displaying 256 simultaneous colors.

### **Usage**

IMG\_TRUE8, *red\_img*, *grn\_img*, *blu\_img*, *rgb\_img*, *red*, *grn*, *blu*

### **Input Parameters**

*red\_img* — A 2D image representing the red component of a true color image. *red\_img* must be the same size as *grn\_img* and *blu\_img*.

*grn\_img* — A 2D image representing the green component of a true color image.

*blu\_img* — A 2D image representing the blue component of a true color image.

### **Output Parameters**

*rgb\_img* — A pseudo true-color 8-bit image that can be displayed using the TV procedure (see *Discussion*).

*red* — The red component of the color table.

*grn* — The green component.

*blu* — The blue component.

### **Keywords**

None.

### **Discussion**

The image generated by IMG\_TRUE8 is suitable for display on devices with 8-bit planes of color. It is useful when you have Landsat type images that you want to merge into a true color system.

For correct appearance, *rgb\_img* should be displayed in a graphics window with *n* colors allocated to it, where *n* is 256 under UNIX and OpenVMS, and 236 under Windows. For example:

```
WINDOW, 0, Colors = 256
```

Also, the proper color table needs to be loaded by using the command:

```
TVLCT, red, grn, blu, 0
```

where *red*, *grn*, and *blu* are the values obtained from *IMG\_TRUE8*.

Then use the TV procedure to display the image:

```
TV, rgb_img
```

---

**NOTE** On some systems it may be necessary to click in the image window to see the proper colors.

---

## Examples

```
PRO img_demo1
    ; This program displays a pseudo true-color Landsat image on an
    ; 8-bit color system.

winx = 477
winy = 512
    ; Specify the window size.

red_img = BYTARR(winx, winy)
grn_img = BYTARR(winx, winy)
blu_img = BYTARR(winx, winy)
    ; Set up the color components for the true color image.

OPENR, 1, !Data_Dir + 'boulder_red.img'
READU, 1, red_img
CLOSE, 1

OPENR, 1, !Data_Dir + 'boulder_grn.img'
READU, 1, grn_img
CLOSE, 1

OPENR, 1, !Data_Dir + 'boulder_blu.img'
READU, 1, blu_img
CLOSE, 1
    ; Read in the data.

WINDOW, 0, Colors=256, XSize=winx, YSize=winy
    ; Set up the display window.
```

---

**Windows USERS** Under Windows, change 256 to 236.

---

```
IMG_TRUE8, red_img, grn_img, blu_img, $
    rgb_img, red, grn, blu
TVLCT, red, grn, blu, 0
TV, rgb_img
    ; Create and display the true color image.
END
```

---

**UNIX and OpenVMS USERS** To see an example using the same data, except displayed in true 24-bit color on a 24-bit X workstation, see the *PV-WAVE User's Guide*.

---

## See Also

[LOADCT](#)

---

**UNIX and OpenVMS USERS** For a comparison of pseudo- and true-color images, see the *PV-WAVE User's Guide*.

---

---

## ***INDEX\_AND Function***

Standard Library function that computes the logical AND for two vectors of positive integers.

### **Usage**

*result* = INDEX\_AND(*array*<sub>1</sub>, *array*<sub>2</sub>)

### **Input Parameters**

*array*<sub>1</sub> — A vector of positive integers.

*array*<sub>2</sub> — A vector of positive integers.

### **Returned Value**

*result* — A vector containing all elements common to both input arrays. (*result* is not unique'd.)

## Keywords

None.

## Examples

```
PM INDEX_AND( [2,0,3], [1,2,0,2] )
```

## See Also

[INDEX\\_OR](#), [WHEREIN](#)

---

## ***INDEX\_CONV Function***

Converts one-dimensional indices to  $n$ -dimensional indices, or  $n$ -dimensional indices to 1D indices.

### Usage

```
 $j = \text{INDEX\_CONV}( a, i )$ 
```

### Input Parameters

$a$  — An  $n$ -dimensional array.

$i$  — A vector of  $m$  one-dimensional indices into  $a$ , or an  $m$ -by- $n$  array of  $m$   $n$ -dimensional indices into  $a$ .

### Returned Value

$j$  — An  $m$ -by- $n$  array of  $m$   $n$ -dimensional indices into  $a$  (if  $i$  is one-dimensional) or a vector of  $m$  one-dimensional indices into  $a$  (if  $i$  is two-dimensional).

### Example

```
a = INDGEN( 2, 3 ) & j = INDEX_CONV( a, INDGEN(6) ) & PM, j
      0          0
      1          0
      0          1
      1          1
      0          2
```

```
      1          2
PM, INDEX_CONV( a, j )
      0
      1
      2
      3
      4
      5
```

---

## ***INDEX\_OR Function***

Standard Library function that computes the logical OR for two vectors of positive integers.

### **Usage**

*result* = INDEX\_OR(*array*<sub>1</sub>, *array*<sub>2</sub>)

### **Input Parameters**

*array*<sub>1</sub> — A vector of positive integers.

*array*<sub>2</sub> — A vector of positive integers.

### **Returned Value**

*result* — A vector consisting of all elements contained in either of the input arrays (*result* is unique'd).

### **Keywords**

None.

### **Examples**

```
PM, INDEX_OR( [2,0,3], [1,2,0,2] )
```

### **See Also**

[INDEX\\_AND](#), [WHEREIN](#)

---

## INDGEN Function

Returns an integer array with the specified dimensions.

### Usage

*result* = INDGEN(*dim*<sub>1</sub>, ..., *dim*<sub>*n*</sub>)

### Input Parameters

*dim*<sub>*i*</sub>— The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### Returned Value

*result* — An initialized integer array. If the resulting array is treated as a one-dimensional array, then its initialization is given by the following:

$$\text{array}(i) = i, \text{ for } i = 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right)$$

### Keywords

None.

### Example

This example creates a 4-by-2 integer array.

```
a = INDGEN(4, 2)
; Create an integer array.
```

```
INFO, a
VARIABLE      INT      = Array(4, 2)
PRINT, a
   0   1   2   3
   4   5   6   7
```

### See Also

[BINDGEN](#), [CINDGEN](#), [DINDGEN](#), [FINDGEN](#), [LINDGEN](#), [SINDGEN](#)

---

## INFO Procedure

Displays information on many aspects of the current PV-WAVE session.

### Usage

INFO, *expr*<sub>1</sub>, ... , *expr*<sub>*n*</sub>

### Input Parameters

*expr*<sub>*i*</sub> — The expressions specifying the type of information to be displayed. These expressions are interpreted differently, depending on the keyword selected. If no keyword is selected, INFO displays basic information for its parameters.

### Keywords

**Breakpoints** — Displays the breakpoint table containing the name of the program module, line number and source file name of each breakpoint. If *Breakpoints* is set to a specified variable, the information is stored in a string array of the variable name rather than displayed.

**Calls** — Displays the current procedure's call stack containing the name of the program module, source file name, and line number. If *Calls* is set to a specified variable, the information is stored in a string array of the variable name rather than displayed. The first array element contains the information about the caller of the INFO command, the second element contains information about its caller, and so on.

If *Level* is specified with *Calls*, the information displayed is for the current routine on the level specified.

---

**NOTE** *Calls* is useful for programs that require traceback information.

---

**Depth** — Displays the depth of the current procedure's call stack. If *Depth* is set to a specified variable, the depth of the procedure call stack is stored in a long scalar of the variable name rather than displayed.

**Device** — Lists all available graphics devices, and displays information about the currently selected graphics device. If *Device* is set to a specified variable, the list of all available graphics devices is stored in a string array of the variable name rather than displayed.

**Files** — Displays information about file units depending on the input parameters supplied in the calling sequence. If input parameters are supplied, the value for *Files* is taken to be integer file-unit numbers, and information on the specified units

is displayed. If no input parameters are supplied, information about all open file units is displayed.

If *Files* is set to a specified variable, the names of all currently open files are stored in a string array of the variable name rather than displayed.

If *Names* is used, *Files* only displays information for the specified files.

**Full** — If nonzero, displays all information about specified variable elements including structures, lists, and associative arrays.

**Functions** — Displays a list of all currently available user functions. If *Functions* is set to a specified variable, the function names are stored in a string array of the variable name rather than displayed.

**Keys** — Displays current function key definitions as set using the DEFINE\_KEY procedure:

If input parameters are supplied, *Keys* must be scalar strings containing the names of function keys, and information on the specified keys is displayed.

If no input parameters are supplied, information on all function keys is displayed.

**Level** — Specifies the level of the program for which information is to be displayed. This keyword is used in conjunction with the keywords *Calls*, *Parameters*, *Routines*, *Traceback*, *Upvar*, and *Variables*.

If  $n \geq 0$ , the level is counted from the \$MAIN\$ level to the current procedure.

If  $n < 0$ , the level count is relative, counting from the current procedure back to the \$MAIN\$ level.

**Memory** — Reports the amount of dynamic memory currently in use by the PV-WAVE session, and the number of times dynamic memory has been allocated and deallocated. If *Memory* is set to a specified variable, it stores the information in a three-element long array of the variable name.

**Names** — Specifies patterns to be matched against strings which are to be displayed. This keyword is used in conjunction with other keywords. Patterns specified by *Names* can contain the following wild card characters.

\* (asterisk) — Matches any string.

? (question mark) — Matches any character.

**Parameters** — Displays the list of all parameters for the current procedure or function. The procedure or function must be specified as an input parameter for INFO, or *Level* must be used; otherwise, *Parameters* defaults to the \$MAIN\$ program level. If *Parameters* is set to a specified variable, the parameter names are stored in a string array of the variable name rather than displayed.

If *Level* is specified, only information about the procedure or function on

the specified level is displayed.

If *Names* is used, only parameters with names matching those specified are displayed.

**Procedures** — Displays a list of all procedures available in the current session. If *Procedures* is set to a specified variable, the procedure names are stored in a string array of the variable name rather than displayed.

If *Names* is used, only procedures with names matching those specified are displayed.

**Recall\_Commands** — Displays the currently saved commands. Input parameters are ignored.

**Routines** — Displays a list of all compiled procedures and functions with their parameter names. Keywords accepted by each module are enclosed in quotation marks. If *Routines* is set to a specified variable, the procedure and function names are stored in a string array of the variable name rather than displayed. The parameter names associated with the functions and procedures are not stored in the string array.

If *Level* is specified, only information about the procedures or functions on the specified level is displayed.

If *Names* is used, only the procedures and functions with names matching those specified are displayed.

**Structures** — Displays information on the structure of variables depending on the input parameters specified in the calling sequence. If input parameters are supplied, the structure of those expressions is displayed. If no input parameters are supplied, all currently defined structures are shown.

**Sysstruct** — Displays information on all system structures (structures that begin with '!'). This keyword is a subset of the *Structures* keyword.

**System\_Variables** — Displays information on all system variables. Input parameters are ignored. If *System\_Variables* is set to a specified variable, the system variable names are stored in a string array rather than displayed.

If *Names* is used, only the system variables with names matching those specified are displayed.

**Traceback** — Displays the call stack of the current procedure, which contains the program module name, source file name, and the line number. If *Traceback* is set to a specified variable, the information is stored in a string array of the variable name rather than displayed. The first array element contains the information about the caller of the INFO command, the second element contains information about its caller, and so on.

If *Level* is specified, only information about the procedure or function on

the specified level is displayed.

***Upvar*** — Displays the name of a variable from the previous program level that was passed as a parameter into the procedure or function that calls INFO. If *Upvar* is a string array, the name of the variable passed between the program levels is stored rather than displayed. If the variable doesn't exist on the previous program level, an empty string is returned.

*Upvar* can display the name of a variable on other program levels, if the *Level* keyword is also specified.

***Userstruct*** — Displays information on the regular user-defined structures (structures that do not begin with '!'). This keyword is a subset of the *Structures* keyword.

***Variables*** — Displays a list of all variables of the current function or procedure. The procedure or function must be specified either by using the input parameter of the INFO command, or by using *Level*; otherwise, *Variables* defaults to the \$MAIN\$ program level. If *Variables* is set to a specified variable, the procedure or function variable names are stored in a string array rather than displayed.

If *Level* is specified, only information about the procedure or function on the specified level is displayed.

If *Names* is used, only variables with names matching those specified are displayed.

## Discussion

You select information on a specific area by specifying the appropriate keyword from the above list. Only one keyword may be specified at a time.

If no input parameters or keywords are specified, INFO shows the current nesting of procedures and functions, all current variables at the current program level, and open files.

## See Also

[DEFINE\\_KEY](#), [DOC\\_LIBRARY](#)

For more information and examples, see the *PV-WAVE Programmer's Guide*.

---

## ***INTARR Function***

Returns an integer vector or array.

## Usage

*result* = INTARR(*dim*<sub>1</sub>, ... , *dim*<sub>*n*</sub>)

## Input Parameters

*dim*<sub>*i*</sub> — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

## Returned Value

*result* — An integer vector or array with the dimensions specified by *dim*<sub>*i*</sub>.

## Keywords

*Nozero* — If *Nozero* is nonzero, the normal zeroing is not performed. This causes INTARR to execute faster.

## Discussion

Normally, INTARR sets every element of the result to zero.

## Example

```
result = INTARR(2, 3)
PRINT, result
  0  0
  0  0
  0  0
```

## See Also

[BYTARR](#), [FLTARR](#), [INDGEN](#), [LONARR](#)

---

## INTERPOL Function

Standard Library function that performs a linear interpolation of a vector using either a regular or irregular grid.

## Usage

$result = \text{INTERPOL}(v, n)$

This form is used with a regular grid.

$result = \text{INTERPOL}(v, x, u)$

This form is used with an irregular grid.

## Input Parameters

For a regular grid:

$v$  — The dependent values of the vector that is to be interpolated. Must be one-dimensional. Can be of any data type except string.

$n$  — The number of interpolated points.

For an irregular grid:

$v$  — The dependent values of the vector that is to be interpolated. Must be one-dimensional. Can be of any data type except string.

$x$  — The independent values of the vector that is to be interpolated. Must have the same number of values as  $v$ , and must be monotonic, either increasing or decreasing.

$u$  — The independent values at which interpolation is to occur. Is not necessarily monotonic.

## Returned Value

For a regular grid:

$result$  — A vector containing  $n$  points interpolated from vector  $v$ .

For an irregular grid:

$result$  — A vector containing the same number of values as  $u$ .

## Keywords

None.

## Discussion

The result returned by INTERPOL differs depending on whether a regular or an irregular grid was used, as described below.

- **Regularly-gridded vectors.** The vector resulting from INTERPOL used with a regular grid is calculated in the following way, using the FIX function:

$$result(i) = v(j) + (j - FIX(j)) (v(j + 1) - v(j))$$

where

$$j = i(m-1)/(n-1)$$

$i$  is in the range  $0 \leq i \leq (n-1)$

$m$  is the number of points in the input vector  $v$ .

The output grid horizontal coordinates (abscissae values) are calculated in the following way, using the FLOAT function:

$$abscissa\_value(i) = FLOAT(i) / m$$

where  $i$  is in the range  $0 \leq i < (n - 1)$ .

- **Irregularly-gridded vectors.** The vector resulting from INTERPOL used with an irregular grid has the same number of elements as  $j$  and is calculated in the following way, using the FIX function:

$$result(i) = v(j) + (j - FIX(j)) (v(j + 1) - v(j))$$

where  $j = u(i)$ .

## Example

INTERPOL can be used in signal processing to reconstruct the signal between original samples of data. For example, suppose you have 5 sample data readings, [0, 5, 10, 5, 0], but you need 15 samples.

To get these additional samples, enter the following command:

```
out_vector = INTERPOL(in_vector, 15)
```

where `in_vector` has a value of [0, 5, 10, 5, 0] and

`out_vector` yields the following values:

```
[0, 1.42857, 2.85714, 4.28571, 5.71429, 7.14286, 8.57143, 10.0000,
 8.57143, 7.14286, 5.71429, 4.28571, 2.85714, 1.42857, 0]
```

The data still starts and ends with zero, and the maximum is still ten, but all the points in between have been recalculated.

## See Also

[BILINEAR](#), [SPLINE](#)

---

## ***INTERPOLATE Function***

Standard Library function that interpolates scattered data at scattered locations.

### **Usage**

*result* = INTERPOLATE(*d*, *x*)

### **Input Parameters**

*d* — An (m, n+1) array of m datapoints in n independent variables and one dependent variable; *d*(\*,*n*) is the dependent variable.

*x* — A (p,n) array specifying p interpolation points.

### **Returned Value**

*result* — A 1d array of values of the dependent variable at points *x*.

### **Keywords**

*r* — A scalar specifying the order of the weighting function. The dependent variable at an interpolation point is computed as a weighted average of the variable over all datapoints. The weighting function is  $1/e^r$  where *e* is the Euclidean distance between the interpolation point and the datapoint. *r* defaults to 2

### **Example**

```
x = findgen(51) / 50      &  plot, x, INTERPOLATE([[0,1],[2,3]],x)
```

### **See Also**

[GRIDN](#)

---

## ***INTRP Function***

Standard Library function that interpolates an array along one of its dimensions.

## Usage

$result = \text{INTRP}(a, n, x)$

## Input Parameters

$a$  — An array.

$n$  — An integer ( $\geq 0$ ) designating the dimension to interpolate.

$x$  — A one-dimensional array giving the coordinates at which to interpolate.

## Returned Value

$result$  — The array of interpolated slices perpendicular to dimension  $n$ .

## Keywords

$z$  — a strictly increasing 1d array of coordinates for dimension  $n$  (defaults to the indices into this dimension).

## Example

See `wave/lib/user/examples/intrp_ex`.

## See Also

[REBIN](#), [RESAMP](#)

---

## ***INVERT Function***

Returns an inverted copy of a square array.

## Usage

$result = \text{INVERT}(array [, status])$

## Input Parameters

$array$  — A two-dimensional square array. May be of any data type except string.

## Output Parameters

$status$  — (optional) The name of a scalar variable used to accumulate errors from singular or near-singular arrays. Possible values are:

- 0 Successful completion.
- 1 A singular array, indicating that the inversion is invalid.
- 2 A warning that a small pivot element was used and it is likely that significant accuracy was lost.

## Returned Value

*result* — An inverted copy of *array*.

## Keywords

None.

## Discussion

An input array of double-precision floating-point data type returns a result of identical type. An input array of any other type yields a result of single-precision floating-point data type.

Errors are accumulated in the optional *status* parameter, or the math error status indicator. This latter status may be checked using the `CHECK_MATH` function.

---

**CAUTION** Unless double-precision floating-point values are used for *array*, round off and truncation errors may occur, resulting in imprecise inversion.

---

`INVERT` uses the Gaussian elimination method (whose objective is the transformation of the given system into an equivalent system with upper-triangular coefficient matrix).

## See Also

[CHECK\\_MATH](#), [DETERM](#), [LUBKSB](#), [LUDCMP](#), [TRANSPOSE](#)

---

## ***ISASKEY*** Function

Matches a key name in a given associative array.

## Usage

*result* = `ISASKEY`(*asarr*, *key*)

## Input Parameters

*asarr* — The name of an associative array.

*key* — A string to be matched against the key names in the given associative array.

## Returned Value

*result* — A value indicating success or failure of the match.

- < 0 Indicates the string matches the key name in the associative array.
- 0 Indicates no match is found.

## Keywords

None.

## Discussion

Use this function to determine if a particular key name exists in a given associative array. A key name is the name associated with an element in the associative array. To create an associative array, use the ASARR function.

## Example

ISASKEY is used to determine if the associative array contains the element named *struct*. If it does, then the value of *struct* is replaced with zero. The INFO command is used to verify the change.

```
asar1 = ASARR('byte', 1B, 'float', 2.2, 'string', '3.3', $
           'struct', {,a:1, b:lindgen(2)})
IF ISASKEY(asar1, 'struct') THEN asar1('struct') = 0
INFO, asar1, /Full
ASAR1 AS. ARR = Associative Array(4)
  byte BYTE = 1
  struct INT = 0
  float FLOAT = 2.20000
  string STRING = '3.3'
```

## See Also

[ASARR](#), [ASKEYS](#), [LIST](#)

---

## ISHFT Function

Performs the bit shift operation on bytes, integers, and longwords.

### Usage

*result* = ISHFT(*p*<sub>1</sub>, *p*<sub>2</sub>)

### Input Parameters

*p*<sub>1</sub> — The scalar or array to be shifted.

$p_2$  — The scalar containing the number of bit positions and direction of the shift.

## Returned Value

*result* — The shifted scalar or array value.

## Keywords

None.

## Discussion

If  $p_2$  is positive,  $p_1$  is left-shifted  $p_2$  bit positions, with 0 bits filling vacated positions. If  $p_2$  is negative,  $p_1$  is right-shifted, again with 0 bits filling vacated positions.

## Example

In this example, ISHFT is used to multiply and divide each element of a five-element vector by powers of 2.

```
a = BYTARR(5)
      ; Create and initialize a five-element byte array.
FOR i = 0, 4 DO a(i) = 4 * i
PRINT, a
      0  4  8  12  16
PRINT, a, ISHFT(a, -2)
      0  4  8  12  16
      0  1  2  3  4
      ; Divide each element of A by 4.
PRINT, a, ISHFT(a, 1)
      0  4  8  12  16
      0  8  16  24  32
      ; Multiply each element of A by 2.
```

## See Also

[SHIFT](#)

---

## **JACOBIAN Function**

Standard Library function that computes the Jacobian of a function represented by  $n$   $m$ -dimensional arrays

### **Usage**

$j = \text{jacobian}(f)$

### **Input Parameters**

$f$  — an  $n$ -element list of  $m$ -dimensional arrays all of the same dimensions  $d$ ;  $f$  represents a  $n$ -valued function of  $m$  variables.

### **Returned Value**

$j$  — an  $n$ -element list of  $m$ -element lists of  $m$ -dimensional arrays:  $(j(p))(q)$  is the  $m$ -dimensional array (of dimensions  $d$ ) which represents the derivative of the  $p^{\text{th}}$  dependent variable with respect to the  $q^{\text{th}}$  independent variable.

### **Keywords**

$x$  —  $m$ -element list of vectors defining the independent variables; by default,  $x(i) = \text{findgen}(d(i))$ .

### **Example**

```
f = list( randomu(s,10,20), randomu(s,10,20), randomu(s,10,20) )
j = jacobian( f )
for p=0,2 do for q=0,1 do pm, same( (j(p))(q), derivn(f(p),q) )
```

### **See Also**

[CURVATURES](#), [DERIVN](#), [EUCLIDEAN](#), [NORMALS](#)

---

## **JOURNAL Procedure**

Provides a record of an interactive session by saving in a file all text entered from the terminal in response to a prompt.

## Usage

JOURNAL [, *param*]

## Input Parameters

*param* — (optional) A string parameter whose use depends on whether journaling is in progress when JOURNAL is called, and whether *param* is explicitly set:

- If journaling is not in progress and *param* is supplied, *param* sets the name of the journal file into which the session's commands will be written.
- If journaling is not in progress and *param* is not supplied, the default journal file named `wavesave.pro` is used.
- If journaling is in progress and *param* is supplied, the contents of the *param* string is written directly into the currently open journal file.
- If journaling is in progress and *param* is not supplied, the current journal file is closed and the logging process is terminated.

## Keywords

*Nobuffer* — If present and nonzero, output lines will be written immediately to the journal file without the normal file buffering.

## Discussion

The first call to JOURNAL starts the logging process. The read-only system variable `!Journal` is set to the file unit into which all session commands are written. Once the logging is initiated, a call to JOURNAL with no parameters closes the log file and terminates the logging process. If logging is in effect and a parameter is supplied, the parameter is simply written to the journal file.

## See Also

[RESTORE](#), [SAVE](#)

Also see the *PV-WAVE User's Guide*.

---

## ***JUL\_TO\_DT Function***

Converts a Julian day number to a date/time variable.

## Usage

*result* = JUL\_TO\_DT(*julian\_day*)

## Input Parameters

*julian\_day* — A Julian day number or array of Julian day numbers.

## Returned Value

*result* — A date/time variable containing the converted data.

## Keywords

None.

## Discussion

The date/time value is interpreted as a day in a series of days that begins on September 14, 1752. For example, 2 is equated with September 15, 1752. The decimal part of the Julian day indicates the time as a portion of the day. For example, for May 1, 1992 at 8:00 a.m, the Julian day is 84702.333.

## Example

```
dt = JUL_TO_DT(87507)
    ; Converts the Julian day 87507 into a date/time variable.
print, dt
    { 1992 4 15 0 0 0.00000 87507.000 0 }
```

## See Also

[SEC\\_TO\\_DT](#), [STR\\_TO\\_DT](#), [VAR\\_TO\\_DT](#)

For more information, see the *PV-WAVE User's Guide*.

---

## **KEYWORD\_SET Function**

Tests if an input expression has a nonzero value.

## Usage

*result* = KEYWORD\_SET(*expr*)

## Input Parameters

*expr* — The expression to be tested. Usually a named variable.

## Returned Value

*result* — A nonzero value, if *expr* is defined and nonzero.

## Keywords

None.

## Discussion

KEYWORD\_SET is especially useful in user-written procedures and functions when you want to process keywords that can be interpreted as being either true (keyword is present and nonzero) or false (keyword was not used, or was set to zero).

It can also be used to test whether an expression evaluates to zero, or whether a variable has been set.

## Example 1

Assume you type the following commands:

```
IF KEYWORD_SET(x) THEN PRINT, 'It is set' ELSE PRINT, 'Not set'
```

You will see:

```
Not set
```

because the value of *x* has not been initialized. On the other hand, if you set *x* to a specific value, say 2, you will see:

```
x = 2
```

```
IF KEYWORD_SET(x) THEN PRINT, 'It is set' ELSE PRINT, 'Not set'  
It is set
```

## Example 2

The following user-defined routine uses KEYWORD\_SET to make a call to print the results of a squaring operation:

```
FUNCTION SQUARE_IT, Value, Print_Flag = Print
```

```

    Squared_Val = Value * Value
    IF (KEYWORD_SET(Print)) THEN PRINT, Squared_Val
    RETURN, Squared_Val
END

```

To run this routine, enter the following commands at the WAVE> prompt:

```

.RUN SQUARE_IT.PRO
y = SQUARE_IT(5)
PRINT, y
    25
y = SQUARE_IT(10, /Print_Flag)
    100

```

## See Also

[N\\_ELEMENTS](#), [N\\_PARAMS](#), [PARAM\\_PRESENT](#)

## LCM Function

Standard Library function that returns the least common multiple of some integers greater than 1.

### Usage

*result* = LCM(*i*)

### Input Parameters

*i* — An array of integers greater than 1.

### Returned Value

*result* — An integer, the least common multiple of the integers *i*.

### Keywords

None.

### Example

```
pm, LCM( [3,2,4] )
```

## See Also

[FACTOR](#), [GCD](#), [PRIME](#)

---

## LEEFILT Function

Standard Library function that performs image smoothing by applying the Lee Filter algorithm.

### Usage

```
result = LEEFILT(image [, n, sigma])
```

### Input Parameters

*image* — A 1D, 2D, or 3D array containing a signal; signal-interleaved signals; an image; image-interleaved images; or a volume.

*n* — (optional) The value of  $2n + 1$  is used for the side of the filter box. The side of the filter box must be smaller than the smallest dimension of *image*. (Default: 5)

*sigma* — (optional) The estimate of the standard deviation. The value must be a positive. (Default: 5)

---

**NOTE** If *sigma* is negative, you will be prompted for a value to be typed in, the value will be displayed, and the filtered image will be displayed with the TVSCL command. This cycle will continue until a zero value of *sigma* is entered.

---

### Returned Value

*result* — A two-dimensional array containing the smoothed image.

### Keywords

*Edge* — A scalar string indicating how edge effects are handled. (Default: 'copy') Valid strings are:

'zero' — Sets the border of the output image to zero.

'copy' — Copies the border of the input image to the output image. (Default)

***Intleave*** — A scalar string indicating the type of interleaving of 2D input signals containing signal-interleaved signals; and 3D input arrays containing image-interleaved images, or a volume. Valid strings and the corresponding interleaving methods are:

'signal' — The 2D input *image* array arrangement is  $(x, p)$  for  $p$  signal-interleaved signals of length  $x$ .

'image' — The 3D *image* array arrangement is  $(x, y, p)$  for  $p$  image-interleaved images of  $x$ -by- $y$ .

'volume' — The input image array is treated as a single entity.

## Discussion

LEEFILT performs image smoothing by applying the Lee Filter algorithm. This algorithm assumes that the sample mean and variance of a value is equal to the local mean and variance of all values within a fixed range surrounding it. LEEFILT smooths additive signal noise by generating statistics in a local neighborhood and comparing them to the expected values.

Since LEEFILT is not very computationally expensive, it can be used for near real-time image processing. It can be used on signals as well as images.

## See Also

[MEDIAN](#), [SMOOTH](#), [TVSCL](#)

For details on the Lee Filter, see the article by Jong-Sen Lee, “Digital Image Enhancement and Noise Filtering by Use of Local Statistics,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Volume PAMI-2, Number 2, pages 165-168, March 1980.

---

## LEGEND Procedure

Standard Library procedure that lets you put a legend on a plot or graph.

## Usage

LEGEND, *label* [, *col*, *lintyp*, *psym*, *data\_x*, *data\_y*, *delta\_y*]

## Input Parameters

***label*** — A row of labels, one for each data line. This parameter may be characters or numbers.

***col*** — (optional) An array containing the color to be used for each row of the legend. If *col* is omitted (or if there are fewer colors than labels), the system variable !P.Color is used.

***lintyp*** — (optional) An array containing the values to be used in drawing each line of the legend. If *lintyp* is omitted (or if there are fewer line types than labels), the system variable !P.Linestyle is used.

***psym*** — (optional) An array containing the value of the plot symbols to be used in the legend. The plot symbols correspond to the values of the system variable !P.Psym. If *psym* is omitted (or if there are fewer plot symbols than labels), the system variable !P.Psym is used.

***data\_x*** — (optional) The *x*-coordinate of the upper-left corner of the legend in data coordinates. If *data\_x* is omitted, you are prompted for a value.

***data\_y*** — (optional) The *y*-coordinate of the upper-left corner of the legend in data coordinates. If *data\_y* is omitted, you are prompted for a value.

***delta\_y*** — (optional) The vertical spacing between the lines of the legend, expressed in data coordinates. If *delta\_y* is omitted, you are prompted for a value.

## Keywords

None.

## Discussion

You have control of the color, line type, and plotting symbol for each row of the legend using the available keywords. The size of the text is controlled by the system variable !P.Charsize.

## See Also

[PLOT](#), [XYOUTS](#)

For more information, see [Chapter 4, System Variables](#).

---

## LINDGEN Function

Returns a longword integer array with the specified dimensions.

### Usage

*result* = LINDGEN(*dim*<sub>1</sub>, ... , *dim*<sub>*n*</sub>)

### Input Parameters

*dim*<sub>*i*</sub>— The dimensions of the result. This parameter may be any scalar expression, and can have up to eight dimensions specified.

### Returned Value

*result*— An initialized longword integer array. If the resulting array is treated as a one-dimensional array, then its initialization is given by:

$$array(i) = \text{LONG}(i), \text{ for } i = 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right)$$

### Keywords

None.

### Example

This example creates a 4-by-2 longword integer array.

```
a = LINDGEN(4, 2)
      ; Create longword integer array.

INFO, a
VARIABLE      LONG      = Array(4, 2)
PRINT, a
      0   1   2   3
      4   5   6   7
```

### See Also

[BINDGEN](#), [CINDGEN](#), [DINDGEN](#), [FINDGEN](#), [INDGEN](#), [INTARR](#),  
[SINDGEN](#)

---

## LINKNLOAD Function

Provides simplified access to external routines in shareable images and Dynamic Link Libraries (DLLs).

### Usage

```
result = LINKNLOAD(object, symbol [, param1, ..., paramn])
```

### Input Parameters

*object* — A string specifying the filename, optionally including file path, of the DLL or shared object file to be linked and loaded.

*symbol* — A string specifying the function name symbol entry point to be invoked in the DLL or shared object file.

*param<sub>i</sub>* — (optional) The data to be passed as a parameter to the function. Any data type except structure can be used.

### Returned Value

*result* — A scalar whose default is assumed to be of type longword unless it is specified with one of the keywords described below.

### Keywords

*Default* — (VMS only) Changes the default object file specification. On VMS, there are two ways to specify the object parameter. It may be a logical name (defined in the system logical table with the /EXEC attribute) including the full path to the file. Or the object parameter may be a filename, without any directory or extension. In the latter case, the default file specification of SYS\$SHARE:.EXE is applied to the object file name. You may use the *Default* keyword to change this. For example:

```
result = LINKNLOAD('EXAMPLE', $
    Default = 'SYS$LOGIN:[LIBS].EXE', symbol, ...)
```

*D\_Value* — Specifies that the returned scalar is of type double-precision floating point.

*F\_Value* — Specifies that the returned scalar is of type single-precision floating point.

***Nocall*** — (UNIX, Windows only) If present and nonzero, LINKNLOAD will not call the function defined by the object and symbol parameters. LINKNLOAD will still try to find the object module and load it. Thus, you can use this keyword to do just the loading of the object module. If used with *Unload*, the object module is unloaded without calling the function. This is useful because you can recompile the object module, unload the old object module, and reload the new one without exiting the PV-WAVE session.

***S\_Value*** — Specifies that the returned scalar is of type string.

***Unload*** — (UNIX, Windows only) If present and nonzero, LINKNLOAD unloads the object module immediately before returning from the LINKNLOAD command.

***Value*** — (byte array) Allows you to specify which parameters should be passed by value, instead of by reference, which is the default. *Value* must be a byte array with one element for each parameter in the call. A parameter is passed by value if the corresponding byte in the *Value* array is non-zero. Array parameters must always be passed by reference.

***Verbose*** — (UNIX, Windows only) If present and nonzero, LINKNLOAD prints status information when an object module is being loaded or unloaded.

***Vmscall*** — When present and nonzero, tells PV-WAVE to use the LIB\$CALLG system routine to call the object. This makes parameter passing significantly more convenient for FORTRAN programmers working in an OpenVMS environment. This keyword is ignored on UNIX platforms.

***Vmsstrdesc*** — When present and nonzero, tells PV-WAVE to pass string parameters as OpenVMS FORTRAN string descriptors. This keyword is ignored on UNIX platforms.

---

**NOTE** In general, LINKNLOAD cannot know what language an object module being called was created from. The *Vmsstrdesc* keyword tells LINKNLOAD that your object module was created from FORTRAN code and that it is expecting string descriptors.

---

## Discussion

LINKNLOAD provides a simple, yet powerful, mechanism for dynamically invoking your own code from PV-WAVE on all of its supported operating systems. For more information on the keywords *Unload*, *Nocall*, and *Verbose*, refer to the following file:

**(UNIX)**       \$VNI\_DIR/wave/demo/interapp/linknload/  
              lnl\_newkeywords.doc

**(OpenVMS)**   VNI\_DIR: [WAVE.DEMO.INTERAPP.LINKNLOAD]  
              LNL\_NEWKEYWORDS.DOC

---

**Windows USERS** The LINKNLOAD function provides simplified access to external routines in Dynamic Link Libraries (DLLs). LINKNLOAD calls a function in a DLL and returns a scalar value. Parameters are passed through PV-WAVE to the specified external function by reference, thus allowing the external function to alter values of PV-WAVE variables. It is the simplest method for attaching your own C code to PV-WAVE.

---

**UNIX and OpenVMS USERS** LINKNLOAD calls a function in an external sharable object and returns a scalar value. It is the simplest method for attaching your own C code to PV-WAVE.

Parameters are passed through PV-WAVE to the specified external function by reference, thus allowing the external function to alter values of PV-WAVE variables.

---

**CAUTION** Be careful to ensure that the number, type, and dimension of the parameters passed to the external function match what it expects (this can most easily be done from within PV-WAVE before calling LINKNLOAD). Furthermore, the length of string parameters must not be altered and multi-dimensional arrays are flattened to one-dimensional arrays.

---

## Accessing the Data in PV-WAVE Variables

Two methods exist for accessing the results generated by PV-WAVE in a user-written application called with LINKNLOAD.

One of these methods is to use the `wavevars` function and the other is to use the C-callable or FORTRAN-callable programming interface.

---

**NOTE** For detailed information on these methods, see the *PV-WAVE Application Developer's Guide*.

---

## Programming Notes

---

**UNIX USERS** For AIX, the symbol entry point must be specified when the external shareable object is built, by using the `-e` flag, and thus the function symbol parameter to LINKNLOAD has no effect on AIX.

---

If you run PV-WAVE and call LINKNLOAD and then relink your C function (or C wrapper) and try to call LINKNLOAD again in the same session, PV-WAVE will crash. You must exit and then re-run PV-WAVE for the newly linked C routine to work.

Variables that are shared between PV-WAVE and a C function must be created by PV-WAVE and their size can not be modified by the C function.

It is possible to pass a constant as a parameter to a C function from PV-WAVE via LINKNLOAD, but of course the C function can not pass a value back via that parameter.

Although `wavevars` returns pointers to the data associated with PV-WAVE's variables, it should be kept in mind that these addresses must be treated as "snapshots" because the data pointer associated with a particular PV-WAVE variable may change after execution of certain PV-WAVE system commands.

Using LINKNLOAD, care must be taken to ensure that the number, type, and dimension of the parameters passed to the function match what the function expects (this can most easily be done from within PV-WAVE before calling LINKNLOAD). Furthermore, the length of string parameters must not be altered and multi-dimensional arrays are flattened to one-dimensional arrays.

## UNIX/OpenVMS Examples

The following PV-WAVE code demonstrates how to invoke a C function using LINKNLOAD. For more information on compiling shareable objects, see the *PV-WAVE Application Developer's Guide*.

In this example, parameters are passed to the C external function using the conventional (*argc*, *argv*) UNIX strategy. *argc* indicates the number of data pointers which are passed from PV-WAVE within the array of pointers called *argv*. The pointers in *argv* can be cast to the desired type as the following program demonstrates.

You can find the following listed file in:

```
$WAVE_DIR/util/linknload/example.c
```

```

#include <stdio.h>
typedef struct complex {
    float r, i;
} complex;
long WaveParams(argc,argv)
    int argc;
    char *argv[];
{
    char *b;
    short *s;
    long *l;
    float *f;
    double *d;
    complex *c;
    char **str;
    if (argc != 7) {
        fprintf(stderr,"wrong # of parameters\n");
        return(0);
    }

    b = ((char **)argv) [0];
    s = ((short **)argv) [1];
    l = ((long **)argv) [2];
    f = ((float **)argv) [3];
    d = ((double **)argv) [4];
    c = ((complex **)argv) [5];
    str = ((char ***)argv) [6];
    fprintf(stderr,"%d %d %ld %g %g <%g%gi> '%s'\n", (int)
        b[0],(int)s[0],(long)l[0], f[0],d[0],c[0].r,c[0].i,str[0]);
    return(12345);
}

```

### ***Accessing the External Function with LINKNLOAD***

The following PV-WAVE code demonstrates how the C function defined in the previous discussion could be invoked.

```

ln = LINKNLOAD('example.so','WaveParams', $
    byte(1),2,long(3), float(4),double(5), $
    complex(6,7),'eight')

```

The resulting output is:

```

1 2 3 4 5 <6,7i> 'eight'

```

Using the INFO command, you can see that LINKNLOAD returns the scalar value 12345, as expected.

```
INFO, ln
      LN      LONG      =      12345
```

The example program works with both scalars and arrays since the actual C program above only looks at the first element in the array and since PV-WAVE collapses multi-dimensional arrays to one-dimensional arrays:

```
ln = LINKNLOAD('example.so', 'WaveParams', $
              [byte(1)], [[2,3], [4,5]], [long(3)], $
              [float(4)], [double(5)], [complex(6,7)], $
              ['eight'])
```

The resulting output is:

```
1 2 3 4 5 <6,7i> 'eight'
```

## Windows Examples

Example programs showing how LINKNLOAD is used to pass parameters from PV-WAVE to an external C function and return results to PV-WAVE can be found in the directory:

```
(Windows) <wavedir>\demo\interapp\win32\linknload
```

Where <wavedir> is the main PV-WAVE directory.

This directory contains an example C program, a PV-WAVE procedure file, a makefile, and a README file. See the README file for details on running the example.

For more information on the example and on LINKNLOAD, see the *PV-WAVE Application Developer's Guide*.

## Other Examples

For examples showing the use of the keywords *Vmscall*, *Vmsstrdesc*, and *Value*, refer to the following files online:

```
(Windows) <wavedir>\demo\interapp\win32\linknload\call*
```

Where <wavedir> is the main PV-WAVE directory.

## See Also

[SIZE](#), [wavevars](#) (in the *PV-WAVE Application Developer's Guide*)

---

## **LIST Function**

Creates a list array.

### **Usage**

*result* = LIST(*expr*<sub>1</sub>, ... *expr*<sub>*n*</sub>)

### **Input Parameters**

*expr*<sub>*i*</sub> — One or more expressions or variables.

### **Returned Value**

*result* — A variable of type list.

### **Keywords**

None.

### **Discussion**

A list is an array of expressions or variables. Each element in a list can have a unique data type and value. The elements in a list can be accessed with subscripts, in much the same way that elements of an array are accessed.

The elements of a list can be any of the eight basic PV-WAVE data types, or other structures or arrays of structures, and other lists or associative arrays. Lists can also be used as structure fields.

### **Example**

This example creates a simple list with four elements: a byte value, a floating-point value, a string, and a structure. Both INFO and PRINT are used to show the contents and structure of the list.

```
lst = LIST(1B, 2.2, '3.3', {,a:1, b:lindgen(2)})  
    ; Create the list.  
INFO, lst, /Full  
    ; Display information about the list.  
LST LIST = List(4)  
    BYTE = 1
```

```

FLOAT = 2.20000
STRING = '3.3'
STRUCT = ** Structure $1, 2 tags, 24 length:
  A INT 1
  B LONG Array(2)
PRINT, lst
  ; Print the value of the list.
{ 1      2.20000 3.3{      1      0      1}}

```

## See Also

[ASARR](#), [ASKEYS](#), [ISASKEY](#)

## LISTARR Function

Returns a list.

### Usage

*result* = LISTARR(*number\_elements*, [*value*])

### Input Parameters

*number\_elements* — The number of elements the created list should contain. Must be a scalar expression.

*value* — (optional) The value with which to initialize each element of the list. May be any data type.

### Returned Value

*result* — A list with the requested number of elements.

### Keywords

None.

### Discussion

Values in the list are initialized to 0L unless *value* is specified.

## Example

```
INFO, LISTARR( 2, 10 ), /Full
```

---

## ***LN03 Procedure (UNIX/OpenVMS)***

Standard Library procedure that opens or closes an output file for LN03 graphics output. The file can then be printed on an LN03 printer.

### Usage

```
LN03 [, filename]
```

### Input Parameters

*filename* — (optional) The name of the file that will contain the LN03 graphics output.

### Keywords

None.

### Example

To open a file for LN03 output, enter the following command:

```
LN03, 'myfile'
```

where `myfile` is the name of the file that will be sent to the LN03 printer.

To close the output file, call the procedure without a parameter

---

## ***LOADCT Procedure***

Standard Library procedure that loads a predefined color table.

### Usage

```
LOADCT [, table_number]
```

### Input Parameters

*table\_number* — (optional) A number between 0 and 15; each number is associated with a predefined color table.

## Keywords

*Ctfile* — Specifies a string containing the name of a color table file to load.

*Silent* — If present and nonzero, suppresses the message indicating that the color table is being loaded.

## Discussion

Predefined color tables are stored in the file `colors.tbl`. There are 16 predefined color tables, with indices ranging from 0 to 15, as shown in the following table.

To see a menu listing of these color tables, call `LOADCT` with no parameters.

Number	Name
0	Black and White Linear
1	Blue/White
2	Green/Red/Blue/White
3	Red Temperature
4	Blue/Green/Red/Yellow
5	Standard Gamma-II
6	Prism
7	Red/Purple
8	Green/White Linear
9	Green/White Exponential
10	Green/Pink
11	Blue/Red
12	16 Level
13	16 Level II
14	Steps
15	PV-WAVE Special

## Examples

```
LOADCT, 3  
; Loads the Red Temperature color table.
```

```
LOADCT, 11, /Silent  
; Loads the Blue/Red color table without displaying a message.
```

## See Also

[TVLCT](#), [COLOR\\_EDIT](#), [MODIFYCT](#), [STRETCH](#), [TEK\\_COLOR](#), [WgCtTool](#)

For more information, including a comparison of LOADCT and TVLCT, see the *PV-WAVE User's Guide*.

---

## **LOADCT\_CUSTOM Procedure**

Loads a predefined custom color table.

### **Usage**

LOADCT\_CUSTOM [, *table\_number*]

### **Input Parameters**

*table\_number* — (optional) An integer number representing the index of the color table to load, from 0 to 31. If omitted, a menu of the available color tables is displayed, and you are prompted to enter a color table number.

### **Keywords**

*Silent* — If present and non-zero, suppresses the colortable message.

### **Discussion**

LOADCT\_CUSTOM loads the selected custom color table by reading the custom color table file located in your home (login) directory. This file is called `.wg_colors` or `wg_colors`. The procedure `WgCeditTool` can be used to create this custom color table file.

The custom color table file has a similar structure to the system color table file `colors.tbl`. The custom file can contain up to 32 (0 – 31) color tables.

The title of each table is contained in the first 32-by-32 character bytes. The colors loaded into the display are saved in the common block `COLORS`. If the current device has fewer than 256 colors, the color table data are interpolated to cover the number of colors in the device.

### **See Also**

[LOADCT](#)

---

## **LOAD\_HOLIDAYS Procedure**

Makes the value of the !Holiday\_List system variable available to the date/time routines.

### **Usage**

LOAD\_HOLIDAYS

### **Parameters**

None.

### **Keywords**

None.

### **Discussion**

Run LOAD\_HOLIDAYS after you:

- Restore a PV-WAVE session in which you used the CREATE\_HOLIDAYS function. Running this procedure makes the restored !Holiday\_List system variable available to the date/time routines.
- Directly change the value of the !Holiday\_List system variable, instead of using the CREATE\_HOLIDAYS procedure.

This procedure is called by the CREATE\_HOLIDAYS function. Thus, CREATE\_HOLIDAYS both creates holidays and makes them available to the date/time routines.

---

**NOTE** If all weekdays have been defined as weekend days, an error results.

---

### **Example**

This example shows how you might directly modify the system variable !Holiday\_List and then run LOAD\_HOLIDAYS to make the change available. Assume that December 26 has been erroneously defined as a holiday and that it is the first date/time structure in !Holiday\_List. To change this holiday to December 25:

```
!holiday_list(0).DAY = byte(25)
```

```
; Manually change the Day field of the first date/time structure in  
; !Holiday_List to the 25th. The Day field of this structure contains a  
; byte value.
```

---

**CAUTION** Whenever you directly modify the date in a !DT structure, set the `Recalc` field (the last field in the !DT structure) to 1. This causes the Julian date to be recalculated. If the Julian date is not recalculated, plots or date/time calculations that use the modified variable may be inaccurate.

---

```
!holiday_list(0).RECALC = byte(1)  
; Manually set the Recalc field of the structure to 1. This field  
; contains a byte value.  
  
LOAD_HOLIDAYS  
; Run LOAD_HOLIDAYS so the new holiday value will take effect.
```

## See Also

[CREATE\\_HOLIDAYS](#), [CREATE\\_WEEKENDS](#), [LOAD\\_WEEKENDS](#)

For more information, see the *PV-WAVE User's Guide*.

---

## **LOAD\_OPTION Procedure**

Explicitly loads an Option Programming Interface (OPI) optional module.

### **Usage**

`LOAD_OPTION`, *option\_name*

### **Input Parameters**

*option\_name* — (string) The name of the Option to be loaded.

### **Keywords**

**Load\_Now** — If specified and nonzero, all the procedures/functions from the Option are loaded. By default, the procedures/functions from the Option are loaded when they are referenced for the first time.

## Discussion

The `LOAD_OPTION` procedure explicitly loads an OPI option. OPI options can be loaded explicitly by any PV-WAVE user. These optional modules can be written in C or FORTRAN, and can contain new system functions or other primitives. For detailed information on creating OPI options, see the *PV-WAVE Application Developer's Guide*.

## Example

```
LOAD_OPTION, 'SAMPLE'
```

## See Also

[OPTION\\_IS\\_LOADED](#), [SHOW\\_OPTIONS](#), [UNLOAD\\_OPTION](#)

---

## **LOADRESOURCES Procedure**

Loads resources from a resource file.

### Usage

```
LOADRESOURCES, file
```

### Input Parameters

*file* — The name of the resource file to be loaded.

### Returned Value

None.

### Keywords

*Appdir* — A string that specifies the application directory name. This is the directory in which the application searches for resource files, string resource files, and icon files. (Default: 'vdatools')

*Subdir* — A string specifying a resource file subdirectory. (Default: !Lang, whose default is 'american').

## Discussion

By default, the function looks for *file* first in directories specified by the environment variable `WAVE_RESPATH`.

---

**UNIX USERS** The `WAVE_RESPATH` environment variable is a colon-separated list of directories, similar to the `WAVE_PATH` environment variable in **PV-WAVE**. If not found in a `WAVE_RESPATH` directory, the directory `<wavedir>/xres/!Lang/vdatools` is searched, where `<wavedir>` is the main **PV-WAVE** directory and `!Lang` represents the value of the `!Lang` system variable (`!Lang` default is 'american').

---

**OpenVMS USERS** The `WAVE_RESPATH` logical is a comma-separated list of directories and text libraries, similar to the `WAVE_PATH` logical in **PV-WAVE**. If not found in a `WAVE_RESPATH` directory, the directory `<wavedir>:[XRES.!Lang.VDATOOLS]` is searched, where `<wavedir>` is the main **PV-WAVE** directory and `!Lang` represents the value of the `!Lang` system variable (`!Lang` default is 'american').

---

**Windows USERS** The `WAVE_RESPATH` environment variable is a semicolon-separated list of directories, similar to the `WAVE_PATH` environment variable in **PV-WAVE**. If not found in a `WAVE_RESPATH` directory, the directory `<wavedir>\xres\!Lang\vdatools` is searched, where `<wavedir>` is the main **PV-WAVE** directory and `!Lang` represents the value of the `!Lang` system variable (`!Lang` default is 'american').

---

If *Subdir* alone is specified, the file is searched for in:

**(UNIX)** `<wavedir>/xres/subdir/vdatools`  
**(OpenVMS)** `<wavedir>:[XRES.SUBDIR.VDATOOLS]`  
**(Windows)** `<wavedir>\xres\subdir\vdatools`

Where `<wavedir>` is the main **PV-WAVE** directory.

If only *Appdir* is specified, the application searches for resources in the following directory:

**(UNIX)** `<wavedir>/xres/!Lang/appdir`  
**(OpenVMS)** `<wavedir>:[XRES.!Lang.APPDIR]`  
**(Windows)** `<wavedir>\xres\!Lang\appdir`

Where `<wavedir>` is the main **PV-WAVE** directory.

If both *Subdir* and *Appdir* are specified, the application searches for resources in the following directory:

**(UNIX)**        <wavedir>/xres/subdir/appdir

**(OpenVMS)** <wavedir>:[XRES.SUBDIR.APPDIR]

**(Windows)** <wavedir>\xres\subdir\appdir

Where <wavedir> is the main PV-WAVE directory.

If the file to be loaded is not already in the resource database, it is loaded and added to the resource database list of files.

---

**NOTE** LOADRESOURCES keeps a list of the loaded files so that files aren't redundantly loaded.

---

## Example

These calls load resources and strings for a Printer Setup dialog box. This example assumes that the resource and string files are located in:

**(UNIX)**        <wavedir>/xres/american/vdatools

**(OpenVMS)** <wavedir>:[XRES.AMERICAN.VDATOOLS]

**(Windows)** <wavedir>\xres\american\vdatools

Where <wavedir> is the main PV-WAVE directory.

```
LOADRESOURCES, 'printsetup.ad'
```

```
    ; Load resources.
```

```
LOADSTRINGS, 'printsetup.ads'
```

```
    ; Load the strings.
```

## See Also

[BUILDRESOURCEFILENAME](#), [LOADSTRINGS](#),  
[WwResource](#) (in the *PV-WAVE Application Developer's Guide*)

---

## LOADSTRINGS Procedure

Lloads strings from a resource file.

## Usage

LOADSTRINGS, *file*

## Input Parameters

*file* — The name of the string resource file to be loaded.

## Keywords

**Appdir** — A string that specifies the application directory name. This is the directory in which the application searches for resource files, string resource files, and icon files. (Default: 'vdatools')

**Subdir** — Specifies a subdirectory in which to look for the resource file. (Default: !Lang, whose default is 'american')

## Discussion

By default, the function looks for *file* first in directories specified by the environment variable WAVE\_RESPATH.

---

**UNIX USERS** The WAVE\_RESPATH environment variable is a colon-separated list of directories, similar to the WAVE\_PATH environment variable in PV-WAVE. If not found in a WAVE\_RESPATH directory, the directory <wavedir>/xres/!Lang/vdatools is searched, where <wavedir> is the main PV-WAVE directory and !Lang represents the value of the !Lang system variable (!Lang default is 'american').

---

**OpenVMS USERS** The WAVE\_RESPATH logical is a comma-separated list of directories and text libraries, similar to the WAVE\_PATH logical in PV-WAVE. If not found in a WAVE\_RESPATH directory, the directory <wavedir>:[XRES.!Lang.VDATOOLS] is searched, where <wavedir> is the main PV-WAVE directory and !Lang represents the value of the !Lang system variable (!Lang default is 'american').

---

**Windows USERS** The WAVE\_RESPATH environment variable is a semicolon-separated list of directories, similar to the WAVE\_PATH environment variable in PV-WAVE. If not found in a WAVE\_RESPATH directory, the directory <wavedir>\xres\!Lang\vdatools is searched, where <wavedir> is the main PV-WAVE directory and !Lang represents the value of the !Lang system variable (!Lang default is 'american').

---

If *Subdir* alone is specified, the file is searched for in:

(UNIX) <wavedir>/xres/subdir/vdatools

(OpenVMS) <wavedir>:[XRES.SUBDIR.VDATOOLS]

**(Windows)** <wavedir>\xres\subdir\vdatoools

Where <wavedir> is the main PV-WAVE directory.

If only *Appdir* is specified, the application searches for resources in the following directory:

**(UNIX)** <wavedir>/xres/!Lang/appdir

**(OpenVMS)** <wavedir>:[XRES.!Lang.APPDIR]

**(Windows)** <wavedir>\xres\!Lang\appdir

Where <wavedir> is the main PV-WAVE directory.

If both *Subdir* and *Appdir* are specified, the application searches for resources in the following directory:

**(UNIX)** <wavedir>/xres/subdir/appdir

**(OpenVMS)** <wavedir>:[XRES.SUBDIR.APPDIR]

**(Windows)** <wavedir>\xres\subdir\appdir

Where <wavedir> is the main PV-WAVE directory.

If the file to be loaded is not already in the resource database, it is loaded and added to the resource database list of files.

This procedure functions as a wrapper to the LOADRESOURCES procedure with the *Strings* keyword set.

## Example

These calls load resources and strings for a Printer Setup dialog box. This example assumes that the resource and string files are located in:

**(UNIX)** <wavedir>/xres/american/vdatoools

**(OpenVMS)** <wavedir>:[XRES.AMERICAN.VDATOOLS]

**(Windows)** <wavedir>\xres\american\vdatoools

Where <wavedir> is the main PV-WAVE directory.

```
LOADRESOURCES, 'printsetup.ad'  
    ; Load resources.
```

```
LOADSTRINGS, 'printsetup.ads'  
    ; Load the strings.
```

## See Also

[BUILDRESOURCEFILENAME](#), [LOADRESOURCES](#), [STRLOOKUP](#)

---

## ***LOAD\_WEEKENDS Procedure***

Makes the value of the !Weekend\_List system variable available to the date/time routines.

### **Usage**

LOAD\_WEEKENDS

### **Parameters**

None.

### **Keywords**

None.

### **Discussion**

Run this procedure after you:

- Restore any PV-WAVE session in which you used the CREATE\_WEEKENDS function. Running this procedure makes the restored !Weekend\_List system variable available to the date/time routines.
- Directly change the value of the !Weekend\_List system variable, instead of using the CREATE\_WEEKENDS procedure.

This procedure is called by the CREATE\_WEEKENDS function. Thus, CREATE\_WEEKENDS both creates weekends and makes them available to the date/time routines.

---

**NOTE** If all weekdays have been defined as weekend days, an error results.

---

### **Example**

```
CREATE_WEEKENDS, 'sat'  
    ; Create the !Weekend_List system variable and define Saturday  
    ; as a weekend.  
  
PRINT, !Weekend_List  
    0  0  0  0  0  0  1  
    ; Current contents of !Weekend_List system variable.
```

```
!Weekend_List = [1, 0, 0, 0, 0, 0, 1]
                ; Manually add Sunday to the weekend list.

LOAD_WEEKENDS
                ; Run LOAD_WEEKENDS so the new weekend value will take effect.
```

## See Also

[CREATE\\_HOLIDAYS](#), [CREATE\\_WEEKENDS](#), [LOAD\\_HOLIDAYS](#)

For more information, see the *PV-WAVE User's Guide*.

---

## LONARR Function

Returns a longword integer vector or array.

### Usage

```
result = LONARR(dim1, ... , dimn)
```

### Input Parameters

*dim<sub>i</sub>* — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### Returned Value

*result* — A longword integer vector or array.

### Keywords

*Nozero* — If *Nozero* is nonzero, the normal zeroing is not performed. This causes LONARR to execute faster.

### Discussion

Normally, LONARR sets every element of the result to zero.

### Example

This example creates a 4-by-2 longword integer array. Note that all elements of the array are initialized to 0L.

```
a = LONARR(4, 2)
```

```

; Create a longword integer array.
INFO, a
      VARIABLE      LONG      = Array(4, 2)
PRINT, a
      0  0  0  0
      0  0  0  0

```

## See Also

[BYTARR](#), [FLTARR](#), [INTARR](#), [LINDGEN](#)

## LONG Function

Converts an expression to longword integer data type.

Extracts data from an expression and places it in a longword integer scalar or array.

### Usage

*result* = LONG(*expr*)

This form is used to convert data.

*result* = LONG(*expr*, *offset*, [*dim*<sub>1</sub>, ... , *dim*<sub>*n*</sub>])

This form is used to extract data.

### Input Parameters

To convert data:

*expr* — The expression to be converted.

To extract data:

*expr* — The expression from which to extract data.

*offset* — The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim*<sub>*i*</sub> — (optional) The dimensions of the result. The dimensions may be any scalar expression with up to eight dimensions specified.

### Returned Value

For data conversion:

*result* — A copy of *expr* converted to longword integer data type.

For data extraction:

*result* — If *offset* is used, LONG does not convert *result*, but allows fields of data extracted from *expr* to be treated as longword integer data. If no dimensions are specified, the result is scalar.

## Keywords

None.

## Discussion

Conversion usage:

If the values of *expr* are not within the range of a long integer, a misleading result occurs and a message may be displayed.

For example, suppose  $A = 2.0 \times 10^{31} + 2$ . The following commands,

```
B = LONG(A)
```

```
C = LONG(-A)
```

produce the erroneous results of

```
B = 2147483647
```

```
C = -2147483648
```

In addition, PV-WAVE does not check for overflow during conversion to longword integer data type.

## Example

In this example, LONG is used in two ways. First, LONG is used to convert a single-precision, floating-point array to type longword. Next, LONG is used to extract a subarray from the longword array created in the first step.

```
a = FINDGEN(6) + 0.5
```

```
    ; Create a single-precision, floating-point vector of length 6. Each  
    ; element has a value equal to its one-dimensional subscript plus 0.5.
```

```
PRINT, a
```

```
0.500000 1.500000 2.500000 3.500000 4.500000 5.500000
```

```
b = LONG(a)
```

```
    ; Convert a to type longword.
```

```
INFO, b
```

```
VARIABLE      LONG      = Array(6)
```

```
    ; Note that the floating-point numbers in a were truncated by LONG.
```

```
PRINT, b
```

```
0      1      2      3      4      5
```

```
    ; Extract the last four elements of b, and place them in a.
```

```

c = LONG(b, 8, 2, 2)
    ; Specify a 2-by-2 long array.
INFO, c
    VARIABLE      LONG      = Array(2, 2)
PRINT, c
    2            3
    4            5

```

## See Also

[BYTE](#), [COMPLEX](#), [DOUBLE](#), [FIX](#), [FLOAT](#), [LINDGEN](#), [LONARR](#)

For more information on data extraction, see the *PV-WAVE Programmer's Guide*.

## LUBKSB Procedure

Solves the set of  $n$  linear equations  $Ax = b$ . (LUBKSB must be used with the procedure LUDCMP to do this.)

### Usage

LUBKSB,  $a$ ,  $index$ ,  $b$

### Input Parameters

$a$  — The LU decomposition of a matrix, created by LUDCMP. Parameter  $a$  is not modified by calling this procedure.

$index$  — A vector, created by LUDCMP, containing the row permutations effected by the partial pivoting.

$b$  — On input,  $b$  contains the vector on the right-hand side of the equation. Must be of data type FLOAT; other data types will cause incorrect output.

### Output Parameters

$b$  — On output,  $b$  is replaced by the solution, vector  $x$ .

### Keywords

None.

## Discussion

LUBKSB must be used in conjunction with LUDCMP.

## Example

```
Function INNER_PROD, v1, v2
IF N_ELEMENTS(v1) NE 3 THEN goto, error1
IF N_ELEMENTS(v2) NE 3 THEN goto, error1
sum = 0.0
FOR i=0, 2 DO BEGIN
    sum = sum + v1(i) * v2(i)
END
RETURN, sum

error1:
    PRINT, 'vectors not 3d'
    RETURN, 0
END

arr = FINDGEN(3, 3)
arr(*, 0) = [1.0, -1.0, 3.0]
arr(*, 1) = [2.0, 1.0, 3.0]
arr(*, 2) = [3.0, 3.0, 1.0]
; Solutions to triplet of equations:
; x + 2y + 3z = 3
; -x + y + 3z = 0
; 3x + 3y + z = 6
rightvec = [3.0, 0.0, 6.0]

PRINT, ' '
PRINT, 'solving system '
PRINT, arr, ' * [x,y,z] = ', rightvec
PRINT, ' '
PRINT, ' '
tarr = arr
rvec = rightvec
LUDCMP, tarr, index, b
LUBKSB, tarr, index, rvec
PRINT, '... solution for ', rightvec, ' is ', rvec
s1 = rvec
PRINT, ' '
PRINT, ' '
PRINT, INNER_PROD(arr(0,*), s1)
```

```
PRINT, INNER_PROD(arr(1,*), s1)
PRINT, INNER_PROD(arr(2,*), s1)
PRINT, ' '
PRINT, ' '
END
```

## See Also

[LUDCMP](#), [MPROVE](#)

LUBKSB is based on the routine of the same name in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988. It is used by permission.

---

## LUDCMP Procedure

Replaces a real  $n$ -by- $n$  matrix,  $a$ , with the LU decomposition of a row-wise permutation of itself. For complex matrices, see the *PV-WAVE: IMSL Mathematics Reference* procedure LUFAC.

### Usage

LUDCMP,  $a$ ,  $index$ ,  $d$

### Input Parameters

$a$  — An  $n$ -by- $n$  matrix.

### Output Parameters

$a$  — On output,  $a$  is replaced by the LU decomposition of a row-wise permutation of itself.

$index$  — The vector which records the row permutation effected by the partial pivoting. The values returned for  $index$  are needed in the call to LUBKSB.

$d$  — An indicator of the number of row interchanges:

- +1 Indicates the number was even.
- 1 Indicates the number was odd.

### Keywords

None.

## Example

The preferred method of solving the linear set of equations  $A\mathbf{x} = \mathbf{B}$  is:

```
LUDCMP, A, indx, D
```

; Decompose square matrix A.

```
LUBKSB, A, indx, B
```

; Use LUBKSB function for forward and back substitution, replacing B with the result x.

## See Also

[LUBKSB](#), [MPROVE](#)

LUDCMP is based on the routine of the same name in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988. It is used by permission.

---

## **MAKE\_ARRAY Function**

Returns an array of specified type, dimensions, and initialization. It provides the ability to create an array dynamically whose characteristics are not known until run time.

### **Usage**

*result* = MAKE\_ARRAY([*dim*<sub>1</sub>,... , *dim*<sub>*n*</sub>])

### **Input Parameters**

*dim*<sub>*i*</sub> — (optional) The dimensions of the result. This may be any scalar expression with up to eight dimensions specified.

### **Returned Value**

*result* — An array of specified type, dimensions, and initialization.

### **Keywords**

*Byte* — If nonzero, sets the type of the result to byte.

*Complex* — If nonzero, sets the type of the result to complex single-precision floating-point.

*Dcomplex* — If nonzero, sets the type of the result to complex double-precision floating-point.

*Dimension* — A vector of 1 to 8 elements specifying the dimensions of the result.

*Double* — If nonzero, sets the type of the result to double-precision floating-point.

*Float* — If nonzero, sets the type of the result to single-precision floating-point.

*Index* — The resulting array is initialized with each element set to the value of its one-dimensional index.

*Int* — If nonzero, sets the type of the result to integer.

*Long* — If nonzero, sets the type of the result to longword integer.

*Nozero* — If nonzero, the resulting array is not initialized.

*Size* — A longword vector specifying the type and dimensions of the result. It consists of the following elements:

- The first element is equal to the number of dimensions of *Value*, and is zero if *Value* is scalar.
- The next elements contain the size of each dimension.
- The last two elements contain the type code and the number of elements in *Value*, respectively. Valid type code values are listed below for the *Type* keyword.

***String*** — If nonzero, sets the type of the result to string.

***Type*** — Sets the type of the result to be the type code entered

Type Code	Data Type
1	Byte
2	Integer
3	Longword integer
4	Single-precision floating-point
5	Double-precision floating-point
6	Complex single-precision
7	String
12	Complex double-precision

***Value*** — Initializes each element of the resulting array with the given value. Can be of any scalar type, including structure types.

## Discussion

The result type is taken from the *Value* keyword unless one of the other keywords that specify a type is also used. In that case, *Value* is coerced to be the type specified by this other keyword prior to initializing the resulting array.

---

**NOTE** The resulting type cannot be specified if *Value* is a structure.

---

If *Value* is specified, all elements in the resulting array are set to *Value*. If *Value* is not specified, all elements are set to zero. If *Index* is specified, each element is set to its index. If *Nozero* is specified, the resulting array is not initialized.

## Example

In this example, three different methods are used to create and initialize a 4-by-3 longword integer array.

```
a = MAKE_ARRAY(Size = [2, 4, 3, 3, 12], $
             Value = 5)
      ; The type of a is determined by the Size keyword.
```

```
INFO, a
      VARIABLE      LONG      = Array(4, 3)
```

```
PRINT, a
      5      5      5      5
      5      5      5      5
      5      5      5      5
```

```
b = MAKE_ARRAY(4, 3, Type = 3, Value = 5)
      ; The type of b is determined by the Type keyword.
```

```
INFO, b
      VARIABLE      LONG      = Array(4, 3)
```

```
PRINT, b
      5      5      5      5
      5      5      5      5
      5      5      5      5
```

; The type of c is determined by the *Value* keyword.

```
c = MAKE_ARRAY(4, 3, Value = 5L)
```

```
INFO, c
      VARIABLE      LONG      = Array(4, 3)
```

```
PRINT, c
      5      5      5      5
      5      5      5      5
      5      5      5      5
```

## See Also

[BINDGEN](#), [BYTARR](#), [CINDGEN](#), [COMPLEXARR](#), [DBLARR](#), [DINDGEN](#),  
[FINDGEN](#), [FLTARR](#), [INDGEN](#), [INTARR](#),  
[LINDGEN](#), [LONARR](#), [SINDGEN](#), [SIZE](#)

---

## MAP Procedure

Plots a map dataset.

### Usage

MAP

### Input Parameters

None.

### Keywords

*Axes* — If present and nonzero, draws coordinate axes on the map plot.

---

**CAUTION** The *Axes* keyword labels the points where meridians or parallels cross the border of a plot. Because of inaccuracies of the reverse projection algorithms used to accomplish this, the labeling can sometimes be omitted or be incorrect for wide area plots and certain projection methods. In general, when a smaller area is plotted, the labeling is correct.

---

**Background** — Specifies a solid background color when the *Projection* keyword is set to 99 (3D mapping onto sphere). If set to -1, creates a light source shaded background. The default is 0 (zero), no background.

**Boundary** — If specified, plots the boundary calculated by the *Exact* keyword. The color used is set by the *Gridcolor* keyword. This keyword has no effect if the *Exact* keyword is not set. *Boundary* is most useful in debugging, or if you wish to identify the possibly non-rectangular closed area specified by your map bounds. (Default: off)

**Center** — (float) Array containing a longitude and latitude value that defines the center of the geographic area to be displayed. This keyword is an alternative to using the *Range* keyword. Use this keyword in conjunction with the *Zoom* keyword to specify the center of the area to zoom in on. Default: [-50.0, 0.0].

**Color** — (integer) Specifies the plot color for lines. Default is !P.Color. If set to -1, the colors defined in the dataset are used. This keyword can specify a scalar or an array containing a different color for each line segment or polygon. See the *Discussion* section for more information.

**Data** — (string) Specifies the name of the map dataset to plot. Datasets provided with PV-WAVE include World Dataset II (`world_db`) and USGS Digital Line Graph Dataset (`usgs_db`). (Default: `world_db`)

**Exact** — If specified, MAP attempts to fit the map area exactly to the latitude/longitude lines specified by the *Range* or *Center/Zoom* keywords. The aspect ratio of the resulting map is also correct, and the largest map possible that fits in the window/position is produced. The *Exact* keyword projects points along the latitude/longitude boundary specified by the *Range* or *Center/Zoom* keywords and uses the minimum and maximum values of the projected data to establish the data coordinate system. The *Exact* keyword usually results in very precise map ranges, but can fail for some projections if a singularity (like one of the poles) exists within the range specified. If such a singularity exists, then the default range calculation is used. (Default: off)

**File\_Path** — (string) Specifies the path to a file that will contain the data extracted from the selected dataset. The data is stored in a binary XDR format, and can be read in again using the *Read\_Path* keyword.

**Filled** — If present and nonzero, uses the MAP\_POLYFILL procedure to create a polygon-filled map rather than a line (vector) plot. Of the two map datasets supplied with PV-WAVE, only the `usgs_db` dataset can be filled. To fill a state, you must set the COUNTY field to -1 (using the *Select* keyword). To fill multiple states, you must make separate calls to MAP.

**GridColor** — (integer) Specifies the line color of the grid overlay. The *GridColor* default is !P.Color.

**GridLat** — (integer) Specifies the default spacing of latitude lines, measured in degrees. (Default: 10 degrees)

**GridLines** — If present and nonzero, overlays a grid on the map projection.

**GridLong** — (integer) Specifies the default spacing of longitude lines, measured in degrees. (Default: 15 degrees)

**GridStyle** — (integer) Specifies the linestyle of the grid overlay, as shown in the following table:

Gridstyles

Index	X Windows Style	Windows Style
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes

### Gridstyles (Continued)

Index	X Windows Style	Windows Style
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

The *Gridstyle* default is !P.Linestyle.

**Image** — Specifies an image (2D array) to be warped around the map projection and displayed with the map lines. If the image array is not of type byte, it will be automatically passed through the BYTSCS procedure before being plotted. The image will be warped to exactly cover the area specified using the *Range* keyword.

**Parameters** — (double) Specifies an array of up to 10 elements containing parameters to be passed to a map projection. The only built-in projection methods that use these parameters are the conic projections. The parameters currently used by the built-in projections are defined as follows:

parameter(0) — The first secant intersection.  
(Default: 33 degrees north)

parameter(1) — The second secant intersection. (Default: 45 degrees north)

**Position** — (float) The position of the map within the plot device in normal coordinates. For example, `Position = [.1, .1, .9, .9]` leaves a border around the map 1/10 the screen size. The *Position* default is the value of !P.Position.

**Projection** — (integer) Specifies the type of projection (see the *Discussion* section for a table of projection types). (Default: 1, equidistant cylindrical)

**Radius** — (float) Specifies a 2D array of the same size as the one specified by the *Image* keyword. The radius array will be used to modify the shape of the sphere used with `Projection=99`. The size of the array should be normalized to values between zero and one for best results. This keyword can be used to create a sphere with geographical relief displayed. The map data may not line up exactly with the radius of the sphere.

**Range** — (float) [*lon1, lat1, lon2, lat2*] Array of four points representing the lower-left and upper-right longitude/latitude coordinates of the region to be displayed. Used to specify the exact geographical area to display. An alternative method of area selection is to use the *Center* and *Zoom* keywords.

**Read\_Path** — (string) Used to specify the name of a file created with the *File\_Path* keyword. The data contained in this file is projected and plotted. Specifying this keyword causes the *Data*, *Select*, and *Resolution* keywords to be ignored.

**Resolution** — (integer) The effect of this keyword depends on which map dataset is used, the default `world_db` dataset or the `usgs_db` dataset:

(If the `world_db` dataset is used, the default) Provides a way to speed up the projection and plotting of large datasets. Specifies the number of data points to skip when a dataset is plotted. For example, `Res=30` means skip every 30th point in the dataset. Specifying a larger value for this keyword results in faster plotting times, but poorer image quality. Specifying smaller values results in slower plotting times, but gives improved image quality. The default value is chosen based on the range of the map and is optimized for the 300,000 points in the `world_db` dataset. A reasonable range of values for the `world_db` dataset is between one and 30.

(If `Data='USGS_DB'`) Resolution is not a measure of how many points to skip when sampling, as is done with the default map dataset, but a measure of the minimum distance (in latitude/longitude) between points. *Resolution* values should range between 0.01 and 0.2 for good results. The sampling is not used for county boundaries, or for Hawaii and Alaska. The code to sample the data is slow, so should be used with the *File\_Path* keyword of the MAP procedure to save the sampled map, which can be later displayed with the *Read* keyword of the MAP procedure. (Default: off)

**Save** — If present and nonzero, saves the 3D scaling parameters in !P.T to allow overplotting of other data. This keyword is only used when the *Projection* keyword is set to 99 (3D mapping onto a sphere).

**Select** — (unnamed structure) Specifies a subset of the dataset to plot. See the *Discussion* for more information on this keyword.

**Stretch** — If present and non-zero causes the plot area to be scaled to the actual range of data points extracted from the dataset. This stretches any subset of data to occupy the entire plot area. (See the *Discussion* section for more information.)

**User** — (string) Specifies the name of a user-defined projection procedure. If this keyword is used, then the *Projection* keyword is automatically set to -1.

**Zoom** — (float) Specifies a factor by which the display is magnified or reduced around a center point specified with the *Center* keyword. For example, a factor 2 magnifies the region surrounding the center point by two times. The default value is 1.0.

## Standard Plotting Keywords

The following standard plotting keywords are used with MAP. See [Chapter 3, Graphics and Plotting Keywords](#) for more information.

LineStyle	Line_Fill	Spacing	Threshold
Pattern	NoData	Orientation	
Fill_Pattern	NoErase	Thick	

---

## Discussion

When you use the *Center* and *Zoom* keywords to specify map bounds, if the center latitude is either  $-90$  or  $90$ , it is assumed that you want a polar view, and the longitude range is forced to  $-180$  to  $180$ . In other words the *Zoom* parameter will only affect the range of latitudes about the pole and not the longitude range.

### Map Projections

You can produce the following map projections with the MAP procedure. To specify a projection, set the *Projection* keyword to the corresponding map projection index number.

PV-WAVE Map Projections

Index	Projection	Index	Projection
1	Equidistant Cylindrical	11	Oblique Azimuthal Equidistant Oblique
2	Lambert Conformal Conic	12	Polar Azimuthal Equidistant Oblique
3	Cylindrical Mercator	13	Polar Azimuthal Equal-Area
4	Sinusoidal	14	Oblique Azimuthal Equal-Area
5	Albers Equal-Area Conic	15	Transverse Mercator
6	Polyconic	16	Mollweide (Ellipsoid)
7	Polar Stereographic	99	Satellite (3D mapping onto a sphere)
8	Oblique Stereographic	-1	User-defined projection (automatically set if the <i>User</i> keyword is supplied)
9	Oblique Orthographical	0	No projection
10	Polar Orthographical		

---

## ***Map Stretch***

By default the data area used by MAP is a rectangle scaled to the values specified in the *Range* or the *Center* and *Zoom* keywords (converted to radians). This usually gives good results, but there is sometimes a border around the map due to the particular mapping projection algorithm being used. Specifying *Stretch* resizes the plot area to the actual range of the projected data extracted and causes the map to always fill the entire plot area. This can change the aspect ratio of the map (height to width ratio), but is useful when a map needs to exactly cover a specified area on the device, such as when a cylindrically projected map is displayed over an image which was displayed using the TV command.

## ***Subsetting Datasets***

The *Select* keyword is used to specify subset criteria for the map dataset. Datasets included with PV-WAVE include World Databank II (`world_db`) and USGS Digital Line Graph (`usgs_db`) datasets.

---

**NOTE** The World Databank II dataset is a subset of a public domain dataset provided by the U.S. Department of Commerce, merged with updated country data from the National Imagery and Mapping Agency (NIMA).

---

You can subset the `world_db` dataset by passing an unnamed structure via the *Select* keyword. The following table lists the possible fields and tags of this structure:

Fields and Tags

<b>Fields</b>	<b>Tags</b>
GROUP	CIL — Coastlines, islands, lakes
	BDY — International boundaries
	PBY — Primary/First Order (internal) boundaries such as, U.S. States).
	RIV — Major rivers
AREA	As of Version 7.0, this field is obsolete. If AREA is specified, it is ignored.

---

---

**NOTE** If the field is set to a null string, all tags are plotted.

---

You can subset the `usgs_db` dataset by passing an unnamed structure via the `Select` keyword. The following table lists the possible fields and tags of this structure

#### Fields and Tags

Fields	Tags
STATE	One or more FIPS codes for the states to plot (two-letter state abbreviations can also be used — AL, AK, AZ, AR, CA, etc.).
COUNTY	One or more FIPS codes for the counties to plot. (-1 draws all counties; 0 draws no counties).

For example:

```
MAP, Select={, GROUP:['CIL'], AREA:'ASIA' }  
    ; World Databank II data is subsetted to plot coastlines, islands, and lakes in Asia.  
  
MAP, Data='usgs_db', Range=[-81,39,-75,43], $  
    /Gridlines, Gridstyle=2, Gridlat=1, $  
    Gridlong=5, Select={,STATE:'PA',COUNTY:-1}  
    ; Plot USGS Digital Line Graph data of Pennsylvania.
```

---

**NOTE** If `COUNTY` is anything but 0, only the first state specified can be plotted. To plot multiple states/counties, you must make separate calls to `MAP`.

---

---

**CAUTION** This USGS data is from the USGS 1:2,000,000 Scale Digital Line Graph CD. Note that some errors exist in the original data. For instance, when filling counties along coastlines, the filling will be inexact because the original polygon data for each county did not include the coastline portion.

---

### **Map Colors**

The colors used when the `Color` keyword is set to -1 are colors that are assigned within the map dataset. If you want to use the dataset colors, you may have to use a custom color palette or the `TEK_COLOR` procedure. The colors returned by the datasets that are provided with `PV=WAVE` are as follows:

Color assignments in the `WORLD_DB` dataset:

- Coastlines/Islands/Lakes — Color = 1
- Rivers — Color = 2
- International Boundaries — Color = 3
- Primary Boundaries — Color = 4

Color assignments in the USGS\_DB dataset:

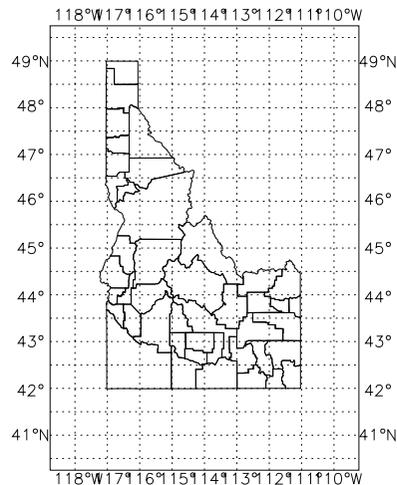
- For State plots (COUNTY tag field = 0): Color = State FIPS code
- For County plots: Color = County FIPS code

The colors are modulus the number of display colors if the FIPS codes exceed !D.N\_Colors.

## Examples

This example produces a filled map of Idaho. Keywords are used to set the map range, select the map dataset, subset the dataset, add gridlines and axes.

```
!P.position = [0.3, 0.1, 0.7, 0.9]
TEK_COLOR
MAP, Range = [-118.0, 40.5, -110.0, 50.0], $
    Data = 'usgs_db', $
    Select = {, state:'ID', county:-1}, $
    Thick = 2, /Axes, /Gridlines, $
    Gridcolor = 10, Gridstyle = 1, $
    Gridlong = 1, Gridlat = 0.5
```



**Figure 2-30** A map of Idaho plotted from the USGS Digital Line Graph Dataset. County boundaries are also plotted.

### ***File\_Path and Read\_Path Keywords***

The *File\_Path* and *Read\_Path* keywords allow you to create a data subset which can be read and reused without re-reading and re-subsetting the entire dataset.

These keywords can greatly increase the performance of your application. To use these keywords, specify a file pathname when you select a subset of the data, as in this example:

```
MAP, DATA='world_db', RANGE=[-150, 30, 30, 70], $  
    SELECT={, GROUP:'cil'}, $  
    RESOLUTION=20, FILE_PATH='mymap.dat'
```

You can now plot this map or subareas of this map without reading the entire world\_db dataset, as in this example:

```
MAP, Range=[-150, 30, 30, 70], Projection=4,$  
    Read_Path='mymap.dat'
```

## See Also

[MAP\\_CONTOUR](#), [MAP\\_PLOTS](#), [MAP\\_POLYFILL](#), [MAP\\_REVERSE](#),  
[MAP\\_VELOVECT](#), [MAP\\_XYOUTS](#), [USGS\\_NAMES](#)

For more information on mapping projections and in-depth discussions of algorithms and uses of the projections used with the MAP procedure, refer to:

*Map Projections Used by the U.S. Geological Survey*, Geological Survey Bulletin 1532, John P. Snyder, Second Edition, 1983.

*An Album of Map Projections*, U.S. Geological Survey Professional Papers 1453, John P. Snyder and Philip M. Voxland, 1989.

Both are available from:

USGS ESIC: Open File Report Sales

Box 25286, Building 810

Denver Federal Center

Denver, CO USA 80225

Phone: (303) 236-7476

FAX: (303) 236-4031

---

## MAP\_CONTOUR Procedure

Draws a contour plot from longitude/latitude data stored in a 2D array.

### Usage

MAP\_CONTOUR, z [, x, y]

### Input Parameters

*z* — A two-dimensional array containing the longitude/latitude values that make up the contour surface.

*x* — (optional) A vector specifying the longitude coordinates for the contour surface.

*y* — (optional) A vector specifying the latitude coordinates for the contour surface.

### Keywords

*Cartesian* — When this keyword is set, MAP\_CONTOUR first projects the grid of *z* values, grids the resulting projected points to a regular grid and performs the contouring in cartesian (data coordinate) space. This keyword allows you to get labeled contours and contours that can encircle a pole in a polar map. (Default: off)

*File\_Path* — (string) The file pathname for a temporary file used by MAP\_CONTOUR. This keyword is optional. See the *Discussion* section for more information.

*Filled* — If present and nonzero, fills the contours.

*Iter,*

*Nghbr* — These keywords are used in conjunction with the *Cartesian* keyword and control parameters to the FAST\_GRID3 function, which is used to perform gridding of the *z* grid data after applying the map projection.

See the FAST\_GRID3 documentation for a description of these parameters.

*Pattern* — A rectangular array of pixels giving the fill pattern.

### Standard Plotting Keywords

The following standard plotting keywords are used with MAP\_CONTOUR. See [Chapter 3, Graphics and Plotting Keywords](#) for more information

C_Annotation	C_Thick	NLevels
C_Colors	Font	Path_Filename
C_Charsize	Follow	Pattern
C_Labels	Levels	Spline
C_Linestyle	Max_Values	

---

## Discussion

If  $x$  and  $y$  are provided, the contour is plotted as a function of the longitude and latitude locations specified by  $x$  and  $y$ . Otherwise, the contour is assumed to occupy the entire range specified in the last call to the MAP procedure.

Each element of  $x$  specifies the longitude coordinate for a column of  $z$ . For example,  $X(0)$  specifies the longitude coordinate for  $Z(0, *)$ .

Each element of  $y$  specifies the latitude coordinate for a row of  $z$ .

In addition to line plot overlays, the *Filled* and *Pattern* keywords allow contours to be filled.

Before contours can be filled, they must be closed. This is usually accomplished by padding the data with zeros or some other value outside the range of the data, as the following code fragment illustrates, where  $m$  and  $n$  represent the dimensions of the original array.

```
new_array = REPLICATE(MIN(array1), m+2, n+2)
;Create a background array that is two elements larger than
;the original array, array1
new_array(1, 1) = array1
;Insert original data into the new array.
```

---

**TIP** You cannot use the *Filled* keyword to fill contours when the projection is set to 99 (Satellite View); however, it is possible to place filled contours on a satellite projection. To do this, create a 2D image using CONTOUR and CONTOURFILL, or by using the CONTOUR2 procedure with the *Fill* keyword, and then use the *Image* keyword with the MAP procedure to wrap the image on the globe.

---

The MAP\_CONTOUR procedure creates a temporary file when it is called that by default is named `wvctmp.dat` and placed in your current directory. You can override this filename and path using the *File\_Path* keyword with MAP\_CONTOUR.

This allows you to specify a string containing the path to a temporary file. This file is deleted when MAP\_CONTOUR is finished plotting.

## Example

In this simple example, a 2D array of data is created and superimposed as contours on a map projection.

```
data = HANNING(20,20)
      ; Create a 2D array of data.
MAP, Projection = 4, range = [-150, 30, 30, 70]
MAP_CONTOUR, data, NLevels=10
```

## See Also

[CONTOUR](#), [MAP](#), [POLYFILL](#)

---

## MAP\_PLOTS Procedure

Plots vectors or points (specified as longitude/latitude data) on the current map projection.

### Usage

```
MAP_PLOTS, x, y [, outx, outy]
```

### Input Parameters

**x** — A scalar or vector providing the longitude coordinates of the points to be connected.

**y** — A scalar or vector providing the latitude coordinates of the points to be connected.

### Output Parameters

**outx** — (optional) Returns an array containing the projected longitude data in data coordinates (radians).

**outy** — (optional) Returns an array containing the projected latitude data in data coordinates (radians).

## Keywords

**Cylinder** — If present and nonzero, specifies that lines be drawn treating longitude and latitude as Cartesian coordinates. This results in a straight line on a cylindrical projection. This keyword is useful if you want to draw your own latitude lines, since they will appear to conform to the current projection, rather than as straight lines or great circle lines.

**Distance** — Returns in a named variable the distance between the points provided as input to MAP\_PLOTS. If two points are provided, *Distance* returns a scalar result. If multiple points are provided as input, *Distance* returns an array of distance values.

**Km** — If present and nonzero, specifies that the distance returned by the *Distance* keyword be measured in kilometers. This is the default.

**Miles** — If present and nonzero, specifies that the distance returned by the *Distance* keyword be measured in miles. Default is kilometers.

**NoCircle** — If present and nonzero, specifies that straight lines be plotted between points. If set equal to zero (the default), MAP\_PLOTS draws great circle lines between any two specified points.

## Standard Plotting Keywords

The following standard plotting keywords are used with MAP\_PLOTS. See [Chapter 3, Graphics and Plotting Keywords](#), for more information.

Color	Nodata	Symsize
Linestyle	Psym	Thick

## Discussion

A valid data coordinate system (i.e., projection) must be established before MAP\_PLOTS is called. (A call to MAP can be used to establish this coordinate system.) Also note that a PV-WAVE graphics window must be open and selected when the call to MAP\_PLOTS is made for the procedure to work correctly.

The coordinates for MAP\_PLOTS must be given in longitude and latitude form.

The MAP\_PLOTS routine can plot lines as straight lines or as great circle lines, which appear on most projections as curved lines and represent the minimum distance between two points on the globe.

---

**CAUTION** The great circle lines depend on the accuracy of trigonometric functions, which for some very small longitude and latitude values or values close to 90 degrees can result in errors that can cause the great circle lines to be incorrectly drawn.

---

## Example

This example plots a map, then draws a great circle line between two points on the map.

```
MAP, Range = [-150, 30, 30, 70]
MAP_PLOTS, [-105.3, -0.1], [40.0, 51.5], $
    Distance = d, /Miles, Color = 5, $
    Psym = -2, Thick = 2
```

## See Also

[MAP](#)

---

## MAP\_POLYFILL Procedure

Fills the interior of a region of the display enclosed by an arbitrary 2D polygon.

### Usage

```
MAP_POLYFILL, x, y
```

### Input Parameters

*x* — A vector providing the longitude coordinates of the points to be connected.

*y* — A vector providing the latitude coordinates of the points to be connected.

### Standard Graphics Keywords

The MAP\_POLYFILL keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

<a href="#">Color</a>	<a href="#">Linestyle</a>	<a href="#">Orientation</a>	<a href="#">Spacing</a>
<a href="#">Fill_Pattern</a>	<a href="#">Line_Fill</a>	<a href="#">Pattern</a>	<a href="#">Thick</a>

## Z-buffer Specific Keywords

These keywords allow you to warp images over 2D or 3D polygons; the keywords are valid only when the Z-buffer device is active. For more information these keywords, refer to the description of the POLYFILL procedure.

Image_Coordinates	Mip
Image_Interpolate	Threshold

## Discussion

The polygon is defined by a list of connected vertices stored in *x* and *y*. The coordinates must be given in longitude/latitude form.

MAP\_POLYFILL uses various filling methods:

- solid fill
- parallel lines
- a pattern contained in an array
- hardware-dependent fill pattern

**Solid Fill Method** — Most devices can fill with a solid color. Solid fill is performed using the line fill method for devices that don't have this hardware capability. Keywords that specify a method are not required for solid filling.

**Line Fill Method** — Filling using parallel lines is device independent and works on all devices that can draw lines. Cross-hatching may be obtained with multiple fillings of differing orientations. The spacing, linestyle, orientation, and thickness of the filling lines may be specified using the corresponding keywords. The *Line\_Fill* keyword selects this filling style, but is not required if either the *Orientation* or *Spacing* keywords are present.

**Patterned Fill Method** — The method of patterned filling and the usage of various fill patterns is hardware dependent. The fill pattern array may be directly specified with the *Pattern* keyword for some output devices. If this keyword is omitted, the polygon is filled with the hardware-dependent pattern index specified by the *Fill\_Pattern* keyword.

## See Also

[MAP](#), [POLYFILL](#)

---

## ***MAP\_REVERSE Procedure***

Converts output from routines like `CURSOR` and `WtPointer` from device, normal, or data coordinates to longitude and latitude coordinates.

### **Usage**

`MAP_REVERSE`, *x*, *y*, *lon*, *lat*

### **Input Parameters**

*x* — A variable containing the value (e.g., column of the current cursor position) to convert to a longitude value.

*y* — A variable containing the value (e.g., row of the current cursor position) to convert to a latitude value.

### **Output Parameters**

*lon* — A named variable to receive the calculated longitude value.

*lat* — A named variable to receive the calculated latitude value.

### **Keywords**

*Data* — If present and nonzero, specifies that data coordinates are the input (the default).

*Device* — If present and nonzero, specifies that device coordinates are the input.

*Normal* — If present and nonzero, specifies that normal coordinates are the input.

### **Discussion**

`MAP_REVERSE` allows you to create mapping applications that permit user interaction. Input (*x*, *y*) is received from a procedure that reads the cursor position, such as `CURSOR`. `MAP_REVERSE` converts these values to longitude and latitude values in the current projection.

You cannot use this routine when the *Projection* keyword is set to 99 (3D Mapping onto Sphere) because it is not a projection, but a true 3D representation of the data.

## Example

The following commands print the longitude and latitude of any point on a map that you click on with the mouse.

```
MAP
CURSOR, x, y
MAP_REVERSE, x, y, lon, lat
PRINT, 'Longitude = ', lon, 'Latitude = ', lat
```

## See Also

[CURSOR](#), [MAP](#)

WtPointer (in the *PV-WAVE Application Developer's Guide*)

---

## MAP\_VELOVECT Procedure

Draws a two-dimensional velocity field plot on a map, with each directed arrow indicating the magnitude and direction of the field.

### Usage

```
MAP_VELOVECT, u, v [, x, y]
```

### Input Parameters

*u* — The X component of the two-dimensional field. Must be a two-dimensional array of the same size as *v*.

*v* — The Y component of the two-dimensional field. Must be a two-dimensional array of the same size as *u*.

*x* — (optional) The abscissa values. Must be a vector of longitude coordinates. The size of *x* must equal the first dimension of *u* and *v*.

*y* — (optional) The ordinate values. Must be a vector of latitude coordinates. The size of *y* must equal the second dimension of *u* and *v*.

### Keywords

*Color* — Sets the color index of vector arrows. If this keyword is omitted, !P.Color specifies the color index.

**Dots** — If present and nonzero, places a dot at the position of the missing data. Otherwise, nothing is drawn for missing points. *Dots* is only valid if the *Missing* keyword is also specified.

**Length** — A length factor. The default value is 1.0, which makes the longest (*u*, *v*) vector have a length equal to the length of a single cell.

**Missing** — A 2D array with the same size as the *u* and *v* arrays. It is used to specify that specific points have missing data.

If the magnitude of the vector at (*i*, *j*) is less than the corresponding value in *Missing*, then the data is considered to be valid. Otherwise, the data is considered to be missing.

Thus, one way to set up a *Missing* array is to initialize all elements to some large value:

```
missing_array = FLTARR(n, m) + 1.0E30
```

Then, if point (*i*, *j*) is a missing point, set the corresponding element to a negative value:

```
missing_array(i, j) = -missing_array(i, j)
```

## Discussion

MAP\_VELOVECT draws a two-dimensional velocity field plot. The arrows indicate the magnitude and the direction of the field.

If missing values are present, you can use the *Missing* keyword to specify that they be ignored during the plotting, or the *Dots* keyword to specify that they be marked with a dot.

## Example

For an example of the MAP\_VELOVECT procedure, refer to the following program:

```
(UNIX)      $VNI_DIR/mapping-1_1/demo/map_test6.pro
```

```
(OpenVMS)  VNI_DIR: [MAPPING-1_1.DEMO]map_test6.pro
```

```
(Windows)  %VNI_DIR%\mapping-1_1\demo\map_test6.pro
```

## See Also

[MAP](#)

---

## MAP\_XYOUTS Procedure

Draws text on the currently selected graphics device starting at the designated map coordinate.

### Usage

MAP\_XYOUTS, *x*, *y*, *string*

### Input Parameters

*x* — Parameter *x* is the longitude at which the output string should start.

*y* — Parameter *y* is the latitude at which the output string should start.

*string* — The scalar string containing the text that is to be output to the display surface. If this parameter is not of string type, it is converted prior to use.

### Standard Plotting Keywords

The following standard plotting keywords are used with MAP\_XYOUTS. See [Chapter 3, Graphics and Plotting Keywords](#), for more information.

Charsize	Charthick	Color	Alignment
Font	Orientation	Width	

### Discussion

MAP\_XYOUTS is machine-dependent when you are using hardware fonts. This means that on two different machines, the same commands may produce text that does not appear the same. To guarantee similar appearance, use software fonts.

---

**UNIX and OpenVMS USERS** You may notice that under the X Window System the size of the software fonts varies from device to device. When you start PV-WAVE, the PV-WAVE hardware font is set to the current hardware font of the X server. Not all X servers will have the same default font size because users can reconfigure the default font and the default font can differ between X servers. Therefore, you may discover that the hardware font size, and therefore the software font size, may vary across different workstations. You can avoid this by explicitly setting the X font using the DEVICE procedure. For example:

## Example

```
MAP, RANGE = [-150, 30, 30, 70]
    ; Plot a map given the specified range.
MAP_XYOUTS, -105.3, 40.0, 'Boulder', Color=5,$
    Charsize = 1.5, Charthick = 2
    ; Label the city of Boulder, Colorado.
```

## See Also

[MAP](#), [USGS\\_NAMES](#)

---

## MAX Function

Returns the value of the largest element in an input array.

### Usage

*result* = MAX(*array* [, *max\_subscript*])

### Input Parameters

*array* — The array to be searched.

*max\_subscript* — (optional) The subscript of the maximum element in *array*:

- If supplied, *max\_subscript* is converted to a long integer containing the index of the largest element in *array*.
- If *max\_subscript* is not supplied, the system variable !C is set to the index of the largest element in the array.

### Returned Value

*result* — The value of the largest element in *array*. The result is given in the same data type as *array*. If the *Dimension* keyword is used, then the values of *result*, *max\_subscript*, and *Min* will all have the structure of the input array, but with dimension *n* collapsed.

## Keywords

**Dimension** — An integer ( $n \geq 0$ ) designating the dimension over which the maximum is taken.

**Min** — Used to specify a variable to hold the value of the minimum array element.

---

**TIP** If you need to find both the minimum and maximum array values, use this keyword to avoid scanning the array twice with separate calls to MAX and MIN.

---

### Example 1

```
x = [22, 40, 9, 12]
PRINT, MAX(x)
    40
```

### Example 2

```
x = [3, 4, 5, 6, 7, 8, 9]
maxval = MAX(x, maxindex, Min=minval)
PRINT, maxval
    9
PRINT, maxindex
    6
PRINT, minval
    3
```

### Example 3

```
a = [ [1,1,3,2], [3,4,1,3], [3,0,1,0], [0,1,2,0] ] & PM, a
    1      3      3      0
    1      4      0      1
    3      1      1      2
    2      3      0      0
PM, MAX( a, d=0 )
    3      4      3      2
PM, MAX( a, d=1 )
    3
    4
    3
```

```

a = INDGEN( 1, 2, 3, 4, 5 )
INFO, MAX( a, d=2 )
<Expression>      INT      = Array(1, 2, 1, 4, 5)

```

## See Also

[!C](#), [AVG](#), [EXTREMA](#), [MEDIAN](#), [MIN](#)

---

## MEDIAN Function

Finds the median value of an array, or applies a one- or two-dimensional median filter of a specified width to an array.

### Usage

```
result = MEDIAN(array [, width])
```

### Input Parameters

**array** — The array to be processed. May be of any size, dimension, and data type, except string.

**width** — (optional) The length of the one- or two-dimensional neighborhood to be used for the median filter. Must be a scalar value, greater than 1 and less than the smaller of the dimensions of *array*. The neighborhood will have the same number of dimensions as *array*.

### Returned Value

**result** — The median value for *array*, or *array* after a median filter has been applied to it.

- If *width* is specified and *array* is of a byte data type, the result is also a byte type. All other types are converted to single-precision floating-point, and the result is floating-point. If *width* is used, *array* can have only one or two dimensions.
- If *width* is not specified, *array* may have any valid number of dimensions. It is converted to single-precision floating-point, and the median value is returned as a floating-point value.

## Keywords

**Average** — If specified, and the number of elements in *array* is even, and *result* returns the average of the two middle values.

---

**NOTE** The *Average* keyword is ignored when filtering a byte array.

---

**Edge** — A scalar string indicating how edge effects are handled. (Default: 'copy') Valid strings are:

'zero' — Sets the border of the output image to zero.

'copy' — Copies the border of the input image to the output image.  
(Default)

**Same\_Type** — If set, the output image is the same data type as the input image.

## Discussion

The MEDIAN function supports multi-layer band interleaved images. When the input *array* is 3D and the *width* parameter is given, it is automatically treated as an array of images, *array(m, n, p)*, where *p* is the number of *m*-by-*n* images. Each image is then operated on separately and an array of the result images is returned.

Median smoothing replaces each point with the median of the one- or two-dimensional neighborhood of the given width. It is similar to smoothing with a boxcar or average filter, but does not blur edges larger than the neighborhood. In addition, median filtering is effective in removing “salt and pepper” noise (isolated high or low values).

The scalar median is simply the middle value, which should not be confused with the average value (e.g., the median of [1, 10, 4] is 4, while the average is 5).

## Example

This example exhibits median filtering of an image that has been corrupted with noise spikes. (For this procedure to run, PV-WAVE:IMSL Mathematics must be running.)

```
OPENR, unit, FILEPATH('head.img', Subdir = 'data'), /Get_Lun
      ; Open the file containing the human head dataset.

head = BYTARR(512, 512)
      ; Create an array large enough to hold the dataset.

READU, unit, head
FREE_LUN, unit
      ; Read the data, then close the file and free the file unit number.
```

```

slice = CONGRID(REFORM(head), 256, 256, /Interp)
    ; Use REFORM to remove degenerate dimensions, then resize the
    ; image using the CONGRID function.

WINDOW, 0, Xsize = 768, Ysize = 256
    ; Create a window large enough to display three images of the
    ; size of the original image.

LOADCT, 3
    ; Load the red temperature color table.

TVSCL, slice, 0
    ; Display the original image in the leftmost portion of the window.

HIST_EQUAL_CT, slice
    ; Histogram equalize the color table.

.RUN

FOR i = 0, 253, 6 DO BEGIN
    rows = RANDOM(256) GT 0.5
    k = RANDOM(1)
    h = 3 * (k(0) GT 0.5)
    FOR j = h, 253, 6 DO BEGIN
        IF rows(j) THEN slice(i:i + 2, j:j + 2) = 0
    ENDFOR
    ; This FOR loop creates 3-by-3 isolated holes in the image.
ENDFOR

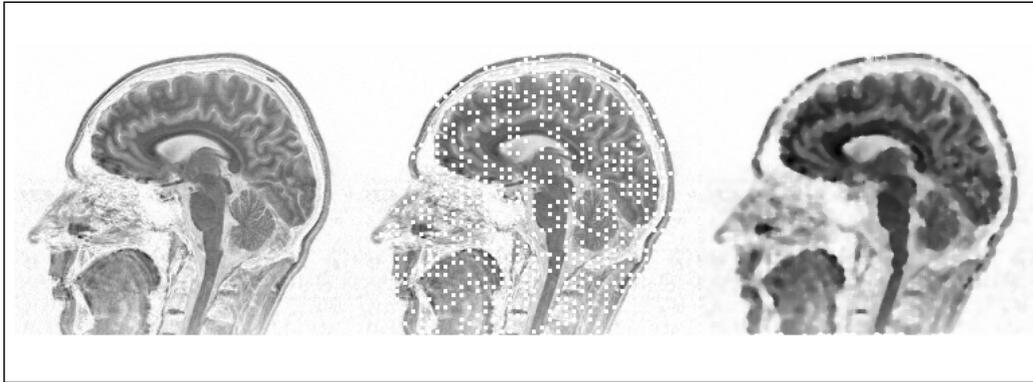
END

TVSCL, slice, 256, 0
    ; Display the image with holes in the center portion of the window.

filtered = MEDIAN(slice, 5)
    ; Median filter the image with holes.

TVSCL, filtered, 512, 0
    ; Display the median filtered image in the rightmost portion of the window.

```



**Figure 2-31** Original image (left), corrupted image (center), and median filtered image (right).

## See Also

[AVG](#), [LEEFILT](#), [MAX](#), [MIN](#), [SMOOTH](#)

---

## MESH Function

Defines a polygonal mesh object that can be used by the RENDER function.

### Usage

*result* = MESH(*vertex\_list*, *polygon\_list*)

### Input Parameters

*vertex\_list* — A double-precision floating-point array of 3D points; the array's size is 3 \* number\_of\_vertices.

*polygon\_list* — A longword integer 1D array defining the polygons; the array's size is number\_of\_polygons \* number\_of\_edges.

For more information on these two parameters, see .

### Returned Value

*result* — A structure that defines an object consisting of multiple polygons.

## Keywords

**Color** — A 256-element double-precision floating-point vector containing the color (intensity) coefficients of the object. The default is `Color (*) = 1.0`. For more information, see the section the section *Defining Color and Shading* in Chapter 7 of the *PV-WAVE User's Guide*.

**Kamb** — A 256-element double-precision floating-point vector containing the ambient (flat shaded) coefficients. The default is `Kamb (*) = 0.0`. For more information, see the section the section *Ambient Component* in Chapter 7 of the *PV-WAVE User's Guide*.

**Kdiff** — A 256-element double-precision floating-point vector containing the diffuse reflectance coefficients. The default is `Kdiff (*) = 1.0`. For more information, see the section the section *Diffuse Component* in Chapter 7 of the *PV-WAVE User's Guide*.

**Ktran** — A 256-element double-precision floating-point vector containing the specular transmission coefficients. The default is `Ktran (*) = 0.0`. For more information, see the section the section *Transmission Component* in Chapter 7 of the *PV-WAVE User's Guide*.

**Materials** — A byte array of size `number_of_polygons` defining the materials list. The purpose of this keyword is similar to that of the *Decal* keyword for quadric objects. Its use permits the specification of properties for each polygon where each polygon specifies an index into the *Color*, *Kamb*, *Kdiff*, and *Ktran* property arrays.

For more information, see the section *Defining Object Material Properties* in Chapter 7 of the *PV-WAVE User's Guide*.

**Transform** — A 4-by-4 double-precision floating-point array containing the local transformation matrix whose default is the identity matrix. For more information, see the section the section *Setting Object and View Transformations* in Chapter 7 of the *PV-WAVE User's Guide*.

## Discussion

MESH can be used by the RENDER function to render collections of 3D polygons, such as iso-surfaces, or spatial-structural data.

---

**NOTE** Any non-coplanar polygons in a mesh are automatically reduced to triangles by RENDER.

---

The *Transform* keyword can be specified to alter the scaling, as well as the orientation and position of the polygons defined by MESH.

## Examples

```
vertices = [[-1.0, -1.0, 1.0],$,  
            [-1.0, 1.0, 1.0],$,  
            [ 1.0, 1.0, 1.0],$,  
            [ 1.0, -1.0, 1.0],$,  
            [-1.0, -1.0, -1.0],$,  
            [-1.0, 1.0, -1.0],$,  
            [ 1.0, 1.0, -1.0],$,  
            [ 1.0, -1.0, -1.0]]  
polygons = [4, 0, 1, 2, 3,$  
            4, 4, 5, 1, 0,$  
            4, 2, 1, 5, 6,$  
            4, 2, 6, 7, 3,$  
            4, 0, 3, 7, 4,$  
            4, 7, 6, 5, 4]  
T3D, /Reset, Rotate=[15, 30, 45]  
cube = MESH(vertices, polygons, Transform=!P.T)  
TV, RENDER(cube)
```

## See Also

[CONE](#), [CYLINDER](#), [RENDER](#), [SPHERE](#), [VOLUME](#)

For more information, see the section *Ray-tracing* in Chapter 7 of the *PV-WAVE User's Guide*.

---

## MESSAGE Procedure

Issues error and informational messages using the same mechanism employed by PV-WAVE system routines.

### Usage

MESSAGE, *text*

### Input Parameters

*text* — A text string containing the message.

## Keywords

***Continue*** — If present and nonzero, causes MESSAGE to return after issuing the error instead of taking the action specified by the ON\_ERROR procedure. This keyword is useful when it is desirable to report an error and then continue processing.

***Informational*** — If present and nonzero, specifies that the message is simply informational text, rather than an error, and that processing is to continue. In this case, !Err, !Error, and !Err\_String are not set. The !Quiet system variable controls the printing of informational messages.

***Ioerror*** — Indicates that the error occurred while performing I/O. In this case, the action specified by the ON\_IOERROR procedure is executed instead of that specified by ON\_ERROR.

***Noname*** — Usually, the message includes the name of the issuing routine at the beginning. If *Noname* is present and nonzero, this name is omitted.

***Noprefix*** — Usually, the message includes the message prefix string at the beginning (as specified by the !Msg\_Prefix system variable). If *Noprefix* is present and nonzero, this prefix is omitted.

***Noprint*** — If present and nonzero, causes actions to proceed quietly, without the message being printed to the screen. The error system variables are updated as usual.

***Traceback*** — If present and nonzero, provides a traceback message giving the location at which MESSAGE was called. This traceback message follows the output error message.

## Discussion

By default, MESSAGE halts execution of your routine; messages are issued as an error and PV-WAVE takes the action specified by the ON\_ERROR procedure. However, if you specify either the *Continue* or the *Informational*, processing of your routine continues uninterrupted.

As a side-effect of issuing the error, the system variables !Err and !Error are set and the text of the error message is placed in the system variable !Err\_String.

## Example 1

Assume the statement:

```
MESSAGE, 'Unexpected value encountered.'
```

is executed in a procedure named `CALC`. This would cause `CALC` to halt after the following message was issued:

```
% CALC: Unexpected value encountered.
```

## Example 2

Assume the statement:

```
MESSAGE, 'Value is greater than 1000; ' + $  
        'you will lose some accuracy.', /Noname,$  
        /Noprefix, /Informational
```

is executed in a procedure named `VERIFY`. This would cause the following message to be issued:

```
Value is greater than 1000; you will lose some accuracy.
```

and execution would continue to the next line of `VERIFY`.

## See Also

[!Err](#), [!Err\\_String](#), [!Msg\\_Prefix](#), [HAK](#), [ON\\_ERROR](#), [ON\\_IOERROR](#), [WAIT](#)

For more information, see [Chapter 4, System Variables](#).

---

## MIN Function

Returns the value of the smallest element in *array*.

### Usage

```
result = MIN(array [, min_subscript])
```

### Input Parameters

*array* — The array to be searched.

*min\_subscript* — (optional) The subscript of the smallest element in *array*:

- If supplied, *min\_subscript* is converted to a long integer containing the one-dimensional subscript of the smallest element.
- If *min\_subscript* is not supplied, the system variable `!C` is set to the one-dimensional subscript of the smallest element.

## Returned Value

**result** — The value of the smallest element in *array*. The result is given in the same data type as *array*. If the *Dimension* keyword is used, then the values of *result*, *min\_subscript*, and *Max* will all have the structure of the input array, but with dimension *n* collapsed.

## Keywords

**Max** — Used to specify a variable to hold value of the largest array element.

**Dimension** — An integer ( $n \geq 0$ ) designating the dimension over which the minimum is taken.

---

**TIP** If you need to find both the minimum and maximum array values, use the *Max* keyword to avoid having to scan the array twice using separate calls to MAX and MIN.

---

### Example 1

```
x = [22, 40, 9, 12]
PRINT, MIN(x)
    9
```

### Example 2

```
x = [3, 4, 5, 6, 7, 8, 9]
minval = MIN(x, minindex, Max=maxval)
PRINT, minval
    3
PRINT, minindex
    0
PRINT, maxval
    9
```

### Example 3

```
a = [ [1,1,3,2], [3,4,1,3], [3,0,1,0], [0,1,2,0] ]    &    PM, a
    1      3      3      0
    1      4      0      1
    3      1      1      2
```

```

      2      3      0      0
PM, MIN( a, d=0 )
      1      1      0      0
PM, MIN( a, d=1 )
      0
      0
      1
      0

a = INDGEN( 1, 2, 3, 4, 5 )
INFO, MIN( a, d=2 )
<Expression>      INT      = Array(1, 2, 1, 4, 5)

```

## See Also

[AVG](#), [EXTREMA](#), [MAX](#), [MEDIAN](#)

## MINIMIZE Function

Standard Library function that minimizes a real valued function of  $n$  real variables.

### Usage

$x = \text{MINIMIZE}(f, l, u, g, i, y)$

### Input Parameters

$f$  — A string specifying a user supplied function to be minimized. Input is a  $(m,n)$  array of  $m$  points in  $n$ -space ( $m$  variable); output is a  $(m,p+1)$  array  $b$ , where  $p$  is the number of constraints,  $b(*,0)$  contains the objective function values at each of the  $m$  input points, and  $b(*,j)$  contains the corresponding values of the  $j$ 'th constraint. All constraints must be of the form  $c(x) \leq 0$ .

$l$  —  $n$ -element vector of lower bounds for the independent variables

$u$  —  $n$ -element vector of upper bounds for the independent variables

$g$  —  $n$ -element vector giving an initial guess for the solution

$i$  — An integer limit on the number of iterations

## Returned Value

*x* — The n-element solution vector

*y* — (optional) A (p+1)-element vector containing the objective function value at *x* followed by the constraint values at *x*.

## Keywords

*d* — A string specifying a user-supplied gradient function. Input is the n-element vector at which to calculate the gradient(s). Output is a (n,p+1) array that contains the objective function gradient followed by the constraint gradients.

*s* — An (n,2) array where *s*(\*,0) is the maximum allowable step and *s*(\*,1) is the minimum allowable step. The default is [ [(u-1)/100], [(u-1)/1000] ].

## Examples

See `wave/lib/user/examples/minimize_ex*.pro`.

---

## MODIFYCT Procedure

Standard Library procedure that lets you replace one of the PV-WAVE color tables (defined in the `colors.tbl` file) with a new color table.

## Usage

MODIFYCT, *table*, *name*, *red*, *green*, *blue*

## Input Parameters

*table* — The color table number to change. The numbers range from 0 to 15.

*name* — The name of the modified color table. The string may be a maximum of 32 characters.

*red* — The red color gun vector. It contains 256 elements.

*green* — The green color gun vector. It contains 256 elements.

*blue* — The blue color gun vector. It contains 256 elements.

## Keywords

*Ctfile* — Specifies a string containing the name of a color table file to load.

## Discussion

Since any changes to the system color tables will affect all users, this procedure should be reserved for a single individual at a PV-WAVE site with authorization to make system modifications. It is also a good idea to make a copy of the `colors.tbl` file prior to using `MODIFYCT`.

## See Also

[HLS](#), [HSV](#), [HSV\\_TO\\_RGB](#), [LOADCT](#), [RGB\\_TO\\_HSV](#)

For more information on customizing color tables, see .

---

## MOLEC Function

Standard Library function that creates an image of a ball and stick molecular model.

## Usage

*result* = MOLEC(*filename*)

## Input Parameters

*filename* — The name of an ASCII file describing the molecular model. Line 1 in the file consists of a single integer designating the number (*m*) of atoms. Each of the lines 2-(*m*+1) contains 7 floats describing an atom in terms of centroid, normalized RGB color components, and diameter. Line *m*+2 consists of a single integer equal to the number (*n*) of bonds; each of the lines (*m*+3)-(*m*+*n*+2) contains 6 floats describing a bond as endpoint1 followed by endpoint2.

## Returned Value

*result* — A 24-bit image of the molecular model.

## Keywords

*h* — A scale factor for adjusting atom size. The default is  $h=1.0$ .

*s* — A 2-element vector specifying image size. The default is  $s=[500, 500]$ .

*v* — A 3-by-4 double-precision floating-point array used to override the autogeneration of the view to that specified as: [viewpoint, top\_left\_viewplane, bottom\_left\_viewplane, bottom\_right\_viewplane]. *v* and !P.T control the 3d view.

*k* — (output) A 3-by-4 double-precision floating-point array used to return the automatically calculated view as: [viewpoint, top\_left\_viewplane, bottom\_left\_viewplane, bottom\_right\_viewplane].

## Examples

```
T3D, /Reset & TV, MOLEC(!data_dir+'molec.dat',h=0.6), true=3
```

---

## MOMENT Function

Standard Library function that computes moments of an array.

### Usage

*result* = MOMENT(*a*, *i*)

### Input Parameters

*a* — An array of *n* dimensions.

*i* — A vector of *n* non-negative real numbers defining moment order.

### Returned Value

*result* — A scalar double equal to:

$$\sum_{J_{n-1}} \dots \sum_{J_0} a(J_0 \dots J_{n-1}) J_0^{i(0)} \dots J_{n-1}^{i(n-1)}$$

### Keywords

None.

## Examples

```
a = BYTARR( 600, 500 )
x = INDEX_CONV( a, LINDGEN(N_ELEMENTS(a)) )
a( WHERE( (x(*,0)-200)^2+(x(*,1)-300)^2 LT 100^2 ) ) = 255
TV, a
CENTROID = [MOMENT(a, [1,0]),MOMENT(a, [0,1])] / MOMENT(a, [0,0])
PLOTS, CENTROID, /DEVICE, COLOR=0, PSYM=2
```

---

## MONTH\_NAME Function

Standard Library function that returns a string or string array containing the names of the months contained in a date/time variable.

### Usage

```
result = MONTH_NAME(dt_var)
```

### Input Parameters

*dt\_var* — A date/time variable.

### Returned Value

*result* — A string array containing the name(s) of the months.

### Keywords

None.

### Discussion

The names of the months are defined as string values in the system variable !Month\_List.

### Example

```
dttoday = TODAY()
PRINT, dttoday
{ 1992 4 1 6 12 57.0000 87493.259 0}
```

```
                ; Create a variable that contains date/time data for today's date.  
m = MONTH_NAME(dttoday)  
PRINT, m  
    April
```

### See Also

[!Day\\_Names](#), [!Month\\_Names](#), [DAY\\_NAME](#), [DAY\\_OF\\_WEEK](#)

For more information, see the section *Working with Date/Time Data* in Chapter 8 of the *PV-WAVE User's Guide*.

---

## MOVIE Procedure

Standard Library procedure that shows a cyclic sequence of images stored in a three-dimensional array.

### Usage

MOVIE, *images* [, *rate*]

### Input Parameters

*images* — A three-dimensional byte array of image data, consisting of *nframes* images, each dimensioned *n*-by-*m*. Thus, the *images* array is (*n*, *m*, *nframes*). This array should be stored with the top row first (i.e., *Order*=1) for maximum efficiency.

*rate* — (optional) The initial rate, in approximate frames per second. If *rate* is omitted, the inter-frame delay is set at 0.01 second.

### Keywords

*Order* — Specifies the image ordering:

- 1 Orders the images from top down (the default).
- 0 Orders the images from bottom up.

### Discussion

The images are displayed in the lower-left corner of the currently selected window.

The rate of display varies with the make of computer, amount of physical memory, and number of frames.

Available memory also restricts the maximum amount of data that can be displayed in a loop.

### **Example**

This example uses `MOVIE` to cycle through cross-sections of a human head. The `Order` keyword is used to specify that the images are ordered bottom up. The cross-sections are contained in a byte array dimensioned (80, 100, 57).

```
OPENR, unit, FILEPATH('headspin.dat', Subdir = 'data'), /Get_Lun
      ; Open the file containing the cross sections.

head = BYTARR(256, 256, 32)
      ; Create a three-dimensional byte array large enough to contain all cross-sections.

READU, unit, head
      ; Read the images.

FREE_LUN, unit
      ; Close the file and free the file unit number.

LOADCT, 15
      ; Load a color table.

MOVIE, head, Order = 0
      ; Cycle through the images.
```

### **See Also**

[TV](#), [TVSCL](#)

---

## **MPROVE Procedure**

Iteratively improves the solution vector,  $\mathbf{x}$ , of a linear set of equations,  $A\mathbf{x} = \mathbf{b}$ . (You must call the LUDCMP procedure before calling MPROVE.)

### **Usage**

MPROVE, *a*, *alud*, *index*, *b*, *x*

### **Input Parameters**

*a* — An  $n$ -by- $n$  matrix containing the coefficients of the linear equation  $A\mathbf{x} = \mathbf{b}$ .

*alud* — The LU decomposition of  $A$ , an  $n$ -by- $n$  matrix, as returned by LUDCMP.

*index* — The vector of permutations involved in the LU decomposition of  $A$ , as returned by LUDCMP.

*b* — An  $n$ -element vector containing the right-hand side of the set of equations.

*x* — An  $n$ -element vector. On input, it contains the initial solution of the system.

### **Output Parameters**

*x* — An  $n$ -element vector. On output, *x* contains the improved solution.

### **Keywords**

None.

### **See Also**

[LUBKSB](#), [LUDCMP](#)

MPROVE is based on the routine of the same name in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988. It is used by permission.

---

## **MSWORD\_CGM\_SETUP Procedure**

Sets up the CGM driver so that CGM files saved from PV-WAVE can be imported easily into Microsoft Word.

### **Usage**

MSWORD\_CGM\_SETUP

### **Input Parameters**

None.

### **Keywords**

None.

### **Discussion**

For information on the CGM driver, see *Appendix B: Output Devices and Window Systems*.

### **Example**

```
SET_PLOT, 'CGM'  
DEVICE, File='myplot.cgm'  
MSWORD_CGM_SETUP  
PLOT, dist(20)  
DEVICE, /Close
```

---

## NAVIGATOR Procedure

Starts the Navigator.

### Usage

NAVIGATOR

### Keywords

**ConfigFile** — A string containing the name of a previously saved Navigator configuration file. You can specify a filename or a complete path name. If you specify a filename only, the file must be in the current working directory.

**Horizontal** — If this keyword is specified, the VDA Tool icons are arranged horizontally in the Navigator window.

**Highres** — Reconfigures the Navigator button bar to display one row of buttons. This option is only suitable for larger monitors. By default, the Navigator starts with two rows of buttons.

**Lowres** — This keyword has been deprecated. It has no effect.

**Position** — Specifies in pixels, the  $x$  and  $y$  (horizontal and vertical, respectively) coordinates for the starting location of the upper-left corner of the Navigator window.

**Template** — A string containing the name of a template file.

**Vertical** — If this keyword is specified, the VDA Tool icons are arranged vertically in the Navigator window.

### Discussion

By default, the Navigator displays with two rows of icons. If you wish to display a long, narrow Navigator with one row of icons, use the *Highres* keyword.

The Navigator combines multiple VDA Tools into a single application. The Navigator provides tools for:

- importing and exporting data
- displaying 2D plots, images, histograms, surfaces, and contour plots
- animating data
- viewing and subsetting tables of data

- saving and restoring sessions
- viewing and selecting variables

Template files are saved with the Navigator **File=>Save Template As** function. A template contains information that allows you to restore a Navigator with the customized defaults that were set when the template file was saved.

Configuration files are saved with the Navigator **File=>Save Configuration** function.

---

**NOTE** For information on how to use the Navigator, use online Help. Select the **On Window** command from the Navigator Help menu to bring up Help on the Navigator.

---

## Example

This command opens the Navigator with a configuration file and a template file created during a previous Navigator session.

```
NAVIGATOR, ConfigFile='myconfig.sav', Template='mynav.tpl'
```

## See Also

See the *PV-WAVE Tutorial* for a lesson on using the Navigator.

---

## NEIGHBORS Function

Standard Library function that finds the neighbors of specified array elements.

### Usage

*result* = NEIGHBORS(*a*, *i*)

### Input Parameters

*a* — An array of *n* dimensions.

*i* — An *m*-element vector of *m* one-dimensional indices into *a*.

## Returned Value

*result* — An (m,\*) array of one-dimensional indices into *a*: *result*(j,\*) contains *i*(j) and its neighbors.

## Keywords

*k* — A positive integer (less than or equal to *n*) defining connectivity. Two array cells are neighbors if they share a common boundary point, and if their centroids are within  $\sqrt{k}$  of each other. The default is *k* = 1, which implies neighbors share a common face.

## Examples

```
a = INDGEN( 8, 9 ) & pm, a
pm, fix(NEIGHBORS(a, [0,4,26,47,71]))
pm, fix(NEIGHBORS(a, [0,4,26,47,71], k=2))
```

## See Also

[BLOB](#), [BLOBCOUNT](#), [BOUNDARY](#)

---

## ***N\_ELEMENTS Function***

Returns the number of elements contained in any expression or variable.

### Usage

*result* = N\_ELEMENTS(*expr*)

### Input Parameters

*expr* — The expression for which the number of elements will be returned.

### Returned Value

*result* — The number of elements contained in any expression or variable.

Scalar expressions always have one element. The number of elements in an array is equal to the product of its dimensions. If *expr* is an undefined variable, N\_ELEMENTS will return zero.

## Keywords

None.

## Example

In this example, `N_ELEMENTS` is used to determine the number of elements in a two-dimensional array.

```
a = INDGEN(3, 2)
    ; Create a 3-by-2 integer array.
PRINT, a
    0      1      2
    3      4      5
PRINT, N_ELEMENTS(a)
    6
    ; Display the number of elements in a.
DELVAR, a
    ; Delete the variable a.
INFO, a
    VARIABLE      UNDEFINED = <Undefined>
PRINT, N_ELEMENTS(a)
    0
```

## See Also

[N\\_PARAMS](#), [N\\_TAGS](#), [PARAM\\_PRESENT](#), [SIZE](#), [TAG\\_NAMES](#)

---

## ***NINT Function***

Converts input to the nearest integer.

### Usage

*result* = `NINT(x)`

### Input Parameters

**x** — A scalar or array of any **PV-WAVE** variable type, usually float or double.

## Keywords

*Long* — If present and non-zero, NINT returns a long instead of a short (FIX) integer.

## Returned Value

*result* — The nearest integer to the input value.

## Discussion

Instead of truncating the input (as FIX does), first the input is rounded by adding or subtracting 0.5 (depending on whether the input is greater or less than zero), and then it is truncated.

If the input is out of the range of integers (for example, if you pass in 1.0d33), an error message will result and NINT returns garbage.

Add  $\pm 0.5$  to the input and convert that to a short integer using FIX. If the *Long* keyword is used, it's converted via long. If the input is a FIX, then it's just passed back. If it's a long, it's also passed back. Strings are converted to bytes before the rounding. In the case of complex values, their magnitude is taken. Structures are not allowed.

## Examples

```
PRINT, NINT(5.1)
5
; Round 5.1 to the nearest integer, which is 5.

PRINT, NINT(5.6)
6
; Round 5.6 up to 6.

PRINT, NINT(-1.9)
-2
; The nearest integer to -1.9 is -2.

PRINT, NINT([0.1, -20.9, 50.9])
0      -21      51
; An array of input floating point numbers returns a similar
; array of the nearest integers.

PRINT, NINT(200000.1)
3392
;NINT returns incorrect results when the input is out of
```

; the range of integers.  
PRINT, NINT(200000.1, /Long)  
    200000  
        ; Floating point numbers which are out of the range of fix  
        ; type integers but in the range of long type integers should  
        ; be rounded using the *Long* keyword.

## See Also

[FIX](#), [SMALL\\_INT](#)

---

## ***NORMALS* Function**

Standard Library function that computes unit normals on a parametrically defined surface.

### **Usage**

$n$  = normals ( $j$ )

### **Input Parameters**

$j$  — The Jacobian (computed by the JACOBIAN function) on the surface.

### **Returned Value**

$n$  — A 3-element list of 2-dimensional arrays of the same size as those in  $j$ :  $n(i)$  is the array describing the distribution of the  $i^{\text{th}}$  component of the unit normal.

### **Keywords**

None.

### **Example**

See `wave/lib/user/examples/normals_ex.pro`.

## See Also

[CURVATURES](#), [EUCLIDEAN](#), [JACOBIAN](#)

---

## ***N\_PARAMS* Function**

Returns the number of non-keyword parameters used in calling a PV-WAVE procedure or function.

### **Usage**

*result* = N\_PARAMS()

### **Parameters**

None.

### **Returned Value**

*result* — The number of non-keyword parameters used in calling a PV-WAVE procedure or function.

### **Keywords**

None.

### **Discussion**

N\_PARAMS is used to determine if user-written procedures or functions were called with positional (non-keyword) parameters.

### **See Also**

[N\\_ELEMENTS](#), [N\\_TAGS](#), [PARAM\\_PRESENT](#), [SIZE](#), [TAG\\_NAMES](#)

For more information on functions and procedures, see .

---

## ***N\_TAGS* Function**

Returns the number of structure tags contained in any expression.

### **Usage**

*result* = N\_TAGS(*expr*)

### **Input Parameters**

*expr* — The expression for which the number of structure tags will be returned.

### **Returned Value**

*result* — The number of structure tags contained in any expression.

### **Keywords**

None.

### **Discussion**

Expressions which are not of structure type are considered to have zero tags. N\_TAGS does not search for tags recursively, so if *expr* is a structure containing nested structures, only the number of tags in the outermost structure are counted.

### **Example**

In this example, a structure with three fields is created. Function N\_TAGS is applied to the structure and to each field of the structure. The first two fields of the structure are not of type structure. The third field of the structure is of type structure, with two fields.

```
b = {example, t1: [1, 2, 3], t2: 7.0, $
      t3: {field3, t3_t1: 99L, t3_t2: [2, 4, 6]}}
```

```
INFO, b, /Structure
```

```
** Structure EXAMPLE, 3 tags, 32 length:
```

```
T1 INT          Array(3)
T2 FLOAT        7.00000
T3 STRUCT      -> FIELD3 Array(1)
```

```
      ; Create the structure.
```

```
PRINT, N_TAGS (b)
      3
      ; Display the number of tags in b.
PRINT, N_TAGS (b.t1)
      0
      ; Display the number of tags in each field of b.
PRINT, N_TAGS (b.t2)
      0
PRINT, N_TAGS (b.t3)
      2
```

## See Also

[N\\_ELEMENTS](#), [N\\_PARAMS](#), [SIZE](#), [STRUCTREF](#), [TAG\\_NAMES](#)

---

## ***ON\_ERROR Procedure***

Determines the action taken when an error is detected inside a user-written procedure or function.

### **Usage**

ON\_ERROR, *n*

### **Input Parameters**

*n* — An integer that specifies the action to take. Valid values are:

- 0 Stop at the statement in the procedure that caused the error. (This is the default action.)
- 1 Return all the way back to the main program level.
- 2 Return to the caller of the program unit which established the ON\_ERROR condition.
- 3 Return to the program unit which established the ON\_ERROR condition.

### **Keywords**

***Continue*** — If specified and nonzero, program execution resumes at the statement specified by one of the following input parameters:

- 0 Continue in the procedure that caused the error.
- 1 Return to the main program level \$MAIN\$ and continue.
- 2 Return to the calling routine that established the ON\_ERROR condition and continue.
- 3 Return to the program unit that established the ON\_ERROR condition and continue.

### **Example**

In this example, a procedure named PROC1 calls a procedure, PROC2, which conditionally calls either procedure PROC3 or PROC4. Procedure PROC3 contains an error. A call to PRINT from within PROC3 is intended, but a typographical error results in a nonexistent procedure, PAINT, being called. The ON\_ERROR proce-

dure, which is called from PROC2, is passed different input parameter values in this example to exhibit the action taken when the error in PROC3 is encountered.

The following is a listing of the procedures used in this example:

```
PRO PROC1
FOR i = 1, 4 DO BEGIN
    PROC2, i
ENDFOR
END

PRO PROC2, j
ON_ERROR, 0
    ; Invoke the ON_ERROR procedure at this point.
IF (j MOD 2) EQ 0 THEN BEGIN
    PROC3, j
ENDIF ELSE BEGIN
    PROC4, j
ENDELSE
END

PRO PROC3, k
PAINT, k, ' is even.'
    ; The call to a nonexistent procedure, PAINT, occurs here.
END

PRO PROC4, m
PRINT, m, ' is odd.'
END
```

Note that ON\_ERROR has the input parameter 0 in PROC2. If the procedures are placed in the file `errex.pro` in your working directory, all of them can be compiled with the following command:

```
.RUN errex
```

### ***Error Condition 0***

Next, run the PROC1 procedure and examine the results:

```
PROC1

    1 is odd.

% Attempt to call undefined
% procedure/function: PAINT.
% Execution halted at PROC3 <errex.pro(21)>.
% Called from PROC2 <errex.pro(13)>.
```

```
% Called from PROC1 <errex.pro(5)>.
% Called from $MAIN$.
```

Examine which procedure PV-WAVE returned to by looking at the current nesting of procedures and functions with the INFO command:

```
INFO, /Traceback
% At PROC3 <errex.pro(22)>.
% Called from PROC2 <errex.pro(13)>.
% Called from PROC1 <errex.pro(5)>.
% Called from $MAIN$.
```

PV-WAVE stopped at the PROC3 procedure, where the error occurred. This is where PV-WAVE would have stopped if ON\_ERROR had not been called.

### ***Error Condition 1***

If the call to ON\_ERROR in PROC2 is changed from

```
ON_ERROR, 0
```

to

```
ON_ERROR, 1
```

then the commands

```
RETALL
.RUN errex
```

need to be executed to return to the main program level and recompile the file containing the example procedures.

To execute PROC1 again and check where PV-WAVE returns after the error in PROC3, issue the following command:

```
PROC1
    1 is odd.

% Attempt to call undefined
% procedure/function: PAINT.
% Execution halted at PROC3 <errex.pro(21)>.
% Called from PROC2 <errex.pro(13)>.
% Called from PROC1 <errex.pro(5)>.
% Called from $MAIN$.

INFO, /Traceback
% At $MAIN$.
```

Note that PV-WAVE returned to the main program level.

### ***Error Condition 3***

Next, the call to ON\_ERROR in PROC2 is changed from

```
ON_ERROR, 1
```

to

```
ON_ERROR, 3
```

and the file containing the example procedures is recompiled and executed with the commands:

```
RETALL
```

```
.RUN errex
```

Issue the following command to execute PROC1 again:

```
PROC1
```

```
    1 is odd.  
% Attempt to call undefined  
% procedure/function: PAINT.  
% Execution halted at PROC3 <errex.pro(21)>.  
%   Called from PROC2 <errex.pro(13)>.  
%   Called from PROC1 <errex.pro(5)>.  
%   Called from $MAIN$.
```

To see where PV-WAVE has returned, issue the following command:

```
INFO, /Traceback  
% Called from PROC2 <errex.pro(13)>.  
%   Called from PROC1 <errex.pro(5)>.  
%   Called from $MAIN$.
```

Note that this time, PV-WAVE returned to PROC2, which is the program unit that made the call to ON\_ERROR.

### ***Error Condition 0 with Continuation***

Next, the call to ON\_ERROR in PROC2 is changed from

```
ON_ERROR, 3
```

to

```
ON_ERROR, 0, /Continue
```

and the file containing the example procedures is recompiled and executed with the commands:

```
RETALL
```

```
.RUN errex
```

Issue the following command to execute PROC1 again:

```
PROC1
    1 is odd.
% Attempt to call undefined
% procedure/function: PAINT.
% Error occurred at PROC3 <errex.pro(21)>.
%   Called from PROC2 <errex.pro(13)>.
%   Called from PROC1 <errex.pro(5)>.
%   Called from $MAIN$.
    3 is odd.
% Attempt to call undefined
% procedure/function: PAINT.
% Error occurred at PROC3 <errex.pro(21)>.
%   Called from PROC2 <errex.pro(13)>.
%   Called from PROC1 <errex.pro(5)>.
%   Called from $MAIN$.
```

To see where PV-WAVE has returned, issue the following command:

```
INFO, /Traceback
%   At $MAIN$.
```

Note that PV-WAVE has returned to the main program level.

### ***Error Condition 1 with Continuation***

If the call to ON\_ERROR in PROC2 is changed from

```
ON_ERROR, 0, /Continue
```

to

```
ON_ERROR, 1, /Continue
```

and the file containing the example procedures is recompiled and executed with the commands:

```
RETAILL
.RUN errex
```

Issue the following command to execute PROC1 again:

```
PROC1
    1 is odd.
% Attempt to call undefined
```

```

% procedure/function: PAINT.
% Error occurred at PROC3 <errex.pro(21)>.
%     Called from PROC2 <errex.pro(13)>.
%     Called from PROC1 <errex.pro(5)>.
%     Called from $MAIN$.

```

To see where PV-WAVE has returned, issue the following command:

```

INFO, /Traceback
%     At $MAIN$.

```

Note that execution continued on the main program level.

### ***Error Condition 2 with Continuation***

Finally, the call to ON\_ERROR in PROC2 is changed from

```
ON_ERROR, 1, /Continue
```

to

```
ON_ERROR, 2, /Continue
```

and the file containing the example procedures is recompiled and executed with the commands:

```

RETALL
.RUN errex

```

Issue the following command to execute PROC1 again:

```

PROC1
    1 is odd.
% Attempt to call undefined
% procedure/function: PAINT.
% Error occurred at PROC3 <errex.pro(21)>.
%     Called from PROC2 <errex.pro(13)>.
%     Called from PROC1 <errex.pro(5)>.
%     Called from $MAIN$.
    3 is odd.

```

To see where PV-WAVE has returned, issue the following command:

```

INFO, /Traceback
%     Called from $MAIN$.

```

---

**NOTE** In this example, execution continued in PROC3 after the error, and PROC1 finished executing.

---

```

PRO PROC1
FOR i = 1, 4 DO BEGIN
    PROC2, i
ENDFOR
END

PRO PROC2, j
ON_ERROR, 0
    ; Invoke the ON_ERROR procedure at this point.
IF (j MOD 2) EQ 0 THEN BEGIN
    PROC3, j
ENDIF ELSE BEGIN
    PROC4, j
ENDELSE
END

PRO PROC3, k
PAINT, k, ' is even.'
    ; The call to a nonexistent procedure, PAINT, occurs here.
END

PRO PROC4, m
PRINT, m, ' is odd.'
END

```

Note that ON\_ERROR has the argument 0 in PROC2. If the procedures are placed in the file `errex.pro` in your working directory, all of them can be compiled with the following command:

```
.RUN errex
```

Next, invoke the top-level procedure and examine the results:

```

PROC1
    1 is odd.
% Attempt to call undefined
% procedure/function: PAINT.
% Execution halted at PROC3 <errex.pro(21)>.
%     Called from PROC2 <errex.pro(13)>.
%     Called from PROC1 <errex.pro(5)>.
%     Called from $MAIN$.

```

Examine which procedure PV-WAVE returned to by looking at the current nesting of procedures and functions with the INFO command:

```
INFO, /Traceback
```

```

% At PROC3 <errex.pro(22)>.
%   Called from PROC2 <errex.pro(13)>.
%   Called from PROC1 <errex.pro(5)>.
%   Called from $MAIN$.

```

PV-WAVE stopped at the PROC3 procedure, where the error occurred. This is where PV-WAVE would have stopped if ON\_ERROR had not been called.

If the call to ON\_ERROR in PROC2 is changed from

```
ON_ERROR, 0
```

to

```
ON_ERROR, 1
```

then the commands

```

RETALL
.RUN errex

```

need to be executed to return to the main program level and recompile the file containing the example procedures.

To execute PROC1 again and check where PV-WAVE returns after the error in PROC3, issue the following command:

```

PROC1
    1 is odd.

% Attempt to call undefined
% procedure/function: PAINT.

% Execution halted at PROC3 <errex.pro(21)>.
%   Called from PROC2 <errex.pro(13)>.
%   Called from PROC1 <errex.pro(5)>.
%   Called from $MAIN$.

INFO, /Traceback
% Called from $MAIN$.

```

Note that PV-WAVE returned to the main program level.

As a final example, the call to ON\_ERROR in PROC2 is changed from

```
ON_ERROR, 1
```

to

```
ON_ERROR, 3
```

and the file containing the example procedures is recompiled and executed with the commands:

```
RETALL
.RUN errex
```

Issue the following command to execute PROC1 again:

```
PROC1
    1 is odd.
% Attempt to call undefined
% procedure/function: PAINT.
% Execution halted at PROC3 <errex.pro(21)>.
%   Called from PROC2 <errex.pro(13)>.
%   Called from PROC1 <errex.pro(5)>.
%   Called from $MAIN$.
```

To see where PV-WAVE returned, issue the following command:

```
INFO, /Traceback
% Called from PROC2 <errex.pro(13)>.
%   Called from PROC1 <errex.pro(5)>.
%   Called from $MAIN$.
```

Note that this time, PV-WAVE returned to PROC2, which is the program unit that made the call to ON\_ERROR.

## See Also

[ON\\_IOERROR](#), [OPEN \(UNIX/OpenVMS\)](#), [OPEN \(Windows\)](#), [READ](#), [RETALL](#), [RETURN](#), [STOP](#), [WRITEU](#)

For more information, see the section *Error Handling in Procedures* in Chapter 9 of the *PV-WAVE Programmer's Guide*.

Additional information can be found in .

---

## ***ON\_ERROR\_GOTO Procedure***

Specifies a statement to jump to if an error occurs in the current procedure.

### **Usage**

`ON_ERROR_GOTO, label`

### **Input Parameters**

*label* — The name of a label statement to jump to.

---

**NOTE** Do not put a colon after this parameter.

---

### **Keywords**

None.

### **Discussion**

The `ON_ERROR_GOTO` procedure transfers program control to the point in the program specified by the *label* parameter after an error occurs. The *label* parameter specifies a label, which is an identifier followed by a colon. A label may exist on a line by itself. Labels are explained in the section *Statement Types* in Chapter 4 of the *PV-WAVE Programmer's Guide*.

---

**NOTE** The label name `null` has a special use with this procedure. If the name of the label is `null`, the effect of `ON_ERROR_GOTO` is canceled and normal processing continues.

---

If an error occurs, an error code is stored in the system variable `!Err`. In addition, the text of the error message is stored in `!Err_String`.

### **Example**

This example demonstrates how `ON_ERROR_GOTO` is used to control program flow after an error is detected.

```
PRO Proc1
    ON_ERROR_GOTO, Proc1_Failed
    ; If an error occurs here, go to the statement label Proc1_Failed.
```

```
ON_ERROR_GOTO, null
    ; The effect of ON_ERROR_GOTO is canceled, normal error processing is in
    ; effect.
RETURN
Proc1_Failed:
PRINT, !Err, !Err_String
END
```

## See Also

[ON\\_ERROR](#)

---

## ON\_IOERROR Procedure

Specifies a statement to jump to if an I/O error occurs in the current procedure.

### Usage

```
ON_IOERROR, label
```

### Input Parameters

*label* — The statement to jump to. Note that *label* is not a string variable, but is rather the statement label (without the trailing colon).

### Keywords

None.

### Discussion

Normally when an input/output error occurs, an error message is printed and program execution is stopped. If ON\_IOERROR is called and an I/O related error later occurs in the same procedure activation, control is transferred to the designated statement with the error code stored in the system variable !Err. The text of the error message is contained in the system variable !Err\_String.

### Example

```
ON_IOERROR, null
    ; The effect of ON_IOERROR is canceled by using null as the label.
```

## See Also

[!Err](#), [!Err\\_String](#), [ON\\_ERROR](#), [OPEN \(UNIX/OpenVMS\)](#), [OPEN \(Windows\)](#), [READ](#), [RETALL](#), [RETURN](#), [STOP](#), [WRITEU](#)

Additional information can be found in .

For more information on system variables, see [Chapter 4, System Variables](#). For more information on statement labels, see the section *Statement Labels* in Chapter 4 of the *PV-WAVE Programmer's Guide*. For background information, see .

---

## ***OPEN Procedures (UNIX/OpenVMS)*** ***(OPENR, OPENU, OPENW)***

Open a specified file for input/output:

- OPENR (OPEN Read) opens an existing file for input only.
- OPENU (OPEN Update) opens an existing file for input and output.
- OPENW (OPEN Write) opens a new file for input and output.

---

**CAUTION** When you use OPENW to create a new file under UNIX, if the file exists, it is truncated and its old contents destroyed. Under OpenVMS, a new file with the same name and a higher version number is created.

---

### Usage

OPENR, *unit*, *filename* [, *record\_length*]

OPENU, *unit*, *filename* [, *record\_length*]

OPENW, *unit*, *filename* [, *record\_length*]

### Input Parameters

*unit* — The logical unit number to be associated with the opened file.

*filename* — The name of the file to be opened. The following differences exist between the UNIX and OpenVMS versions of PV-WAVE regarding wildcard characters and file extensions:

- Under UNIX, the name may contain any wildcard characters recognized by the shell specified by the SHELL environment variable. However, it is faster not to

use wildcards because PV-WAVE doesn't use the shell to expand filenames unless it has to. No wildcard characters are allowed under OpenVMS.

- Under OpenVMS, filenames that do not have a file extension are assumed to have the .DAT extension. No such processing of filenames occurs under UNIX.

**record\_length** — (optional — OpenVMS Only) Specifies the file record size in bytes. This parameter is required when creating new fixed-length files, and is optional when opening existing files:

- If present when creating variable length record files, it specifies the maximum allowed record size.
- If present and no file organization keyword is specified, fixed-length records are implied.

---

**NOTE** Due to limitations in RMS, the length of records must always be an even number of bytes. Therefore, odd record lengths are automatically rounded up to the nearest even boundary.

---

## Keywords

**Append** — If present and nonzero, causes the file to be opened with the file pointer at the end of the file, ready for data to be appended. (Normally, the file is opened with the file pointer at the beginning of the file.) Under UNIX, use of *Append* prevents OPENW from truncating existing file contents.

**Block** — (OpenVMS only) If present and nonzero, specifies that the file should be processed using RMS block mode. In this mode, most RMS processing is bypassed and PV-WAVE reads and writes to the file in disk block units. Such files can be accessed only via unformatted I/O commands.

Files created in block mode can be accessed only in block mode. Block mode files are treated as an uninterpreted stream of bytes in a manner similar to UNIX stream files.

---

**NOTE** With some controller/disk combinations, RMS does not allow transfer of an odd number of bytes.

---

**Default** — (OpenVMS only) A scalar string providing a default file specification from which missing parts of *filename* are taken. For example, to make .log be the

default file extension and open a new file named `data`, you might enter the following when you open the file:

```
OPENW, 'data', Default='.log'
```

**Delete** — If present, causes the file to be deleted when it is closed.

---

**CAUTION** *Delete* will cause the file to be deleted even if it was opened for read-only access. In addition, once a file is opened with this keyword, there is no way to cancel its operation.

---

**Error** — If present and nonzero, specifies a named variable into which the error status should be placed. (If an error occurs in the attempt to open *filename*, PV-WAVE normally takes the error handling action defined by `ON_ERROR` and/or `ON_IOERROR`.)

The `OPEN` procedures always return to the caller without generating an error message when *Error* is present. A nonzero error status indicates that an error occurred. The error message can then be found in the system variable `!Err_String`.

For example, statements similar to the following can be used to detect errors:

```
OPENR, 1, 'demo.dat', Error=err
      ; Try to open the file demo.dat.

IF (err NE 0) then PRINTF, -2, !Err_String
      ; If err is nonzero, something happened, so print the error message
      ; to the standard error file (logical unit -2).
```

**Extendsize** — (OpenVMS only) If present and nonzero, specifies the number of disk blocks by which *filename* should be extended.

File extension is a relatively slow operation, and it is desirable to minimize the number of times it is done. In order to avoid the unacceptable performance that would result from extending a file a single block at a time, OpenVMS extends its size by a default number of blocks in an attempt to trade a small amount of wasted disk space for better performance.

- *Extendsize* is often used in conjunction with the *Initialsize* and *Truncate\_On\_Close* keywords.

**Fixed** — (OpenVMS only) Specifies that the file has fixed-length records. The *Record\_Size* parameter is required when opening new fixed-length files. For example:

```
OPENW, 1, 'data', /Fixed, 512
```

**Fortran** — (OpenVMS only) Specifies that FORTRAN-style carriage control will be used when you create a new file.

**F77\_Unformatted** — (UNIX only) If present and nonzero, specifies that PV-WAVE is to read and write extra information in the same manner as F77. This allows data to be processed by both FORTRAN and PV-WAVE.

Unformatted, variable-length record files produced by UNIX FORTRAN programs contain extra information along with the data in order to allow the data to be properly recovered. This is necessary because FORTRAN is based on record-oriented files, while UNIX files are simple byte streams that do not impose any record structure.

**Get\_Lun** — If present and nonzero, calls the GET\_LUN procedure to set the value of *unit* before the file is opened. Thus, the two statements:

```
GET_LUN, unit
OPENR, unit, 'data.dat'
```

can be written as:

```
OPENR, unit, 'data.dat', /Get_Lun
```

**Initialsize** — (OpenVMS only) Specifies the initial size of the file allocation in blocks. This keyword is often used in conjunction with the *Extendsize* and *Truncate\_On\_Close* keywords.

**Keyed** — (OpenVMS only) Specifies that the file has indexed organization.

**List** — (OpenVMS only) Specifies that carriage return carriage control will be used when you create a new file. If no other carriage control keyword is specified, *List* is the default.

**More** — (Under OpenVMS, allowed only with stream files.) If present and nonzero, and the specified *filename* is a terminal, formats all output to the specified *unit* in a manner similar to the UNIX `more` command. Output pauses at the bottom of each screen, at which point you can press one of the following keys:

<Space>	Causes the next page of text to be displayed.
<Return>	Causes the next line of text to be displayed.
<Q>	Suppresses all following output.
<H>	Displays the list of available options at this point.

---

For example, the following statements show how to output a file named `text.dat` to the terminal:

```
OPENR, inunit, 'test.dat', /Get_Lun
; Open the text file.
```

```

OPENW, outunit, '/dev/tty', /Get_Lun, /More
    ; Open the terminal as a file.
line = '' & readf, inunit, line
    ; Read the first line.
while not eof(inunit) do begin
    printf, outunit, line
    readf, inunit, line
        ; While there is text left, output it.
ENDWHILE
FREE_LUN, inunit & FREE_LUN, outunit
    ; Close the files and deallocate the units.

```

**None** — (OpenVMS only) Specifies that explicit carriage control will be used when you create a new file. This means that OpenVMS does not add any carriage control information to the file, and you must explicitly add any desired carriage control to the data being written to the file.

**Print** — (OpenVMS only) If present, sends the file to SYS\$PRINT (the default system printer) when it is closed.

**Segmented** — (OpenVMS only) Specifies that the file has OpenVMS FORTRAN-style segmented records. Segmented records allow logical records to exist with record sizes that exceed the maximum possible physical record sizes supported by OpenVMS. Segmented record files are useful primarily for passing data between FORTRAN and PV-WAVE programs.

**Shared** — (OpenVMS only) If present, allows other processes read and write access to the file in parallel with PV-WAVE. If *Shared* is not present, read-only files are opened for read sharing and read/write files are not shared.

---

**CAUTION** It is not a good idea to allow shared write access to files open in RMS block mode. In block mode, OpenVMS cannot perform the usual record locking which avoids file corruption. It is therefore possible for multiple writers to corrupt a block mode file. This same warning also applies to fixed-length-record disk files, which are also processed in block mode.

---

**Stream** — (OpenVMS only) Specifies that the file will be opened in stream mode.

**Submit** — (OpenVMS only) If present, specifies that the file will be submitted to SYS\$BATCH (the default system batch queue) when it is closed.

**Truncate\_On\_Close** — (OpenVMS only) If present, causes any unused disk space allocated to the file to be freed when the file is closed. This keyword can be used to get rid of excess allocations caused by the *Extendsize* and *Initialsize* keywords.

*Truncate\_On\_Close* has no effect if the *Shared* keyword is present, or if the file is open for read-only access.

**Variable** — (OpenVMS only) Specifies that the file has variable-length records. If the *record\_size* parameter is present, this keyword specifies the maximum record size. Otherwise, the only limit is that imposed by RMS (32,767 bytes). If no file organization is specified, variable-length records are the default.

**Width** — Specifies the desired width for output. If this keyword is not present, PV-WAVE uses the following rules to determine where to break lines:

- If the output file is a terminal, the terminal width is used.
- Under OpenVMS, if the file has fixed-length records or a maximum record length, the record length is used.
- If neither condition above applies, a default of 80 columns is used.

**XDR** — (Under OpenVMS, allowed only with stream files.) If present and non-zero, opens the file for unformatted XDR (eXternal Data Representation) input/output via the READU and WRITEU procedures.

Using this keyword makes binary data portable across different machine architectures by reading and writing all data in a standard format. When a file is open for XDR access, the only I/O data transfer procedures that can be used with it are READU and WRITEU.

If you open an XDR file with OPENU, you must use the *Append* keyword in the OPENU call to move the file pointer to the end of the file. By default, when an existing XDR file is opened with OPENU, the I/O transfer is set for writing, and the file pointer is set to the beginning of the file.

## Example

This example uses OPENR to open the `head.img` file for reading. This file is in the subdirectory `data`, under the main PV-WAVE distribution directory. The image is read from the file, and the file is closed.

```
OPENR, unit, FILEPATH('head.img', Subdir = 'data'), /Get_Lun
      ; Open head.img for reading.

ct = BYTARR(512, 512)
      ; Create a 256-by-256 byte array to hold the image.

READU, unit, ct
      ; Read the image.

FREE_LUN, unit
      ; Close head.img and free the file unit number associated with it.
```

```
TVSCL, ct  
; Display the image.
```

## See Also

[!Err](#), [!Err\\_String](#), [FREE\\_LUN](#), [GET\\_LUN](#), [ON\\_ERROR](#),  
[ON\\_IOERROR](#), [POINT\\_LUN](#), [READ](#), [WRITEU](#)

For background information, see Chapter 8, *Working with Data Files*, in the *PV-WAVE Programmer's Guide*.

For more information on XDR, see .

---

## ***OPEN Procedures (Windows)*** ***(OPENR, OPENU, OPENW)***

Open a specified file for input/output:

- OPENR (OPEN Read) opens an existing file for input only.
- OPENU (OPEN Update) opens an existing file for input and output.
- OPENW (OPEN Write) opens a new file for input and output.

### Usage

OPENR, *unit*, *filename*

OPENU, *unit*, *filename*

OPENW, *unit*, *filename*

### Input Parameters

*unit* — The logical unit number to be associated with the opened file.

*filename* — The name of the file to be opened.

### Keywords

*Append* — If present and nonzero, causes the file to be opened with the file pointer at the end of the file, ready for data to be appended. (Normally, the file is opened with the file pointer at the beginning of the file.)

*Delete* — If present, causes the file to be deleted when it is closed.

---

**CAUTION** *Delete* will cause the file to be deleted even if it was opened for read-only access. In addition, once a file is opened with this keyword, there is no way to cancel its operation.

---

**Error** — If present and nonzero, specifies a named variable into which the error status should be placed. (If an error occurs in the attempt to open *filename*, PV-WAVE normally takes the error handling action defined by ON\_ERROR and/or ON\_IOERROR.)

The OPEN procedures always return to the caller without generating an error message when *Error* is present. A nonzero error status indicates that an error occurred. The error message can then be found in the system variable !Err\_String.

For example, statements similar to the following can be used to detect errors:

```
OPENR, 1, 'demo.dat', Error=err
    ; Try to open the file demo.dat.

IF (err NE 0) then PRINTF, -2, !Err_String
    ; If err is nonzero, something happened, so print the error message
    ; to the standard error file (logical unit -2).
```

**Get\_Lun** — If present and nonzero, calls the GET\_LUN procedure to set the value of *unit* before the file is opened. Thus, the two statements:

```
GET_LUN, unit
OPENR, unit, 'data.dat'
```

can be written as:

```
OPENR, unit, 'data.dat', /Get_Lun
```

**String\_Xdr** — If present and nonzero, interprets the XDR strings in READU, WRITEU procedures as standard XDR strings (string length encoded as an unsigned integer) followed by the “length” bytes of the string. By default, XDR strings in PV-WAVE are interpreted as string length encoded as two unsigned integers followed by the “length” bytes of the string.

**Width** — Specifies the desired width for output. If this keyword is not present, PV-WAVE uses the following rules to determine where to break lines:

- If the output file is a terminal, the terminal width is used.
- If the output file is not a terminal, a default of 80 columns is used.

**XDR** — If present and nonzero, opens the file for unformatted XDR (eXternal Data Representation) input/output via the READU and WRITEU procedures.

Using this keyword makes binary data portable across different machine architectures by reading and writing all data in a standard format. When a file is open for XDR access, the only I/O data transfer procedures that can be used with it are READU and WRITEU.

### **Example**

This example uses OPENR to open the `head.img` file for reading. This file is in the subdirectory `data`, under the main PV-WAVE distribution directory. The image is read from the file, and the file is closed.

```
OPENR, unit, FILEPATH('head.img', $
    Subdir = 'data'), /Get_Lun
    ; Open head.img for reading.
ct = BYTARR(512, 512)
    ; Create a 256-by-256 byte array to hold the image.
READU, unit, ct
    ; Read the image.
FREE_LUN, unit
    ; Close head.img and free the file unit number associated with it.
TVSCL, ct
    ; Display the image.
```

### **See Also**

[FREE\\_LUN](#), [GET\\_LUN](#), [ON\\_ERROR](#), [ON\\_IOERROR](#),  
[POINT\\_LUN](#), [READ](#), [WRITEU](#)

System Variables: [!Err](#), [!Err\\_String](#)

For background information, see Chapter 8, *Working with Data Files*, in the *PV-WAVE Programmer's Guide*.

For more information on XDR, see .

---

## **OPENURL Procedure**

Opens a file on the Internet to be accessed (through Java) using PV-WAVE.

### **Usage**

OPENURL, *url*, *Unit = unit*

### **Input Parameters**

*url* — A string containing a Uniform Resource Locator (URL) for the document (file) to be read.

### **Keywords**

*Proxy* — A string containing the host name of a proxy server (firewall) if required by your network; or, of the form 'hostname:nn', where nn is a number specifying the port number of the proxy server.

*Unit* — The unit returned from the call, as for the PV-WAVE OPENR, and the READF and READU procedures.

### **Discussion**

---

**NOTE** Java software must be available in your path and its location must be in the operating-system search path before you start PV-WAVE. (You should be able to run a Java program by typing `java` and the program name at the OS prompt.)

---

The OPENURL procedure spawns a Java process to open and read an URL anywhere on the Internet. It then passes the data to an open unit, which can be processed by a READF or READU procedure call.

---

**Windows USERS** Make sure that one of the environment variables TMP or TEMP is set to a writable directory to enable temporary file writing.

---

### **Example**

For other examples of using PV-WAVE across the Internet, see the demonstrations under the following directories:

(UNIX) <wavedir>/demo/web  
(OpenVMS) <wavedir>:[DEMO.WEB]  
(Windows) <wavedir>\demo\web

where <wavedir> is the main PV-WAVE directory.

This example procedure uses OPENURL, keeping the file in *unit* while reading and writing the file to standard output until no more strings are found.

```
PRO URL_DEMO, url = url
IF N_ELEMENTS(url) EQ 0 THEN $
    url = 'http://www.vni.com'
    ; Verify that the URL exists.
OPENURL, url, unit = unit
str = ''
ON_IOERROR, done
WHILE 1 DO BEGIN
    READF, unit, str
    PRINT, str
ENDWHILE
    ; Reading and writing the content of the URL file.
done:
END
```

## See Also

[HTML\\_\\* routines](#), [OPEN procedures](#), [READ procedures](#)

---

## ***O*PLOT Procedure**

Plots vector data over a previously drawn plot.

### **Usage**

OPLOT,  $x$  [,  $y$ ]

### **Input Parameters**

$x$  — A vector. If only one parameter is supplied,  $x$  is plotted on the  $y$ -axis as a function of point number.

$y$  — (optional) A vector. If two parameters are supplied,  $y$  is plotted as a function of  $x$ .

### **Keywords**

OPLOT keywords let you control many aspects of the plot's appearance. These keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

Background	Noclip	T3d	[XYZ]Tickformat
Channel	Nodata	Thick	[XYZ]Ticklen
Charsize	Noerase	Tickformat	[XYZ]Tickname
Charthick	Normal	Ticklen	[XYZ]Ticks
Clip	Nsum	Title	[XYZ]Tickv
Color	Polar	[XYZ]Charsize	[XYZ]Title
Data	Position	[XYZ]Gridstyle	[XYZ]Type
Device	Psym	[XYZ]Margin	YLabelCenter
Font	Save	[XYZ]Minor	YNozero
Gridstyle	Solid_Psym	[XYZ]Range	ZValue
Linestyle	Subtitle	[XYZ]Style	

---

### **Discussion**

OPLOT differs from PLOT only in that it does not generate a new axis. Instead, it uses the scaling established by the most recent call to PLOT and simply overlays a plot of the data on the existing axis. Each call to PLOT establishes the plot window (the region of the display enclosed by the axes), the axis types (linear or log), and the scaling. This information is saved in the system variables !P, !X, and !Y, and used by subsequent calls to OPLOT.

## Example

In this example, 10 random points are plotted as square markers using the PLOT procedure. Procedure OPLOT is then used to plot a cubic spline interpolant to the random points. The square markers at the random points remain in the plotting window when the interpolant is plotted because OPLOT draws over what is already in the plotting window. This example uses the PV-WAVE:IMSL Mathematics Toolkit and PV-WAVE:IMSL Statistics Toolkit functions CSINTERP, RANDOMOPT, RANDOM, and SPVALUE.

```
RANDOMOPT, Set = 45321
      ; Create a vector of 10 random values.
x = RANDOM (10)
PLOT, x, Psym = 6
      ; Plot the random points as a function of vector index. Use square
      ; marker symbols to represent the data points.
pp = CSINTERP(FINDGEN(10), x)
ppval = SPVALUE(FINDGEN(100)/10, pp)
      ; Compute the cubic spline interpolant.
OPLOT, FINDGEN(100)/10, ppval
      ; Plot the interpolant over the marker symbols (see ).
```

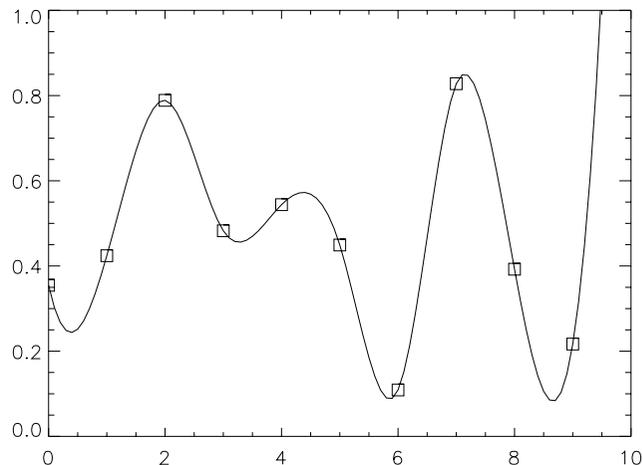


Figure 2-32 Overplotting a plot.

## See Also

[OPLOTERR](#), [PLOT](#), [PLOTERR](#)

For more information, see Chapter 4, *Displaying 2D Data*, in the *PV-WAVE User's Guide*.

---

## ***O*PLOTERR Procedure**

Standard Library procedure that overplots symmetrical error bars on any plot already output to the display device.

### **Usage**

`O`PLOTERR, *x*, *y*, *error* [, *psym*]

### **Input Parameters**

*x* — A real vector containing the *x*-coordinates of the data to plot. If not present, *x* is assumed to be a vector of the same size as *y* and to have integer values beginning at 0 and continuing to the size of *y* – 1.

*y* — A real vector containing the *y* coordinates of the data to plot.

*error* — A vector containing the symmetrical error bar values at every element in *y*.

*psym* — (optional) Specifies the plotting symbol to use. It corresponds to the system variable !Psym. If not specified, the default is 7 (the symbol “X”).

### **Keywords**

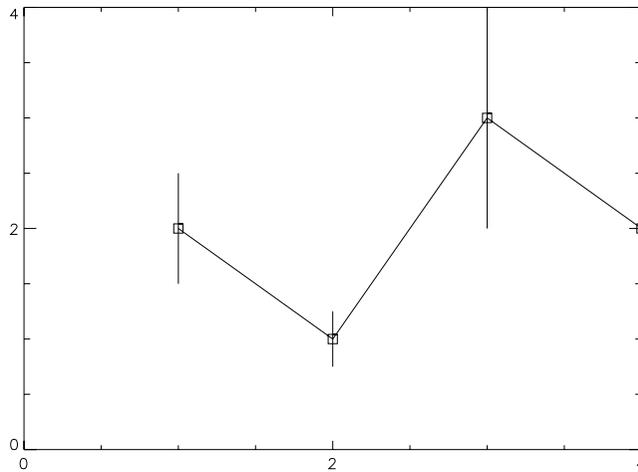
None.

### **Example 1**

To plot error bars over the *x* and *y* vectors, with symbols at the data values, use the following commands:

```
x = [1, 2, 3, 4]
y = [2, 1, 3, 2]
PLOT, x, y
error = [0.5, 0.25, 1, 0]
psym = 6
O
```

This produces the plot shown in .

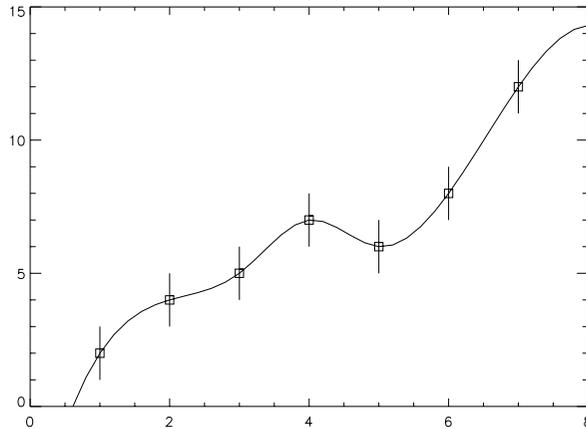


**Figure 2-33** In this example, OPLOTERR was used to plot error bars over the x and y vectors, using the square symbol at the data values.

## Example 2

This example plots a B-spline interpolant to some scattered data. The data points, along with error bars extending one unit on either side of the data points, are then plotted on top of the interpolant. This example uses the PV-WAVE:IMSL Mathematics Toolkit function BSINTERP.

```
x = INDGEN(7) + 1
    ; Generate the abscissas.
y = [2, 4, 5, 7, 6, 8, 12]
    ; Create a vector of ordinates.
bs = BSINTERP(x, y)
    ; Compute the B-spline interpolant.
bsval = SPVALUE(FINDGEN(100)/5, bs)
    ; Plot the interpolant.
PLOT, FINDGEN(100)/5, bsval, XRange = [0, 8], YRange = [0, 13]
err = MAKE_ARRAY(7, Value = 1)
    ; Create the error bar vector.
OPLOTERR, x, y, err, 6
    ; Overplot the data points, using square marker symbols and the
    ; error bars (see ).
```



**Figure 2-34** Scattered data interpolant with overplotted data points and symmetric error bars.

## See Also

[ERRPLOT](#), [PLOTERR](#)

System Variables: [!P.Psym](#)

## ***OPTION\_IS\_LOADED*** Function

Checks if a specified Option Programming Interface (OPI) optional module is currently loaded.

### Usage

*result* = `OPTION_IS_LOADED` (*option\_name*)

### Input Parameters

*option\_name* — (string) The name of the OPI option to check.

### Output Parameters

*result* — A value indicating the status of the given OPI option.

- 1 Indicates the OPI option is currently loaded.
- 0 Indicates the OPI option is not loaded.

## Keywords

None.

## Discussion

OPI options can be loaded explicitly by any PV-WAVE user using the `LOAD_OPTION` procedure. These optional modules can be written in C or FORTRAN, and can contain new system functions or other primitives. For detailed information on creating OPI options, see the *PV-WAVE Application Developer's Guide*.

## Example

```
IF NOT OPTION_IS_LOADED('SAMPLE') THEN $
    LOAD_OPTION, 'SAMPLE'
```

## See Also

[LOAD\\_OPTION](#), [SHOW\\_OPTIONS](#), [UNLOAD\\_OPTION](#)

---

## ***ORDER\_BY Function***

Sorts the rows in a PV-WAVE table variable to create a new table.

### Usage

```
result = ORDER_BY(in_table, 'col_1 [ASC | DESC] [, col_2 [ASC | DESC]] ...
[, col_n [ASC | DESC]]')
```

---

**NOTE** The entire second parameter is a string and must be enclosed in quotes. Also, note that the vertical bar (|) means “or” in this usage. In this case, you can use either *ASC* or *DESC*, but not both.

---

### Input Parameters

*in\_table* — An input PV-WAVE table variable to be sorted.

*col\_i* — The column(s) on which the sort is to be performed. If more than one column is specified, then result is sorted first by *col\_1*, then *col\_2*, and so on.

**ASC** — Requires that the rows of the result are sorted in ascending order for that column. If no sort order is specified, **ASC** is the default.

**DESC** — Requires that the rows of the result are sorted in descending order for that column.

## Returned Value

**result** — The resulting PV-WAVE table variable, containing the same rows as *in\_table*, but in the requested sort order.

## Keywords

None.

## Discussion

The **ORDER\_BY** function produces similar output to the *Order By* option of the **QUERY\_TABLE** function, but **ORDER\_BY** has a more compact and convenient syntax.

## Example

Consider the following table variable (*prop\_trx*), which contains information about property transactions. Property type, location, amount, and transaction date are all recorded.

TRX_ID	PROP_TYPE	PROP_XGRID	PROP_YGRID	TRX_AMT	TRX_DATE
1	2BR_HOUSE	1.45096	2.34159	142166.00	04/20/2005 02:00:37
2	3BR_CONDO	1.30454	1.12332	108730.00	05/23/2010 11:06:15
3	4+BR_HOUSE	0.41492	0.67620	273141.00	08/05/1997 06:22:53
4	4+BR_HOUSE	3.52749	0.91387	267949.00	02/25/2010 06:55:33
5	2BR_HOUSE	1.56556	0.46332	193346.00	02/22/1997 12:32:35
6	1BR_HOUSE	2.33187	2.94950	128165.00	04/01/1993 09:46:35
7	4+BR_HOUSE	1.74596	3.32184	229348.00	05/20/2005 00:15:39
8	3BR_HOUSE	4.38503	1.13107	203425.00	07/11/2001 11:17:37
9	4+BR_HOUSE	4.20096	0.90143	376974.00	05/04/1991 23:06:53
10	STUDIO	0.50743	1.31675	39148.00	05/26/1999 19:01:01

We execute an **ORDER\_BY** function, to sort first by property type, and then by transaction date:

```
prop_trx = ORDER_BY( prop_trx, 'prop_type, trx_date')
```

Now, the rows in *prop\_trx* have been reordered:

TRX_ID	PROP_TYPE	PROP_XGRID	PROP_YGRID	TRX_AMT	TRX_DATE
6	1BR_HOUSE	2.33187	2.94950	128165.00	04/01/1993 09:46:35
5	2BR_HOUSE	1.56556	0.46332	193346.00	02/22/1997 12:32:35
1	2BR_HOUSE	1.45096	2.34159	142166.00	04/20/2005 02:00:37
2	3BR_CONDO	1.30454	1.12332	108730.00	05/23/2010 11:06:15
8	3BR_HOUSE	4.38503	1.13107	203425.00	07/11/2001 11:17:37
9	4+BR_HOUSE	4.20096	0.90143	376974.00	05/04/1991 23:06:53
3	4+BR_HOUSE	0.41492	0.67620	273141.00	08/05/1997 06:22:53
7	4+BR_HOUSE	1.74596	3.32184	229348.00	05/20/2005 00:15:39
4	4+BR_HOUSE	3.52749	0.91387	267949.00	02/25/2010 06:55:33
10	STUDIO	0.50743	1.31675	39148.00	05/26/1999 19:01:01

Notice that, for properties of the same type, the rows are ordered by ascending date.

## See Also

[GROUP\\_BY](#), [QUERY\\_TABLE](#), [UNIQUE](#)

For more information on [BUILD\\_TABLE](#), see .

For information on reading data into variables, see .

---

## **PADIT Function**

Pads an array with variable thickness.

### **Usage**

$c = \text{PADIT}( a, [b] )$

### **Input Parameters**

$a$  — An  $n$ -dimensional array.

$b$  — (optional) A scalar with which to pad the array; if  $b$  is omitted then each pad layer is a copy of its underlying layer.

### **Returned Value**

$c$  — The padded version of parameter  $a$ .

### **Keywords**

$t$  — An  $n$ -by-2 array of integers:  $t(m,0)$  and  $t(m,1)$  are the bottom and top pad thicknesses for dimension  $m$  of  $a$ ; the default is to pad each side with one layer.

### **Example**

```
a = INDGEN( 2, 3 )      &      PM, a
      0      2      4
      1      3      5
PM, PADIT( a )
      0      0      2      4      4
      0      0      2      4      4
      1      1      3      5      5
      1      1      3      5      5
PM, PADIT( a, 6 )
      6      6      6      6      6
      6      0      2      4      6
      6      1      3      5      6
      6      6      6      6      6
```

### **See Also**

[VOL\\_PAD](#)

---

## ***PALETTE Procedure***

Standard Library procedure that lets you interactively create a new color table based on the RGB color system.

### **Usage**

PALETTE [, *colors\_out*]

### **Input Parameters**

None.

### **Output Parameters**

*colors\_out* — (optional) Contains the color values of the final color table in the form of a two-dimensional array that has the number of colors in the color table as the first dimension and the integer 3 as the second dimension.

The values for red are stored in the first row, the values for green are stored in the second row, and those for blue in the third row; in other words:

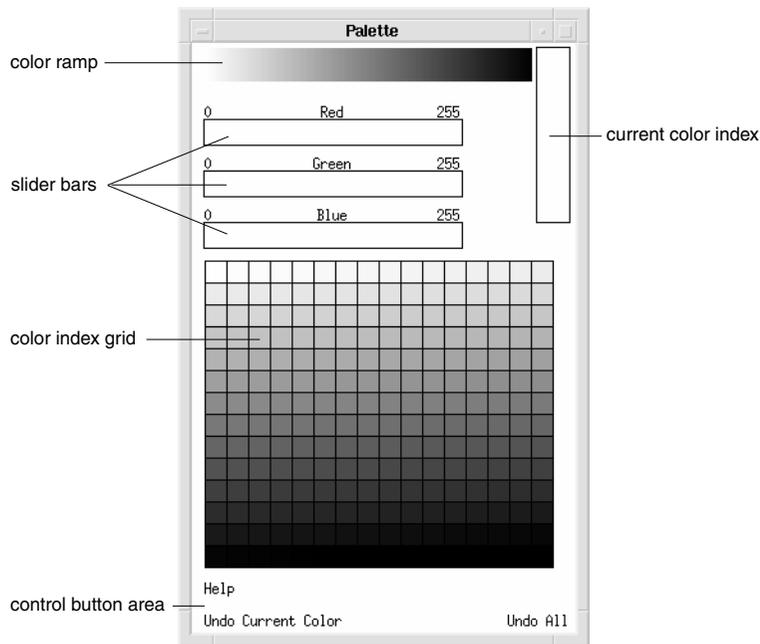
$$red = colors\_out(*, 0)$$
$$green = colors\_out(*, 1)$$
$$blue = colors\_out(*, 2)$$

### **Keywords**

None.

### **Discussion**

PALETTE works only on displays with window systems. It creates an interactive window that lets you use the mouse to create a new color table. This window is shown in .



**Figure 2-35** The PALETTE window lets you use the mouse to create a new color table interactively.

The PALETTE window contains the following items:

- **Color Ramp** — Displays the current color table.
- **Slider Bars** — Used to adjust the color values of the selected color index.
- **Color Index Grid** — Contains a representation of each color index that comprise the current color table. There are three rectangular slider bars for the red, green, and blue color vectors. The grid is used to select a color index to adjust and to create a new color ramp. The number of color indices that appear in the grid depend on the number of colors selected with the WINDOW procedure.
- **Control Button Area** — Contains buttons for selecting **Help**, **Undo Current Color**, and **Undo All**. To select a button, click it with the left mouse button.

The **Help** button displays information about the procedure in the main PV-WAVE window

The **Undo All** button resets all color indices to their default values.

The **Undo Current Color** button resets the currently selected color index to its default value.

You can modify any number of color indices, or produce an interpolation between two modified color indices. The default color table is the currently loaded color table.

For information on using the PALETTE window, select the **Help** button and follow the online instructions provided.

## Example 1

```
TVSCL, LINDGEN(256, 256)
```

```
PALETTE, rgb_array
```

— Modify the color table to your liking and exit the procedure. —

```
SAVE, filename = 'my_colortable', rgb_array
```

```
LOADCT, 2
```

```
RESTORE, 'my_colortable'
```

```
rgb_array = REFORM(rgb_array, $  
    N_ELEMENTS(rgb_array)/3, 3)
```

```
TVLCT, rgb_array(*, 0), rgb_array(*, 1), rgb_array(*, 2)
```

## Example 2

```
TVSCL, LINDGEN(256, 256)
```

```
PALETTE
```

— Modify the color table to your liking and exit the procedure. —

```
TVLCT, r, g, b, /Get
```

```
SAVE, filename = 'my_colortable_2', r, g, b
```

```
LOADCT, 8
```

```
RESTORE, 'my_colortable_2'
```

```
TVLCT, r, g, b
```

## See Also

[COLOR\\_CONVERT](#), [COLOR\\_EDIT](#), [COLOR\\_PALETTE](#),  
[MODIFYCT](#), [WgCeditTool](#)

For more ideas about what can be done with color tables, see .

---

## ***PARAM\_PRESENT Function***

Tests if a parameter was actually present in the call to a procedure or function.

### **Usage**

*result* = PARAM\_PRESENT(*parameter*)

### **Input Parameters**

*parameter* — One of the formal parameters as given in the function or procedure definition.

### **Returned Value**

*result* — A nonzero value (true) if the parameter was present in the call to the current procedure or function. Returns a zero value (false) if the parameter was not present.

### **Keywords**

None.

### **Discussion**

PARAM\_PRESENT compliments the KEYWORD\_SET and N\_ELEMENTS functions. PARAM\_PRESENT lets you distinguish between the two cases in which KEYWORD\_SET returns FALSE, and the two cases when N\_ELEMENTS returns zero (0).

#### ***With KEYWORD\_SET***

The KEYWORD\_SET function returns FALSE:

- 1) When the keyword was set to zero or an undefined variable.
- 2) When the keyword was not used in the call.

PARAM\_PRESENT distinguishes between these cases by returning TRUE in case 1 and FALSE in case 2.

## ***With N\_ELEMENTS***

The N\_ELEMENTS function returns zero (0) in two cases;

- 1) When the keyword or parameter was present but is an undefined variable.
- 2) When the keyword or parameter was not present in the call.

PARAM\_PRESENT distinguishes between these two cases by returning TRUE in case 1 and FALSE in case 2.

## **Example**

The following example demonstrates the expected results when the functions PARAM\_PRESENT, KEYWORD\_SET, and N\_ELEMENTS are used in a procedure.

```
PRO test, Key = k
  IF (PARAM_PRESENT(k)) THEN $
    PRINT, 'Key is present' $
  ELSE PRINT, 'Key is not present'
  IF (KEYWORD_SET(k)) THEN $
    PRINT, 'Key is set' $
  ELSE PRINT, 'Key is NOT SET'
  PRINT, 'Number of elements in Key = ', N_ELEMENTS(k)
END

WAVE> test
  Key is not present
  Key is NOT SET
  Number of elements in Key = 0

WAVE> test, Key = 0
  Key is present
  Key is NOT SET
  Number of elements in key = 1

WAVE> test, Key = a
  Key is present
  Key is NOT SET
  Number of elements in key = 0

WAVE> a = 10

WAVE> test, Key = a
  Key is present
  Key is SET
  Number of elements in Key = 1
```

## See Also

[KEYWORD\\_SET](#), [N\\_ELEMENTS](#), [N\\_PARAMS](#)

For more information on parameter checking, see .

---

## **PARSEFILENAME Procedure**

Extracts specified parts of a full file pathname.

### Usage

PARSEFILENAME, *pathname*

### Input Parameters

*pathname* — A string containing a full file pathname.

### Keywords

*Extension* — If set, returns the filename extension (for example, .pro).

*Filename* — If set, returns a string containing the file root name with its extension (for example, boulder.img).

*Fileroot* — If set, returns a string containing the filename without its extension.

*Path* — If set, returns a string containing the full pathname without the filename.

*Separator* — If set, specifies a single-char string to use as a directory separator. The default separator is platform-specific:

(UNIX)        ' / '

(OpenVMS)    ' ] '

(Windows)    ' \ '

### Discussion

The PARSEFILENAME procedure takes a string containing a path and/or a filename and returns its constituent components (path, filename, file root, and extension) in variables specified using the keywords.

For example, on a UNIX system the file `file_mine.pro` consists of the file root `file_mine` and the extension `pro`.

## Example

The following example demonstrates PARSEFILENAME with a UNIX pathname:

```
full = '/home/work/data/testimage.tif'
PARSEFILENAME, full, Fileroot = root, $
    Path = path, Filename = name, $
    Extension = extension
INFO, path, root, name, extension

    PATH          STRING      = '/home/work/data/'
    ROOT          STRING      = 'testimage'
    NAME          STRING      = 'testimage.tif'
    EXTENSION     STRING      = 'tif'
```

## See Also

[CHECKFILE](#)

---

## PIE Procedure

Displays data as a pie chart.

### Usage

PIE, *data* [, *labels*]

### Input Parameters

*data* — An array containing the data to plot as a pie chart.

*labels* — (optional) A string array of labels for the legend. The array must contain the same number of elements as *data*. If this parameter is not supplied, no legend is displayed.

### Keywords

*Colors* — Sets an array of color indices specifying colors for the slices. If there are more slices than array elements, the colors are repeated. By default, the colors are chosen sequentially from the color table, excluding black and white. If *Line\_Fill* is specified, the default color is !P.Color.

**Line\_Fill** — If nonzero, a line fill algorithm is used to fill the slices. This keyword can also be set to an array of ones and zeroes to apply line filling to specific slices. (Default: no line filling)

**Fill\_Thick** — A floating point scalar or array specifying the thickness of fill lines. If set to a scalar value, all slices receive the same line thickness. If set to an array, line thicknesses are mapped, in sequential slices, to the value of each array index. A thickness of 1 is normal, two is double-wide, and so on. (Default: !P.Thick)

---

**NOTE** This keyword has no effect unless *Line\_Fill* is set.

---

**Fill\_Orientation** — A floating-point scalar or array specifying the orientation of fill lines, in degrees counterclockwise from the horizontal. If set to a scalar, the orientation of fill lines in all slices is the same. If set to an array, the orientations are mapped, in sequential slices, to the value of each array index. (Default: each slice is set to a unique angle)

---

**NOTE** This keyword has no effect unless *Line\_Fill* is set.

---

**Fill\_Linestyle** — An integer or integer array specifying the style of fill lines. If set to a scalar, all slices are filled with the same linestyle. If set to an array, the linestyle of each slice is mapped, in sequential slices, to the value of each array index. (Default: !P.Linestyle)

---

**NOTE** This keyword has no effect unless *Line\_Fill* is set.

---

Valid linestyle indices are shown in the following table:

---

Index	X Windows Style	Windows Style
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

---

**Fill\_Spacing** — A floating point scalar or array specifying the space, in centimeters, between fill lines. If a set to a scalar, the spacing between lines in all slices is the same. If set to an array, the spacing for each slice is mapped, in sequential slices, to the value of each array index. (Default: one percent of the width of the graphics window)

---

**NOTE** This keyword has no effect unless *Line\_Fill* is set.

---

**Fill\_Background** — An integer or integer array specifying the background color behind fill lines. If set to a scalar, the specified color applies to all of the slices. If set to an array, the colors for each slice are mapped, sequentially, to the value of each array index. Set this keyword to -1 to specify no background. (Default: no color behind fill lines)

---

**NOTE** This keyword has no effect unless *Line\_Fill* is set.

---

**Text\_Color** — An integer specifying the color index for the text used in titles and legends. (Defaults: !P.Color)

**Outline\_Color** — An integer specifying the color index used for the outline of the pie slices and for the box around the legend. (Default: !P.Color)

**NoLegend\_Box** — If nonzero, do not draw a box around the legend.

**Legend\_Position** — Specifies the corner in which to place the legend. Choices are:

- 0 Northwest corner (Default)
- 1 Northeast corner
- 2 Southwest corner
- 3 Southeast corner

---

**TIP** If you do not want a legend to appear, do not specify the *labels* input parameter.

---

**Percent\_Label** — If nonzero, each slice is labeled with its percentage of the total pie. If the slice is too small so that the text won't fit, then it is not labeled. You cannot specify both the *Percent\_Label* and *Value\_Label* keywords.

**Value\_Label** — If nonzero, each slice is labeled with its value from the *data* array. If the slice is too small so that the text won't fit, then it is not labeled. You may not specify both the *Percent\_Label* and *Value\_Label* keywords.

**Label\_Colors** — Colors used for the slice labels, if one of *Percent\_Label* or *Value\_Label* is set. May be a scalar or an array. If set to a scalar, the specified color applies to all of the slices. If set to an array, the colors for each slice are mapped, sequentially, to the value of each array index. (Default: !P.Color)

**Explode** — An array of ones and zeroes specifying which (if any) slices should be “exploded” (drawn away from the center of the pie circle). This array must have the same number of elements as *data*. (Default: not exploded)

**Offset\_Explode** — A scalar specifying the offset from the center of the circle for exploded slices. The value is a fraction of the circle radius. (Default: 0.1)

**Shadow\_Color** — An integer specifying the color index for a shadow under the circle. (Default: no shadow)

**Offset\_Shadow** — A scalar specifying the offset from the center of the pie circle for the shadow. The value is a fraction of the circle radius. (Default: 0.05)

**Slice\_Start\_Ang** — A floating point value specifying the angle, in degrees clockwise from the vertical, where to start drawing the first slice—the slice represented by *data*(0). (Default: 0 (vertical)).

**Slice\_Reverse\_Direction** — If nonzero, draws slices in counterclockwise order rather than in the default direction, clockwise.

Other PIE keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

---

<a href="#">Charsize</a>	<a href="#">Noerase</a>	<a href="#">Title</a>
<a href="#">Font</a>	<a href="#">Subtitle</a>	<a href="#">[XY]Margin</a>

## Discussion

PIE and PIE\_CHART produce similar looking graphics. PIE includes a legend option and always ensures that label text and graphics fall within the plot window.

A legend for the colored pie slices is displayed in one corner, and labels (percent or data value) may be displayed on the slices.

If the legend occupies a significant portion of the plot area, there may not be enough room for the pie chart. The pie may be drawn very small, or it may overlap the legend. Use a smaller character size or a larger window.

Some combinations of shadowing and explosion may cause the pie to be slightly smaller than you may consider optimal. Sometimes, the shadow may overlap the

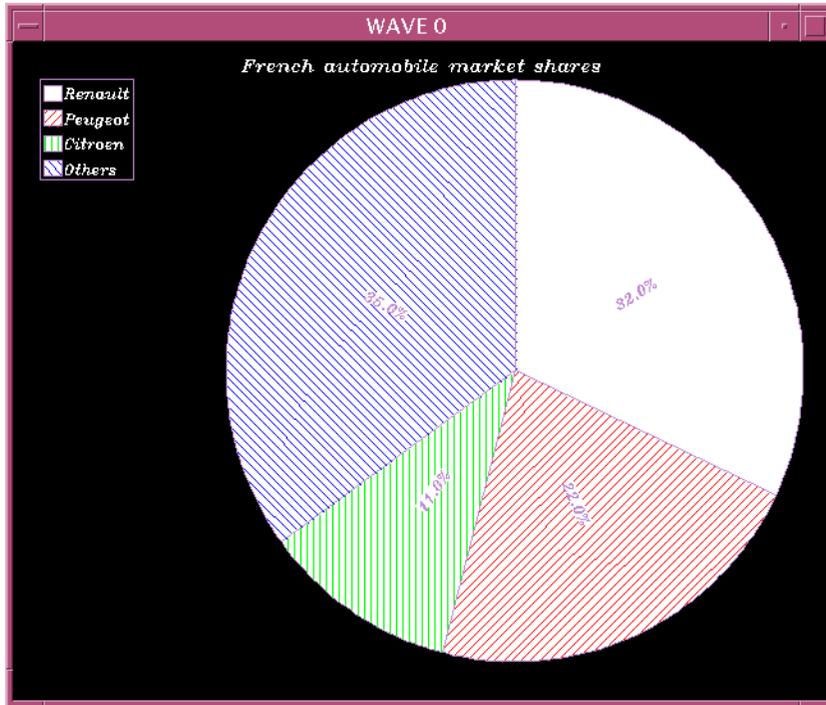
corner of the legend box. Try moving the legend to the left side (set the keyword *Legend\_Position* to 0 or 2).

Negative values in *data* produce unwanted plotting results (some slices are plotted backwards); therefore, the absolute value of *data* is plotted. All labels, however, will display the correct negative values.

You can use !P.Multi, !P.Position, and !P.Region with PIE.

## Examples

```
; Create data
data = FLTARR(4)
data(0) = 32
data(1) = 22
data(2) = 11
data(3) = 35
; Create labels
strg = STRARR(4)
strg(0)='Renault'
strg(1)='Peugeot'
strg(2)='Citroen'
strg(3)='Others'
title = '!18French automobile market shares'
; Create pie chart
PIE, data, strg, /Line_fill, Title=title, Fill_background=1, $
  /Percent_label, Text_color=1
```



**Figure 2-36** A pie chart with line-filled slices, a legend, and a title. Each slice is labeled with its percentage of the total pie.

## See Also

[PIE\\_CHART](#)

---

## ***PIE\_CHART Procedure***

Displays data as a pie chart.

### **Usage**

`PIE_CHART, data, xcenter, ycenter, radius`

### **Input Parameters**

*data* — An array of data values to be displayed as a pie chart.

***xcenter, ycenter*** — Specifies the *x* and *y* coordinates of the center of the pie chart, by default in normal coordinates.

***radius*** — Specifies the size of the radius of the pie, by default in normal coordinates.

## Keywords

***Device*** — If nonzero, the pie chart is drawn in device coordinates. (Default: normal coordinates)

***Label*** — A string array specifying the labels for each slice. The array must contain the same number of elements as *data*. If a slice is too small to display text, the label is not drawn. (Default: no labels)

***Font*** — An integer specifying a PV-WAVE font command. For example, 3 is the command for the font Complex Roman. Do not put an exclamation mark (!) in front of the font command. For a complete list of font commands, see *Chapter 10: Using Fonts* in the *PV-WAVE User's Guide*. (Default: current font)

***Charsize*** — Sets the overall character size for annotation. A *Charsize* of 1.0 is the normal; a size of 2.0 is twice as big, and so on. (Default: !P.Charsize)

***Tposition*** — An integer or integer array specifying where to draw labels for the individual slices.

- 0 Draw the label inside the slice.
- 1 Draw the label outside the slice. Draw the label text horizontally.
- 2 Draw the label outside the slice. Align the angle of the label text with the slice.

***Tcolor*** — An integer array specifying colors for the labels. The colors for each label are mapped, sequentially, to the value of each array index.

***Tperct*** — If nonzero, the percentage for each slice is displayed. (Default: the percentage is not displayed)

***Tvalue*** — If nonzero, the value of each slice is displayed. (Default: the values are not displayed)

***Tborder*** — If nonzero, a border is drawn around each label. (Default: no border)

***Tbord\_color*** — An integer specifying the color index for the label border. (Default: !P.Color)

**Shade** — A floating point value specifying the percent of displacement for a shadow under the pie chart. By default, the shadow is offset in the direction of 315 degrees. (Default: zero displacement)

**Explode** — A floating point scalar or array specifying how far from the center to offset or “explode” the pie slices. This value is specified as a percentage of the pie radius. If a scalar, all slices are offset by the same amount. If set to an array, the offset for each slice is mapped, sequentially, to the value of each array index. (Default: no offset)

**Color** — For a description of this keyword, see Chapter 3, *Graphics and Plotting Keywords*.

## Discussion

PIE and PIE\_CHART produce similar looking graphics. PIE includes a legend option and always ensures that label text and graphics fall within the plot window.

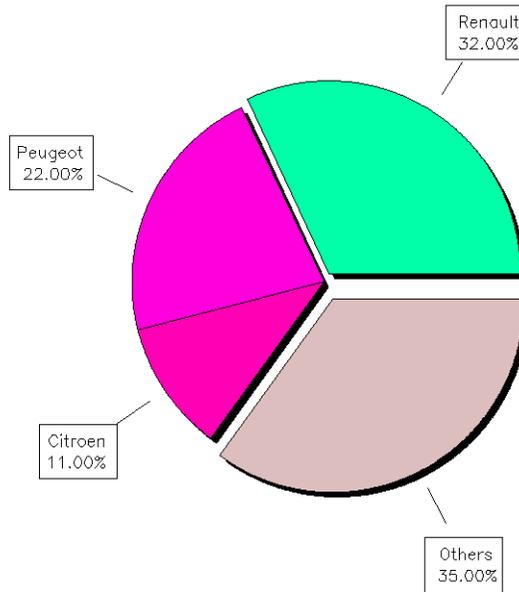
This procedure creates a pie chart at a specified position within the graphics window. You can add text labels and colors to individual slices of the pie. In addition, pie slices can be offset or “exploded” from the center, and slices can be given shadows for a three-dimensional look.

You can display up to 30 slices; however fewer than 15 is recommended for the best display. Labels are not displayed in any slices that are too small. If data for a particular slice represents less than 0.01% of the overall dataset, the slice is not drawn and an informational message appears.

## Examples

```
data = fltarr(4)
data(0) = 32
data(1) = 22
data(2) = 11
data(3) = 35
strg = strarr(4)
; Create labels
; -----
strg(0)='Renault'
strg(1)='Peugeot'
strg(2)='Citroen'
strg(3)='Others'
title = '!18French automobile market shares'
; Create the window
; -----
WINDOW, /Free, Xsize=800, Ysize=800
; Initialize a white background
; -----
loadct,13
!p.color=0
!P.Background=!d.n_colors-1
ERASE,!P.Background
; Plot the chart
; -----
PIE_CHART, data, .5, .5, .25, /Tperc, Explode=[.05,0,0,.1], $
    Shade=0.04, Tpos=[1,1,1,1], /Tborder, Color=[60,120,140,200],$
    Label=strg, Font='3', Charsize=1.5
XYOUTS, .5,.9, Title, Charsize=3, Charthick=2, Align=.5, $
    /Normal, Color=0
```

## *French automobile market shares*



**Figure 2-37** An exploded pie chart with labels and a title.

### **See Also**

[PIE](#)

---

## ***PLOT Procedures*** ***(PLOT, PLOT\_IO, PLOT\_OI, PLOT\_OO)***

Produce a 2D graph of vector parameters:

PLOT produces a simple XY plot.

PLOT\_IO produces an XY plot with logarithmic scaling on the y-axis.

PLOT\_OI produces an XY plot with logarithmic scaling on the x-axis.

PLOT\_OO produces an XY plot with logarithmic scaling on both the x- and y-axes.

### **Usage**

PLOT,  $x$  [,  $y$ ]

PLOT\_IO,  $x$  [,  $y$ ]

PLOT\_OI,  $x$  [,  $y$ ]

PLOT\_OO,  $x$  [,  $y$ ]

### **Input Parameters**

$x$  — A vector. If only one parameter is supplied,  $x$  is plotted on the y-axis as a function of point number.

$y$  — (optional) A vector. If two parameters are supplied,  $y$  is plotted as a function of  $x$ .

### **Keywords**

Keywords let you control many aspects of a plot's appearance. Valid keywords for the four PLOT procedures are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

Background	Noclip	Week_Boundary
Box	Nodata	[XYZ]Charsize
Channel	Noerase	[XYZ]Gridstyle
Charsize	Normal	[XYZ]Margin
Charthick	Nsum	[XYZ]Minor
Clip	Polar	[XYZ]Range

Color	Position	[XYZ]Style
Compress	Psym	[XYZ]Tickformat
Data	Solid_Psym	[XYZ]Ticklen
Device	Start_Level	[XYZ]Tickname
DT_Range	Subtitle	[XYZ]Ticks
Font	Symsize	[XYZ]Tickv
Gridstyle	T3d	[XYZ]Title
Linestyle	Thick	[XYZ]Type
Max_Levels	Tickformat	YLabelCenter
Month_Abbr	Ticklen	YNozero
	Title	ZValue

---

## Example 1

This example shows the most common way to use the PLOT procedure.

```
x = FINDGEN(37) * 10
y = SIN(x * !Dtor)
    ; Create a sine wave from 0 to 360 degrees.

PLOT, y
    ; Plot the sine wave using the default values for the plotting
    ; keywords.

PLOT, x, y, XRange=[0, 360], Title = 'SIN(X)', $
    XTitle = 'degrees', YTitle = 'sin(x)'
    ; Plot the sine wave against the x values, change the range of
    ; the x-axis, and add labels. Notice that PV=WAVE rounds the
    ; requested range values on the axis to values that give a nice
    ; looking plot—in this case 0 to 400.

PLOT, x, y, XRange = [0, 360], XStyle = 1, $
    XTicks = 6, XMinor = 6, XTitle = $
    '!8degrees', YTitle = '!8sin(x)!3', $
    Title = '!17SIN(X)'
    ; Use keywords to get the desired style and to add fonts to
    ; the labels.

PLOTS, [0, 360], [0, 0]
    ; Plot a base line at 0.
```

```

TEK_COLOR
    ; Set up a predefined color table called tek_color.
POLYFILL, x(0:18), y(0:18), Color = 6
    ; Fill under the positive half of the curve with solid magenta.
z = COS(x * !Dtor)
    ; Calculate the cosine of x.
OPLOT, x, z, Linestyle = 2, Color = 3, Thick =4
    ; Plot the cosine using a thick, dashed line and a different color.

```

## Example 2

This example creates a logarithmic plot.

```

x = [100, 5000, 20000, 50000, 70000L]
y = [100, 10000, 1100000L, 100000L, 200000L]
    ; Create the data.
PLOT_OO, x, y, Ticklen = 0.5, Gridstyle = 1,$
    Tickformat = '(I7)', Title = 'TEST PLOT', $
    YRange = [1.e2, 1.e7], YStyle = 1
    ; Plot the data on a log-log plot, with dotted grid lines. Use a
    ; format of I7 for the plot labels.
max_y = MAX(y)
    ; Get the maximum value of y.
x_max_y = x(WHERE(y EQ max_y))
    ; Get the value for x when y is at its maximum.
XYOUTS, x_max_y(0), max_y+1.e5, 'Test Max', $
    Alignment = 0.5
    ; Print Test Max centered at the maximum point.

```

## Example 3

In this example, the cubic spline interpolant to the function

$$f(x) = 1000\sin(1/x^2) + \cos(10x)$$

over the interval [1, 21] is plotted using PLOT\_IO. This example uses the PV-WAVE:IMSL Mathematics function CSINTERP.

```

x = FINDGEN(101)/5 + 1
    ; Generate the abscissas.
y = 1000 * SIN(1/x^2) + COS(10 * x)
    ; Generate the function values.

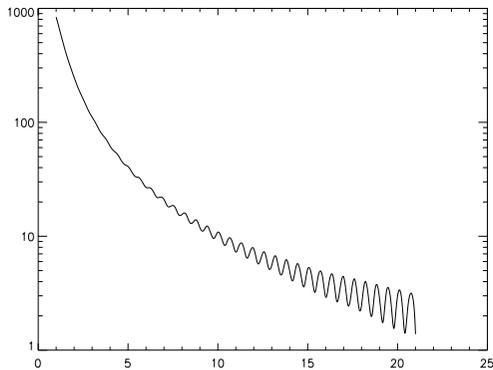
```

```

pp = CSINTERP(x, y)
; Compute the cubic spline interpolant.

ppval = SPVALUE(FINDGEN(1001)/50 + 1, pp)
PLOT_IO, FINDGEN(1001)/50 + 1, ppval
; Plot the result with logarithmic scaling on the y-axis (see
; ).

```



**Figure 2-38** A plot of the cubic spline interpolant of function  $f(x)$  using PLOT\_IO.

## Example 4

This example uses the PV-WAVE:IMSL Mathematics SPVALUE function.

```

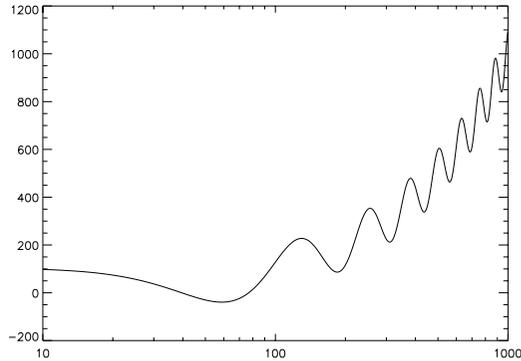
x = FINDGEN(100) * 10
; Generate the abscissas.

y = x + 100 * COS(0.05 * x)
; Generate the function values.

pp = CSINTERP(x, y)
; Compute the cubic spline interpolant.

ppval = SPVALUE(FINDGEN(1000), pp)
PLOT_OI, FINDGEN(1000), ppval, XRange = [10, 1000]
; Plot the result with logarithmic scaling on the x-axis
; (see ).

```



**Figure 2-39** Semi-logarithmic scaling of  $f(x)$  using PLOT\_OI.

## Example 5

This example creates a polar plot.

```
theta = (FINDGEN(200)/100) * !Pi
r = 2 * SIN(4 * theta)
      ; Create the data.

PLOT, r, theta, /Polar, XStyle=4, YStyle=4, $
      Title='POLAR PLOT TEST'
      ; Display a polar plot, disabling the box-style axes.

AXIS, 0, 0, XAxis=0
      ; Draw an x-axis at the point 0, 0 with tick marks going down.

AXIS, 0, 0, YAxis=0
      ; Draw a y-axis at the point 0, 0 with tick marks going left.
```

## Example 6

This example creates a plot with multiple axes.

```
temperature = [50., 40., 35., 60., 40.]
pressure = [1025, 1020, 1015, 1026, 1022]
      ; Create the data.

PLOT, temperature, YRange=[20., 70], $
      YTitle = 'Degrees Fahrenheit', $
      XTitle = 'Sample Number', $
      Title = 'Sample Data', XMargin = [8, 8], $
      YStyle = 8, Color = 16
      ; Plot temperature against scale on the left axis.

AXIS, YAxis=1, YRange=[1000, 1040], YStyle=1,$
```

```

        YTitle = 'Air Pressure', /Save, Color = 16
        ; Create the axis for air pressure.
OPlot, pressure, Linestyle = 2, Color = 6
        ; Plot air pressure against scale on the right axis.
LEGEND, ['temperature', 'air pressure'], $
        [16, 6], [0, 2], [0, 0], 2.4, 1005, 2
        ; Display a legend.

```

## Example 7

This example creates a plot with a Date/Time *x*-axis.

```

x = VAR_TO_DT(1992,1,1)
        ; Create a date-time array with the first day of 1992.
x = DTGEN(x, 12, /Month)
        ; Create an array of date lines with one value for each month.
y = RANDOMU(seed, 12) * 1000
        ; Create an array of data values ranging from 0 to 1000.

PRINT, !Month_Names
hold_month_names = !Month_Names
        ; Print the current names for months, and then save them.
FOR i = 0, 11 DO !Month_Names(i) = $
        STRMID(!Month_Names(i), 0, 3)
        ; Change the names to be 3-letter abbreviations for the months.

ylabels = STRARR(6)
FOR i = 0, 5 DO ylabels(i) = '$' + $
        STRTRIM(string(100L * i * 2), 2)
        ; Create labels on the y-axis that start with a '$'.

PLOT, x, y, /Month_Abbr, /Box, $
        Title='Test Date Plot', YRange=[0, 1000], $
        YStyle=1, YTickname=ylabels, YTicks=5, $
        YTitle='In thousands', YGridstyle=1, $
        YTicklen=0.5
        ; Plot the data.

!Month_Names = hold_month_names
        ; Restore !Month_names to the original.

```

## See Also

[AXIS](#), [OPlot](#), [OPlotERR](#), [PlotERR](#), [Plots](#), [XYOUTS](#)

For background information, see Chapter 4, *Displaying 2D Data*, in the *PV-WAVE User's Guide*.

---

## **PLOTERR Procedure**

Standard Library procedure that plots data points with accompanying symmetrical error bars.

### **Usage**

PLOTERR, [*x*,] *y*, *error*

### **Input Parameters**

*x* — (optional) A real vector containing the *x*-coordinates of the data to plot. If not present, *x* is assumed to be a vector of the same size as *y* and to have integer values beginning at 0 and continuing to the size of *y* – 1.

*y* — A real vector containing the *y*-coordinates of the data to plot.

*error* — A vector containing the error bar values of every point to be plotted.

### **Keywords**

*Psym* — The plotting symbol to use. If not specified, the default is 7 (the symbol “X”). *Psym* corresponds to the system variable !Psym. See [Chapter 3, Graphics and Plotting Keywords](#), for a complete description of the *Psym* graphics keyword.

*Type* — Specifies the type of plot to produce. Valid values are:

- 0 X Linear, Y Linear (the default)
- 1 X Linear, Y Log
- 2 X Log, Y Linear
- 3 X Log, Y Log

### **Discussion**

PLOTERR produces a plot of *y* versus *x*, with error bars drawn from *y* – *error* to *y* + *error*.

## Example

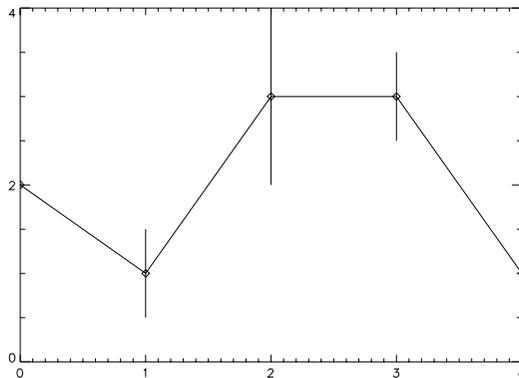
Assume that vector `y` contains the data values to be plotted, and that `error` is the vector containing the error bar values. The commands to plot the data points with accompanying symmetrical error bars are:

```
y = [2, 1, 3, 3, 1]
error = [0.0, 0.5, 1.0, 0.5, 0.0]
PLOTERR, y, error, Psym = 4
```

To overplot a line through the error bar, enter the following command:

```
OPLLOT, y
```

This produces the plot shown in :



**Figure 2-40** In this example, PLOTERR was first used to plot data points with their accompanying symmetrical error bars, and then OPLLOT was used to overplot a line through the error bar.

## See Also

[ERRPLOT](#), [OPLLOT](#), [OPLLOTERR](#), [PLOT](#)

---

## ***PLOT\_FIELD Procedure***

Standard Library procedure that plots a two-dimensional velocity field.

### **Usage**

`PLOT_FIELD`,  $u$ ,  $v$

### **Input Parameters**

$u$  — A 2D array giving the field vector at each point along the  $x$ -axis.

$v$  — A 2D array giving the field vector at each point along the  $y$ -axis.

### **Keywords**

***Aspect*** — The aspect ratio of the final plot; that is, the ratio of the length of  $x$ -axis to the length of the  $y$ -axis. (Default: 1)

***Length*** — The length of the longest field vector, expressed as a fraction of the plotting area. (Default: 0.1)

$N$  — The number of arrows to draw. (Default: 200)

***Title*** — A string containing the title of the plot. The default title is “Velocity Field”.

***YLabelCenter*** — Controls whether the top and bottom major tick labels on a  $Y$  axis will be positioned within the boundaries of the axis box or centered across from the corresponding major tick.

### **Discussion**

`PLOT_FIELD` picks  $N$  points at random and traces a path from each point along the field. The length of the path is proportional to *Length* and the field vector magnitude at that point.

---

**CAUTION** Extra care must be taken if you run the `PLOT_FIELD` and `VEL` procedures in the same `PV-WAVE` session. Each procedure calls a routine named `ARROWS`, but the `ARROWS` routines are slightly different. If you get an error in the `ARROWS` routine when you are using `PLOT_FIELD`, recompile `PLOT_FIELD` (by typing `.run PLOT_FIELD`), and then try again.

---

## Example

```
u = FLTARR(21, 21)
```

```
v = FLTARR(21, 21)
```

```
; Create the arrays.
```

```
.RUN
```

```
; After you type .RUN, the prompt WAVE> will change to a dash (-)
```

```
; to indicate that you may enter a complete program unit.
```

```
FOR j = 0, 20 DO BEGIN
```

```
FOR i = 0, 20 DO BEGIN
```

```
x = 0.05 * FLOAT(i)
```

```
z = 0.05 * FLOAT(j)
```

```
u(i, j) = -SIN(!Pi * x) * COS(!Pi * z)
```

```
v(i, j) = COS(!Pi * x) * SIN(!Pi * z)
```

```
ENDFOR
```

```
ENDFOR
```

```
END
```

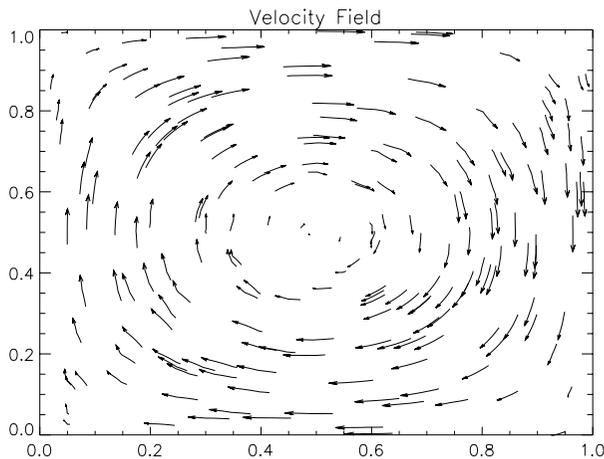
```
; This procedure puts values into the array. The last END exits you
```

```
; from programming mode, compiles and executes the procedure,
```

```
; and then returns you to the WAVE prompt.
```

```
PLOT_FIELD, u, v, Title = 'Velocity Field'
```

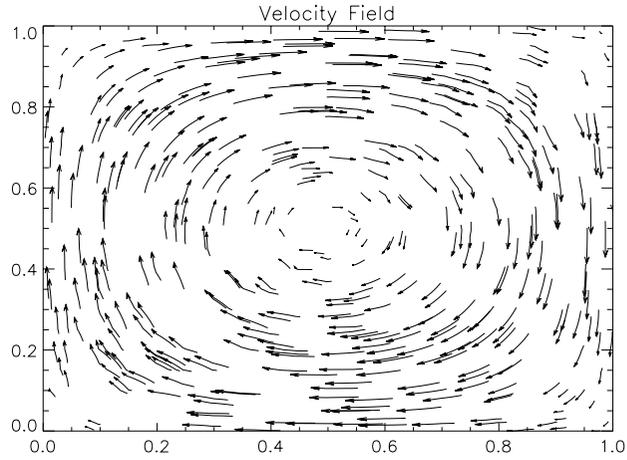
```
; Display the velocity field with default values ().
```



**Figure 2-41** Velocity field displayed with default values.

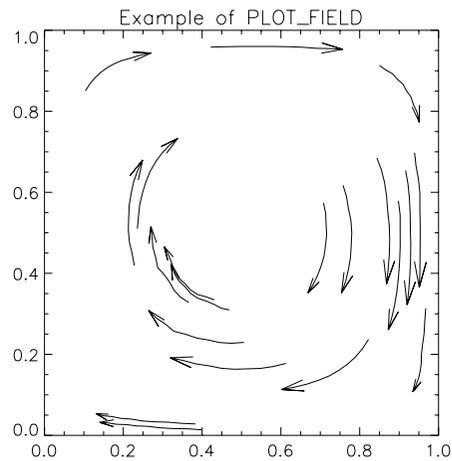
```
PLOT_FIELD, u, v, N = 400, Title = 'Velocity Field'
```

```
; Display the velocity field using 400 arrows (see ).
```



**Figure 2-42** Velocity field displayed with 400 arrows.

```
PLOT_FIELD, u, v, Aspect = .7, Length = .4, $
      N = 20, Title = 'Example of PLOT_FIELD'
      ; Display the velocity field with individual modifications (shown
      ; in ).
```



**Figure 2-43** Velocity field modified with various keywords.

## See Also

[VEL](#), [VELOVECT](#)

---

## ***PLOT\_HISTOGRAM Procedure***

Plots a histogram.

### **Usage**

`PLOT_HISTOGRAM`, *variable*

### **Input Parameters**

*variable* — A 1D array containing histogram data. The array cannot be of type complex, string, or structure.

### **Keywords**

*Axiscolor* — (integer) Specifies the index of the axis color.

*Binsize* — Specifies the width of the bins displayed in the histogram.

*Fillcolor* — (integer) Specifies the index of the color used to fill the histogram.

*Filled* — If present and nonzero, the histogram is filled with color.

*Noaxis* — If present and nonzero, no axes are drawn.

*Stepped* — If present and nonzero, the histogram is plotted as “steps” rather than as “bars”.

*Title* — A string containing a title for the histogram plot.

*Xmax* — The maximum value for which histogram data is plotted. Any data that falls above this value will be clipped.

*Xmin* — The minimum value for which histogram data is plotted. This corresponds to the leftmost point on the *x*-axis where the plot begins. By default, this minimum is set to zero. If there are negative values in your histogram data, you may need to adjust this value to shift the data to the left. Otherwise, the plot will start at the origin.

*Xverts* — A 1D array containing the *x*-vertices of the bottom of each bar in the histogram plot is returned. Two paired-elements are returned for each bar.

## Additional Keywords

Additional keywords let you control many aspects of a plot's appearance. Valid keywords for the PLOT\_HISTOGRAM procedure are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

Background	Noerase	[XY]Ticklen
Clip	Thick	[XY]Title
Color	[XY]Range	[XY]Type
Nodata	[XY]Style	

## Discussion

A histogram is a graph that allows you to visualize quantitative trends in large amounts of data. Histograms are different from bar charts because each “bar” in a histogram represents the results of a statistical sampling of the data, whereas each bar in a bar chart represents a discrete data element. In a histogram, data points do not map to “bars” on the graph one-to-one as they do in a bar chart.

Each “bar” in a histogram is called a *bin* and the width of each bin represents a range in the independent variable's values. The height of each bin represents the number of data points in the original variable that fall within the bin width — that is, that fall within the specified range of the independent variable.

This routine is used to render the graphics for the WzHistogram VDA Tool.

Data suitable for use as input to this procedure can be produced with the HISTOGRAM function. For example:

```
hist_data = HISTOGRAM(original_data)
PLOT_HISTOGRAM, hist_data
```

The vertices for the eighth bar in a graph ( $b=8$ ), can be obtained by using the following values returned by the *Xverts* keyword:

```
Xverts(i) and Xverts(i+1), where i = b*2
```

## Example

The following commands generate and plot histogram data. Keywords are used to color the histogram plot.

```
dist_data = DIST(20)
hist_data = HISTOGRAM(dist_data)
```

TEK\_COLOR

PLOT\_HISTOGRAM, hist\_data, /Filled, Color=3, Axiscolor=2

## See Also

[HISTOGRAM](#), [WzHistogram](#)

---

## ***PLOTS Procedure***

Plots vectors or points on the current graphics device in either two or three dimensions.

### **Usage**

PLOTS, *x* [, *y* [, *z*]]

### **Input Parameters**

*x* — A vector parameter providing the *x*-coordinates of the points to be connected.

If only one parameter is specified, *x* must be an array of either two or three vectors: (2,\*) or (3,\*). In this special case, *x*(0,\*) is taken as the *x* values, *x*(1,\*) is taken as the *y* values, and *x*(2,\*) is taken as the *z* values.

*y* — (optional) A vector parameter providing the *y*-coordinates of the points to be connected.

*z* — (optional) If present, a vector parameter providing the *z*-coordinates of the points to be connected. If *z* is not specified, *x* and *y* are used to draw lines in two dimensions. *z* has no effect if the keyword *T3d* is not specified and the system variable !P.T3d = 0.

### **Keywords**

Keywords let you control many aspects of the plot's appearance. PLOTS keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

<a href="#">Channel</a>	<a href="#">Fill_Pattern</a>	<a href="#">Orientation</a>	<a href="#">Spacing</a>
<a href="#">Clip</a>	<a href="#">Linestyle</a>	<a href="#">Pattern</a>	<a href="#">Symsize</a>

Color	Line_Fill	PClip	T3d
Data	Noclip	Psym	Thick
Device	Normal	Solid_Psym	Z

---

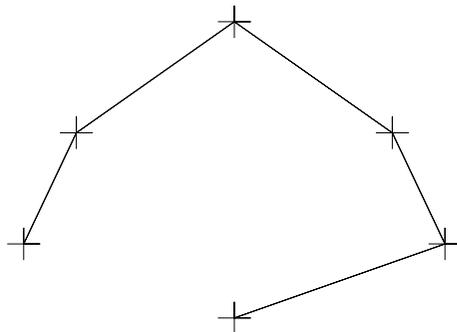
## Discussion

A valid data coordinate system must be established before PLOTS is called. (A call to PLOT can be used to establish this coordinate system.) Also note that a PV-WAVE window must be open and selected when the call to PLOTS is made for the procedure to work correctly.

The coordinates for PLOTS can be given in data, device, or normalized form using the *Data*, *Device*, or *Normal* keywords.

## Example 1

```
WINDOW
xdata = [.1, .2, .5, .8, .9, .5]
ydata = [.3, .6, .9, .6, .3, .1]
PLOTS, xdata, ydata
PLOTS, xdata, ydata, /Normal
PLOTS, xdata, ydata, Symsize=5.0, Psym=-1,$
      /Normal
```



**Figure 2-44** Connected lines drawn with PLOTS.

## Example 2

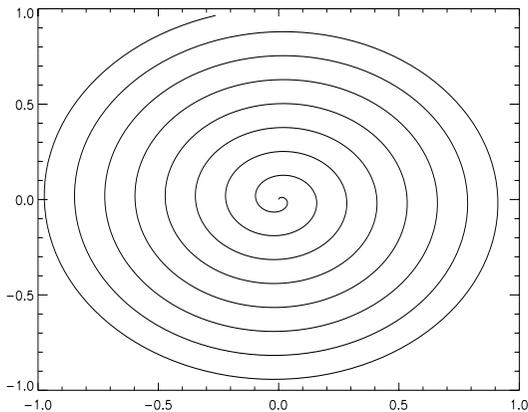
This example plots a curve in  $R^2$ . Notice that PLOT is first called to set up the two-dimensional coordinate system. Normally, PLOT would be called to perform the entire plot, but for demonstration purposes, PLOTS is used to complete the plot once the coordinate system has been defined.

```
PLOT, FINDGEN(2), /Nodata, $
    XRange = [-1, 1], YRange = [-1, 1]
    ; Set up the axes using PLOT.

p = FINDGEN(1000)/999
    ; Generate data.

x = p * SIN(50 * p)
y = p * COS(50 * p)

PLOTS, x, y
```



**Figure 2-45** Plot of curve in  $R^2$  using PLOTS .

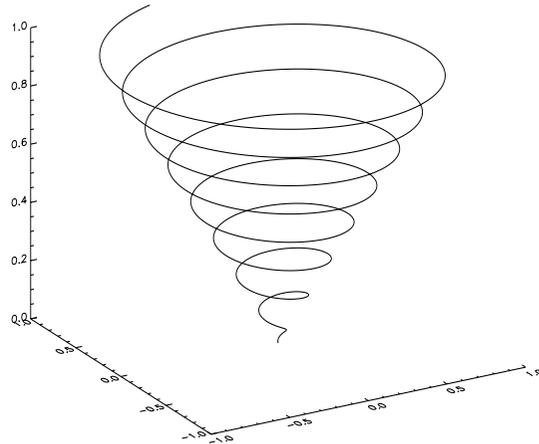
## Example 3

This example plot a curve in  $R^3$ . Notice that SURFACE is called initially to set up the three-dimensional coordinate system. It is also important to note that keywords *Nodata* and *Save* are used when calling SURFACE, and keyword *T3d* is used in the call to PLOTS.

```
SURFACE, FINDGEN(2, 2), /Nodata, /Save, $
    XRange = [-1, 1], YRange = [-1, 1], $
    ZRange = [0, 1]

z = FINDGEN(1000)/999
```

```
x = z * SIN(50 * z)
y = z * COS(50 * z)
PLOTS, x, y, z, /T3d
```



**Figure 2-46** Line plot in  $\mathbb{R}^3$ .

## See Also

[OPLOT](#), [PLOT](#)

For additional examples, see the section *Clipping PV-WAVE Graphics* in Chapter 4 of the *PV-WAVE User's Guide* and the section *Procedure Used to Draw a House* in Chapter 5 of the *PV-WAVE User's Guide*.

---

## PM Procedure

Performs formatted output of matrices to the standard output stream (logical file unit  $-1$ ).

### Usage

PM,  $expr_1$ , ...,  $expr_n$

### Input Parameters

$expr_i$  — Expression to be output.

## Keywords

**Format** — If not specified, PV-WAVE uses its default rules for formatting the output. Keyword *Format* allows the format to be specified in precise detail, using FORTRAN-style specifications.

**Title** — If present, specifies a character string to be used as the title of the output matrix. Only one title is printed, regardless of the number of expressions sent to PM.

## Discussion

Procedure PM formats output to the standard output stream of matrices stored in the PV-WAVE linear algebra matrix-storage mode. This procedure is designed to be used when working with matrices read by the procedures RM or RMF or other matrices using the PV-WAVE matrix-storage scheme.

By using keywords, the form of the output matrix can be customized. Keyword *Title* can be used to specify a character string to be used as a title. Keyword *Format* is provided to allow for explicitly formatted output.

## Example

In this example, a 2-by-3 matrix is read using the matrix-reading procedure RM and the results are printed using the matrix-printing procedure PM.

```
RM, a, 2, 3
    ; Read in a 2-by-3 matrix.
row 0: 11 22 33
row 1: 40 50 60

PM, a
    11.0000      22.0000      33.0000
    40.0000      50.0000      60.0000
    ; Output the matrix to standard output.
```

## See Also

[PMF](#), [RM](#), [RMF](#)

See [for more information](#).

---

## PMF Procedure

Performs formatted output of matrices to a specified file unit.

### Usage

PMF, *unit*, *expr*<sub>1</sub>, ..., *expr*<sub>n</sub>

### Input Parameters

*unit* — File unit to which the output is sent.

*expr*<sub>i</sub> — Expression to be output.

### Keywords

*Format* — If not specified, PV-WAVE uses its default rules for formatting the output. Keyword *Format* allows the format to be specified in precise detail, using FORTRAN-style specifications.

*Title* — Specifies a character string to be used as the title of the output matrix. Only one title is printed, regardless of the number of expressions sent to PMF.

### Discussion

Procedure PMF formats output to a specified file unit of matrices stored in the PV-WAVE linear algebra matrix-storage mode. This procedure is designed to be used when working with matrices read by the procedures RM or RMF or other matrices using the PV-WAVE matrix-storage scheme.

Using keywords, the form of the output matrix can be customized. Keyword *Title* can be used to specify a character string to be used as a title. Keyword *Format* is provided to allow for explicitly formatted output.

### Example

This example reads a 2-by-3 matrix using the matrix-reading procedure RM and prints the results to standard output, (*unit* = -1), using the matrix-printing procedure PMF.

```
RM, a, 2, 3
      ; Read matrix.
row 0: 11 22 33
```

```
row 1: 40 50 60
PMF, -1, a
    11.0000    22.0000    33.0000
    40.0000    50.0000    60.0000
; Output the matrix to standard output.
```

## See Also

[PM](#), [RM](#), [RMF](#)

See [for more information](#).

---

## ***POINT\_LUN Procedure***

Allows the current position of the specified file to be set to any arbitrary point in the file.

### **Usage**

`POINT_LUN, unit, position`

### **Input Parameters**

***unit*** — The file unit (logical unit number) for which the file position will be set. This keyword can be set to either *unit* or *-unit*. If *-unit* is specified, the current position of the file pointer is returned in the output parameter *position*.

***position*** — A positive integer specifying the position of the file pointer as a byte offset from the start of the file.

### **Output Parameters**

***position*** — If *-unit* is specified, the current position of the file pointer (in bytes) is returned in the *position* parameter. See the *Example* section.

### **Keywords**

None.

## Discussion

POINT\_LUN is for the PV-WAVE programmer who wants explicit control over positioning for reading or writing within a given file. It is seldom used for general file I/O operations.

---

**OpenVMS USERS** To use POINT\_LUN to specify a byte offset in an OpenVMS file, you must use the *Block* keyword in the OPEN procedure when you open the file. For example:

```
OPENR, 1, 'File.dat', /Block
POINT_LUN, 1, 25B
```

---

## Example

In this example, POINT\_LUN is used to move about within an unformatted file of integers.

```
a = INDGEN(100)
    ; Create an integer vector of length 100 that is initialized to the values
    ; of its one-dimensional subscripts.

OPENW, unit, 'ptlun.dat', /Get_Lun
    ; Open a file called ptlun.dat for writing.

WRITEU, unit, a
    ; Write the 100-element integer vector as unformatted binary data to
    ; the file ptlun.dat.

POINT_LUN, -unit, pos
PRINT, 'Current offset into ptlun.dat is', $
    pos, ' bytes.'
    Current offset into ptlun.dat is 200 bytes.
    ; Retrieve and display the current position within ptlun.dat.

POINT_LUN, unit, 0
    ; Rewind the file to the beginning.

b = INTARR(2)
    ; Read two integers from the beginning of the file into a two-element
    ; integer array, b.

READU, unit, b
PRINT, b
    0      1

POINT_LUN, -unit, pos
    ; Since two integers were just read, each of length 2 bytes, the
```

```
    ; current position within the file should be 4 bytes offset from
    ; beginning. Retrieve current position.

PRINT, 'Current offset into ptlun.dat is', pos, ' bytes.'
    Current offset into ptlun.dat is 4 bytes.

FREE_LUN, unit
    ; Close the file and free the file unit number.
```

## See Also

[FREE\\_LUN](#), [FSTAT](#), [GET\\_LUN](#), [OPEN \(UNIX/OpenVMS\)](#),  
[OPEN \(Windows\)](#), [READ](#), [WRITEU](#)

For more information, see the section *Positioning File Pointers* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

---

## ***POLY* Function**

Standard Library function that evaluates a polynomial function of a variable.

### **Usage**

*result* = POLY(*x*, *coefficients*)

### **Input Parameters**

*x* — The variable that is evaluated. May be a scalar or array.

*coefficients* — A vector containing one more element than the degree of the polynomial function. These elements are the coefficients of the polynomial equation that is used by POLY.

### **Returned Value**

*result* — The values calculated from the polynomial function of *x*.

### **Keywords**

None.

## Discussion

POLY evaluates a polynomial function of a variable according to the formula:

$$result = c_0 + c_1x + c_2x^2 + \dots + c_{n-1}x^{n-1}$$

where  $n$  is the dimension of the polynomial  $c$ , and  $c_0$  through  $c_{n-1}$  are the *coefficients*.

POLY returns a variable with the same dimensions as  $x$ . Each element of the result is equal to the computed value of  $c_0 + c_1x + c_2x^2 + c_ix^i$  for each element of  $x$ .

The POLY function can be used in conjunction with POLY\_FIT and POLYFITW, which both return the coefficients of a polynomial function fitted through data.

## Example

```
x = 2
c = [1, 2, 3]
func = POLY(x, c)
PRINT, func
      17
```

## See Also

[POLY\\_2D](#), [POLY\\_FIT](#), [POLYFITW](#)

---

## ***POLY\_2D Function***

Performs polynomial warping of images.

### Usage

```
result = POLY_2D(array, coeffx, coeffy [, interp [, dimx, ..., dimy]])
```

### Input Parameters

**array** — The array to be processed. Must be two-dimensional. Can be of any basic type except string.

*coeff<sub>x</sub>* — The  $x$  coefficients.

*coeff<sub>y</sub>* — The  $y$  coefficients.

*interp* — (optional) If present and nonzero, specifies that the bilinear interpolation method is to be used in the resampling. Otherwise, the nearest neighbor method is used.

*dim<sub>x</sub>* — (optional) If present, specifies the number of columns in the resulting array. Otherwise, the output has the same number of columns as *array*.

*dim<sub>y</sub>* — (optional) If present, specifies the number of rows in the resulting array. Otherwise, the output has the same number of rows as *array*.

## Returned Value

*result* — A two-dimensional array containing the warped image. It is of the same data type as *array*.

## Keywords

*Missing* — Specifies the output value for points whose  $x'$ ,  $y'$  is outside the bounds of *array*. (If this keyword is not used, these values are extrapolated from the nearest pixel of *array*.)

## Discussion

POLY\_2D performs a geometrical transformation in which the resulting array is defined by:

$$g(x,y) = f(x', y') = f[a(x,y), b(x,y)]$$

where  $g(x,y)$  represents the pixel in the output image at coordinate  $(x, y)$ , and  $f(x', y')$  is the pixel at  $(x', y')$  in the input image that is used to derive  $g(x, y)$ .

The functions  $a(x, y)$  and  $b(x, y)$  are polynomials in  $x$  and  $y$  of degree  $n$ , and specify the spatial transformation:

$$x' = a(x, y) = \sum_{i=0}^n \sum_{j=0}^n \text{coeff}_{x_i} x^i y^j$$

$$y' = b(x, y) = \sum_{i=0}^n \sum_{j=0}^n \text{coeff}_{y,i,j} x^j y^i$$

where  $n$  is the degree of the polynomials that are being used to produce the warping, and  $\text{coeff}_x$  and  $\text{coeff}_y$  are arrays containing the polynomial coefficient. Each array must contain  $(n + 1)^2$  elements.

For example, for a linear transformation,  $\text{coeff}_x$  and  $\text{coeff}_y$  must contain four elements and may be either a 2-by-2 array or a 4-element vector.  $C_{x,i,j}$  contains the coefficient used to determine  $x'$ , and is the weight of the term  $x^j y^i$ .

The nearest neighbor interpolation method is not linear, because new values that are needed are merely set equal to the nearest existing value of *image*. For more information on bilinear interpolation, see the BILINEAR function.

Bilinear interpolation can be time-consuming. For example, it requires about twice as much time as does the nearest neighbor method, even if you are working with a linear case (in other words, if  $n$  equals 1).

---

**TIP** The POLYWARP function may be used to fit  $(x', y')$  as a function of  $(x, y)$  and return the coefficient arrays  $C_x$  and  $C_y$ .

---

The POLY\_2D function supports multi-layer band interleaved images. When the input array is three-dimensional, it is automatically treated as an array of images,  $\text{array}(m, n, p)$ , where  $p$  is the number of  $m$  by  $n$  images. Each image is then operated on separately and an array of the result images is returned.

## Example

Some simple linear (degree one) transformations are shown in the following table:

$P_{0,0}$	$P_{1,0}$	$P_{0,1}$	$P_{1,1}$	$Q_{0,0}$	$Q_{1,0}$	$Q_{0,1}$	$Q_{1,1}$	Effect
0	0	1	0	0	1	0	0	Identity
0	0	0.5	0	0	1	0	0	Stretch x by a factor of 2
0	0	1	0	0	2.0	0	0	Shrink y by a factor of 2
z	0	0	0	0	0	0	0	Shift left by z pixels

$P_{0,0}$	$P_{1,0}$	$P_{0,1}$	$P_{1,1}$	$Q_{0,0}$	$Q_{1,0}$	$Q_{0,1}$	$Q_{1,1}$	Effect
0	1	0	0	0	0	1	0	Transpose

## See Also

[BILINEAR](#), [POLY](#), [POLY\\_FIT](#), [POLYWARP](#)

For details on interpolation methods. see .

---

## ***POLY\_AREA Function***

Standard Library function that returns the area of an  $n$ -sided polygon, given the vertices of the polygon.

### Usage

*result* = POLY\_AREA( $x$ ,  $y$ )

### Input Parameters

$x$  — An  $n$ -element real vector containing the  $x$ -coordinates of each vertex in the polygon.

$y$  — An  $n$ -element real vector containing the  $y$ -coordinates of each vertex in the polygon.

### Returned Value

*result* — The area of the polygon, returned as a floating-point value.

### Keywords

None.

### Discussion

POLY\_AREA assumes that the input polygon has  $n$  vertices with  $n$  sides, and that the edges connect the vertices in the order  $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n), (x_1, y_1)]$ . The last vertex must be connected to the first.

## Example 1

```
x = [1, 3, 2]
y = [1, 1, 4]
area = POLY_AREA(x, y)
PRINT, area
      3.00000
```

## Example 2

```
x = [2, 4, 4, 2]
y = [1, 1, 2, 2]
PRINT, POLY_AREA(x, y)
      2.00000
```

## See Also

[POLYFILL](#), [POLYFILLV](#), [POLYSHADE](#)

---

## ***POLY\_C\_CONV Function***

Returns a list of colors for each polygon, given a polygon list and a list of colors for each vertex.

### **Usage**

```
result = POLY_C_CONV(polygon_list, colors)
```

### **Input Parameters**

*polygon\_list* — An array containing a list of polygons. For more information, see the section *Vertex Lists and Polygon Lists* in Chapter 7 of the *PV-WAVE User's Guide*.

*colors* — A vertex-based color list.

## Returned Value

**result** — An integer vector containing the list of colors, one color for each input polygon. (In other words, the result contains the same number of colors as the number of polygons in *polygon\_list*.)

## Keywords

None.

## Discussion

POLY\_C\_CONV is useful for converting a vertex-based color list to a polygon-based list. For example, the SHADE\_VOLUME procedure returns a list of polygon colors when the *Shades* keyword is used. This list has one color per vertex. The POLY\_PLOT procedure, however, requires a color list with one color per polygon.

## Example

This example program displays data warped onto an irregular sphere.

```
PRO sphere_demo2

radii = RANDOMU(s, 60, 60)
radii = SMOOTH((radii + 1.0), 2)
POLY_SPHERE, radii, 60, 60, vertex_list, $
    polygon_list
    ; Define the sphere as a list of polygons.

p_colors = BYTSCL(DIST(60), Top=127)
    ; Define the shading colors for the sphere.

WINDOW, 0, Colors=128, XPos=16, YPos=384
LOADCT, 1
CENTER_VIEW, Xr=[-2.0, 2.0], $
    Yr=[-2.0, 2.0], Zr=[-2.0, 2.0], $
    Ax=(-75.0), Az=(-90.0), Zoom=0.99
    ; Set up the viewing window and load the color table.

TVSCL, POLYSHADE(vertex_list, polygon_list, $
    /T3d, Shade=p_colors)
    ; Construct a shaded surface representation of the data and
    ; display it using POLYSHADE.

WINDOW, 1, Colors = 128, XPos = 256, YPos = 64
    ; Create a new window for a new plot.

pg_num = POLY_COUNT(polygon_list)
```

```

; Count the number of polygons in the sphere.
vertex_list = POLY_NORM(vertex_list)
vertex_list = POLY_TRANS(vertex_list, !P.T)
vertex_list = POLY_DEV(vertex_list, 640, 512)
; Transform the polygon vertices from data coordinates to device
; coordinates.

p_colors = POLY_C_CONV(polygon_list, p_colors(*))
; Convert the colors from a vertex-based list to a polygon-based
; list.

POLY_PLOT, vertex_list, polygon_list, $
pg_num, 640, 512, p_colors, 0, -1
; Plot the sphere using POLY_PLOT.

END

```

For another POLY\_C\_CONV example, see the `poly_demo1` demonstration program in:

```

(UNIX)      <wavedir>/demo/ar1
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows) <wavedir>\demo\ar1

```

Where <wavedir> is the main PV-WAVE directory.

## See Also

[POLY\\_PLOT](#), [POLYSHADE](#), [SHADE\\_VOLUME](#)

## ***POLY\_COUNT Function***

Returns the total number of polygons contained in a polygon list.

### **Usage**

```
result = POLY_COUNT(polygon_list)
```

### **Input Parameters**

*polygon\_list* — An array containing a list of polygons. For more information, see the section *Vertex Lists and Polygon Lists* in Chapter 7 of the *PV-WAVE User's Guide*.

## Returned Value

*result* — The total number of polygons contained in the specified polygon list.

## Keywords

None.

## Discussion

The value returned by POLY\_COUNT is suitable for input as the *pg\_num* parameter used in the POLY\_PLOT procedure.

## Example

See the [Example](#) section in the description of the POLY\_C\_CONV routine.

For another example, see the `sphere_demo3` demonstration program in:

(UNIX) `<wavedir>/demo/ar1`

(OpenVMS) `<wavedir>:[DEMO.ARL]`

(Windows) `<wavedir>\demo\ar1`

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[POLY\\_PLOT](#)

---

## ***POLY\_DEV Function***

Returns a list of 3D points converted from normal coordinates to device coordinates.

### **Usage**

*result* = POLY\_DEV(*points* [,*winx*, *winy*])

### **Input Parameters**

*points* — A (3, *n*) array of points (or vertices) to transform.

*winx*, *winy* — (optional) The maximum *x* and *y* dimension, respectively, in device coordinates. If these parameters are omitted, the values of !D.X\_Size and !D.Y\_Size are used.

## Returned Value

*result* — The list of 3D points that has been converted from normal coordinates to device coordinates. The list is in long integer format.

## Keywords

None.

## Example

The program in this example displays a perspective view of a surface from a view-point within the data.

```
PRO poly_demo1
winx = 1000
winy = 750
    ; Specify the window size.
imgx = 477
imgy = 512
    ; Specify the image size.
elev_dat = BYTARR(imgx, imgy)
OPENR, 1, !Data_Dir + 'bldr_elev.dat'
READU, 1, elev_dat
CLOSE, 1
    ; Read in the elevation image data.
landsat = BYTARR(imgx, imgy)
OPENR, 1, !Data_Dir + 'bldr_img7.dat'
READU, 1, landsat
CLOSE, 1
    ; Read in the Landsat image data.
imgx = 120
imgy = 125
elev_dat = CONGRID(FLOAT(elev_dat), imgx, $
    imgy, /Interp)
    ; Shrink the elevation data to a 120-by-125 array.
landsat = BYTSCL(CONGRID(FLOAT(landsat), $
```

```

        imgx, imgy, /Interp), Top = 127)
        ; Shrink the Landsat image to a 120-by-125 array and scale the
        ; image into the range of 0 – 127.
zscale = 0.08
        ; Define the Z compression factor.

viewpoint = [105.0, 70.0, (5.0 * zscale)]
viewvector = [-10.0, -2.5, -0.75]
perspective = 0.06
izoom = 11.0
viewup = [0.0, 1.0]
viewcenter = [0.5, 0.5]
xr = [0, (imgx - 1)]
yr = [0, (imgy - 1)]
zr = [MIN(elev_dat), MAX(elev_dat)]
        ; Define the view parameters.

elev_dat = elev_dat * zscale
        ; Compress the elevation data.

SET_VIEW3D, viewpoint, viewvector, $
        perspective, izoom, viewup, $
        viewcenter, winx, winy, xr, yr, zr
        ; Set up a 3D view based on eye-point and view vector.

PRINT, "Building polygons ..."
POLY_SURF, elev_dat, vertex_list, $
        polygon_list, pg_num
        ; Generate the polygons representing the surface.

vertex_list = vertex_list
PRINT, "Normalizing coordinates ..."
vertex_list = POLY_NORM(vertex_list)
        ; Convert the polygon vertices from data coordinates to normal
        ; coordinates.

PRINT, "Transforming coordinates ..."
vertex_list = POLY_TRANS(vertex_list, !P.T)
        ; Transform the new coordinates.

PRINT, "Changing to device coordinates ..."
vertex_list = POLY_DEV(vertex_list, winx, $
        winy)
        ; Convert the normal coordinates to device coordinates.

WINDOW, XSize=winx, YSize=winy, XPos=10, $
        YPos=50, Colors=128
        ; Set up a new window for plotting.

PRINT, "Plotting ..."

```

```

landsat = POLY_C_CONV(polygon_list, landsat)
      ; Create an array containing one color for each polygon.
POLY_PLOT, vertex_list, polygon_list, $
      pg_num, winx, winy, landsat, -1, -1
      ; Plot the surface.

END

```

For other examples of POLY\_DEV, see the `sphere_demo2`, `sphere_demo3`, and `vol_demo4` demonstration programs in:

```

(UNIX)      <wavedir>/demo/ar1
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows)  <wavedir>\demo\ar1

```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[POLY\\_NORM](#), [POLY\\_TRANS](#)

For more information, see the section *Coordinate Conversion* in Chapter 7 of the *PV-WAVE User's Guide*, and the section *Three Graphics Coordinate Systems* in Chapter 4 of the *PV-WAVE User's Guide*.

## ***POLYFILL Procedure***

Fills the interior of a region of the display enclosed by an arbitrary 2D or 3D polygon.

### **Usage**

POLYFILL,  $x$  [,  $y$  [,  $z$ ]]

### **Input Parameters**

$x$  — A vector parameter providing the  $x$ -coordinates of the points to be connected.

If only one parameter is specified,  $x$  must be an array of either two or three vectors: (2,\*) or (3,\*). In this special case,  $x(0,*)$  is taken as the  $x$  values,  $x(1,*)$  is taken as the  $y$  values, and  $x(2,*)$  is taken as the  $z$  values.

$y$  — (optional) A vector parameter providing the  $y$ -coordinates of the points to be connected.

*z* — (optional) If present, a vector parameter providing the *z*-coordinates of the points to be connected. If *z* is not present, *x* and *y* are used to draw lines in two dimensions. *z* has no effect if the keyword *T3d* is not specified and the system variable !P.T3d = 0.

## Keywords

Keywords let you control many aspects of the plot's appearance. POLYFILL keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

<a href="#">Channel</a>	<a href="#">Fill_Pattern</a>	<a href="#">Normal</a>	<a href="#">Spacing</a>
<a href="#">Clip</a>	<a href="#">Linestyle</a>	<a href="#">Pattern</a>	<a href="#">Symsize</a>
<a href="#">Color</a>	<a href="#">Line_Fill</a>	<a href="#">PClip</a>	<a href="#">T3d</a>
<a href="#">Data</a>	<a href="#">Noclip</a>	<a href="#">Psym</a>	<a href="#">Thick</a>
<a href="#">Device</a>	<a href="#">Orientation</a>		

---

## Z-buffer Specific Keywords

These keywords allow you to warp images over 2D or 3D polygons; the keywords are valid only when the Z-buffer device is active. For more information on the Z-buffer, see [Appendix B, Output Devices and Window Systems](#).

***Image\_Coordinates*** — To warp an image over a polygon, pass the image into POLYFILL with the *Pattern* keyword, and specify a (2, *n*) array containing the image space coordinates that correspond to each of the *n* vertices with the *Image\_Coordinates* keyword.

***Image\_Interpolate*** — When present and nonzero, specifies that bilinear interpolation is used instead of the nearest-neighbor method of sampling.

***Mip*** — When present and nonzero, produces improved transparency by Maximum Intensity Projection. Rather than setting an arbitrary threshold value, a pixel is set in the Z-buffer, regardless of its depth, if its intensity is greater than the current pixel in the buffer.

***Threshold*** — Pixels less than the *Threshold* value are not drawn, producing a transparent effect.

## Discussion

The polygon is defined by a list of connected vertices stored in *x*, *y*, and *z*. The coordinates can be given in data, device, or normalized form using the *Data*, *Device*, or *Normal* keywords.

POLYFILL uses various filling methods:

- solid fill
- parallel lines
- a pattern contained in an array
- hardware-dependent fill pattern

**Solid Fill Method** — Most devices can fill with a solid color. Solid fill is performed using the line fill method for devices that don't have this hardware capability. Keywords that specify a method are not required for solid filling.

**Line Fill Method** — Filling using parallel lines is device independent and works on all devices that can draw lines. Cross-hatching may be obtained with multiple fillings of differing orientations. The spacing, linestyle, orientation, and thickness of the filling lines may be specified using the corresponding keywords. The *Line\_Fill* keyword selects this filling style, but is not required if either the *Orientation* or *Spacing* keywords are present.

**Patterned Fill Method** — The method of patterned filling and the usage of various fill patterns is hardware dependent. The fill pattern array may be directly specified with the *Pattern* keyword for some output devices. If this keyword is omitted, the polygon is filled with the hardware-dependent pattern index specified by the *Fill\_Pattern* keyword.

## Example 1

In this example, POLYFILL is used to create and fill a square, triangle, and pentagon with different patterns. Device coordinates are used for these polygons.

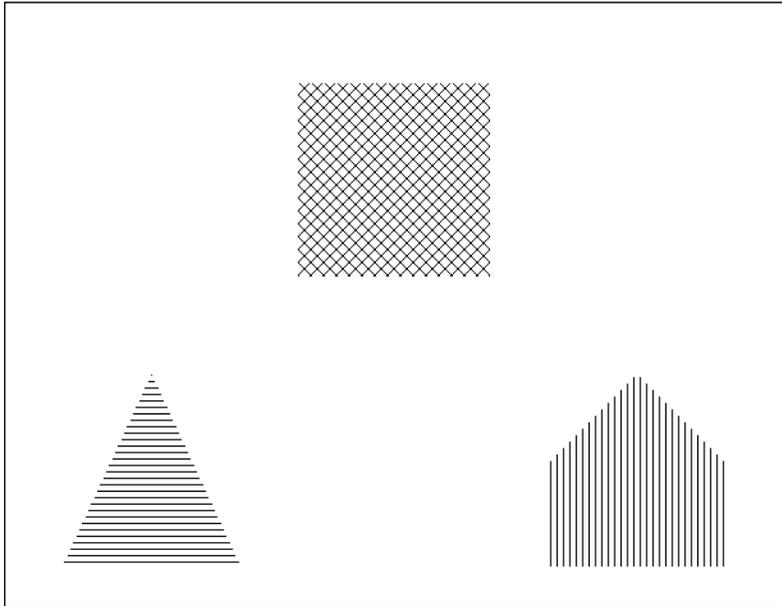
```
a = INTARR(10, 10)
    ; Create fill pattern for square. This will be an X pattern.
FOR i = 0, 9 DO a(i, i) = !D.N_Colors - 1
a = a + ROTATE(a, 1)
POLYFILL, [225, 375, 375, 225], $
    [275, 275, 425, 425], /Device, Pattern = a
    ; Create square and fill it with the pattern in variable a
    ; ()
b = INTARR(10, 10)
```

```

; Create fill pattern (horizontal lines) for triangle.
b(*, 2) = !D.N_Colors - 1
b(*, 7) = !D.N_Colors - 1
POLYFILL, [40, 180, 110], [50, 50, 200], $
    /Device, Pattern = b
    ; Create triangle and fill it with the pattern in variable b
    ; ().

c = INTARR(10, 10)
; Create fill pattern (vertical lines) for pentagon.
c(2, *) = !D.N_Colors - 1
c(7, *) = !D.N_Colors - 1
POLYFILL, [420, 560, 560, 490, 420], $
    [50, 50, 130, 200, 130], /Device, $
    Pattern = c
    ; Create pentagon and fill it with the pattern in variable c
    ; ().

```



**Figure 2-47** Pattern-filled polygons.

## Example 2

This example uses POLYFILL to draw the faces of cubes that are stacked on top of each other in pyramid fashion. The width and length of the base of the stack is equal to *numcubes*. This value is an argument passed to the procedure that draws the cubes.

The SURFACE procedure is used to establish a three-dimensional transformation matrix that determines the view. The *T3d* keyword is used with POLYFILL so that the transformation matrix established by SURFACE is used.

```
PRO cubes, numcubes
    ; Argument numcubes is the number of cubes to place along the base.

COMMON com1, size, colr1, colr2, colr3

size = 1.0D/numcubes
    ; Determine size, which is the width, height, and length of each cube.
    ; Note that everything is normalized to lie between 0 and 1.

LOADCT, 3
    ; Load red temperature color table.

SURFACE, FLTARR(2, 2), /Nodata, XStyle = 4, $
    YStyle = 4, ZStyle = 4, XRange = [0, 1], $
    YRange = [-1, 0], ZRange = [0, 1], /Save
    ; Establish three-dimensional transformation using SURFACE.
    ; The Nodata keyword permits the use of a dummy
    ; two-dimensional array to be passed to SURFACE. Setting
    ; XStyle, YStyle, and ZStyle to 4 causes the axes to be invisible.
    ; The ranges are set here, and the transformation is saved.

colr1 = FIX(!D.N_Colors/2.0)
colr2 = (FIX((!D.N_Colors - colr1)/2.0) + $
    colr1) MOD !D.N_Colors
colr3 = (FIX((!D.N_Colors - colr2)/2.0) + $
    colr1) MOD !D.N_Colors
    ; Determine available colors to use.

z = 0

FOR i = FIX(numcubes), 1, -1 DO BEGIN
    x = (FIX(numcubes) - i) * size
    y = -size
    z = z + size
    ; Draw the cubes by layer, starting at the bottom.
    FOR j = 1, i - 1 DO BEGIN
        draw_cube, x, y, z
    ; Draw the cubes along the left edge of the current layer.
```

```

        y = y - size
    ENDFOR
    x = (i - 1) * size + x
    FOR j = 1, i DO BEGIN
        DRAW_CUBE, x, y, z
        x = x - size
        ; Draw the cubes along the right edge of the current layer.
    ENDFOR
ENDFOR
END

PRO draw_cube, x, y, z
; Draw a cube.

COMMON com1, size, colr1, colr2, colr3
left_face, x, y, z, colr1
right_face, x, y, z, colr2
top_face, x, y, z, colr3
END

PRO left_face, x, y, z, colr
COMMON com1, size
POLYFILL, [x, x, x, x], [y, y, y + size, $
    y + size], [z, z - size, z - size, z], $
/T3d, Color = colr
; Use POLYFILL to draw the left face of a cube.

END

PRO right_face, x, y, z, colr
COMMON com1, size
POLYFILL, [x, x + size, x + size, x], $
    [y, y, y, y], [z, z, z - size, z - size], $
/T3d, Color = colr
; Use POLYFILL to draw the right face of a cube.

END

PRO top_face, x, y, z, colr
COMMON com1, size
POLYFILL, [x, x + size, x + size, x], $
    [y, y, y + size, y + size], [z, z, z, z], $
/T3d, Color = colr
; Use POLYFILL to draw the top face of a cube.

END

```

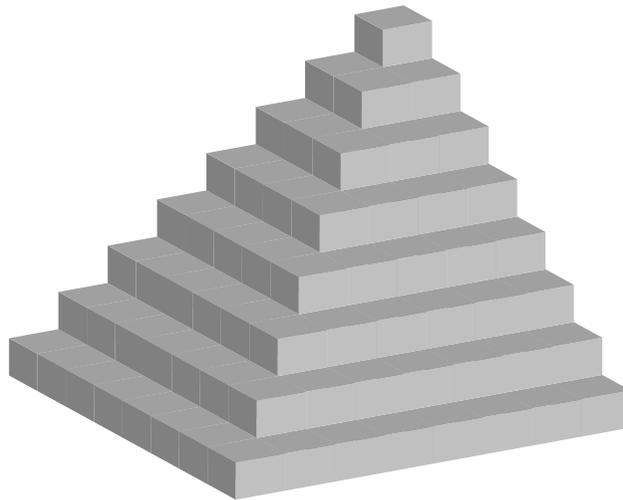
If this procedure is contained in a file named `cubes.pro` in your directory, it can be compiled with the following command:

```
.RUN cubes
```

The number of levels in the pyramid height is equal to the number passed to `cubes`. If the procedure is run with the command:

```
cubes, 8
```

then a pyramid with eight levels is created (see ).



**Figure 2-48** Example of three-dimensional polygon filling.

### See Also

[POLY\\_AREA](#), [CONTOUR2](#)

---

## ***POLYFILLV Function***

Returns a vector containing the subscripts of the array elements contained inside a specified polygon.

### **Usage**

```
result = POLYFILLV(x, y, sx, sy [, run_length])
```

### **Input Parameters**

*x* — A vector containing the *x* subscripts of the vertices that define the polygon.

*y* — A vector containing the *y* subscripts of the vertices that define the polygon.

*sx* — The number of columns in the array surrounding the polygon.

*sy* — The number of rows in the array surrounding the polygon.

*run\_length* — (optional) If present and nonzero, returns a vector of run lengths, rather than subscripts. Each element with an even subscript result contains the length of the run, and the following element contains the starting index of the run.

For large polygons, using *run\_length* can save considerable space.

### **Returned Value**

*result* — A vector containing the subscripts of the array elements contained inside a polygon defined by *x* and *y*.

### **Keywords**

None.

### **Discussion**

POLYFILLV is useful in defining, analyzing, and displaying regions of interest within a two-dimensional array.

The *x* and *y* parameters are vectors containing the subscripts of the vertices that define the polygon in the coordinate system of the two-dimensional *sx*-by-*sy* array.

The *sx* and *sy* parameters define the number of columns and rows in the array enclosing the polygon. At least three points must be specified, and all points should lie within the limits:

$$0 \leq x_i < sx \quad \text{and} \quad 0 \leq y_i < sy \quad \text{for all } i$$

The polygon is defined by connecting each point with its successor and the last point with the first.

## Example

This example determines the mean and standard deviation of the elements within a triangular region defined by the vertices at pixel coordinates (100, 100), (200, 300), and (300, 100), inside a 512-by-512 array called `data`.

```
x = [100, 200, 300]
y = [100, 300, 100]
    ; Define triangle's coordinates.

p = data(POLYFILLV(x, y, 512, 512))
    ; Get the subscripts of the elements in the polygon.

std = STDEV(p, mean)
    ; Use the STDEV function to obtain the mean and standard deviation
    ; of the selected elements.
```

## See Also

[POLY\\_AREA](#)

---

## ***POLY\_FIT Function***

Standard Library function that fits an *n*-degree polynomial curve through a set of data points using the least-squares method.

### Usage

```
result = POLY_FIT(x, y, deg [, yft, ybd, sig, mat])
```

### Input Parameters

*x* — The vector containing the independent *x*-coordinates of the data.

*y* — The vector containing the dependent y-coordinates of the data. Must have the same number of elements as *x*.

*deg* — The degree of the polynomial to be fitted to the data.

## Output Parameters

*yft* — (optional) The vector containing the calculated y values.

*ybd* — (optional) The vector containing the error estimate of each point. (The error estimate is equal to one standard deviation.)

*sig* — (optional) The standard deviation of the function, expressed in the units of the y direction.

*mat* — (optional) The correlation matrix of the coefficients.

## Returned Value

*result* — The vector containing the coefficients of the polynomial equation which best approximates the data.

## Keywords

None.

## Discussion

POLY\_FIT uses a least-squares method for determining the equation of the curve, which minimizes the error at each point of the curve. This function is useful for showing the relationship between two variables.

POLY\_FIT returns a vector with a length of *deg* + 1. For example, if you had requested a polynomial of degree 3, the fitted curve would have the equation:

$$f(x) = \text{result}(3)x^3 + \text{result}(2)x^2 + \text{result}(1)x + \text{result}(0)$$

The *yft* parameter is in a format that can be readily displayed as a curve alongside the input curve, thereby allowing you to compare the two curves.

## Example

```
x = FINDGEN(9)
y = [5., 4., 3., 2., 2., 3., 5., 6., 7.]
; Create the data.
```

```

TEK_COLOR
    ; Load a color table.

PLOT, x, y, Title='POLY_FIT EXAMPLE'
    ; Plot the data.

coeff_1_deg = POLY_FIT(x, y, 1, yfit)
    ; Fit with a first-order polynomial.

OPLOT, x, yfit, Color=3
    ; Overplot the calculated values on the original plot.

coeff_3_deg = POLY_FIT(x, y, 3, yfit)
    ; Fit with a third-order polynomial.

OPLOT, x, yfit, Color=2
    ; Overplot the calculated values on the original plot.

coeff_5_deg = POLY_FIT(x, y, 5, yfit)
    ; Fit with a fifth-order polynomial.

OPLOT, x, yfit, Color=6
    ; Overplot the calculated values on the original plot.

labels = ['Original data', $
        'Fit with first order polynomial', $
        'Fit with third order polynomial', $
        'Fit with fifth order polynomial']

LEGEND, labels, [255, 3, 2, 6], [0, 0, 0, 0], $
    [0, 0, 0, 0], 4., 1.5, .3
    ; Put a legend on the plot.

```

## See Also

[CURVEFIT](#), [FUNCT](#), [GAUSSFIT](#), [POLYFITW](#), [REGRESS](#), [SVDFIT](#)

---

## ***POLYFITW* Function**

Standard Library function that fits an  $n$ -degree polynomial curve through a set of data points using the least-squares method.

### **Usage**

*result* = POLYFITW(*x*, *y*, *wt*, *deg* [, *yft*, *ybd*, *sig*, *mat*])

### **Input Parameters**

*x* — The vector containing the independent  $x$ -coordinates of the data.

*y* — The vector containing the dependent  $y$ -coordinates of the data. Must have the same number of elements as  $x$ .

*wt* — The vector of weighting factors for determining the weighting of the least-squares fit. Must have the same number of elements as  $x$ . Normalize this parameter for best results.

*deg* — The degree of the polynomial to be fitted to the data.

### **Output Parameters**

*yft* — (optional) The vector containing the calculated  $y$  values.

*ybd* — (optional) The vector containing the error estimate of each point. (The error estimate is equal to one standard deviation.)

*sig* — (optional) The standard deviation of the function, expressed in the units of the  $y$  direction.

*mat* — (optional) The correlation matrix of the coefficients.

### **Returned Value**

*result* — The vector containing the coefficients of the polynomial equation that best approximates the data. It has a length of  $deg + 1$ .

### **Keywords**

None.

## Discussion

POLYFITW is similar to the POLY\_FIT function, except that it permits the weighting of data points. Weighting is useful when you want to correct for potential errors in the data you are fitting to a curve. The weighting factor, *wt*, adjusts the parameters of the curve so that the error at each point of the curve is minimized. For more information, see the section [Weighting Factor on page 180](#) in Volume 1 of this *Reference*.

## Example

```
x = FINDGEN(9)
y = [5., 4., 3., 2., 2., 3., 5., 6., 7.]
    ; Create the data.

TEK_COLOR
PLOT, x, y, Title='POLYFITW EXAMPLE'
    ; Load a color table and plot the original data.

wt = FLTARR(9) + 1.0
coeff_no_wt = POLYFITW(x, y, wt, 1, yfit)
    ; Fit with a first-order polynomial, without weighting.

OPLOT, x, yfit, Color=3
    ; Overplot the calculated values on the original plot.

wt = 1.0/y
coeff_stat_wt = POLYFITW(x, y, wt, 1, yfit)
    ; Fit with statistical weighting.

OPLOT, x, yfit, Color=2
    ; Overplot the calculated values on the original plot.

labels=['Original data', 'Fit with no weighting', $
        'Fit with statistical weighting']

LEGEND, labels, [255, 3, 2], [0, 0, 0], $
        [0, 0, 0], 4., 1.5, .3
    ; Put a legend on the plot.
```

## See Also

[CURVEFIT](#), [FUNCT](#), [GAUSSFIT](#), [POLY\\_FIT](#), [REGRESS](#), [SVDFIT](#)

---

## ***POLY\_MERGE Procedure***

Merges two vertex lists and two polygon lists together so that they can be rendered in a single pass.

### **Usage**

*POLY\_MERGE*, *vertex\_list1*, *vertex\_list2*, *polygon\_list1*,  
*polygon\_list2*, *vert*, *poly*, *pg\_num*

### **Input Parameters**

*vertex\_list1* — The first vertex list.

*vertex\_list2* — The second vertex list.

*polygon\_list1* — The first polygon list.

*polygon\_list2* — The second polygon list.

---

**NOTE** For more information, see the section *Vertex Lists and Polygon Lists* in Chapter 7 of the *PV-WAVE User's Guide*.

---

### **Output Parameters**

*vert* — A new variable consisting of *vertex\_list1* and *vertex\_list2* merged together.

*poly* — A new variable consisting of *polygon\_list1* and *polygon\_list2* merged together and modified so that it is compatible with *vert*.

*pg\_num* — The total number of polygons in the merged list.

### **Keywords**

*Edge1* — A vector containing the edge colors for the first polygon list.

*Edge2* — A vector containing the edge colors for the second polygon list.

*Edge\_List* — The edge colors for the merged list.

*Fill1* — A vector containing the fill colors for the first polygon list.

*Fill2* — A vector containing the fill colors for the second polygon list.

*Fill\_List* — The fill colors for the merged list.

***Opaque1*** — A vector containing the translucency factors for the first polygon list. (A translucency factor of 0 is completely clear. The higher the translucency factor, the more opaque the polygon.)

***Opaque2*** — A vector containing the translucency factors for the second polygon list.

***Opaque\_List*** — The translucency factors for the merged list.

## Discussion

The merged lists returned by POLY\_MERGE are suitable for input into POLYSHADE or POLY\_PLOT, where they may be rendered in a single pass.

## Example

See the `sphere_demo3` demonstration program in:

(UNIX) `<wavedir>/demo/arl`

(OpenVMS) `<wavedir>:[DEMO.ARL]`

(Windows) `<wavedir>\demo\arl`

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[POLY\\_PLOT](#), [POLYSHADE](#)

---

## ***POLY\_NORM Function***

Returns a list of 3D points converted from data coordinates to normal coordinates.

### Usage

*result* = POLY\_NORM(*points*)

### Input Parameters

*points* — A (3, *n*) array of points (or vertices) to transform.

## Returned Value

*result* — The list of 3D points that has been converted from data coordinates to normal coordinates.

## Keywords

None.

## Discussion

POLY\_NORM uses the system variables !X.S, !Y.S, and !Z.S to do the conversion. (These system variables are described in [Chapter 4, \*System Variables\*](#).)

## Example

See the [Example](#) section in the description of the POLY\_C\_CONV routine.

For other examples, see the poly\_demo1, sphere\_demo3, and vol\_demo4 demonstration programs in:

(UNIX)        <wavedir>/demo/arl

(OpenVMS)   <wavedir>:[DEMO.ARL]

(Windows)   <wavedir>\demo\arl

Where <wavedir> is the main PV-WAVE directory.

## See Also

[POLY\\_DEV](#), [POLY\\_TRANS](#)

For more information, see the section *Coordinate Conversion* in Chapter 7 of the *PV-WAVE User's Guide*, and the section *Three Graphics Coordinate Systems* in Chapter 4 of the *PV-WAVE User's Guide*.

---

## ***POLY\_PLOT Procedure***

Renders a given list of polygons.

### **Usage**

*POLY\_PLOT*, *vertex\_list*, *polygon\_list*, *pg\_num*, *winx*, *winy*,  
*fill\_colors*, *edge\_colors*, *poly\_opaque*

### **Input Parameters**

*vertex\_list* — A (3, *n*) array containing the 3D coordinates of each vertex. Must be in device coordinates.

---

**TIP** To obtain device coordinates from data coordinates, use the *POLY\_NORM*, *POLY\_TRANS*, and *POLY\_DEV* functions, in that order.

---

*polygon\_list* — An array containing the number of sides for each polygon and the subscripts into the *vertex\_list* array.

For more information on the above two parameters, see .

*pg\_num* — The total number of polygons to plot.

*winx*, *winy* — The *x* and *y* dimensions, respectively, of the current plot window in device coordinates.

---

**NOTE** The *winx* and *winy* parameters are ignored if the *Image* keyword is present.

---

*fill\_colors* — The color(s) to fill the polygons with:

- If *fill\_colors* contains fewer than *pg\_num* elements, then all polygons are filled with the color specified by the first element in *fill\_colors*.
- Otherwise, each polygon is filled with the corresponding color found in *fill\_colors(i)*.
- To prevent polygon fill, set *fill\_colors* to -1. For example, *fill\_colors* could contain:

255 Fill the first polygon with color 255.

200 Fill the second polygon with color 200.

-1 Don't fill the third polygon.

0 Fill the fourth polygon with color 0.

**edge\_colors** — The color(s) to draw the polygon edges with. *edge\_colors* works in the same manner as *fill\_colors*.

To suppress the plotting of polygon edges, set *edge\_colors* to  $-1$ .

---

**NOTE** Polygon edges are not plotted if the *Image* keyword is present.

---

**poly\_opaque** — The translucency factor to use for plotting each polygon:

- If *poly\_opaque* contains fewer than *pg\_num* elements, then all polygons are plotted with the translucency factor specified by the first element in *poly\_opaque*.
- Otherwise, each polygon is plotted with the corresponding translucency factor found in *poly\_opaque(i)*.
- To prevent translucency, set *poly\_opaque* to  $-1$ .

A translucency factor of 0 is completely clear. The higher the translucency factor, the more opaque the polygon is. If the maximum value found in *Image* is 255 and if the maximum color value found in *fill\_colors* is also 255, then a translucency factor of 255 is completely opaque.

---

**NOTE** The *Image* keyword must be used for *poly\_opaque* to take effect.

---

## Keywords

**Image** — On input, a 2D array containing the image on which to plot the polygons:

- If *Image* is not present, then the polygons are plotted immediately as generated in the current graphics window.
- If *Image* is present, then no polygon edges are plotted and the *winx* and *winy* parameters are ignored.

On output, *Image* contains the original image with the polygons plotted on it. This image may then be displayed using TV or other similar routines.

**ZClip** — If this keyword is present and nonzero, then polygons that do not have at least one vertex in front of the view point are not plotted.

## Discussion

POLY\_PLOT renders a list of polygons. It is slower than the alternative procedure POLYSHADE, but it is more flexible:

- POLY\_PLOT can draw the edges of the polygons, unlike POLYSHADE.
- POLY\_PLOT does not fail if one or more polygons have a vertex outside the current plot window, unlike POLYSHADE.

POLY\_PLOT uses a simple back-to-front sorting method to determine the polygon plotting order. It does not render polygons with light source shading, but it can plot opaque and translucent polygons. You can also specify the fill color and edge color for each polygon.

## Example

See the [Example](#) section in the description of the POLY\_DEV function.

For other examples, see the `sphere_demo2`, `sphere_demo3`, and `vol_demo4` demonstration programs in:

```
(UNIX)      <wavedir>/demo/ar1  
(OpenVMS)  <wavedir>:[DEMO.ARL]  
(Windows)  <wavedir>\demo\ar1
```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[POLY\\_C\\_CONV](#), [POLY\\_DEV](#), [POLY\\_NORM](#), [POLYSHADE](#), [POLY\\_TRANS](#)

---

## **POLYSHADE Function**

Constructs a shaded surface representation of one or more solids described by a set of polygons.

### **Usage**

*result* = POLYSHADE(*vertices*, *polygons*)

*result* = POLYSHADE(*x*, *y*, *z*, *polygons*)

### **Input Parameters**

***vertices*** — A (3, *n*) array containing the *x*-, *y*-, and *z*-coordinates of each vertex. Coordinates may be in either data or normalized coordinates, depending on which keywords are present.

***x*, *y*, *z*** — The *x*-, *y*-, and *z*-coordinates of each vertex may alternatively be specified as three individual array expressions; *x*, *y*, and *z* must all contain the same number of elements.

***polygons*** — An integer or longword array containing the indices of the vertices of each polygon. The vertices of each polygon should be listed in counterclockwise order when observed from outside the surface. The vertex description of each polygon is a vector of the form [*n*, *i*<sub>0</sub>, *i*<sub>1</sub>, ... , *i*<sub>*n*-1</sub>], and the array *polygons* is the concatenation of the lists of each polygon.

For example, to render a pyramid consisting of four triangles, *polygons* will contain 16 elements, made by concatenating four 4-element vectors of the form [*3*, *V0*, *V1*, *V2*]. *V0*, *V1*, and *V2* are the indices of the vertices describing each triangle.

### **Returned Value**

***result*** — A 2D byte array containing the shaded image.

### **Keywords**

***Data*** — Indicates that the vertex coordinates are in data units, the default coordinate system.

***Mesh*** — When present and nonzero, a wire-frame mesh is drawn over the polygons, and they are not shaded.

**Normal** — Indicates that coordinates are in normalized units, within the 3D (0,1) cube.

**Poly\_Shades** — Similar to the *Shades* keyword, except one shade per polygon is passed to POLYFILL rather than one shade per vertex.

**Shades** — An array expression, of the same number of elements as vertices, containing the color index at each vertex. The shading of each pixel is interpolated from the surrounding *Shades* values. For most displays, this keyword should be scaled into the range of bytes. If this keyword is omitted, light source shading is used.

**T3d** — Enables the 3D to 2D transformation contained in the homogeneous 4-by-4 matrix !P.T. If this keyword is set, the system variable !P.T must contain a valid transformation matrix.

**XSize** — The number of columns in the output image array. If omitted, sets the number of columns equal to the *x* resolution of the currently selected display device.

**YSize** — The number of rows in the output image array. If omitted, sets the number of rows equal to the *y* resolution of the currently selected display device.

---

**CAUTION** If you are using a PostScript or other high resolution graphics device, you should explicitly specify the *XSize* and *YSize* parameters. Making the output image of full device size (the default) will result in an insufficient memory error.

---

## Discussion

Note that you must set up a 3D coordinate system prior to calling POLYSHADE.

POLYSHADE constructs the shaded surface using the scan line algorithm. The shading model is a combination of diffuse reflection and depth cueing. Polygons are shaded in one of two ways:

- With constant shading, where each polygon is given a constant intensity.
- With Gouraud shading, where the intensity is computed at each vertex and then interpolated over the polygon.

---

**TIP** Use the SET\_SHADING procedure to control the direction of the light source and other shading parameters.

---

### Example

Function POLYSHADE is often used in conjunction with procedure SHADE\_VOLUME for volume visualization. This example creates a volume dataset and renders an isosurface from that dataset.

```
vol = FLTARR(20, 20, 20)
    ; Create a 3D single-precision, floating-point array.

FOR x = 0, 19 DO FOR y = 0, 19 $
    DO FOR z = 0, 19 DO $
        vol(x, y, z) = SQRT((x-10)^2 + (y-10)^2 + $
            (z-10)^2) + 1.5 * COS(z)
    ; Create the volume dataset.

SHADE_VOLUME, vol, 7, v, p
    ; Find the vertices and polygon at a contour level of 7.

SURFACE, FLTARR(2, 2), /Nodata, /Save, $
    XRange = [0, 20], YRange = [0, 20], $
    ZRange = [0, 20], XStyle = 4, YStyle = 4, $
    ZStyle = 4
    ; Set up an appropriate 3D transformation.

image = POLYSHADE(v, p, /T3d)
    ; Render the image. Note that the T3d keyword has been set so that
    ; the 3D transformation established by SCALE3 is used.

TV, image
    ; Display the image.
```



**Figure 2-49** Isosurface at level 7 of volume dataset from example.

## See Also

[RENDER24](#), [SET\\_SHADING](#)

System Variables: [!P.T](#)

For additional information on defining a coordinate system, see .

---

## ***POLY\_SPHERE Procedure***

Generates the vertex list and polygon list that represent a sphere.

### **Usage**

`POLY_SPHERE`, *radius*, *px*, *py*, *vertex\_list*, *polygon\_list*

### **Input Parameters**

*radius* — If *radius* is a scalar value, then all the polygons are generated at this radius.

If *radius* is a 2D (*m*, *n*) array, then the radius of each polygon is generated at the corresponding radius. The *radius* array is scaled to the dimensions (*px*, *py*) before use.

You can use the array returned by the `GRID_SPHERE` function as *radius* values.

*px* — A scalar value specifying the number of polygons around the equator.

*py* — A scalar value specifying the number of polygons around the meridian.

### **Output Parameters**

*vertex\_list* — A (3, *n*) array of polygon vertices.

*polygon\_list* — The list of vertices for each polygon.

For more information, see the section *Vertex Lists and Polygon Lists* in Chapter 7 of the *PV-WAVE User's Guide*.

### **Keywords**

*Degrees* — If *Degrees* is present and nonzero, then the values for *XMin*, *XMax*, *YMin*, and *YMax* are in degrees instead of radians.

***XMax*** — The longitude of the right edge of the portion of the polygon mesh you want to use (where on the sphere the polygon mesh is to be extracted). Should be in the range  $-\pi$  to  $+\pi$  radians ( $-180$  to  $+180$  degrees). The value is assumed to be in radians unless the *Degrees* keyword is set.

If *XMax* is omitted, a longitude of  $\pi$  is mapped to the right edge of the polygon mesh for the entire sphere (when viewed from outside the sphere).

***XMin*** — The longitude of the left edge of the portion of the polygon mesh you want to use (where on the sphere the polygon mesh is to be extracted). Should be in the range  $-\pi$  to  $+\pi$  radians ( $-180$  to  $+180$  degrees). The value is assumed to be in radians unless the *Degrees* keyword is set.

If *XMin* is omitted, a longitude of  $-\pi$  is mapped to the left edge of the polygon mesh for the entire sphere (when viewed from outside the sphere).

***YMax*** — The latitude of the top edge of the polygon mesh. Should be in the range  $-\pi/2$  to  $+\pi/2$  radians ( $-90$  to  $+90$  degrees). The value is assumed to be in radians unless the *Degrees* keyword is set.

If *YMax* is omitted, a latitude of  $\pi/2$  is mapped to the top edge of the polygon mesh for the entire sphere (when viewed from outside the sphere).

***YMin*** — The latitude of the bottom edge of the portion of the polygon mesh you want to use (where on the sphere the polygon mesh is to be extracted). Should be in the range  $-\pi/2$  to  $+\pi/2$  radians ( $-90$  to  $+90$  degrees). The value is assumed to be in radians unless the *Degrees* keyword is set.

If *YMin* is omitted, a latitude of  $-\pi/2$  is mapped to the bottom edge of the polygon mesh for the entire sphere (when viewed from outside the sphere).

## Discussion

The *vertex\_list* and *polygon\_list* generated by POLY\_SPHERE are suitable for use with the POLYSHADE and POLY\_PLOT rendering procedures.

To generate the polygons for a portion of a sphere, rather than an entire sphere, use the *XMin*, *XMax*, *YMin*, and *YMax* keywords. For example, to work with the central portion of the country from a map of the United States, you might use:

```
XMin=-110, XMax=-100, YMin=35, YMax=45
```

## Example

```
PRO sphere_demo1
; This program displays an image warped onto a sphere.
```

```

xval = 512
yval = 512
img = BYTARR(xval, yval)
OPENR, 1, !Data_Dir + 'mandril.img'
READU, 1, img
CLOSE, 1
xval = 128
yval = 128
img = REBIN(img, xval, yval)
      ; Read in the image and shrink it to a 128-by-128 array.
POLY_SPHERE, 1.0, xval, yval, vertex_list, $
      polygon_list
      ; Define the sphere as a list of polygons.
WINDOW, 0, Colors=128
LOADCT, 4
CENTER_VIEW, Ax=(-75.0), Az=(-90.0), Zoom=0.9
      ; Set up the viewing window and load the color table.
TVSCL, POLYSHADE(vertex_list, polygon_list, /T3d, Shade=img)
      ; Display the shaded surface representation of the data warped
      ; onto the sphere.
TVSCL, img
      ; Display the original image in the corner of the window.
END

```

For other examples, see the `grid_demo5`, `sphere_demo2`, and `sphere_demo3` demonstration programs in:

```

(UNIX)      <wavedir>/demo/arl
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows) <wavedir>\demo\arl

```

Where <wavedir> is the main PV-WAVE directory.

## See Also

[GRID\\_SPHERE](#), [POLYSHADE](#), [POLY\\_SURF](#)

---

## ***POLY\_SURF Procedure***

Generates a 3D vertex list and a polygon list, given a 2D array containing  $z$  values.

### **Usage**

`POLY_SURF`, *surf\_dat*, *vertex\_list*, *polygon\_list*, *pg\_num*

### **Input Parameters**

*surf\_dat* — A 2D array containing  $z$  values. The 3D polygon vertices are generated from this data.

### **Output Parameters**

*vertex\_list* — A  $(3, n)$  array containing the 3D coordinates of the polygon vertices.

*polygon\_list* — A 1D array containing the number of sides for each polygon, as well as the subscripts into the *vertex\_list* array for the vertices of each polygon.

*pg\_num* — The total number of polygons defined by *vertex\_list* and *polygon\_list*. This parameter can be used as input into `POLY_PLOT`.

### **Keywords**

None.

### **Discussion**

`POLY_SURF` generates a list of polygons from a 2D array that contains  $Z$  values. All the polygons generated have four sides (and four vertices).

- The *vertex\_list* array returned is suitable for input into the `POLY_TRANS`, `POLY_NORM`, `POLY_DEV`, `POLY_PLOT` and `POLYSHADE` procedures.
- The *polygon\_list* array returned is suitable for input into the `POLY_PLOT` and `POLYSHADE` procedure. For more information, see .

### **Example**

See the Example section in the description of the `POLY_DEV` routine.

### **See Also**

[POLY\\_SPHERE](#)

---

## ***POLY\_TRANS* Function**

Returns a list of 3D points transformed by a 4-by-4 transformation matrix.

### **Usage**

*result* = POLY\_TRANS(*points*, *trans*)

### **Input Parameters**

*points* — A (3, *n*) array of points (or vertices) to transform.

*trans* — A (4, 4) array to transform the points with.

### **Returned Value**

*result* — A list of 3D points transformed by a 4-by-4 transformation matrix.

### **Keywords**

None.

### **Discussion**

You can use the T3D, CENTER\_VIEW, SET\_VIEW3D, and VIEWER procedures to build the transformation matrix. The 4-by-4 matrix most often used is the system viewing matrix !P.T. For more information, see .

### **Example**

See the Example section in the description of the [POLY\\_DEV](#) routine.

For other examples, see the `sphere_demo2`, `sphere_demo3`, and `vol_demo4` demonstration programs in:

(UNIX)        <wavedir>/demo/ar1

(OpenVMS)    <wavedir>:[DEMO.ARL]

(Windows)    <wavedir>\demo\ar1

Where <wavedir> is the main PV-WAVE directory.

### **See Also**

[POLY\\_DEV](#), [POLY\\_NORM](#)

For more information, see the section *Coordinate Conversion* in Chapter 7 of the *PV-WAVE User's Guide*.

---

## **POLYWARP Procedure**

Standard Library procedure that calculates the coefficients needed for a polynomial image warping transformation.

### **Usage**

POLYWARP, *xd*, *yd*, *xin*, *yin*, *deg*, *xm*, *ym*

### **Input Parameters**

*xd* — The vector containing the  $x$ -coordinates to be fit as a function of (*xin*, *yin*).

*yd* — The vector containing the  $y$ -coordinates to be fit as a function of (*xin*, *yin*).

*xin* — The vector containing the independent  $x$ -coordinates. Must have the same number of points as *xd*.

*yin* — The vector containing the independent  $y$ -coordinates. Must have the same number of points as *yd*.

*deg* — The degree of the polynomial to be fitted to the data. The number of coordinate pairs formed by *xin* and *yin* must be greater than or equal to  $(deg + 1)^2$ .

### **Output Parameters**

*xm* — A  $(deg + 1)$ -by- $(deg + 1)$  array containing the coefficients of *xd* as a function of (*xin*, *yin*).

*ym* — A  $(deg + 1)$ -by- $(deg + 1)$  array containing the coefficients of *yd* as a function of (*xin*, *yin*).

### **Keywords**

None.

### **Discussion**

POLYWARP calculates its transformation coefficients using the polynomial least-squares method. It returns the coefficients of the polynomial functions which best approximate the data:

- The *xm* polynomial array pertains to the  $x$  direction.
- The *ym* polynomial array pertains to the  $y$  direction.

POLYWARP determines the coefficients  $A_x(i, j)$  and  $A_y(i, j)$  of these two polynomial functions:

$$X_{dep} = \sum_{i,j}^{deg} A_x(i, j) X_{indep}^j Y_{indep}^i$$

and

$$Y_{dep} = \sum_{i,j}^{deg} A_y(i, j) X_{indep}^j Y_{indep}^i$$

where  $A_x = xm$  and  $A_y = ym$ .

---

**NOTE** The  $xm$  ( $x$ -coefficients) and  $ym$  ( $y$ -coefficients) can be used as input for the POLY\_2D function. POLY\_2D performs the actual warping of the image, using the  $x$  and  $y$  transformation coefficients that you provide.

---

## Example

```
image = BYTSCL(DIST(300))
WINDOW, XSize=300, YSize=300
TV, image
    ; Create a test image and display it.

xin = [100, 200, 200, 100]
yin = [200, 200, 100, 100]
XYOUTS, 100, 200, '0', /Device
XYOUTS, 200, 200, '1', /Device
XYOUTS, 200, 100, '2', /Device
XYOUTS, 100, 100, '3', /Device
    ; Create arrays describing four independent control points and plot
    ; these points on top of the image. These points describe a square,
    ; and represent the position of points in a calibration image.

xd = INTARR(4)
yd = INTARR(4)
FOR i=0,3 DO BEGIN
```

```

PRINT, 'Pick warped point ', i
CURSOR, xt, yt, /Device
WAIT, 0.5
xd(i) = xt
yd(i) = yt
ENDFOR
; Pick four dependent warping control points from the image; these
; points represent the calibration points as actually measured by an
; instrument, which, due to distortion, has warped them.

deg = 1
POLYWARP, xd, yd, xin, yin, deg, xm, ym
; Perform linear (first degree) warping. POLYWARP will return xm
; and ym, the coefficients of the polynomial functions which describes
; this warping.

interp = 1
result = POLY_2D(image, xm, ym, interp)
; Apply the polynomial functions calculated with POLYWARP to the
; first image, using bilinear interpolation.

TVSCL, result
; Display the resulting image, which represents the image after
; correcting for instrument distortion.

```

## See Also

### [POLY\\_2D](#)

For more information on image processing, see Chapter 5, *Displaying 3D Data*, in the *PV-WAVE User's Guide*.

For additional information, see the section *Geometric Transformations* in Chapter 6 of the *PV-WAVE User's Guide*.

---

## ***POPD Procedure***

Standard Library procedure that pops a directory from the top of a last-in, first-out directory stack.

### **Usage**

POPD

### **Parameters**

None.

### **Keywords**

None.

### **Discussion**

POPD changes the current working directory to the directory saved on the top of the directory stack. The stack is maintained by the PUSHHD and POPD procedures. This top directory is then removed from the stack. (If you try to pop from an empty stack, an error message is displayed.)

Directories that have been pushed onto the stack are removed by the POPD procedure. The last directory pushed onto the stack is the first directory popped out of it. There is no limit to how deep directories may be stacked.

### **Example**

In this example, PUSHHD is used to change the current working directory. The current working directory is pushed onto the directory stack before moving to the subdirectory. Procedure POPD is used to change the current working directory to the directory at the top of the directory stack. Thus, you are returned to the original working directory. Procedure PRINTD is used to view the current working directory and the directory stack before and after the execution of PUSHHD and POPD.

### **UNIX Examples**

```
PRINTD
```

```
    ; Display the current working directory and the directory stack.
```

```
PUSHD, 'sub1\data'  
    ; Push the current working directory onto the directory stack, and  
    ; move to the subdirectory /sub1/data.  
  
PRINTD  
    ; Display the current working directory and the directory stack.  
    ; Note that the top of the stack contains the previous working  
    ; directory.  
  
POPD  
    ; Move to the directory at the top of the directory stack. In this case,  
    ; you are moved back to the original working directory.  
  
PRINTD  
    ; Display the current working directory and the directory stack.
```

## OpenVMS Examples

```
PRINTD  
    ; Display the current working directory and the directory stack.  
  
PUSHD, '[.sub1.data]'  
    ; Push the current working directory onto the directory stack, and  
    ; move to the subdirectory [.sub1.data] (OpenVMS).  
  
PRINTD  
    ; Display the current working directory and the directory stack.  
    ; Note that the top of the stack contains the previous working  
    ; directory.  
  
POPD  
    ; Move to the directory at the top of the directory stack. In this case,  
    ; you are moved back to the original working directory.  
  
PRINTD  
    ; Display the current working directory and the directory stack.
```

## Windows Examples

```
PRINTD  
    ; Display the current working directory and the directory stack.  
  
PUSHD, 'sub1\data'  
    ; Push the current working directory onto the directory stack, and  
    ; move to the subdirectory \sub1\data.  
  
PRINTD  
    ; Display the current working directory and the directory stack.  
    ; Note that the top of the stack contains the previous working  
    ; directory.
```

POPD

; Move to the directory at the top of the directory stack. In this case,  
; you are moved back to the original working directory.

PRINTD

; Display the current working directory and the directory stack.

## See Also

[CD](#), [PRINTD](#), [PUSHD](#)

---

## ***PRIME*** Function

Standard Library function that returns all positive primes less than or equal to a scalar input.

### Usage

*result* = PRIME(*value*)

### Input Parameters

*value* — The scalar input.

### Returned Value

*result* — A vector containing all positive primes less than or equal to *value*.

### Keywords

None.

### Examples

PRINT, PRIME(10)

## See Also

[FACTOR](#), [GCD](#), [LCM](#)

---

## **PRINT Procedures**

### **(PRINT, PRINTF)**

Perform output of ASCII data:

- PRINT performs output to the standard output stream (file unit `-1`).
- PRINTF requires the output file unit to be specified.

### **Usage**

PRINT, *expr*<sub>1</sub>, ... , *expr*<sub>*n*</sub>

PRINTF, *unit*, *expr*<sub>1</sub>, ... , *expr*<sub>*n*</sub>

### **Input Parameters**

*unit* — The file unit to which the output will be sent.

*expr*<sub>*i*</sub> — The expressions to be output.

### **Keywords**

*Format* — Allows the format of the output to be specified in precise detail, using a FORTRAN-style specification. FORTRAN-style formats are described in the *PV-WAVE Programmer's Guide*.

If the *Format* keyword is not present, PV-WAVE uses its default rules for formatting the output. These rules are described in the section *Free Format Output* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

### **Example**

In this example, PRINTF is used to write 10 integers to a file. The integers are then read from the file and displayed using the PRINT procedure. The *Format* keyword is used with the PRINT procedure to place one space after each integer as it is written to the screen.

```
nums = INDGEN(10)
      ; Create a 10-element integer vector that is initialized to the values of
      ; its one-dimensional subscripts.

OPENW, unit, 'printex.dat', /Get_Lun
      ; Open the printex.dat file for writing.

PRINTF, unit, nums
      ; Write the integers in nums to the file.
```

```
POINT_LUN, unit, 0
    ; Rewind the file to the beginning.
n = INTARR(10)
    ; Create a 10-element integer vector.
READF, unit, n
    ; Read the contents of printex.dat into n.
PRINT, n, Format = '(10(i1, 1x))'
    0 1 2 3 4 5 6 7 8 9
    ; Display the formatted contents of variable n.
FREE_LUN, unit
    ; Close the file and free the file unit number.
```

## See Also

[DT\\_PRINT](#)

---

## ***PRINTD Procedure***

Standard Library procedure that lists the directories located in the directory stack, and the current working directory.

### **Usage**

PRINTD

### **Input Parameters**

None.

### **Keywords**

None.

### **Example**

See the example for POPD.

## See Also

[CD](#), [POPD](#), [PUSHD](#)

---

## ***PRODUCT Function***

Returns the product of all elements in an array.

### **Usage**

*result* = PRODUCT(*array*)

### **Input Parameters**

*array* — An array.

### **Returned Value**

*result* — A scalar value equal to the product of all the elements in *array*.

### **Keywords**

None.

### **Discussion**

If *array* is of type single- or double-precision floating point, or single- or double-precision complex, the result will be of the same type. If *array* is of any other type, PRODUCT returns a single-precision floating-point result.

### **Examples**

PM, PRODUCT( [ 2, 3, 4 ] )

---

## ***PROFILE Function***

Standard Library function that extracts a profile from an image.

### **Usage**

*result* = PROFILE(*image*)

### **Input Parameters**

*image* — The input image array. May be any type except string or complex.

## Returned Value

**result** — A floating-point vector containing the profile data points. It is of the same data type as *image*.

## Keywords

**Nomark** — If set to 1, inhibits marking the selected line on the image display.

**XStart** — The starting *x* location of the lower-left corner of the image in the window.

**YStart** — The starting *y* location of the lower-left corner of the image in the window.

## Discussion

To use PROFILE, mark two endpoints on the image display with the cursor by clicking on any mouse button. PROFILE then extracts the values of the image elements along a line connecting the endpoints and returns these values as a floating-point vector.

## Example

This example uses the PROFILE function to retrieve a vector of image values.

```
OPENR, unit, FILEPATH('aerial_demo.img', Subdir = 'data'), /Get_Lun
      ; Open the file containing the image.

img = BYTARR(512, 512)
      ; Create an array large enough to hold the image.

READU, unit, img
      ; Read the image data.

WINDOW, 0, XSize = 512, YSize = 512
      ; Create a window to display an image from the file.

TV, img
      ; Display the first image from the file.

HIST_EQUAL_CT, img

vals = PROFILE(img)
      ; Retrieve a profile from the file.

INFO, vals
      ; Examine the type and number of elements in the returned vector.

FREE_LUN, unit
      ; Close the file and free the file unit number.
```

## See Also

[PROFILES](#), [TV](#), [TVSCL](#)

---

## PROFILES Procedure

Standard Library procedure that lets you interactively draw row or column profiles of the image displayed in the current window. The profiles are displayed in a new window, which is deleted when you exit the procedure.

### Usage

PROFILES, *image*

### Input Parameters

*image* — The image array displayed in the current window. May be any data type except string or complex. The profile graphs are made from this array.

### Keywords

*Order* — Controls the direction of *image* transfer. Set to 1 to have *image* written top-down. Set to 0 to have *image* written bottom-up. The default is the current value of the system variable !Order.

*Sx* — The starting *x* value of the image within the window. If omitted, 0 is assumed.

*Sy* — The starting *y* value of the image within the window. If omitted, 0 is assumed.

*Wsize* — The size of the new profile window as a fraction or multiple of the default size, which is 640-by-512.

### Discussion

To use PROFILES, place the cursor in the original image window. Move the cursor so that the row or column profile is updated interactively. Press the left mouse button to toggle between displaying a row or column profile. Press the right mouse button to exit the procedure.

### Example

To create a profile window for the image found in file `cereb_demo .img`, change to the following directory:

(UNIX) <wavedir>/data

(OpenVMS) <wavedir>:[DATA]

(Windows) <wavedir>\data

Where <wavedir> is the main PV-WAVE directory.

To display the image in the window, enter:

```
cereb = BYTARR(512, 512)
```

; Create a 512-by-512 byte array called cereb.

```
OPENR, 1, !Data_Dir + 'cereb_demo.img'
```

; Open the file for reading using a logical unit number of 1.

```
READU, 1, cereb
```

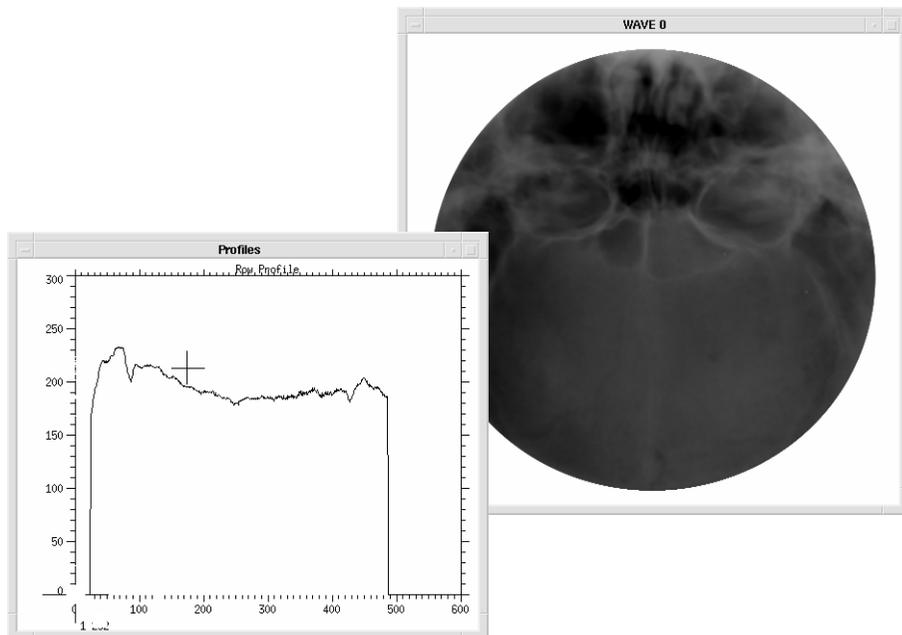
; Read data from the file into the variable cereb.

```
TVSCL, cereb
```

; Display the image.

```
PROFILES, cereb
```

; Display the profile window.



**Figure 2-50** Profile plot taken from the image.

## See Also

[PROFILE](#)

---

## **PROMPT Procedure**

Standard Library procedure that sets the interactive prompt.

### **Usage**

PROMPT, *string*

### **Input Parameters**

*string* — A scalar string defining the new prompt.

### **Keywords**

None.

### **Discussion**

PROMPT sets the interactive prompt to *string*, and also changes the value of the system variable !Prompt to that string.

If no parameter is supplied, the prompt string reverts to WAVE>.

### **Example**

```
WAVE> PROMPT, 'YOU_RANG?> '  
YOU_RANG?> PROMPT  
WAVE>
```

## See Also

System Variables: [!Prompt](#)

---

## **PSEUDO Procedure**

Standard Library procedure that creates a pseudo color table based on the hue, lightness, saturation (HLS) color system.

### **Usage**

PSEUDO, *ltlo*, *lthi*, *stlo*, *sthi*, *hue*, *lp* [, *rgb*]

### **Input Parameters**

*ltlo* — The starting color lightness or intensity, expressed as 0 to 100 percent. Full lightness (the brightest color) is 100 percent.

*lthi* — The ending color lightness or intensity, expressed as 0 to 100 percent.

*stlo* — The starting color saturation, expressed as 0 to 100 percent. Full saturation (undiluted or pure color) is expressed as 100 percent.

*sthi* — The ending color saturation, expressed as 0 to 100 percent.

*hue* — The starting hue. It ranges from 0 to 360 degrees, with red equal to 0, green equal to 120, and blue equal to 240.

*lp* — The number of loops of hue to make in the color cone. Does not have to be an integer.

### **Output Parameters**

*rgb* — (optional) A 256-by-3 integer output array containing the red, green, and blue vector values loaded into the color tables. The red vector is equal to `RGB(*, 0)`, the green vector is `RGB(*, 1)`, and the blue vector is `RGB(*, 2)`.

### **Keywords**

None.

### **Discussion**

The pseudo-color mapping generated by PSEUDO is done by the following three steps:

- ❑ Map the HLS coordinate space to the lightness, absorbance, saturation (LAS) coordinate space.
- ❑ Find  $n$  colors (in this case 256) spread out along a helix that spans this LAS space. These colors are supposedly a near maximal entropy mapping for the eye, given a particular  $n$ .
- ❑ Map the LAS coordinate space into the red, green, blue (RGB) coordinate space.

The result, given  $n$  desired colors, is that  $n$  discrete values are loaded into the red color vector,  $n$  discrete values are loaded into the green color vector, and  $n$  discrete values are loaded into the blue color vector.

## See Also

[HLS](#), [LOADCT](#), [TVLCT](#)

For background information about color systems, see .

---

## ***PUSHD Procedure***

Standard Library procedure that pushes a directory onto the top of a last-in, first-out directory stack.

### **Usage**

`PUSHD` [, *directory*]

### **Input Parameters**

*directory* — A scalar string specifying the path of the new working directory.

If not specified, or if specified as a null string, pushes the current directory onto the stack, and the new working directory is changed to the user's home directory.

### **Keywords**

None.

## Discussion

Directories that have been pushed onto the stack by `PUSHD` can be removed with `POPD`. The last directory pushed onto the stack is the first directory popped out of it. There is no limit to how deep directories may be stacked.

## Example

See the example for `POPD`.

## See Also

[CD](#), [POPD](#), [PRINTD](#)

---

## QUERY\_TABLE Function

Subsets a table created with the BUILD\_TABLE function.

### Usage

```
result = QUERY_TABLE(table,  
' [Distinct] * | col1 [alias] [, ... , coln [alias]]  
[Where cond]  
[Group By colg1 [,... colgn]] |  
[Order By colo1 [direction][, ... , colon [direction]]] ' )
```

Note that the entire second parameter is a string and must be enclosed in quotes. Also, note that the vertical bar (|) means “or” in this usage. For instance, use either “\*” or “col<sub>i</sub> [alias] [, ..., col<sub>n</sub> [alias]]”, but not both.

### Input Parameters

**table** — The original table (created with the BUILD\_TABLE function) on which the query is performed.

**\*** — An optional wildcard character that includes *all* columns from the original table in the resulting table.

**Distinct** — A qualifier that removes duplicate rows from the resulting table.

**col<sub>i</sub>** — The list of columns that you want to appear in the resulting table. Use the asterisk (\*) wildcard character to select *all* columns in the original table. The *col* names can be arguments to the calculation functions used with the Group By clause.

**alias** — Changes the input table’s column name, *col<sub>i</sub>*, to a new name in the output table. If no alias is specified, the input table’s column name is used in the resulting table.

**Where cond** — A clause containing a conditional expression, *cond*, that is used to specify the rows to be placed in the resulting table. The expression can contain Boolean and/or relational operators.

**Group By col<sub>g</sub>** — A clause specifying one or more columns by which the rows are grouped in the resulting table. Normally, the grouped rows are data summaries containing results of calculation functions (Sum, Avg, etc.) applied to the columns. (Group By and Order By clauses are mutually exclusive: they cannot be used in the same function call.)

**Order By** *colo<sub>i</sub>* — Name of the column(s) to be sorted (ordered) in the resulting table. The first column named is sorted first. The second column named is sorted within the primary column, and so on. (Group By and Order By clauses are mutually exclusive: they cannot be used in the same function call.)

**direction** — Either Asc (the default) or Desc. Asc sorts the column in ascending order. Desc sorts the column in descending order. If neither are specified, the column is sorted in ascending order.

## Returned Value

**result** — The resulting table, containing the columns specified by *col<sub>p</sub>*, and the rows specified by the query qualifiers and clauses. If the query result is empty, and no syntax or other errors occurred, the result returned is -1.

## Keywords

None.

## Discussion

Before you can use QUERY\_TABLE, you must create a table with the BUILD\_TABLE function. For details on BUILD\_TABLE, see the discussion of the BUILD\_TABLE function in this chapter. See also the *PV-WAVE User's Guide*.

A table query always produces a new table containing the query results, or -1 if the query is empty.

Any string or numeric constant used in a QUERY\_TABLE call can be passed into the function as a variable parameter. This means that you can use variables for numeric or string values in relational or Boolean expressions. For more information on passing parameters into QUERY\_TABLE, see

---

**NOTE** Within a QUERY\_TABLE call, the Group By and Order By clauses are mutually exclusive. That is, you cannot place both Group By and Order By in the same QUERY\_TABLE call.

---

## ***Boolean and Relational Operators Used in Queries***

The Where clause uses Boolean and relational operators to “filter” the rows of the table. You can specify any of the following conditions within a Where clause. Use parentheses to control the order of evaluation, if necessary.

- **Comparison operators** — = (equal to), <> (not equal to), < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to)
- **Compound search condition** — Not, And, Or
- **Set membership test** — In

See the *Example* section for more information on these operators.

You can also use relational operators (EQ, GE, GT, LE, LT, and NE) in a Where clause instead of the SQL-style operators listed above.

---

**NOTE** When a literal string is used in a comparison, it must be enclosed in quotes—a different set of quotes than those used to delimit the entire QUERY\_TABLE parameter string. For more information on using strings in comparisons, see the *PV-WAVE User's Guide*.

---

### ***Calculation Functions Used with GROUP BY***

The Group By clause is used in conjunction with calculation functions that operate on the values in the specified groupings. Each function takes one column name as its argument. See the *Example* section for examples showing the use of calculation functions with Group By.

The calculation functions used with Group By are the following, where *col* is the name of a column:

- *Avg(col)* — Averages the values that fall within a group.
- *Count(col)* — Counts the number of occurrences of each data value that falls within a group.
- *Max(col)* — Returns the maximum value that falls within a group.
- *Min(col)* — Returns the minimum value that falls within a group.
- *Sum(col)* — Returns the sum of the values that fall within a group.

### **Examples**

For the following examples, assume table called *phone\_data* contains information on company phone calls. This table contains eight columns of phone information: the date, time, duration of call, caller's initials, phone extension, cost of call, area code of call, and number of call.

The table used in these examples and the data used to create it are available to you. Enter the following command at the WAVE> prompt to restore the table and data:

**(UNIX)** RESTORE, !dir+' /data/phone\_example.sav'

**(OpenVMS)** RESTORE, !dir+' [DATA] PHONE\_EXAMPLE.SAV'

**(Windows)** RESTORE, !dir+' \data\phone\_example.sav'

For more information on the structure of this table and more examples, see the *PV-WAVE User's Guide*.

For an example showing the use of the Distinct qualifier, see the *PV-WAVE User's Guide*.

The following examples show how to query this table in various ways using QUERY\_TABLE.

## Example 1

Create a new table containing only the phone extensions, area code, and phone number of each call made.

This example demonstrates a simple table query that produces a three-column subset of the original table.

```
new_table = QUERY_TABLE(phone_data, $
    'EXT, AREA, NUMBER')
```

A portion of the resulting table is organized as follows:

EXT	AREA	NUMBER
311	215	2155554242
358	303	5553869
320	214	2145559893
289	303	5555836
248	617	6175551999

---

**TIP** For information on printing tables, see the *PV-WAVE User's Guide*.

---

## Example 2

Show data on the calls that cost more than one dollar.

This example demonstrates how a Where clause is used to produce a subset of the original table, where all rows that contain a cost value of less than one dollar are filtered out.

```
new_tbl = QUERY_TABLE(phone_data, $
    '* Where COST > 1.0')
```

The following is an excerpt from the resulting table:

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901002	093200	21.40	TAC	311	5.78	215	2155554242
901002	094700	17.44	EBH	320	4.71	214	2145559893
901004	095000	3.77	DJC	331	1.02	512	5125551228

### Example 3

Show the total cost and duration of calls made from each phone extension for the period of time the data was collected.

This example demonstrates the use of the Group By clause. The column specified *after* Group By is the column by which the other specified columns are grouped. The calculation function Sum() is used to return the total cost and duration for each extension in the table.

The following command produces this result:

```
sum_table = QUERY_TABLE(phone_data, $
    'EXT, SUM(COST), SUM(DUR) Group By EXT')
```

This produces the new table, called `sum_table` containing the columns EXT, SUM\_COST, and SUM\_DUR:

EXT	SUM_COST	SUM_DUR
0	0.00000	4.49000
248	0.350000	1.31000
289	0.00000	16.2300
311	5.78000	21.4000
320	4.71000	17.4400
331	1.02000	3.77000

---

**TIP** The cost and duration columns are named in the result table, by default, with the prefix SUM\_. This prevents any confusion with the existing table columns that are already named COST and DUR. You can change these default names by including aliases in the QUERY\_TABLE function call.

---

The INFO command can be used to show the basic structure of this new table:

```
INFO, /Structure, sum_table
** Structure TABLE_GB_2, 3 tags, 12 length:
```

EXT	LONG	0
SUM_COST	FLOAT	0.000000
SUM_DUR	FLOAT	4.49000

The *Structure* keyword is used because tables are represented as an array of structures. For more information, see the *PV-WAVE User's Guide*.

## Example 4

Show the extension, date, and total duration of all calls made from each extension on each date.

This example demonstrates a multiple Group By clause. For example, you can obtain a grouping by extension and by date. The result is a “grouping within a grouping”.

The following command produces the desired result:

```
tbl = QUERY_TABLE(phone_data, $
    'EXT, DATE, Sum(DUR) Group By EXT, DATE')
```

EXT	DATE	SUM_DUR
0	901003	2.33000
0	901004	2.16000
248	901002	1.31000
289	901002	16.2300
311	901002	21.4000
320	901002	17.4400
331	901004	3.77000
332	901003	2.53000

EXT	DATE	SUM_DUR
358	901002	1.05000
370	901003	0.450000
370	901004	0.160000
379	901003	1.53000
379	901004	1.93000
418	901003	0.350000

Note that each multiple grouping produces one summary value. In this case the total duration is calculated for each extension/date grouping. For instance, in the table shown above, the row:

---

370	901003	0.450000
-----	--------	----------

shows the total duration (0.450000) of all calls made from extension 370 on date 901003.

## Example 5

Show the number of calls made from each extension for the period of time the data was collected.

This example demonstrates the Group By clause used with the Count function.

```
cost_sum = QUERY_TABLE(phone_data, $
    'EXT, Count(NUMBER) Group By EXT')
```

The result is a two-column table that contains each extension number and a count value. The count value represents the total number of times each extension number appears in the table.

EXT	COUNT_NUMBER
0	3
248	1
289	1
311	1
320	1
331	1

```

332          1
358          1
370          2
379          2
418          1

```

The parameter specified in the *Count* function has no real effect on the result, because the function is merely counting the number of data values in the primary column (that is, null values are not ignored). You can obtain the same result with:

```

cost_sum = QUERY_TABLE(phone_data, $
    'EXT, Count(DUR) Group By EXT')

```

## Example 6

Sort the *phone\_data* table by extension, in ascending order.

This example demonstrates how the *Order By* clause is used to sort a column in a table.

```

ext_sort = QUERY_TABLE(phone_data, '* Order By EXT')

```

Here is a portion of the resulting table.

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901004	95300	1.360	JAT	0	0.00	303	5553200
901004	94700	0.800	JAT	0	0.00	303	5553200
901003	91600	2.330	JAT	0	0.00	303	5553440
901002	94800	1.310	RLD	248	0.35	617	6175551999
901002	94800	16.23	TDW	289	0.00	303	5555836
901002	93200	21.40	TAC	311	5.78	215	2155554242
901002	94700	17.44	EBH	320	4.71	214	2145559893

## Example 7

Sort the *phone\_data* table by extension, in ascending order, then by cost in descending order.

The table can be further refined by sorting the *COST* field as well.

```

cost_sort = QUERY_TABLE(phone_data, '* Order By EXT, COST DESC')

```

This command produces a table organized like the previous table, except the COST column is now sorted in descending order within each group of extensions. The following illustrates the new table organization:

DATE	TIME	DUR	INIT	EXT	COST	AREA	NUMBER
901003	91600	0.450	MLK	370	0.12	212	2125557956
901004	95100	0.160	MLK	370	0.00	303	5551245
901004	94900	1.930	SRB	379	0.52	818	8185552880
901003	91600	1.530	SRB	379	0.41	212	2125556618

### Example 8

The In operator provides another means of filtering data in a table. This operator tests for membership in a set (one-dimensional array) of values. For example, the following array contains a subset of the initials found in the INIT column of the phone\_data table:

```
nameset = ['TAC', 'KAR', 'OLL', 'ERD']
```

The following QUERY\_TABLE call produces a new table that contains information only on the members of nameset:

```
res = QUERY_TABLE(phone_data, ' * Where INIT In nameset')
```

### See Also

[BUILD\\_TABLE](#), [GROUP\\_BY](#), [ORDER\\_BY](#), [UNIQUE](#)

For more information on QUERY\_TABLE, see the *PV-WAVE User's Guide*.

---

## QUIT Procedure

Standard Library procedure that exits a PV-WAVE session.

### Usage

QUIT

### Parameters

None.

## Keywords

None.

## Discussion

QUIT simply prints a message and then calls the system routine EXIT.

## See Also

[EXIT](#)

---

**NOTE** For more information on exiting PV-WAVE, see the *PV-WAVE User's Guide*.

---

---

## ***RANDOMN Function***

Returns one or more normally distributed floating-point pseudo-random numbers with a mean of zero and a standard deviation of 1.

### **Usage**

*result* = RANDOMN(*seed* [, *dim*<sub>1</sub>, ... , *dim*<sub>n</sub>])

### **Input Parameters**

*seed* — A named variable containing the seed value for random number generation. The initial value of *seed* should be set to different values in order to obtain different random sequences. *seed* is updated by RANDOMN once for each random number generated. If *seed* is undefined, it is derived from the current system time.

*dim*<sub>*i*</sub> — (optional) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### **Returned Value**

*result* — A single scalar or an array of the specified dimensions. Contains normally distributed floating-point pseudo-random numbers with a mean of zero and a standard deviation of 1. The floating-point numbers are in the range of  $-6.0 < x < 6.0$ .

### **Keywords**

None.

### **See Also**

[RANDOMU](#)

---

## ***RANDOMU* Function**

Returns one or more uniformly distributed floating-point pseudo-random numbers over the range  $0 < Y < 1.0$ .

### **Usage**

*result* = RANDOMU(*seed* [, *dim*<sub>1</sub>, ... , *dim*<sub>n</sub>])

### **Input Parameters**

*seed* — A named variable containing the seed value for random number generation. *seed* is updated by RANDOMU once for each random number generated. The initial value of *seed* should be set to different values in order to obtain different random sequences. If *seed* is undefined, it is derived from the current system time.

*dim*<sub>*i*</sub> — (optional) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### **Returned Value**

*result* — Returns a scalar or array of pseudo-random numbers over the range  $0 < Y < 1.0$ . If no dimensions are specified, RANDOMU returns a scalar result.

### **Keywords**

None.

### **Discussion**

Uniform distribution means that, given a large enough sample of randomly-generated numbers, the same quantity of each number will be produced by RANDOMU. In other words, if you were to select a number generated by RANDOMU, you are as likely to pick any one number as another.

### **Example**

This example simulates the result of rolling two dice 10,000 times, and plots the distribution of the total using RANDOMU:

```
PLOT, HISTOGRAM(FIX(6 * RANDOMU(S, 10000)) + $
                FIX(6 * RANDOMU(S, 10000)) + 2)
```

In the above statement, the expression `RANDOMU(S, 10000)` is a 10,000-element floating-point array of random numbers greater or equal to 0 and less than 1. Multiplying this by 6 converts the range to  $0 < Y < 6$ .

Applying the `FIX` function yields 10,000-point integer vectors from 0 to 5, one less than the numbers on one die. This is done twice, once for each die, and then 2 is added to obtain a vector from 2 to 12, the total of two die.

The `HISTOGRAM` function makes a vector in which each element contains the number of occurrences of dice rolls whose total is equal to the subscript of the element.

This vector is plotted by the `PLOT` procedure.

## See Also

[RANDOMN](#)

---

## ***RDPIX Procedure***

Standard Library procedure that displays the  $x$ ,  $y$ , and pixel values at the location of the cursor in the image displayed in the currently active window.

### **Usage**

`RDPIX, image [, x0, y0]`

### **Input Parameters**

*image* — The image array loaded into the current window. May be any type. Pixel values are read from within the borders of the image; they are not read in relation to the full display area of the screen. This avoids scaling difficulties.

*x0* — The  $x$ -coordinate of the lower-left corner of the image displayed in the currently active window.

*y0* — The  $y$ -coordinate of the lower-left corner of the image displayed in the currently active window.

### **Keywords**

None.

## Discussion

The  $x$ ,  $y$ , and pixel values under the cursor position are constantly displayed and updated. Pressing the left or center button makes a new line of output, saving the old line on the display. Pressing the right mouse button exits the procedure.

## Example

```
OPENR, lun, !Data_dir + 'mandril.img', /Get_lun
mandril_img = BYTARR(512, 512)
READU, lun, mandril
    ; Read in the image file.

TV, mandril_img
    ; Display the image.

RDPIX, mandril_img
    ; Press the left or center mouse button to see pixel values, and the
    ; right mouse button to quit.

TV, mandril_img, 100, 100
    ; Display the image offset on the screen by 100 pixels vertically and
    ; horizontally.

RDPIX, mandril_img, 100, 100
    ; Read the pixel values from the offset image. Then press the left or
    ; center mouse button to see pixel values, and right mouse button to
    ; quit.
```

## See Also

[CURSOR](#)

---

## ***READ Procedures*** ***(READ, READF, READU)***

Reads input into variables:

- READ reads ASCII (formatted) input from the standard input stream (file unit 0).
- READF reads ASCII input from a specified file.
- READU reads binary (unformatted) input from a specified file. (No processing of any kind is done to the data.)

## Usage

READ,  $var_1$ , ... ,  $var_n$

READF, *unit*,  $var_1$ , ... ,  $var_n$

READU, *unit*,  $var_1$ , ... ,  $var_n$

## Input Parameters

*unit* — The file unit from which the input will be taken.

$var_i$  — The named variables to receive the input.

## Keywords

*Format* — (READ and READF only). Lets you specify the format of the input in precise detail, using a FORTRAN-style specification. FORTRAN-style formats are described in the *PV-WAVE Programmer's Guide*

If the *Format* keyword is not present, PV-WAVE uses its default rules for formatting the output. These rules are described in .

## Discussion

**READ and READF Procedures** — If the *Format* keyword is not present and READ is called with more than one parameter, and the first parameter is a scalar string starting with the characters '\$(', this initial parameter is taken to be the format specification, just as if it had been specified via the *Format* keyword. This feature is maintained for compatibility with Version 1 of PV-WAVE.

**READU Procedure** — For nonstring variables, the number of bytes required for  $var_i$  is input. For string variables, PV-WAVE reads exactly the number of bytes contained in the existing string.

### Example 1

This example reads a string from the standard input stream using the READ procedure. The value of the string is then displayed.

```
b = ' '
      ; Define a variable with string type.
READ, 'Enter a string: ', b
      Enter a string: This is a string.
      ; Read a string from the terminal.
```

```

PRINT, b
    This is a string.
    ; Display the contents of b.

```

## Example 2

In this example, three integers are read from the standard input stream into a three-element integer array, *nums*, using READ. A file named *readex.dat* is then opened for writing, and the integers in *nums* are written to the file using PRINTF. The file is then closed. The *Format* keyword is used with PRINTF to specify the format of the integers in the file. The *readex.dat* file is then opened for reading, and the integers are read into a three-element integer array using READF with the *Format* keyword. The file is then closed and the values that were read from *readex.dat* are displayed.

```

nums = INTARR(3)
    ; Create a three-element integer array.
READ, 'Enter 3 integers: ', nums
    Enter 3 integers: 3 5 7
    ; Read three integers from the standard input stream.

PRINT, nums
    3      5      7

OPENW, unit, 'readex.dat', /Get_Lun
    ; Open the readex.dat file for writing.

PRINTF, unit, nums, Format = '(3i1)'
    ; Write the integers to the file using a specified format.

FREE_LUN, unit
    ; Close the file and free the file unit number.

OPENR, unit, 'readex.dat', /Get_Lun
    ; Open the readex.dat file for reading.

ints = INTARR(3)
    ; Create a new three-element integer array.

READF, unit, ints, Format = '(3i1)'
    ; Read the three integers from the readex.dat file using the same
    ; specified format as when they were written.

PRINT, ints
    3      5      7
    ; Display the integers read from the file.

FREE_LUN, unit
    ; Close the file and free the file unit number.

```

### Example 3

In this example, READU is used to read an image of a galaxy from the file `whirlpool.img`, which is contained in the subdirectory `data` under the main PV-WAVE distribution directory.

```
OPENR, unit, FILEPATH('whirlpool.img', $
    Subdir = 'data'), /Get_Lun
    ; Open the file galaxy.dat for reading.

a = BYTARR(512, 512)
    ; Create a byte array large enough to hold the galaxy image.

READU, unit, a
    ; Read the image data.

FREE_LUN, unit
    ; Close the file and free the file unit number.
```

### See Also

[GET\\_LUN](#), [OPEN \(UNIX/OpenVMS\)](#), [OPEN \(Windows\)](#)

For more information and examples, see .

For more information on format specification codes, see the *PV-WAVE Programmer's Guide*.

---

## READ\_XBM Procedure

Reads the contents of an X-bitmap (XBM) file into a PV-WAVE variable.

### Usage

`READ_XBM, file, image`

### Input Parameters

*file* — A scalar string indicating the name of the bitmap file.

*image* — A 2D byte array containing the image from the file.

### Keywords

*Background* — Specifies the background color. (Default: !P.Background)

**Color** — Specifies the foreground color. (Default: !P.Color)

**Order** — Determines the y-scan order as follows:

- 0 Scans down from the top
- 1 Scans up from the bottom

## Discussion

The READ\_XBM procedure allows you to use XBM images in PV-WAVE for analysis and display, or to convert to other formats by using the IMAGE\_WRITE function.

## Example

```
READ_XBM, 'my.xbm', x
; Reads in a file into a 2D byte array.

WzImage, x
; Displays the image.
```

## See Also

[HTML\\_IMAGE](#), [IMAGE\\_CREATE](#), [IMAGE\\_READ](#),  
[IMAGE\\_WRITE](#), [WRITE\\_XBM](#)

---

## REBIN Function

Returns a vector or array resized to the given dimensions.

### Usage

```
result = REBIN(array, dim1, ... , dimn)
```

### Input Parameters

**array** — The array to be sampled. Cannot be of string data type. Must have the same number of dimensions as the number of dimension parameters that you supply.

**dim<sub>i</sub>** — The dimension(s) of the resampled array. Must be integral multiples or factors of the original array's dimension(s).

## Returned Value

*result* — The resized (resampled) vector or array.

## Keywords

*Sample* — If present and nonzero, specifies that nearest neighbor sampling is to be used for both magnifying and shrinking operations.

If not present, specifies that bilinear interpolation is to be used for magnifying and that neighborhood averaging is to be used for shrinking. (Bilinear interpolation gives higher quality results, but requires more time.)

## Discussion

The expansion or compression of each dimension is independent of the others; REBIN can expand or compress one dimension while leaving the others untouched.

## Example

This example creates an image of a shaded surface, resizes the image using REBIN, then displays the resized image.

```
x = DIST(20)
    ; Create a 20-by-20 single-precision, floating-point array where each
    ; element is proportional to its frequency.

SHADE_SURF, x, Color = 0
    ; Display a shaded-surface representation of x.

LOADCT, 7
y = TVRD(0, 0, 640, 512)
    ; Get the content of the display subsystem's memory.

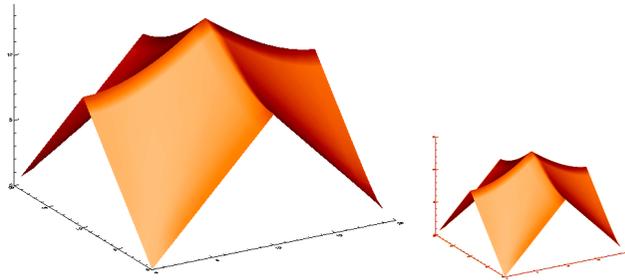
INFO, y
    VARIABLE      BYTE      = Array(640, 512)

z = REBIN(y, 320, 256)
    ; Resize the 640-by-512 array y and place the result in z.

WINDOW, 0, Xsize = 960, Ysize = 512
    ; Create window of size 960-by-512.

TV, y, 0
    ; Display larger image in position 0 of window.
```

```
TV, z, 5  
; Display resized image in position 3 of window.
```



**Figure 2-51** Original image (left); resized image (right).

## See Also

[CONGRID](#), [EXPAND](#), [INTRP](#), [REFORM](#), [REPLV](#), [RESAMP](#)

For more information on resampling and resizing images, see .

For information on interpolation methods, see .

---

## ***REFORM Function***

Reformats an array without changing its values numerically.

### **Usage**

```
result = REFORM(array, dim1, ... , dimn)
```

### **Input Parameters**

***array*** — The array that is to have its dimensions modified. Must have the same total number of elements as specified by the new dimensions.

***dim*<sub>*i*</sub>** — The dimensions of the result. Alternatively, you can specify the dimensions in a vector. See the *Example* section for more information.

### **Returned Value**

***result*** — The reformatted array.

## Keywords

None.

## Discussion

REFORM returns a copy of *array* with the dimensions specified. If no dimensions are specified, then REFORM returns a copy of *array* with all dimensions of size 1 removed. Only the dimensions of *array* are changed; the actual data remains unaltered.

---

**TIP** REFORM is useful for removing degenerate leading dimensions of size one. These leading dimensions can be created when you extract a subarray from an array with more dimensions.

---

## Example

To use REFORM to remove degenerate leading dimensions:

```
a = intarr(10,10, 10)
    ; The variable a is a 3-dimensional array.
b = a(5,*,*)
    ; Extract a "slice" from a.
INFO, b, REFORM(b)
    ; Use INFO to show what REFORM did.
```

Executing the above statements produces:

```
      B          INT =  Array(1, 10, 10)
<Expression> INT =  Array(10, 10)
```

Note that the two statements:

```
b = REFORM(a,200,5)
b = REFORM(a, [200,5])
```

have identical effect. They create a new array *b*, with dimensions of (200,5), from *a*.

## See Also

[CONGRID](#), [REBIN](#)

---

## **REGRESS Function**

Standard Library function that fits a curve to data using the multiple linear regression method.

### **Usage**

*result* = REGRESS(*x*, *y*, *wt* [, *yf*, *a0*, *sig*, *ft*, *r*, *rm*, *c*])

### **Input Parameters**

*x* — An array containing the independent values. Must be two-dimensional of size *m* by *n*, where *m* is the number of coefficients to be computed, and *n* is the number of data points.

*y* — A vector containing the dependent values. Must have *n* elements.

*wt* — A vector of weighting factors for determining the weighting of the multiple linear regression. Must have *n* elements.

### **Output Parameters**

*yf* — (optional) An array containing the calculated values of *y*. It contains *n* number of elements

*a0* — (optional) The constant term (offset) of the output function.

*sig* — (optional) A vector containing the standard deviations for the coefficients.

*ft* — (optional) The value of F in the standard F Test for the goodness of fit.

*r* — (optional) A vector containing the linear correlational coefficients.

*rm* — (optional) The multiple linear correlation coefficient.

*c* — (optional) The value of  $X^2$  in the Chi-Squared test for the goodness of fit.

### **Returned Value**

*result* — A column vector containing the coefficients (  $a_1$  to  $a_m$  ) of the function in *x*.

### **Keywords**

None.

## Discussion

REGRESS performs a multiple linear regression fit to a dataset with a specified function which is linear in the coefficients. The general function is given by:

$$f(x) = a_0 + a_1x_1 + a_2x_2 + \dots + a_mx_m$$

Weighting is useful when you want to correct for potential errors in the data you are fitting to a curve. The weighting factor, *wt*, adjusts the parameters of the curve so that the error at each point of the curve is minimized. For more information, see the section [Weighting Factor on page 180](#), in Volume 1 of this reference.

## Example

```
x = FLTARR(3, 9)
x(0, *) = [0.,1.,2.,3.,4.,10.,13.,17.,20.]
x(1, *) = [0.,3.,6.,9.,12.,15.,18.,19.,20.]
x(2, *) = [0.,4.,8.,12.,13.,14.,15.,18.,20.]
y = [5.,4.,3.,2.,2.,4.,5.,8.,9.]
    ; Create the data.

wt = FLTARR(9) + 1.0
coeff = REGRESS(x,y,wt,yf,a0,sig,ft,r,rm,c)
    ; Perform multiple linear regression with no weighting.

PLOT, yf, title='REGRESS EXAMPLE'
    ; Plot the fitted data.

PRINT, 'Fitted function:'
PRINT, ' f(x) = ',a0,' +', $
coeff(0, 0),' x1 +', $
coeff(0, 1),' x2 +', $
coeff(0, 2),' x3'
PRINT, 'Standard deviations for ' +$
    'coefficients: ', sig
PRINT, 'F Test value:', ft
PRINT, 'Linear correlation coefficients: ', r
PRINT, 'Multiple linear correlation ' + 'coefficient: ', rm
PRINT, 'Chi-squared value: ', c
    ; Print all the output parameters.
```

## See Also

[CURVEFIT](#), [GAUSSFIT](#), [POLY\\_FIT](#), [POLYFITW](#), [SVDFIT](#)

The REGRESS function is adapted from the program REGRES in *Data Reduction and Error Analysis for the Physical Sciences*, by Philip Bevington, McGraw-Hill, New York, 1969.

---

## **RENAME Procedure**

Renames a PV-WAVE variable.

### **Usage**

RENAME, *variable*, *new\_name*

### **Input Parameters**

*variable* — The variable to rename.

*new\_name* — A string specifying the name of the new variable. By default, the new variable is placed at the \$MAIN\$ program level.

### **Keywords**

*Level* — An integer,  $n$ , specifying the level of the program to which to add the renamed variable.

If  $n \geq 0$ , the level is counted from the \$MAIN\$ level to the current procedure.

If  $n < 0$ , the level count is relative, counting from the current procedure back to the \$MAIN\$ level.

### **Discussion**

If the new variable already exists at the specified program level, the existing variable is overwritten.

To rename a variable that exists at a level other than the current level, use the UPVAR command to bind that variable to a local variable.

### **Example 1**

This example is the simplest case, where a variable on the main program level is renamed.

```

orig = 1B
RENAME, orig, 'new'
INFO, new
      new      BYTE      =      1

```

## Example 2

The following program demonstrates how RENAME can be used with the *Level* keyword and the UPVAR procedure to move variables between program levels.

To run this example, follow the steps below.

1. Copy or type the code for the TESTRENAME and TESTLEVEL2 procedures into a file called `testrename.pro`.

```

PRO TESTRENAME
    ; This procedure tests the RENAME procedure first inside the local
    ; program level (level 1), and then by renaming variables on other
    ; program levels.

orig1 = 'Original Level 1 Variable'
INFO, Depth = d
    ; First, create a variable in the local program level, and verify that the
    ; program level is level 1.

RENAME, orig1, 'orig1_new', Level = d
    ; Rename the variable inside the local scope of TESTRENAME.

INFO, orig1_new
    ; Verify that RENAME created the new variable.

UPVAR, 'orig0', orig_level1
    ; Grab a variable from program level 0 ($MAIN$) using UPVAR and
    ; bind that variable to a variable within the local scope.

TESTLEVEL2
    ; Execute the TESTLEVEL2 procedure.

INFO, local_level1
    ; After returning from TESTLEVEL2, verify the existence of the variable
    ; local_level1 and print it.

PRINT, local_level1

END

PRO TESTLEVEL2
INFO, /Depth
    ; First, verify that the local scope is program level 2.

```

```

UPVAR, 'orig_level1', orig_level2, Level = -1
; Use UPVAR to pass the variable orig_level1 from program level 1
; to the current program level 2. Note the use of the Level keyword,
; which causes UPVAR to look one program level down from the current
; level to find the variable.

RENAME, orig_level2, 'new', Level = 2
; Rename the variable that was just passed into program level 2.
; The Level keyword specifies that the renamed variable be placed in
; program level 2.

INFO, new
; Verify that the new variable exists in the local scope.

local_level2 = INDGEN(10) + 222
; Simply create a new variable within the current scope (level 2).

RENAME, local_level2, 'local_level1', Level = -1
; Rename this variable, but put the renamed version on a different
; program level. The Level keyword accomplishes this. It specifies that
; the renamed variable be placed one program level down from the
; current level. The current program level is 2, so the new variable is
; placed in program level 1.

RENAME, new, 'new_main'
; Finally, use RENAME to rename a variable in the current scope (level 2)
; and place the renamed variable in program level 0 ($MAIN$). Note that
; program level 0 is where RENAME places renamed variables by default,
; unless the Level keyword is used.

END

```

**2.** At the WAVE> prompt, compile the test procedures with .RUN.

```

.RUN testrename

% Compiled module: TESTRENAME.

% Compiled module: TESTLEVEL2.

```

**3.** Next, create a main level variable (a string).

```
orig0 = 'Original Level 0 Variable'
```

**4.** Run the procedure TESTRENAME.

```

TESTRENAME

PROGRAM LEVEL          = 1
ORIG1_NEW              STRING = 'Original Level 1 Variable'
PROGRAM LEVEL          = 2
NEW                    STRING = 'Original Level 0 Variable'
LOCAL_LEVEL1          INT     = Array(10)

      222  223  224  225  226  227  228  229  230  231

```

5. Verify the existence of the variable *new\_main*.

```
INFO, new_main
    new_main          STRING      = 'Original Level 0 Variable'
```

## See Also

[ADDVAR](#), [INFO](#), [UPVAR](#)

---

## RENDER Function

Generates a ray-traced rendered image from one or more predefined objects.

### Usage

```
result = RENDER(object1, ..., objectn)
```

### Input Parameters

*object<sub>i</sub>*— A previously-defined object. Valid object types include CONE, CYLINDER, MESH, SPHERE, and VOLUME.

### Returned Value

*result* — A 2D byte array (image) of size X-by-Y.

### Keywords

*Info* — A 3-by-4 double-precision floating-point array used to return the automatically calculated view as: [viewpoint, top\_left\_viewplane, bottom\_left\_viewplane, bottom\_right\_viewplane].

For example, you could define the variable *k* to contain this default view as follows:

```
k = DBLARR(3, 4)
RENDER(object, Info=k)
```

*Lights* — A double-precision floating-point array defining the position and intensity (*x,y,z*, intensity) of all point light sources in the scene. It is of size 4-by-number\_of\_lights.

If this keyword is omitted, then a single light source is defined; this light source coincides with the automatically generated viewer's eye-point.

**Sample** — A long integer containing the number of randomly distributed rays to fire per pixel to perform anti-aliasing. The default is `Sample=1`.

**Scale** — If present, indicates that the resultant image should be scaled prior to conversion to bytes. By default, all generated shaded values are assumed to be in the range  $\{0...1\}$  (see Discussion below).

**Shadows** — If present, indicates that shadow rays should be fired so that all points on all objects are not visible to all light sources. If not present, every point in a scene is visible to each light source.

---

**TIP** For most visualization applications, you will want to omit the *Shadows* keyword, since this causes the ray tracer to run much faster.

---

**Transform** — A 4-by-4 double-precision floating-point array containing the local transformation matrix whose default is the identity matrix.

**View** — A 3-by-4 double-precision floating-point array used to override the auto-generation of the view to that specified. Uses the same format as is used for the *Info* keyword.

**X** — An integer defining the width of the byte image to be returned. Defaults to 256.

**Y** — An integer defining the height of the byte image to be returned. Defaults to 256.

## Discussion

RENDER generates an image from one or more objects using a technique called “ray tracing.” The size of the returned byte image is X-by-Y, where X and Y each default to 256 unless overridden with the *X* and *Y* keywords. The returned image can be displayed using either the TV or TVSCL procedure.

Numerous objects can be rendered in the same scene. RENDER automatically generates the viewing information such that all objects are visible and the observer's viewpoint is on the positive Z axis looking towards the origin into the scene with a slight perspective. The *Transform* or the *View* keyword can be used to alter the default view. For more information, see

The *Lights* keyword can be used to pass in an array of locations and intensities of point light sources. Except for the default light source (when none are specified by

keyword), the light sources specified are not transformed. For best results, the sum of the intensities of light sources should equal 1.

The *Scale* keyword should be used in the following cases to ensure that all objects in the resulting image are in proportionate intensity:

- if the sum of the light source intensities is greater than 1, or
- if there exists a material property in the scene such that  $Color(i) * (Kamb(i) + Kdiff(i) + Ktran(i)) > 1$ .

If all of the values are less than 1, then the *Scale* keyword is not required, but you may wish to view the resultant image using TVSCL to improve the contrast.

By default, shadow rays are turned off and thus all points on all objects are visible to all lights. The firing of shadow rays can be turned on using the *Shadows* keyword.

## Example

```
TV, RENDER (SPHERE ( ) )
```

## See Also

[CONE](#), [CYLINDER](#), [MESH](#), [RENDER24](#), [SPHERE](#), [SHADE\\_VOLUME](#), [TV](#), [TVSCL](#), [VOLUME](#)

For more information, see .

---

## **RENDER24 Function**

Standard Library function that generates a ray-traced rendered 24-bit image of m objects.

### **Usage**

```
result = RENDER24(b)
```

### **Input Parameters**

**b** — A m-element list containing the m objects to render; objects are created using the CONE, CYLINDER, MESH, or SPHERE functions. The objects must be created with default material properties since these properties are controlled with keywords (see below).

## Returned Value

*result* — (n,p,3) byte array containing a 24-bit image of the objects.

## Keywords

*c* — (m,3) array of normalized RGB color components for the objects; by default,  $c(*,*) = 1.0$

*k* — (m,3,3) array of normalized shade components for the objects:  $k(i,j,*)$  contains the ambient, reflective, and transmissive components for  $c(i,j)$ . The sum of the three components must not exceed one. The default is  $k(i,j,*) = [0.0, 1.0, 0.0]$

*v* — (3,4) array used to override the view automatically generated from !p.t. If defined, *v* works like RENDER's *View* keyword; if undefined, *v* works like RENDER's *Info* keyword.

*g* — (4,q) array giving position and intensity for q light sources; the sum of the source intensities  $g(3,*)$  must equal one. The default is a single light source at the viewer's eye

*s* — A 2-element vector specifying image size. The default is [256, 256]

## Example

```
b = LIST( SPHERE(), CYLINDER() ) & b(1).transform(3,2) = 2
c = [ [0,0], [1,0], [0,1] ] & T3D, /reset, ROTATE=[0,50,0]
TV, RENDER24(b,c=c,s=[500,500]), true=3
```

## See Also

[POLYSHADE](#), [RENDER](#)

---

## REPLICATE Function

Forms an array with the given dimensions, filled with the specified scalar value.

### Usage

```
result = REPLICATE(value, dim1, ... , dimn)
```

### Input Parameters

*value* — The scalar value used for filling the resulting array. May be of any scalar type, including scalar structures.

*dim<sub>i</sub>* — The dimensions of the result.

### Returned Value

*result* — An array with the given dimensions, filled with the specified *value*. The resulting data type is that of *value*.

### Keywords

None.

### Example

This example uses REPLICATE to create a 4-by-3 string array. Each element of the array contains the string “string”.

```
strs = REPLICATE("string", 4, 3)
INFO, strs
      STRS          STRING          = Array(4, 3)
PRINT, strs
      string  string  string  string
      string  string  string  string
      string  string  string  string
```

### See Also

[MAKE\\_ARRAY](#), [REPLV](#)

---

## **REPLV Function**

Standard Library function that replicates a vector into an array.

### **Usage**

*result* = REPLV(*vector*, *dim\_vector*, *dim*)

### **Input Parameters**

*vector* — The vector to be replicated.

*dim\_vector* — A vector specifying the dimensions of the output array.

*dim* — An integer ( $\geq 0$ ) designating the dimension to replicate.

### **Returned Value**

*result* — An array of dimensions *dim\_vector*.

### **Keywords**

None.

### **Examples**

```
PM, REPLV( [0,1], [2,4], 0 )
```

```
PM, REPLV( [0,1], [4,2], 1 )
```

```
PM, REPLV( [0,1], [4,2], 0 )
```

### **See Also**

[EXPAND](#), [REBIN](#), [REPLICATE](#)

---

## **RESAMP Function**

Standard Library function that resamples an array to new dimensions.

### **Usage**

*result* = RESAMP(*array*, *dim*<sub>1</sub>, ..., *dim*<sub>*n*</sub>)

### **Input Parameters**

*array* — An array of *n* dimensions.

*dim*<sub>*i*</sub> — Integers (>0) specifying the new dimensions.

### **Returned Value**

*result* — The resampled version of *array*.

### **Keywords**

*Interp* — If set, n-linear interpolation is used instead of the default nearest-neighbor interpolation.

### **Examples**

```
PM, RESAMP( [0,1,2,3], 3 )
```

```
PM, RESAMP( [0,1,2,3], 3, /i )
```

```
PM, RESAMP( [0,1,2,3], 6, /i )
```

```
PM, RESAMP( [[0,1,2,3],[4,5,6,7]], 6, 3, /i )
```

### **See Also**

[INTRP](#), [REBIN](#)

---

## RESTORE Procedure

Restores the PV-WAVE objects saved in a file by the SAVE procedure.

### Usage

RESTORE [, *filename*]

### Input Parameters

*filename* — The name of the file from which the PV-WAVE objects should be restored. If not specified, the `wavesave.dat` file is used.

### Keywords

*Filename* — The name of the file from which the PV-WAVE objects should be restored. If not present, `wavesave.dat` is used. This keyword serves exactly the same purpose as the *filename* parameter; only one of them needs to be provided.

*Verbose* — If present and nonzero, prints an informative message for each restored object.

### Discussion

Saving and restoring the value `-0.0` (negative float zero) is not portable between platforms. This is because of different internal implementations of that number.

### Example

```
SAVE, /All, Filename='mysave.dat'
```

```
    ; Save all local variables and system variables.
```

— User exits and then enters a new PV-WAVE session. —

```
RESTORE, 'mysave.dat'
```

```
    ; Restore all the saved variables.
```

### See Also

[COMPILE](#), [JOURNAL](#), [SAVE](#)

---

**UNIX and OpenVMS USERS** For more information, see .

---

---

## ***RETALL Procedure***

Issues RETURNS from nested routines. Used primarily to recover from errors in user-written procedures and functions.

### **Usage**

RETALL

### **Parameters**

None.

### **Keywords**

None.

### **Discussion**

RETALL issues RETURNS from nested procedures and functions until the main program level is reached. (The name RETALL is an abbreviation for RETURN ALL.)

When an error occurs in a procedure or function, control is left within that routine unless it contains special error handling instructions. Issuing the RETALL command causes control to return to the main level of PV-WAVE.

---

**NOTE** RETALL stops all currently running WAVE Widgets applications. Typing RETALL will often make “disappearing variables” reappear.

---

### **See Also**

[RETURN](#), [STOP](#)

For more information, see .

---

## ***RETURN Procedure***

Returns control to the caller of a user-written procedure or function.

### **Usage**

RETURN [, *expr*]

### **Input Parameters**

*expr* — (optional) Returns the result of the function to the caller. This parameter can only be used when RETURN is called inside a function; it cannot be used in a procedure.

### **Keywords**

None.

### **Example**

```
FUNCTION SQUARE_IT, val
  x = val * val
  RETURN, x
  ; Return the value of x to the calling program.
END
```

### **See Also**

[RETALL](#), [STOP](#)

For more information about the role of the RETURN procedure, see .

---

## REVERSE Function

Standard Library function that reverses a vector or array for a given dimension.

### Usage

```
result = REVERSE(array, [dimension])
```

### Input Parameters

**array** — The vector or array to be reversed.

**dimension** — (optional) The dimension of *array* to be reversed, such as the rows or columns. (Default: reverse rows, the first dimension).

### Returned Value

**result** — The vector or array that has been reversed.

### Keywords

None.

### Discussion

REVERSE is helpful in a variety of applications; one example is its use in obtaining perfectly symmetrical figures, such as an hourglass, by simply creating a portion of the needed figure and then making a reverse image for the rest.

---

**NOTE** Running REVERSE is equivalent to running the ROTATE function with the correct parameter.

---

### Example 1

This example exhibits the result of applying REVERSE to a 4-by-3 integer array.

```
a = INDGEN(4, 3)
    ; Create a 4-by-3 integer array. Each element has a value equal to its
    ; one-dimensional subscript.

PRINT, a
      0      1      2      3
```

```

4      5      6      7
8      9     10     11
PRINT, REVERSE(a)
3      2      1      0
7      6      5      4
11     10     9      8
      ; Reverse the rows of a.
PRINT, REVERSE(a, 2)
8      9     10     11
4      5      6      7
0      1      2      3
      ; Reverse the columns of a.
PRINT, REVERSE(REVERSE(a), 2)
11     10     9      8
7      6      5      4
3      2      1      0
      ; Reverse the columns and rows of a.

```

## Example 2

The following commands first display the image contained in the `scientist3.dat` file, and then rotate and redisplay it:

```

image1 = BYTARR(250, 200)
OPENR, 1, !Data_dir + 'scientist3.dat'
READU, 1, image1
TV, image1, 0
TV, REVERSE(image1), 1

```

## See Also

[ROTATE](#)

---

## ***REWIND Procedure (OpenVMS)***

Rewinds the tape on the designated tape unit.

### **Usage**

REWIND, *unit*

### **Input Parameters**

*unit* — A number between 0 and 9 that specifies the magnetic tape unit to rewind. (Do not confuse this parameter with file logical unit numbers.)

### **Keywords**

None.

### **See Also**

For more information, see .

---

## ***RGB\_TO\_HSV Procedure***

Standard Library procedure that converts from the RGB color system to the HSV color system.

### **Usage**

RGB\_TO\_HSV, *red, green, blue, h, s, v*

### **Input Parameters**

*red* — Red color value(s). These can be scalar or vectors, whose values are short integers in the range 0 to 255.

*green* — Green color value(s). This parameter must have the same number of elements as the *red* input parameter.

*blue* — Blue color value(s).

## Output Parameters

*h* — The resulting hue value, with the same number of elements as *red* and whose value is in the range of 0 to 360.

*s* — The corresponding saturation value in the range of 0 to 1.

*v* — The value of the corresponding value in the range 0 to 1.

## Keywords

None.

## Discussion

RGB\_TO\_HSV converts colors from the RGB (red, green, blue) color system to the HSV (hue, saturation, value) color system.

## See Also

[C\\_EDIT](#), [COLOR\\_CONVERT](#), [COLOR\\_EDIT](#), [HSV\\_TO\\_RGB](#), [LOADCT](#), [MODIFYCT](#), [TVLCT](#), [WgCeditTool](#), [WgCtTool](#)

For background information about color systems, see .

---

## ***RM Procedure***

Reads data into a one- or two-dimensional matrix.

## Usage

RM, *a* [, *rows*, *columns*]

## Input Parameters

*rows* — (optional) Number of rows in the matrix.

*columns* — (optional) Number of columns in the matrix.

## Output Parameters

*a* — Named variable into which the data is stored.

## Keywords

**Complex** — If present and nonzero, creates a single-precision complex matrix.

**Dcomplex** — If present and nonzero, creates a double-precision complex matrix.

**Double** — If present and nonzero, creates a double-precision matrix.

## Description

Procedure RM is used to read data into matrices according to the PV-WAVE matrix-storage scheme. If *rows* and *columns* are not specified, RM attempts to use the current definition of *a* to determine the number of rows and columns of *a*. If *rows* and *columns* are not specified and *a* is undefined or a scalar, an error is issued.

Upon invoking RM, the user is prompted with the row number for which input is expected. The prompt for the next row to be filled does not appear until the current row is filled. If the amount of data input for a particular row is larger than the defined number of columns of *a*, then the extra trailing input is ignored, and the prompt for the next row is given.

The matrix-printing procedures [PM](#) or [PMF](#) must be used to correctly print a matrix read in with RM.

### Example 1: Reading a Simple Matrix

This example reads a 2-by-3 matrix and prints the results using the matrix-printing procedure PM.

```
RM, a, 2, 3
row 0: 11 22 33
row 1: 40 50 60
      ; Read a 2-by-3 matrix.

PM, a
11.0000      22.0000      33.0000
40.0000      50.0000      60.0000
      ; Output the matrix.
```

### Example 2: Reading a Complex Matrix

In this example, a complex matrix is read. Notice that the elements input as integers are promoted to type complex with the value of the imaginary part set to zero.

```
RM, a, 3, 2, /Complex
row 0: (1, 0)      (1, 1)
row 1: 1           -1
```

```

row 2: (10.0, 0) (0, -10)
; Read the matrix; note that keyword Complex is set.

PM, a
( 1.00000, 0.00000) ( 1.00000, 1.00000)
( 1.00000, 0.00000) (-1.00000, 0.00000)
( 10.0000, 0.00000) ( 0.00000, -10.0000)
; Print the result.

```

### Example 3: Reading a Matrix to be Used with LUSOL

In this example, a 3-by-3 matrix and a 3-by-1 matrix are read. These matrices are then used in a call to the PV-WAVE:IMSL Mathematics Toolkit function LUSOL.

```

RM, a, 3, 3
row 0: 1 3 3
row 1: 1 3 4
row 2: 1 4 3
; Read the coefficient matrix.

RM, b, 3, 1
row 0: 1
row 1: 4
row 2: -1
; Read the right-hand side.

x = LUSOL(b, a)
; Call LUSOL to compute the solution.

PM, x
-2.00000
-2.00000
3.00000
; Output the results.

PM, a#x - b
0.00000
0.00000
0.00000

```

### See Also

[PM](#), [PMF](#), [RMF](#)

See [for more information](#).

---

## **RMF Procedure**

Reads data into a one- or two-dimensional matrix from a specified file unit.

### **Usage**

RMF, *unit*, *a* [, *rows*, *columns*]

### **Input Parameters**

*unit* — File unit from which the input is taken.

*rows* — (optional) Number of rows in the matrix.

*columns* — (optional) Number of columns in the matrix.

### **Output Parameters**

*a* — Named variable into which the data is stored.

### **Keywords**

**Complex** — If present and nonzero, creates a single-precision complex matrix.

**Dcomplex** — If present and nonzero, creates a double-precision complex matrix.

**Double** — If present and nonzero, creates a double-precision matrix.

**Format** — Scalar string specifying the precise format of the data to be read. If *Format* is not specified, PV-WAVE uses its default rules for formatting the input. The character string should start with a left parenthesis and end with a right parenthesis. For example:

```
Format = ' (f10.5) '
```

### **Description**

RMF is used to read data from a specified file unit into a matrix according to the PV-WAVE matrix-storage mode. If *rows* and *columns* are not specified, RMF attempts to use the current definition of *a* to determine the number of rows and columns of *a*. If the arguments *rows* and *columns* are not specified and *a* is undefined or a scalar, an error is issued.

The matrix-printing procedures PM or PMF must be used to correctly print a matrix read in with RMF.

### **Example**

This example reads in a 2-by-3 matrix from standard input (*unit* = -1) and prints the results using the matrix-printing procedure PM.

```
RMF, 0, a, 2, 3
  : 11 22 33
  : 40 50 60
    ; Read matrix.

PM, a
  11.0000      22.0000      33.0000
  40.0000      50.0000      60.0000
    ; Output the matrix.
```

### **See Also**

[PM](#), [PMF](#), [RM](#)

See `.for` for more information.

---

## **ROBERTS Function**

Performs a Roberts edge enhancement of an image.

### **Usage**

*result* = ROBERTS(*image*)

### **Input Parameters**

*image* — A two-dimensional array.

### **Returned Value**

*result* — A two-dimensional array of integer data type which contains the edge-enhanced image.

### **Keywords**

*Col* — Computes the column gradient (horizontal line enhancement). (Default: *Col* = 1)

---

**NOTE** For horizontal line enhancement only, you must disable the vertical line enhancement by setting the *Row* keyword to 0.

---

**Edge** — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

'zero' — Sets the border of the output image to zero. (Default)

'copy' — Copies the border of the input image to the output image.

**No\_Clip** — If set, the *result* data type is greater than the *image* data type so that overflow values in *result* are not clipped.

---

**TIP** Use the *No\_Clip* keyword to avoid overflow conditions.

---

**Return** — A scalar string specifying a mathematical function to apply. Valid strings are 'abs', 'phase', and 'value'. The *Return* keyword is used with the *Row* and *Col* keywords per the following table. (Default: 'abs')

	<i>Return</i> = 'abs'	<i>Return</i> = 'phase'	<i>Return</i> = 'value'
<i>Col</i> = 1, <i>Row</i> = 0	ABS(col grad)	Invalid Condition	column gradient
<i>Col</i> = 0, <i>Row</i> = 1	ABS(row grad)	Invalid Condition	row gradient
<i>Col</i> = 1, <i>Row</i> = 1	ABS(row grad)+ ABS(col grad)	ATAN(row grad ÷ col grad), data type is double	Invalid Condition
<i>Col</i> = 0, <i>Row</i> = 0	Invalid Condition	Invalid Condition	Invalid Condition

**Row** — Computes the row gradient (vertical line enhancement). (Default: *Row* = 1)

---

**NOTE** For vertical line enhancement only, you must disable the horizontal line enhancement by setting the *Col* keyword to 0.

---

**Same\_Type** — If set, the *result* is the same data type as *image*; otherwise, the *result* image data type is always integer.

**Zero\_Negatives** — If set, all negative values in *result* are set to zero.

## Discussion

The ROBERTS function supports multi-layer band-interleaved images. When *image* is 3D, it is treated as an array of images *array(m, n, p)*, where *p* is the number of *m*-by-*n* images. Each image in the input array is then operated on separately and an array of the *result* images is returned.

The ROBERTS function performs edge sharpening and isolation on *image*. It returns an approximation to the Roberts edge enhancement operator for images. This approximation is:

$$G_A(j,k) = |F_{j,k} - F_{j+1,k+1}| + |F_{j,k+1} - F_{j+1,k}|$$

The resulting image returned by ROBERTS has the same dimensions as the input *image*.

---

**CAUTION** Because the *result* image is saved in integer format, large original data values will cause overflow. Overflow occurs when the absolute value of the result is larger than 32,767. Use the *No\_Clip* keyword to avoid overflow.

---

## Example

This example uses the ROBERTS function to apply the Roberts edge enhancement operator to an aerial image. The final edge enhanced image is the absolute value of the difference between the original image and the image from the ROBERTS function.

```
OPENR, unit, FILEPATH('aerial_demo.img', Subdir='data'), /Get_Lun
      ; Open the file containing the image.
img = BYTARR(512, 512)
      ; Create an array large enough to hold the image.
READU, unit, img
      ; Read the image data.
FREE_LUN, unit
      ; Close the file and free the file unit number.
WINDOW, 0, Xsize = 1024, Ysize = 512
      ; Create a window large enough to hold two 512-by-512 images.
TV, img
      ; Display the original image in the left-half of the window.
HIST_EQUAL_CT, img
TV, ABS(img - ROBERTS(img)), 1
      ; Display the absolute value of the difference between the original
      ; image and the result of the ROBERTS function in the right-half of
```

; the window.



**Figure 2-52** Original image (left), and Roberts edge enhanced image (right).

## See Also

[CONVOL](#), [SHIFT](#), [SOBEL](#)

For background information, see the section *Image Sharpening* in Chapter 6 of the *PV-WAVE User's Guide*.

---

## ***ROT Function***

Standard Library function that rotates and magnifies (or demagnifies) a two-dimensional array.

### **Usage**

```
result = ROT(image, ang[, mag, xctr, yctr])
```

### **Input Parameters**

**image** — The input image to be manipulated. Can be of any data type except string. Must be two-dimensional.

**ang** — The angle of rotation in degrees clockwise.

**mag** — (optional) The magnification or demagnification factor (see *Discussion*).

**xctr** — (optional) The *x* subscript of the center of rotation. If omitted, *xctr* is equal to the number of columns in *image* divided by 2.

*yctr* — (optional) The y subscript of the center of rotation. If omitted, *yctr* is equal to the number of rows in *image* divided by 2.

## Returned Value

**result** — A rotated and magnified (or demagnified) image. The dimensions are the same as those for the input *image*.

## Keywords

**Interp** — If present and nonzero, specifies that bilinear interpolation is to be used. Otherwise, the nearest neighbor interpolation method is used.

**Missing** — Data value to substitute for pixels in the output image that map outside the input image.

## Discussion

ROT uses the POLY\_2D function to rotate and scale the input image.

The magnification factor can be of integer or floating-point data type. It is specified as follows (with 1 being the default value):

---

<i>mag</i> = 1	no change
<i>mag</i> > 1	causes magnification
<i>mag</i> < 1	causes demagnification

---

For example, if *mag* is set to 0.5, this would result in an image half the size of the original image, and if *mag* is set to 3, this would result in an image three times the size of the original.

---

**TIP** If you only need to rotate an image by 90-degree increments, the ROTATE function is more efficient to use.

---

---

**TIP** If you need a more accurate bilinear interpolation method, use the ROT\_INT function.

---

## Example

This example uses ROT to both rotate and demagnify an image. The image is rotated 135 degrees clockwise, while the demagnification factor is 0.63. Also, pixels that map outside the original image are assigned a value of 0.

```
OPENR, unit, FILEPATH('x2y2.dat', Subdir = 'data'), /Get_Lun
    ; Open the file containing the image.

image = BYTARR(320, 256)
    ; Create an array large enough to hold the image.

READU, unit, image
    ; Read the image.

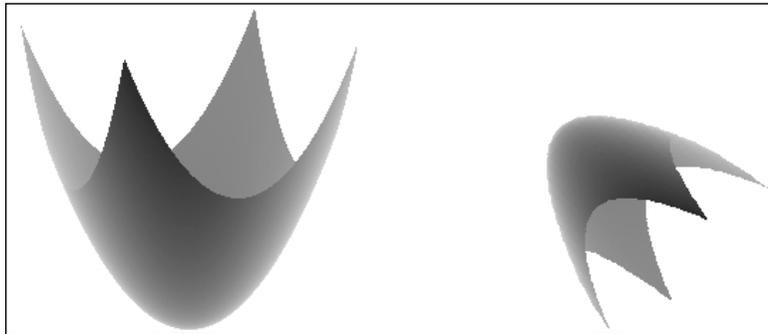
FREE_LUN, unit
    ; Close the file and free the file unit number.

WINDOW, 0, Xsize = 640, Ysize = 256
    ; Create a window large enough to contain two 320-by-256 images.

TV, image, 0
    ; Display the original image in the left-half of the window.

a = ROT(image, 135, 0.63, Missing = 0)
    ; Rotate the image 135 degrees clockwise, and demagnify it by a
    ; factor of 0.63. Also, pixels that map outside the original image are
    ; assigned a value of 0.

TV, a, 1
    ; Display rotated, demagnified image in the right-half of the window.
```



**Figure 2-53** Original image (left); rotated/demagnified image (right).

## See Also

[AFFINE](#), [POLY\\_2D](#), [ROTATE](#), [ROT\\_INT](#)

For information on interpolation methods, see .

---

## **ROTATE Function**

Returns a rotated and/or transposed copy of the input array.

### **Usage**

*result* = ROTATE(*array*, *direction*)

### **Input Parameters**

*array* — The 1D, 2D, or 3D array to be rotated.

*direction* — An integer that specifies the type of rotation to be performed, as shown below:

<b>Direction</b>	<b>Transpose</b>	<b>Rotation Clockwise</b>
0	No	None
1	No	90°
2	No	180°
3	No	270°
4	Yes	None
5	Yes	90°
6	Yes	180°
7	Yes	270°

The input parameter *direction* is taken modulo 8, so a rotation of  $-1$  is the same as 7, 9 is the same as 1, and so forth.

### **Returned Value**

*result* — A copy of *array* that has been rotated and/or transposed by 90-degree increments.

### **Keywords**

None.

## Discussion

The ROTATE function supports multi-layer band interleaved images. When the input array is three-dimensional, it is automatically treated as an array of images,  $array(m, n, p)$ , where  $p$  is the number of  $m$  by  $n$  images. Each image is then operated on separately and an array of the result images is returned.

The resulting array is of the same data type as the input *array*. The dimensions of the result are the same as those of *array* if *direction* is equal to 0 or 2; the dimensions are switched if *direction* is 1 or 3.

---

**TIP** To rotate by amounts other than multiples of 90 degrees, use the functions ROT and ROT\_INT. However, note that ROTATE is more efficient than either of those functions.

---

## Example 1

ROTATE may be used to reverse the order of elements in vectors. For example, to reverse the order of elements in the vector in variable X, use the expression:

```
ROTATE(X, 2)
```

If

```
X = [0, 1, 2, 3]
```

then

```
ROTATE(X, 2) = [3, 2, 1, 0]
```

## Example 2

This example uses ROTATE to rotate an image by 90 degrees counterclockwise and displays the image both before and after the rotation.

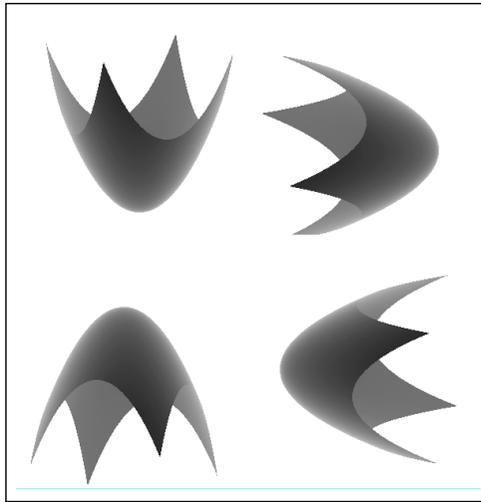
```
OPENR, unit, FILEPATH('x2y2.dat', $
    Subdir = 'data'), /Get_Lun
    ; Open the file containing the image.

img = BYTARR(320, 256)
READU, unit, img
FREE_LUN, unit
    ; Create array to hold the image, read the image, and free the LUN.

WINDOW, 0, Xsize = 640, Ysize = 640
    ; Create a window.

TV, img, 0, 321
TV, ROTATE(img, 1), 321, 257
TV, ROTATE(img, 2), 0, 0
```

```
TV, ROTATE(img, 3), 321, 0  
; Display the image, before, and after rotations.
```



**Figure 2-54** Original image and rotations.

## See Also

[AFFINE](#), [REVERSE](#), [ROT](#), [ROT\\_INT](#)

---

## ***ROT\_INT Function***

Standard Library function that rotates and magnifies (or demagnifies) an image on the display screen.

### **Usage**

```
result = ROT_INT(image, ang [, mag, xctr, yctr])
```

### **Input Parameters**

***image*** — The input image to be manipulated. Can be of any data type except string. Must be two-dimensional.

***ang*** — The angle of rotation in degrees clockwise.

***mag*** — (optional) The magnification or demagnification factor (see *Discussion*).

*xctr* — (optional) The *x* subscript of the center of rotation. If omitted, *xctr* is equal to the number of columns in *image* divided by 2.

*yctr* — (optional) The *y* subscript of the center of rotation. If omitted, *yctr* is equal to the number of rows in *image* divided by 2.

## Returned Value

*result* — A rotated and magnified (or demagnified) image. The dimensions are the same as those for the input *image*.

## Keywords

None.

## Discussion

ROT\_INT calls the function POLY\_2D to rotate and scale the input image.

The magnification factor can be of integer or floating-point data type. It is specified as follows (with 1 being the default value):

---

<i>mag</i> = 1	no change
<i>mag</i> > 1	causes magnification
<i>mag</i> < 1	causes demagnification

---

For example, if *mag* is set to 0.5, this would result in an image half the size of the original image, and if *mag* is set to 3, this would result in an image three times the size of the original.

ROT\_INT uses the bilinear interpolation method to rotate and scale the input image.

---

**TIP** If a faster (but less accurate) method of interpolation is needed, use the related function ROT, which uses a nearest neighbor method.

---

## See Also

[AFFINE](#), [POLY\\_2D](#), [ROT](#), [ROTATE](#)

For information on interpolation methods, see .



---

## **SAME Function**

Standard Library function that tests if two variables are the same.

### **Usage**

*result* = SAME(*x*, *y*)

### **Input Parameters**

*x* — A variable.

*y* — A variable.

### **Returned Value**

*result* — 1 if the two variables are the same (within keyword settings), 0 if not.

---

**NOTE** For Named Structures, the name is not tested, so {a,x:1} is same as {b,x:1}.

---

### **Keywords**

**NoType** — If set, the types of *x* and *y* are ignored. (FINDGEN(5) is same as INDGEN(5).)

**NoDim** — If set, the dimensions of the arrays are ignored. ([1,2,3,4] is same as [[1,2],[3,4]] - same as SAME(a(\*),b(\*).)

**NoVal** — If set, the values in the arrays are ignored. ([1,2,3] is same as [3,4,5].)

### **Examples**

To test for exact match:

*result* = SAME(*a*, *b*)

To test for compatible sizes:

*result* = SAME(*a*, *b*, /NoType, /NoVal)

(Replaces multiple SIZE calls.)

### **See Also**

[SIZE](#)

---

## **SAVE Procedure**

Saves variables or other specified objects in a file for later recovery by RESTORE.

### **Usage**

SAVE [, *var*<sub>1</sub>, ... , *var*<sub>*n*</sub>]

### **Input Parameters**

*var*<sub>*i*</sub> — The named variables that are to be saved.

### **Keywords**

**All** — If nonzero, specifies that everything (common blocks, system variables, local variables, compiled functions, and compiled procedures) should be saved.

**Comm** — If present and nonzero, causes all main level common block definitions to be saved.

**Filename** — The name of the file into which to save the PV-WAVE objects. If this keyword is not present, the file `wavesave.dat` is used.

**Level** — Specifies the level of the program for which data is to be saved. If  $n > 0$ , the level is counted from the \$MAIN\$ level to the current procedure. If  $n < 0$  the level count is relative, counting from the current procedure back to the \$MAIN\$ level. (Default: 0 — \$MAIN\$).

**Routines** — If present and nonzero, saves all procedures and functions that are currently compiled in memory.

**System\_Variables** — If present and nonzero, saves all system variables.

**Variables** — If present and nonzero, saves all current local variables to.

**Verbose** — If present and nonzero, prints an informative message for each saved object.

**Wavepoint** — If present and nonzero, saves PV-WAVE variables so that they can be read into PV-WAVE Point & Click and PV-WAVE Personal Edition. This keyword is disabled for the Digital Alpha Digital UNIX platform.

**XDR** — If present and nonzero, causes the save file to be written in a portable format using XDR (eXternal Data Representation).

---

**UNIX USERS** Under UNIX, XDR is the only supported format, so specifying this keyword is unnecessary.

---

---

**OpenVMS USERS** Under OpenVMS, XDR is used to interchange data with other versions of PV-WAVE. The default is a VAX-specific format that is more efficient to process.

---

## Discussion

Saving and restoring the value  $-0.0$  (negative float zero) is not portable between platforms. This is because of different internal implementations of that number.

## See Also

[COMPILE](#), [JOURNAL](#), [RESTORE](#)

---

**UNIX and OpenVMS USERS** For more information, see

---

---

## ***SCALE3D Procedure***

Standard Library procedure that scales a three-dimensional unit cube into the viewing area.

### **Usage**

SCALE3D

### **Parameters**

None.

### **Keywords**

None.

## Discussion

SCALE3D is useful for certain forms of perspective transformation, although it does not work for all forms.

SCALE3D does not use explicit parameters, but rather modifies the system variable !P.T for use as the implicit input and output parameters. Eight three-dimensional data points are created at the vertices of the three-dimensional unit cube. These eight points are transformed by !P.T. The system is translated to bring the minimum (X, Y, Z) point to the origin, and then scaled to make each coordinate's maximum value equal to 1.

## Example

For an example, see .

## See Also

System Variables: [!P.T](#)

---

## SEC\_TO\_DT Function

Converts any number of seconds into date/time values.

### Usage

*result* = SEC\_TO\_DT(*num\_of\_seconds*)

### Input Parameters

*num\_of\_seconds* — A scalar representing the number of seconds elapsed from the date specified in the system variable !DT\_Base.

### Returned Value

*result* — A date/time variable containing the converted values.

### Keywords

*Base* — A string containing a date, such as “3-27-92”. This is the base date from which the number of seconds is calculated. The default value for *Base* is taken from the system variable !DT\_Base.

*Date\_Fmt* — Specifies the format of the base date, if passed into the function. Possible values are 1, 2, 3, 4, or 5, as summarized in the following table:

Value	Format Description	Examples for May 1, 1992
1	MM*DD*[YY]YY	05/01/92
2	DD*MM*[YY]YY	01-05-92
3	ddd*[YY]YY	122,1992
4	DD*mmm[mmmmmm]*[YY]YY	01/May/92
5	[YY]YY*MM*DD	1992-05-01

where the asterisk (\*) represents one of the following separators: dash (-), slash (/), comma (,), period (.), or colon (:).

For a description of these formats, see .

## Discussion

This function is useful for converting time stamps that count seconds to date/time values. Most time stamps count seconds from an arbitrary base date. For example, the UNIX time stamp counts seconds from January 1, 1970.

### Example 1

This example shows how a value of 20 seconds is represented internally inside a date/time variable after it has been converted with SEC\_TO\_DT. The example uses a base start date of January 1, 1970.

```
date = SEC_TO_DT(20, Base='1-1-70', $
    Date_Fmt=1)
PRINT, date
    { 1970 1 1 0 0 20.0000 79367.000 0 }
```

### Example 2

Assume you have the following dataset which contains time stamps and associated measurements. The file contains data collected from January 1, 1990 to January 5, 1990. The base date for the clock is January 1, 1970.

```
6.3119520e+08    113
6.3128160e+08    768
6.3136800e+08    632
6.3145440e+08    227
6.3154080e+08    224
```

Assume the above file has been read into two variables. The first column is read into a double-precision array called `tarray`. The second column is read into an array called `fluid_level`, which indicates the water level of a lake for each

specified time period. You can convert the date/time data in Column 1 to date/time data with the `SEC_TO_DT` function:

```
dtarray = SEC_TO_DT(tarray, Base='1-1-70')
PRINT, dtarray
{ 1990 1 1 12 0 0.00000 86672.500 0}
{ 1990 1 2 12 0 0.00000 86673.500 0}
{ 1990 1 3 12 0 0.00000 86674.500 0}
{ 1990 1 4 12 0 0.00000 86675.500 0}
{ 1990 1 5 12 0 0.00000 86676.500 0}
```

Notice the `SEC_TO_DT` function creates an array containing a date/time structure for each of the seconds values. The Julian day shown for each date/time is automatically adjusted to reflect the system base date of September 14, 1752.

## See Also

[DT\\_TO\\_SEC](#), [JUL\\_TO\\_DT](#), [STR\\_TO\\_DT](#), [VAR\\_TO\\_DT](#)

System Variables: [!DT\\_Base](#)

For more information, see Chapter 8, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

---

## ***SELECT\_READ\_LUN Procedure (UNIX)***

Waits for input on any of a list of logical unit numbers.

### **Usage**

`SELECT_READ_LUN, luns`

### **Input Parameters**

*luns* — A vector of logical unit numbers.

### **Output Parameters**

*luns* — A vector of logical unit numbers.

### **Keywords**

*Timeout* — Amount of time, in seconds, this procedure should block while waiting for input. This may be a floating-point number. The default is to block forever.

**Widget** — If present and nonzero, selects the X11 socket. If there is no X11 input, then the value of this keyword is set to -1.

## Description

This procedure checks for input on all the logical unit numbers specified in the *luns* vector. When it returns, those unit numbers *without* input are set to -1. By default, this procedure blocks forever waiting for input. Keyword *Timeout* can be used to reduce the time the procedure blocks. This procedure always returns when any one of the listed LUNs has input to be read.

This procedure is an interface to the `select_read_lun` C routine in the `io.so` shared library.

## See Also

[ADD\\_EXEC\\_ON\\_SELECT](#), [DROP\\_EXEC\\_ON\\_SELECT](#),  
[EXEC\\_ON\\_SELECT](#)

---

## **SETDEMO Procedure**

Standard Library procedure that defines key bindings and system variables to run the PV-WAVE demonstration system for the system on which PV-WAVE is started.

## Usage

SETDEMO

## Parameters

None.

## Keywords

None.

## Discussion

SETDEMO is called when PV-WAVE is started, and defines the key bindings and system variables used to run the demonstration system. The message that appears at the start of a session identifying the key bindings is written by this procedure.

---

**NOTE** Some key definitions may not be possible on all platforms.

---

**UNIX and OpenVMS USERS** The SETDEMO procedure is called by the PV-WAVE startup file:

(UNIX)      <wavedir>/bin/wavestartup

(OpenVMS) <wavedir>: [000000.bin]wavestartup.dat

Where <wavedir> is the main PV-WAVE directory.

---

During startup, SETDEMO sets up the key bindings for the PV-WAVE demonstration, accesses the Help system, outputs the PV-WAVE status, and creates a subprocess on your system. It then displays a message indicating that the key bindings were set.

You can use SETDEMO to customize your keys for commonly used commands (through the DEFINE\_KEY routine), to change the message that is printed to the screen when you invoke PV-WAVE, or to change the default key bindings.

## See Also

[DEFINE\\_KEY](#), [SETUP\\_KEYS](#)

---

## **SETENV Procedure (UNIX/Windows)**

Adds or changes an environment string in the process environment.

### **Usage**

SETENV, *environment\_expr*

### **Input Parameters**

*environment\_expr* — A scalar string containing an environment expression to be added to the environment.

### **Keywords**

None.

### **UNIX Example**

```
SETENV, 'SHELL=/bin/sh'
```

### **Windows Example**

To change the HOMEPATH environment variable to point to the directory  
D:\users\chris\utah\_data:

```
r = GETENV('HOMEPATH')
PRINT, r
    D:\users\chris
SETENV, 'HOMEPATH=D:\users\chris\utah_data'
r = GETENV('HOMEPATH')
PRINT, r
    D:\users\chris\utah_data
```

### **See Also**

[ENVIRONMENT](#), [GETENV](#)

---

## SETLOG Procedure (OpenVMS)

Defines a logical name.

### Usage

SETLOG, *logname*, *value*

### Input Parameters

*logname* — A scalar string containing the name of the logical to be defined.

*value* — A string giving the value to which *logname* will be set. If *value* is a string array, *logname* is defined as a multi-valued logical where each element of *value* defines one of the equivalence strings.

### Keywords

*Concealed* — If set, RMS interprets the equivalence name as a device name.

*Confine* — If set, prevents *logname* from being copied from the PV-WAVE process to its spawned subprocesses.

*No\_Alias* — If set, prevents *logname* from being duplicated in the same logical table at an outer access mode. If another logical name with the same name already exists at an outer access mode, it is deleted.

*Table* — A scalar string giving the name of the logical table from which to delete *logname*. If this keyword is not present, the table LNM\$PROCESS\_TABLE is used.

*Terminal* — If set, prevents further iterative logical name translation on the equivalence name from being performed when SETLOG attempts to translate *logname*.

### See Also

[DELETE\\_SYMBOL](#), [DELLOG](#), [GET\\_SYMBOL](#), [SET\\_SYMBOL](#),  
[TRNLOG](#)

For more information on logical names and access modes, see the *VAX/OpenVMS DCL Dictionary*. Information oriented to C programmers can be found in the *OpenVMS System Services Manual*.

---

## **SETNCOPTS Procedure**

Sets the value of the *ncopts* variable and defines the level of error reporting for the NetCDF functions as discussed in the error section of the *NetCDF User's Guide*.

### **Usage**

SETNCOPTS, *new\_ncopts*

### **Input Parameters**

*new\_ncopts* — The new value for the *ncopts* variable. Valid values for the *ncopts* variable are:

0 (zero) — No error messages will be reported.

NC\_FATAL — No error messages will be reported and all errors will be fatal.

NC\_VERBOSE — Error messages will be reported.

NC\_FATAL+NC\_VERBOSE — Error messages will be reported and all errors will be fatal.

NC\_VERBOSE and NC\_FATAL are defined in HDF\_COMMON and HDF\_INIT.

### **Keywords**

*Help* — List the usage for this function.

*Usage* — List the usage for this function. Same as the Help keyword.

### **Discussion**

SETNCOPTS sets the value of the “ncopts” variable thus defining the level of error reporting by the netCDF functions as discussed in the Error Handling section of the *NetCDF User's Guide*.

SETNCOPTS is only valid for the netCDF functionality.

Fatal errors will cause PV-WAVE to exit.

### **Example**

```
SETNCOPTS, 0
ncid = NCOPEX("foo.nc", NC_NOWRITE)
```

```
status = NCCLOSE(ncid)
status = NCREDEF(ncid)
    % NCREDEF: error in HDF return status.
SETNCOPTS, NC_VERBOSE
ncid = NCOPEX("foo.nc", NC_NOWRITE)
status = NCCLOSE(ncid)
status = NCREDEF(ncid)
    ncredef: 0 is not a valid cdfid % NCREDEF: error in HDF return
    status.
```

## See Also

[GETNCERR](#), [GETNCOPTS](#)

Also refer to the *NetCDF User's Guide*.

For more information on using the HDF interface and the calling sequence for the entire suite of HDF base functions, refer to [Appendix A, The PV-WAVE HDF Interface](#).

For a complete list of the HDF convenience routines, refer to [Chapter 1, Functional Summary of Routines](#).

---

## SET\_PLOT Procedure

Specifies the device type used by PV-WAVE graphics procedures.

### Usage

SET\_PLOT, *device*

### Input Parameters

*device* — A scalar string giving the name of the device to use. This parameter is case-insensitive.

### Keywords

*Copy* — If present and nonzero, PV-WAVE's internal color table is copied into the device. This is the preferred method if you are displaying graphics and each color index is explicitly loaded.

**Interpolate** — If present and nonzero, the color table of the new device is loaded by interpolating the old color table to span the new number of color indices. This method works best when displaying images with continuous color ranges.

## Discussion

The following are valid device names. These devices are explained in detail in [Appendix B, Output Devices and Window Systems](#).

### PV-WAVE Device Drivers

Name	Device
CGM	Computer Graphics Metafile format
HP	HPGL device
PCL	PCL device
PM	Pixel Map
PS	PostScript device
REGIS	REGIS terminal
TEK	Tektronix terminal
WIN32	Windows NT
WMF	Windows Metafile
X	X Window System
Z	Z-buffer pseudo device

If the *Copy* keyword parameter is set, the color table copying is straightforward as long as both devices have the same number of color indices. However, if the new device has more colors than the old device, some color indices will be invalid. If the new device has less colors than the old, not all the colors are saved.

## Examples

```
SET_PLOT, 'ps'  
; Send output to a PostScript file.
```

```
SET_PLOT, 'cgm'  
; Send output to a CGM file.
```

## See Also

[DEVICE](#)

System Variables: [!D](#)

---

## **SET\_SCREEN Procedure**

Standard Library procedure that establishes a new position for the rectangular plot area based on input values specified using the device coordinate system.

### **Usage**

SET\_SCREEN, *xmin*, *xmax* [, *ymin*, *ymax*]

### **Input Parameters**

*xmin* — The position of the left edge of the rectangular plot area in device coordinates.

*xmax* — The position of the right edge of the rectangular plot area in device coordinates.

*ymin* — The position of the bottom edge of the rectangular plot area in device coordinates.

*ymax* — The position of the top edge of the rectangular plot area in device coordinates.

### **Keywords**

*Cursor* — Lets you set the region for the rectangular plot area interactively with the mouse:

- If nonzero, lets you set the boundaries by clicking the mouse at the corners of the rectangular plot area that you want to use. Click first to set the lower-left corner, then once again to set the upper-right corner. Any input parameters for SET\_SCREEN are ignored.
- If zero, uses the input parameters as specified.

*Region* — If present, uses the system variable !P.Region to set the plot area. The default is to use !P.Position.

### **Discussion**

SET\_SCREEN is used to set up the area of the display that will be used for plotting. It is identical to the SET\_VIEWPORT procedure, except that the input parameters are in device coordinates for SET\_SCREEN.

If only *xmin* and *xmax* are provided, the values for *ymin* and *ymax* are calculated. Calling SET\_SCREEN with four input parameters is the same as setting the !P.Position system variable or using the *Position* keyword with the plotting commands.

SET\_SCREEN overrides the effect of the system variables !X.Margin and !Y.Margin.

---

**TIP** To find out the current values of your device, type one of the following two commands:

```
INFO, /Structure, !D
PRINT, !D.X_Size, !D.Y_Size
```

---

## Example

```
INFO, /Structure, !D
SET_SCREEN, !D.X_Size/10., !D.X_Size/2.
SET_SCREEN, !D.X_Size/10., !D.X_Size/2.1, $
    !D.Y_Size/1.7, !D.Y_Size/1.1
PLOT, [3, 4, 5], Title='Upper left plot'
SET_SCREEN, !D.X_Size/10., !D.X_Size/2.1, $
    !D.Y_Size/10., !D.Y_Size/2.3
PLOT, [5, 1, 6], Title='Lower left plot', $
    /Noerase
SET_SCREEN, !D.X_Size/1.8, !D.X_Size/1.1, $
    !D.Y_Size/1.7, !D.Y_Size/1.1
PLOT, [3, 4, 5], Title='Upper right plot', $
    /Noerase
SET_SCREEN, !D_X_Size/1.8, !D_X_Size/1.1, $
    !D_Y_Size/10., !D_Y_Size/2.3
PLOT, [2, 4, 3], title='Lower right plot', /Noerase
SET_SCREEN, 1, 1, 1, 1, /Cursor
    ; Click the mouse button on the lower left corner of the desired
    ; plotting area and then again for the desired upper right corner.
PLOT, [3, 5, 4]
SET_SCREEN, !D_X_Size/10., !D_X_Size/1.1, $
    !D_Y_Size/10., !D_Y_Size/1.1
PLOT, [3, 4, 5], $
    Title='Without setting the region keyword'
```

```
SET_SCREEN, !D_X_Size/10., !D_X_Size/1.1, $
           !D_Y_Size/10., !D_Y_Size/1.1, Region
PLOT, [3, 4, 5], Title='With the region keyword set', /Noerase
```

## See Also

[SET\\_VIEWPORT](#)

System Variables: [!P.Position](#), [!P.Region](#)

---

## **SET\_SHADING Procedure**

Modifies the light source shading parameters affecting the output of SHADE\_SURF and POLYSHADE.

### Usage

SET\_SHADING

### Input Parameters

None.

### Keywords

**Gouraud** — Controls the method of shading the surface polygons of the POLYSHADE procedure:

- If set to nonzero (the default), the Gouraud shading method is used.
- Otherwise, each polygon is shaded with a constant intensity.

Gouraud shading interpolates intensities from each vertex along each edge. Then, when scan converting the polygons, the shading is interpolated along each scan line from the edge intensities. Gouraud shading is slower than constant shading, but usually results in a more realistic appearance.

**Light** — A 3-element vector specifying the direction of the light source. The default light source vector is [0, 0, 1], with the light rays parallel to the Z axis.

**Reject** — If set (the default), causes polygons to be rejected as being hidden if their vertices are ordered in a clockwise direction as seen by the viewer.

- You should always set *Reject* when rendering enclosed solids whose original vertex lists are in counterclockwise order.
- You may also choose to set *Reject* when rendering surfaces that are not closed or are not in counterclockwise order, although this may cause shading anomalies at boundaries between visible and hidden surfaces to occur.

## Discussion

SET\_SHADING keywords let you control the light source direction, shading method, and the rejection of hidden surfaces. SET\_SHADING first resets its keywords to their default values. The values specified in the call then overwrite the default values.

## Example

This example creates a spherical volume dataset and then renders two isosurfaces from that dataset. The first isosurface does not use the Gouraud method of shading but instead shades each polygon with a constant intensity. This is achieved by using SET\_SHADING with the *Gouraud* keyword set to 0. The second isosurface uses the Gouraud method of shading, which is achieved by using SET\_SHADING with the *Gouraud* keyword set to 1.

```
sphere = FLTARR(20, 20, 20)
    ; Create a three-dimensional single precision, floating-point array.

FOR x = 0, 19 DO FOR y = 0, 19 DO FOR $
    z = 0, 19 DO sphere(x, y, z) = $
    SQRT((x-10)^2 + (y-10)^2 + (z-10)^2)
    ; Create the spherical volume dataset.

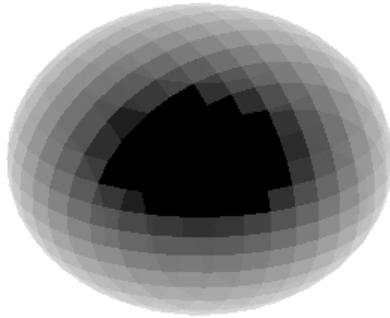
SHADE_VOLUME, sphere, 7, v, p
    ; Find the vertices and polygons at a contour level of 7.

SURFACE, FLTARR(2, 2), /Nodata, /Save, $
    Xrange = [0, 20], Yrange = [0, 20], $
    Zrange = [0, 20], Xstyle = 4, Ystyle = 4, $
    Zstyle = 4
    ; Set up an appropriate three-dimensional transformation.

SET_SHADING, Gouraud = 0
    ; Turn Gouraud shading off.

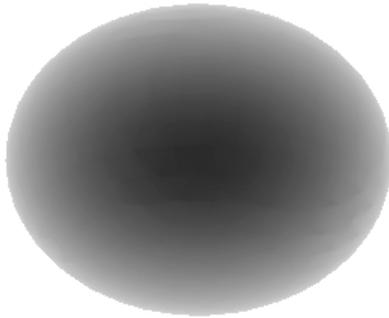
image = POLYSHADE(v, p, /T3d)
    ; Render the image.

TV, image
    ; Display the image.
```



**Figure 2-55** Spherical isosurface with constant intensity shading.

```
SET_SHADING, Gouraud = 1
    ; Turn Gouraud shading on.
image = POLYSHADE(v, p, /T3d)
    ; Render the image.
TV, image
    ; Display the image.
```



**Figure 2-56** Spherical isosurface with Gouraud shading.

## See Also

[POLYSHADE](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#)

---

## **SET\_SYMBOL Procedure (OpenVMS)**

(OpenVMS Only) Defines a DCL interpreter symbol for the current process.

### **Usage**

SET\_SYMBOL, *name*, *value*

### **Input Parameters**

*name* — A scalar string containing the name of the symbol to be defined.

*value* — A scalar string containing the value with which *name* will be defined.

### **Keywords**

*Type* — Indicates the OpenVMS table into which *name* will be placed:

- 1 Specifies the local symbol table (the default).
- 2 Specifies the global symbol table.

### **See Also**

[DELETE\\_SYMBOL](#), [DELLOG](#), [GET\\_SYMBOL](#), [SETLOG](#), [TRNLOG](#)

For more information, see the *VAX/OpenVMS DCL Dictionary*. Information oriented to C programmers can be found in the *OpenVMS System Services Manual*.

---

## **SETUP\_KEYS Procedure**

Standard Library procedure that sets up the functions keys for keyboards known to PV-WAVE.

### **Usage**

SETUP\_KEYS

### **Parameters**

None.

## Keywords

---

**Windows USERS** All of the keywords described below work only under UNIX and OpenVMS. This procedure does not take any keywords under Windows.

---

***App\_Keypad*** — Specifies the escape sequences for the group of keys in the numeric keypad, enabling these keys to be program-med within PV-WAVE.

***Eightbit*** — Indicates that the 8-bit versions of the escape codes should be used instead of the default 7-bit versions when the VT200 function key definitions are entered. *Eightbit* is only valid if the *VT200* keyword is also used.

***HP9000*** — Specifies that function key definitions for an HP-9000 Series 300 keyboard should be established.

The upper right-hand group of four keys (at the same height as the function keys) are called <BLANK1> through <BLANK4>, since they have no written labels.

Keys defined to have labels beginning with a capital <K> belong to the numeric keypad group; for example, <K9> refers to keypad key <9>.

***Mips*** — Specifies that function key definitions for a MIPS RS series keyboard should be established.

***Num\_Keypad*** — Disables programmability of the numeric keypad.

***Sun*** — Specifies that function key definitions for a Sun-4 keyboard should be established.

***VT200*** — Specifies that function key definitions for a VT200 keyboard should be established.

## Discussion

You can examine the function key definitions by running the `SETUP_KEYS` procedure and then typing `INFO, /Keys`. To change a key definition or add a new key definition, use the `DEFINE_KEY` procedure.

## Example

```
SETUP_KEYS, /Sun
; Establishes function key definitions for a Sun-4 keyboard.
```

## See Also

[DEFINE\\_KEY](#), [SETDEMO](#)

System Variables: [!Version](#)

---

## SET\_VIEW3D Procedure

Generates a 3D view, given a view position and a view direction.

### Usage

SET\_VIEW3D, *viewpoint*, *viewvector*, *perspective*, *izoom*, *viewup*, *viewcenter*,  
*winx*, *winy*, *xr*, *yr*, *zr*

### Input Parameters

***viewpoint*** — A three-element vector containing the point from which to view the data in data coordinates.

***viewvector*** — A three-element vector containing the direction to look.

***perspective*** — The perspective projection distance. The smaller the projection distance, the more “severe” the projection is. The larger the projection distance, the more “isometric” the projection is.

---

**TIP** To prevent a perspective projection, set *perspective* to 0 (or less than zero).

---

***izoom*** — The magnification factor for the projection.

***viewup*** — A two-element vector containing the final 2D view up vector. For a “right-side-up” view, set this parameter to [0.0, 1.0].

***viewcenter*** — A two-element vector containing the window location on which to place the viewpoint. It is in normal coordinates and is usually set to [0.5, 0.5].

***winx*, *winy*** — The *x* and *y* dimensions, respectively, of the plot window in device coordinates. Typically, *winx* and *winy* are set to the X and Y size of the current graphics window.

***xr*, *yr*, *zr*** — Two-element vectors containing the minimum and maximum *x*, *y*, and *z* values, respectively, found in the data to be plotted. The minimum value is in *xr(0)*, *yr(0)*, and *zr(0)*; the maximum value in *xr(1)*, *yr(1)*, and *zr(1)*.

## Keywords

None.

## Discussion

SET\_VIEW3D creates a view transformation that preserves the correct aspect ratio of the data, even if the plot window is non-square.

SET\_VIEW3D changes the system viewing matrix !P.T, as well as the system variables, !X.S, !Y.S, and !Z.S, which handle conversion from data coordinates to normal coordinates. (These system variables are described in [Chapter 4, System Variables](#).)

## Examples

See the *Examples* section in the description of the [POLY\\_DEV](#) routine.

## See Also

[CENTER\\_VIEW](#)

---

## **SET\_VIEWPORT Procedure**

Standard Library procedure that establishes a new position for the rectangular plot area based on input values specified using the normalized coordinate system.

### Usage

SET\_VIEWPORT, *xmin*, *xmax* [, *ymin*, *ymax*]

### Input Parameters

*xmin* — The position of the left edge of the rectangular plot area in normal coordinates.

*xmax* — The position of the right edge of the rectangular plot area in normal coordinates.

*ymin* — (optional) The position of the bottom edge of the rectangular plot area in normal coordinates.

*ymax* — (optional) The position of the top edge of the rectangular plot area in normal coordinates.

## Keywords

*Cursor* — Lets you set the region for the rectangular plot area interactively with the mouse:

- If nonzero, lets you set the boundaries by clicking the mouse at the corners of the rectangular plot area that you want to use. Click first to set the lower-left corner, then once again to set the upper-right corner. Any input parameters for SET\_VIEWPORT are ignored.
- If zero, uses the input parameters as specified.

*Region* — If present, uses the system variable !P.Region to set the plot area. The default is to use !P.Position.

## Discussion

SET\_VIEWPORT is used to set up the area of the display that will be used for plotting. It is identical to the SET\_SCREEN procedure, except that the input parameters are in normalized coordinates for SET\_VIEWPORT.

If only *xmin* and *xmax* are provided, the values for *ymin* and *ymax* are calculated. Calling SET\_VIEWPORT with four input parameters is the same as setting the !P.Position system variable or using the *Position* keyword with the plotting commands.

SET\_VIEWPORT overrides the effect of the system variables !X.Margin and !Y.Margin.

## Example

```
SET_VIEWPORT, .2, .5
PLOT, [3, 5, 2]
SET_VIEWPORT, .1, .45, .55, .90
PLOT, [3, 4, 5], Title='Upper left plot'
SET_VIEWPORT, .1, .45, .1, .45
PLOT, [5, 1, 6], Title='Lower left plot', /Noerase
SET_VIEWPORT, .55, .90, .55, .90
PLOT, [3, 4, 5], Title='Upper right plot', /Noerase
```

```
SET_VIEWPORT, .55, .90, .1, .45
PLOT, [2, 4, 3], Title='Lower right plot', /Noerase
SET_VIEWPORT, 1, 1, 1, 1, /Cursor
    ; Click with the mouse button on the lower left corner of the desired
    ; plotting area and then again for the desired upper right corner.
PLOT, [3, 5, 4]
SET_VIEWPORT, .2, .8, .2, .8
PLOT, [3, 4, 5], Title='Without setting the region keyword'
SET_VIEWPORT, .2, .8, .2, .8, /Region
PLOT, [3, 4, 5], Title='With the region keyword set', /Noerase
```

## See Also

[SET\\_SCREEN](#)

System Variables: [!P.Position](#), [!P.Region](#)

---

## SET\_XY Procedure

Standard Library procedure that sets the default data axis range for either the  $x$ - or  $y$ -axis. It should only be used for emulating Version 1 of PV-WAVE.

### Usage

```
SET_XY, xmin, xmax [, ymin, ymax]
```

### Input Parameters

***xmin*** — The minimum default value for the  $x$ -axis.

***xmax*** — The maximum default value for the  $x$ -axis.

***ymin*** — (optional) The minimum default value for the  $y$ -axis.

***ymax*** — (optional) The maximum default value for the  $y$ -axis.

### Keywords

None.

## Discussion

SET\_XY is used for setting the range of data values that will be drawn in the rectangular display window. (Normally the axis range is determined automatically by scanning the data.)

Note that you should only use SET\_XY if you are trying to emulate Version 1 of PV-WAVE. With more recent versions of PV-WAVE, you should use the system variables !X.Range and !Y.Range, or their corresponding keywords.

---

**NOTE** SET\_XY sets the Range, Crange, and S (scaling) fields of the system variables !X and !Y. This may cause subsequent plots of otherwise familiar data to appear skewed.

---

## Example

```
PLOT, [12, 26, 35, 44], [50, 43, 89, 70]
PLOT, [12, 26, 35, 44], [50, 43, 89, 70], $
      XRange=[10, 50], YRange=[40, 100]
SET_XY, 10, 50, 40, 100
PLOT, [12, 26, 35, 44], [50, 43, 89, 70]
```

## See Also

[AXIS](#), [PLOT](#), [SET\\_VIEWPORT](#)

Graphics and Plotting Keywords: [\[XY\]Range](#), [\[XY\]Style](#)

System Variables: [!\[XY\].Range](#), [!\[XY\].Style](#)

For more information, see .

---

## ***SGN Function***

Standard Library function that returns the sign of passed values.

### **Usage**

*result* = SGN(*x*)

### **Input Parameters**

*x* — A scalar or array.

### **Returned Value**

*result* — An integer array of the same size of *x* where each element is:

1 where  $x > 0$

-1 where  $x < 0$

0 where  $x = 0$

### **Keywords**

None.

### **Examples**

PM, SGN( [-0.5, 0, 0.5] )

---

## **SHADE\_SURF Procedure**

Creates a shaded surface representation of a regular or nearly regular gridded surface, with shading from either a light source model or from a specified array of intensities.

### **Usage**

SHADE\_SURF,  $z$  [,  $x$ ,  $y$ ]

### **Input Parameters**

$z$  — A two-dimensional array containing the values that make up the surface. If  $x$  and  $y$  are supplied, the surface is plotted as a function of the X,Y locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of  $z$ .

$x$  — A vector or two-dimensional array specifying the  $x$ -coordinates for the contour surface. If  $x$  is a vector, each element of  $x$  specifies the  $x$ -coordinate for a column of  $z$ .

For example,  $x(0)$  specifies the  $x$ -coordinate for  $z(0, *)$ . If  $x$  is a two-dimensional array, each element of  $x$  specifies the  $x$ -coordinate of the corresponding point in  $z$  ( $x_{ij}$  specifies the  $x$ -coordinate for  $z_{ij}$ ).

$y$  — A vector or two-dimensional array specifying the  $y$ -coordinates for the contour surface. If a vector, each element of  $y$  specifies the  $y$ -coordinate for a row of  $z$ .

For example,  $y(0)$  specifies the  $y$ -coordinate for  $z(*, 0)$ . If  $y$  is a two-dimensional array, each element of  $y$  specifies the  $y$ -coordinate of the corresponding point in  $z$  ( $y_{ij}$  specifies the  $y$ -coordinate for  $z_{ij}$ ).

### **Keywords**

**Image** — The name of a variable into which the image containing the shaded surface is stored. If this keyword is omitted, the image is displayed but not saved.

**Max\_Img\_Size** — For devices with scalable pixels (e.g., postscript), this keyword sets the largest allowed image size created internally to render the shaded surface. Larger values will result in a better quality image but at a cost of greater memory use and larger file size. The minimum value is 100 (100x100), which will generally result in a poor-quality image. (Default: 400)

*Shades* — An array expression, of the same dimensions as *z*, containing the color index at each point. The shading of each pixel is interpolated from the surrounding *Shades* values. For most displays, this parameter should be scaled into the range of bytes. If this keyword is omitted, light source shading is used.

Other keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

Ax	Noclip	[XYZ]Charsize
Az	Nodata	[XYZ]Gridstyle
Background	Noerase	[XYZ]Margin
Channel	Normal	[XYZ]Minor
Charsize	Position	[XYZ]Range
Charthick	Save	[XYZ]Style
Clip	Subtitle	[XYZ]Tickformat
Color	T3d	[XYZ]Ticklen
Data	Thick	[XYZ]Tickname
Device	Tickformat	[XYZ]Ticks
Font	Ticklen	[XYZ]Tickv
Gridstyle	Title	[XYZ]Title
		ZValue

---

## Discussion

SHADE\_SURF is similar to the SURFACE procedure. Given a regular or near-regular grid of elevations, SHADE\_SURF produces a shaded surface representation of the data with the hidden surfaces removed.

If the graphics output device has scalable pixels (e.g., PostScript), then the output image is scaled so that its largest dimension is less than or equal to 400. Use the *Max\_Img\_Size* keyword to increase this size.

When outputting to PostScript devices, the default for the *Background* keyword is white (index 255), rather than !P.Background.

Use the SET\_SHADING procedure to control the direction of the light source and other shading parameters.

---

**CAUTION** If the *T3D* keyword is set, the 3D to 2D transformation matrix contained in !P.T must project the *z*-axis to a line parallel to the device *y*-axis, or errors will occur.

If the *X,Y* grid is not regular or nearly regular, errors in hidden line removal will likely occur. In this case, you should use the *SHADE\_SURF\_IRR* procedure.

*SHADE\_SURF* is not supported on Tektronix terminals or the 4510 Rasterizer. If you try to display shaded image on such a device, *PV-WAVE* may abort. This is because of a limitation in the range of image coordinates available on Tektronix devices. *SHADE\_SURF* is also unsupported on VT240 terminals.

---

### **Example 1**

This example uses *SHADE\_SURF* to display a shaded surface representation of the function:

$$f(x, y) = x \sin(y) + y \cos(x) - \sin\left(\frac{xy}{4}\right)$$

where

$$(x, y) \in \{\mathbb{R}^2 \mid x, y \in [-10, 10]\} \quad .$$

```
x = FINDGEN(101)/5 -10
    ; Create 101-element vector of x-coordinates such that -10 < x < 10.

y = x
    ; Make the vector of y-coordinates the same as the
    ; vector of x-coordinates.

z = FLTARR(101, 101)
    ; Create a 101-by-101 array to hold the function values.

FOR i = 0, 100 DO BEGIN $
    z(i, *) = x(i)*SIN(y) + y*COS(x(i)) $
    - SIN(0.25*x(i)*y)
    ; Insert the function values into z. Note that z is filled
    ; columnwise instead of elementwise.

SHADE_SURF, z, x, y, Ax = 50, XCharsize = 2, $
    YCharsize = 2, ZCharsize = 2
    ; Display the shaded surface. The Ax keyword is used
```

```

; to specify the desired angle of rotation about the x-axis.
; The XChsize, YChsize, and ZChsize keywords are used
; to enlarge the characters used to annotate the axes.

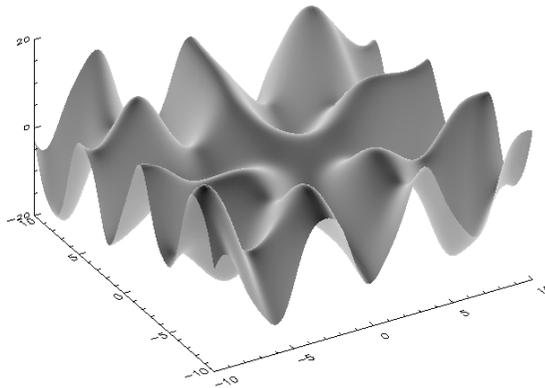
```

```

XYOUTS, 118, 463, $
"f(x, y) = x*sin(y) + y*cos(x) - + sin(x*y)/4)", $
Chsize = 2, /Device
; Place a title in the window. Note that the CURSOR
; procedure with the Device keyword was used to locate
; the proper position for the title.

```

$$f(x, y) = x \sin(y) + y \cos(x) - \sin(xy)/4$$



**Figure 2-57** Shaded surface with title.

### **Example 2**

Users often wish to store the data that describes a surface in a file. Each row of data in the file is a point on the surface so there are three columns of data in the file. The first column contains  $x$ -coordinates of the points, the second column contains  $y$ -coordinates of the points, and the third column contains  $z$ -coordinates of the points.

This example creates data describing the surface defined by the function

$$f(x, y) = xy \cos(0.575xy) - 10(x^2 + y^2)$$

where

$$(x, y) \in \{\mathbb{R}^2 \mid x, y \in [-10, 10]\}$$

and places that data in the file `shsurf.dat` in the columnar format described above. The data is then read from the file `shsurf.dat`, placed in the data structures expected by `SHADE_SURF`, and displayed.

```

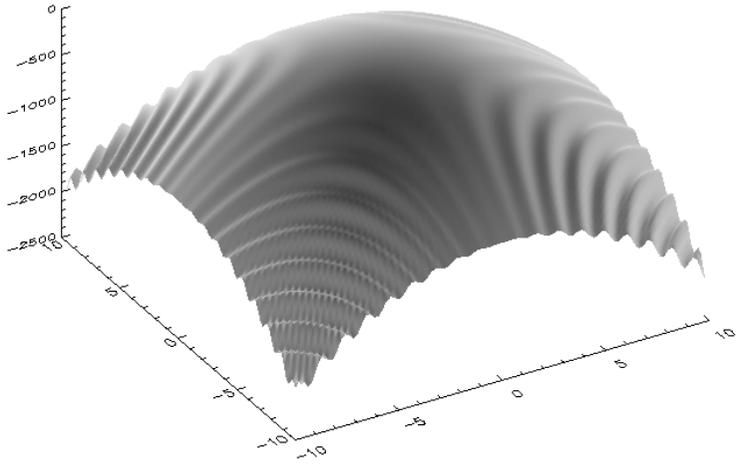
x = FINDGEN(101)/5 - 10
y = x
    ; Create vectors containing the x- and y-
    ; coordinates of the region of the
    ; x-y plane over the surface.
x = x # REPLICATE(1., 101)
y = REPLICATE(1., 101) # y
    ; Convert these vectors into two-dimensional arrays
z = x * y * cos(0.575 * x * y) - 10 * (x^2 + y^2)
    ; Create a z-dimensional array containing the function values on the surface.
OPENW, 1, 'shsurf.dat'
    ; Open the file that is to hold the data describing the surface.
data = FLTARR(3, 101, 101)
    ; Build an array to hold file output data.
data(0, *, *) = x
data(1, *, *) = y
data(2, *, *) = z
    ; Write the x-, y-, and z-coordinates of points on the
    ; surface to this array.
PRINTF, 1, Format = '(3f16.7)', data
    ; Write the array to the file shsurf.dat.
CLOSE, 1
    ; Close the file
    ; NOTE: The commands below could be used during
    ; another wave session, since the data is originating from a file.
data = FLTARR(3, 101, 101)
    ; Create an array large enough to hold the data
    ; contained in shsurf.dat.
OPENR, 1, 'shsurf.dat'
    ; Open shsurf.dat for reading.
READF, 1, data
    ; Read the data from the file.
CLOSE, 1
    ; Close the input file.
x = REFORM(data(0, *, *))
    ; Copy the x-coordinates into x.
y = REFORM(data(1, *, *))

```

```

; Copy the y-coordinates into y.
z = REFORM(data(2, *, *))
; Copy the z-coordinates into z.
SHADE_SURF, z, x, y, Ax = 50, Charsize = 2
; Display the shaded surface of the function. The Ax
; keyword is used to specify the desired angle of
; rotation about the x-axis. The Charsize keywords are used
; to enlarge the characters used to annotate the axes.

```



**Figure 2-58** Shaded surface defined in the example.

### Example 3

```
SHADE_SURF, DIST(100), Ax=60
```

### See Also

[SET\\_SHADING](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

---

## SHADE\_SURF\_IRR Procedure

Standard Library routine that creates a shaded-surface representation of a semiregularly gridded surface, with shading from either a light source model or from a specified array of intensities.

### Usage

SHADE\_SURF\_IRR,  $z$ ,  $x$ ,  $y$

### Input Parameters

$z$  — A two-dimensional array containing the values that make up the surface. If  $x$  and  $y$  are supplied, the surface is plotted as a function of the X,Y locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of  $z$ .

$x$  — A two-dimensional array specifying the  $x$ -coordinates for the contour surface. Each element of  $x$  specifies the  $x$ -coordinate of the corresponding point in  $z$  ( $x_{ij}$  specifies the  $x$ -coordinate for  $z_{ij}$ ).

$y$  — A two-dimensional array specifying the  $y$ -coordinates for each elevation. Each element of  $y$  specifies the  $y$ -coordinate of the corresponding point in  $z$  ( $y_{ij}$  specifies the  $y$ -coordinate for  $z_{ij}$ ).

### Keywords

**Image** — The name of a variable into which the image containing the shaded surface is stored. If this keyword is omitted, the image is displayed but not saved.

**Max\_Img\_Size** — For devices with scalable pixels (e.g., postscript), this keyword sets the largest allowed image size created internally to render the shaded surface. Larger values will result in a better quality image but at a cost of greater memory use and larger file size. The minimum value is 100 (100x100), which will generally result in a poor-quality image. (Default: 400)

**Shades** — An array expression, of the same dimensions as  $z$ , containing the color index at each point. The shading of each pixel is interpolated from the surrounding **Shades** values. For most displays, this parameter should be scaled into the range of bytes. If this keyword is omitted, light source shading is used.

Other keywords let you control many aspects of the plot's appearance. These keywords are listed in the following table. For a description of each keyword, see [Chapter 3, \*Graphics and Plotting Keywords\*](#).

Ax	Noclip	[XYZ]Margin
Az	Nodata	[XYZ]Minor
Background	Noerase	[XYZ]Range
Charsize	Normal	[XYZ]Style
Clip	Position	[XYZ]Tickname
Color	Save	[XYZ]Ticks
Data	T3d	[XYZ]Tickv
Device	Ticklen	[XYZ]Title
Font	[XYZ]Charsize	

---

## Discussion

The input data for `SHADE_SURF_IRR` must be able to be represented as an array of quadrilaterals. This procedure should be used when the  $(x, y, z)$  arrays are too irregular to be drawn by the `SHADE_SURF` procedure, but are still semiregular.

`SHADE_SURF_IRR` is similar to the [SURFACE](#) procedure. Given a semiregular grid of elevations, it produces a shaded surface representation of the data with hidden surfaces removed.

If the graphics output device has scalable pixels (e.g., PostScript), then the output image is scaled so that its largest dimension is less than or equal to 400. Use the `Max_Img_Size` keyword to increase this size.

When outputting to PostScript devices, the default for the `Background` keyword is white (index 255), rather than `!P.Background`.

Use the [SET\\_SHADING](#) procedure to control the direction of the light source and other shading parameters.

---

**NOTE** The `NoErase` keyword is ignored on devices that do not support `TVRD()` — for example, PostScript.

---

---

**CAUTION** If the *T3d* keyword is set, the 3D to 2D transformation matrix contained in !P.T must project the *z*-axis to a line parallel to the device *y*-axis, or errors will occur.

---

## Example

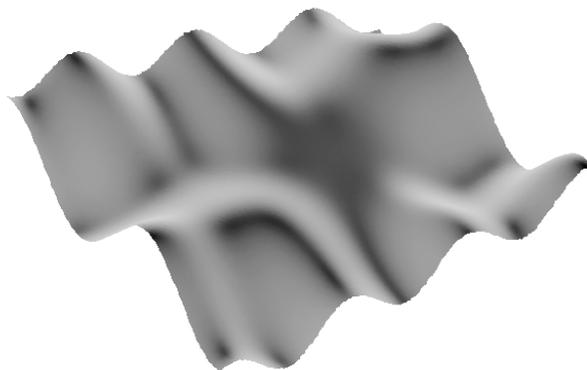
This example uses *SHADE\_SURF\_IRR* to display a shaded surface over an irregular grid. The function defining the surface is:

$$f(x, y) = x \sin(y) + y \cos(x)$$

where

$$(x, y) \in \{\mathbb{R}^2 \mid x \in [-10, 10], y \in [-5, 5]\}$$

```
x = (LINDGEN(200, 100) MOD 200) / 10.0 - 10.0
y = (LINDGEN(200, 100) / 200) / 10.0 - 5.0
; Compute x- and y-components of a 200x100 regular grid.
x = x + (RANDOMN(seed, 200, 100) - 2.0) / 16.0
y = y + (RANDOMN(seed, 200, 100) - 2.0) / 16.0
; Build an irregular grid by perturbing the regular grid
; by a random factor.
z = x * SIN(y) + y * COS(x)
; Compute a two-dimensional array of elevations.
SHADE_SURF_IRR, z, x, y, Ax = 70
; Display the shaded surface. The Ax keyword is used to specify
; the angle of rotation about the x-axis.
```



**Figure 2-59** Shaded surface defined in the example over irregular grid.

### See Also

[SET\\_SHADING](#), [SHADE\\_SURF](#), [SURFACE](#)

---

## ***SHADE\_VOLUME Procedure***

Given a 3D volume and a contour value, produces a list of vertices and polygons describing the contour surface.

### Usage

`SHADE_VOLUME, volume, value, vertex, poly`

### Input Parameters

*volume* — An array with three dimensions containing the dataset to be contoured. If the volume array is dimensioned  $(D_0, D_1, D_2)$ , the resulting vertices range as follows:

- In X, they range between 0 and  $D_0 - 1$ .
- In Y, they range between 0 and  $D_1 - 1$ .
- In Z, they range between 0 and  $D_2 - 1$ .

*value* — A scalar containing the contour value.

## Output Parameters

*vertex* — The name of a variable to receive the vertex array. This variable will be set to a  $(3, n)$  floating-point array, suitable for input to the MESH or POLYSHADE procedure.

*poly* — The name of a variable to receive the polygon list, an  $m$ -element longword array. This list describes the vertices of each polygon and is suitable for input to MESH or POLYSHADE.

## Keywords

*Low* — If present and nonzero, indicates that the low side of the contour surface is to be displayed and that the contour surfaces enclose high data values. Otherwise, it is assumed that the high side of the contour surface is to be displayed and that the contour encloses low data values. If this parameter is incorrectly specified, errors in shading will result.

*Shades* — An array with the same dimensions as *volume*. On input, it contains the user-specified shading color index for each volume element (voxel), and is converted to byte type before use. On output, this input array is replaced by another array containing the shading value for each vertex, contained in *vertex*.

## Discussion

SHADE\_VOLUME computes the polygons that describe a three-dimensional contour surface. Each voxel is visited to find the polygons formed by the intersections of the contour surface and the voxel edges.

You can obtain shading from either a single light-source model or from the values you specify with *Shades*.

The surface produced by SHADE\_VOLUME may then be displayed as a shaded surface with the POLYSHADE procedure.

This routine is limited to processing datasets that will fit in memory.

## Example

The following procedure shades a volume passed as a parameter. It uses SURFACE to establish the viewing transformation. It then calls SHADE\_VOLUME to produce the vertex and polygon lists, and POLYSHADE to draw the contour surface.

```
PRO ShowVolume, vol, thresh, Low=low
    ; Display the contour surface of a volume.
```

```

s = SIZE(vol)
    ; Get the dimensions.
IF s(0) NE 3 THEN PRINT, 'Error'
    ; Flag an error if s is not a 3D array.
SURFACE, FLTARR(2,2), /Nodata, /Save, $
    XRange=[0,s(1)-1], YRange=[0,s(2)-1], ZRange=[0,s(3)-1]
    ; Use SURFACE to establish 3D transformation and
    ; coordinate ranges.
IF N_ELEMENTS(low) EQ 0 THEN low = 0
    ; Make the default be to view the high side of the contour surface.
SHADE_VOLUME, vol, thresh, v, p, Low = low
    ; Produce the vertices and polygons.
TV, POLYSHADE(v, p, /T3D)
    ; Produce the image of the surface and display.
END

```

---

**NOTE** For another example demonstrating SHADE\_VOLUME, see the POLYSHADE function.

---

## See Also

[POLYSHADE](#), [SURFACE](#)

For information on volume visualization, see .

The method used by SHADE\_VOLUME is that described by Klemp, McIrvin and Boyd in “PolyPaint — A Three-Dimensional Rendering Package,” *American Meteorology Society Proceedings*, Sixth International Conference on Interactive Information and Processing Systems, 1990.

This method is similar to the marching cubes algorithm described by Lorensen and Cline in “Marching Cubes: A High Resolution 3D Surface Construction Algorithm,” *Computer Graphics*, Vol. 21, pp. 163-169, 1987.

---

## SHIF Function

Standard Library function that shifts an array along one of its dimensions.

### Usage

*result* = SHIF(*array*, *dimension*, *shift\_amount*)

### Input Parameters

*array* — An array.

*dimension* — An integer ( $\geq 0$ ) designating the shift dimension.

*shift\_amount* — An integer specifying the shift amount.

### Returned Value

*result* — An array of the same dimensions as *array*. *result* is obtained from the input array by a shift of *shift\_amount* elements along dimension *dimension*. The shift is not cyclic; elements behind the shift are unaltered.

### Keywords

*y* — If set, the shift is cyclic.

### Examples

```
PM, SHIF( [0,1,2,3,4,5], 0, 1 )
```

```
PM, SHIF( [0,1,2,3,4,5], 0, -2 )
```

```
PM, SHIF( [0,1,2,3,4,5], 0, -2, /y )
```

```
PM, SHIF( [[0,1,2],[3,4,5],[6,7,8]], 0, 1 )
```

```
PM, SHIF( [[0,1,2],[3,4,5],[6,7,8]], 1, 1 )
```

### See Also

[SHIFT](#)

---

## **SHIFT Function**

Shifts the elements of a vector or array along any dimension by any number of elements.

### **Usage**

*result* = SHIFT(*array*, *shift*<sub>1</sub>, ... , *shift*<sub>*n*</sub>)

### **Input Parameters**

*array* — The array to be shifted.

*shift*<sub>*i*</sub> — The shift parameters (see *Discussion*).

### **Returned Value**

*result* — The shifted array.

### **Keywords**

None.

### **Discussion**

The SHIFT function is used to perform a circular shift upon the elements of a vector or an array. The resulting array has the same dimension and data type as the input array. Shifts are handled similarly, regardless of the number of dimensions in the input array, as detailed below:

- **Shifts on One-Dimensional Arrays** — A shift performed on a one-dimensional array (a vector) shifts the contents of each element to the right or left, depending on the number of elements specified in the second parameter: a positive number shifts elements to the right, while a negative number shifts them to the left.
- **Shifts on Two-Dimensional Arrays** — A shift performed on a two-dimensional array (such as a raster image) is done in a similar way. The contents of entire rows and/or columns are shifted to the rows above or below, or to the columns to the right or left, depending on the number of rows and columns specified by the second and third parameters of the process, respectively.

Positive numbers for the second and third parameters shift rows in an up direction (or columns to the right), while negative numbers shift rows in a down direction (or columns to the left).

- **Shifts on Arrays with More than Two Dimensions** — For arrays of more than two dimensions, the parameter  $shift_n$  specifies the shift applied to the  $n$ th dimension. For example,  $shift_1$  specifies the shift along the first dimension. If you specify 0 for  $shift_n$ , this means that no shift is to be performed along that dimension.

Regardless of the number of dimensions, all shifts are circular, meaning that values that are pushed off the beginning or end of the array by the shift operation are automatically inserted back onto the opposite end of the array. No values in the array are lost.

If only one shift parameter is present and the parameter is an array, the array is treated as a vector.

### **Sample Usage**

Typical uses of the SHIFT function include:

- To force the elements of one array to align with the elements of another array.
- To force the elements of one array to be misaligned with the elements of another array (some statistical analysis techniques require this).
- To line up (or *register*) the edges of an image to match those of another image. This can be used to compensate for an image that was initially digitized out of alignment with respect to the edges of another image.
- To shift the beginning and ending point of a color table in an image.
- To do an edge enhancement technique for images commonly known as *Shift and Difference Edge Enhancement* (see *Example 2* for more information).

### **Example 1**

The following demonstrates one-dimensional shifting using the SHIFT function with a vector:

```
a = INDGEN(5)
    ; Make a small vector.

PRINT, a, SHIFT(a,1), SHIFT(a,-1)
    ; Print the vector, the vector shifted one position to the right, and
    ; the vector shifted one position to the left.
```

Executing these statements gives the result:

0	1	2	3	4
4	0	1	2	3
1	2	3	4	0

Notice how elements of the vector that shift off the end wrap around to the other end. This “wrap around” occurs when shifting arrays of any dimension.

## Example 2

SHIFT can be used to create an edge enhancement technique for images. In this technique, a copy of an array may be shifted by one or more elements (pixels) in any direction, and then subtracted from the original image.

When performed on a binary image (containing only black and white pixels), edges that are opposite the direction of the shift are enhanced.

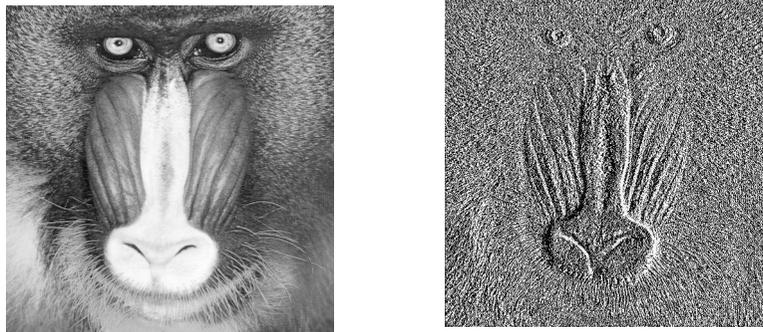
When performed on a gray scale or pseudocolor image, an embossing effect is created opposite the direction of the shift.

By carefully selecting the direction and amount of the shift, you can make certain details (for example, only vertical or horizontal lines) be discernible in an otherwise jumbled image.

Typically, single-element (one pixel) shifts are most pronounced, while any shift beyond ten elements (pixels) tends to start blurring the features in the image.

For example, to shift a mandrill image to highlight the edges to the left of each feature, you would first run the SHIFT function and then subtract the resulting image from the original image to create a third image:

```
shift_mandrill = SHIFT(mandrill, 1, 0)
shift_diff_mand = mandrill - shift_mandrill
```

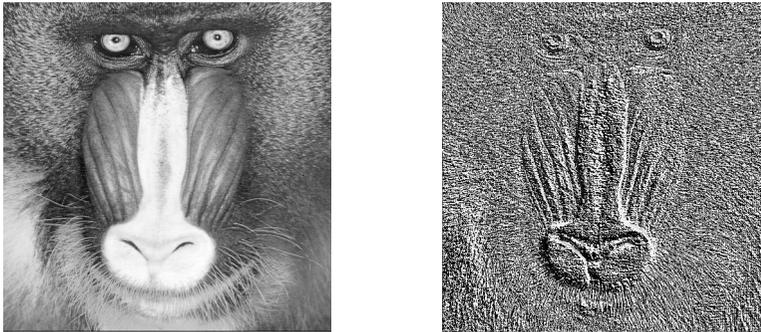


**Figure 2-60** The SHIFT function described above has been used with this 512-by-512 mandrill image for edge enhancement. Notice how the dark edges to the left of each feature in

the image are highlighted by using SHIFT with a positive number for the first parameter (which causes the image to be shifted to the right).

To shift a mandrill image to highlight the edges below and to the left of each feature:

```
shift_mandrill = SHIFT(mandrill, 1, 1)
shift_diff_mand = mandrill - shift_mandrill
```



**Figure 2-61** Notice how the dark edges to the southwest of each feature in the image are now highlighted by using the SHIFT function with a positive number for the first and second parameter (which causes the image to be shifted both up and to the right).

## See Also

[CONVOL](#), [ISHFT](#), [ROBERTS](#), [SHIF](#), [SOBEL](#)

For more information about Shift and Difference Edge Enhancement, see *Digital Image Processing*, by Gregory Baxes, Cascade Press, Denver, 1988.

---

## **SHOW3 Procedure**

Standard Library procedure that displays a two-dimensional array as a combination contour, surface, and image plot. The resulting display shows a surface with an image underneath and a contour overhead.

### **Usage**

SHOW3, *array*

### **Input Parameters**

*array* — The two-dimensional array to display.

### **Keywords**

*Ax* — The angle of rotation, in degrees, about the *x*-axis.

*Az* — The angle of rotation, in degrees, about the *z*-axis.

*Bot\_Image* — The name of the image array to be used as the underneath image.

*C\_Colors* — A vector of color indices whose elements indicate which color to use in drawing the corresponding contour level.

*Interp* — If present and nonzero, specifies that bilinear interpolation is to be used for the pixel display. Otherwise, the nearest neighbor interpolation method is used.

*Sscale* — The reduction scale for the surface. If set to anything other than 1 (the default value), the image size is reduced by the specified factor. If the image dimensions are not an integral multiple of *Sscale*, the image is reduced to the next smaller multiple.

### **Discussion**

You can modify SHOW3, if necessary, to customize the contour and surface commands to your satisfaction (e.g., use different colors, skirts, line styles, and contour levels). See the descriptions of the CONTOUR and SURFACE routines for details.

---

**TIP** When you are displaying larger images (say 50-by-50), the display produced by SHOW3 can become too “busy.” If this happens, try using the SMOOTH and/or REBIN procedures to smooth the surface plot.

---

---

**CAUTION** The SHOW3 procedure is not supported on Tektronix terminals or the 4510 Rasterizer. If you try to display a SHOW3 image on such a device, PV-WAVE may abort. This is because of a limitation in the range of image coordinates available on Tektronix devices.

---

## Example

This example uses the Pike's Peak elevation and snowpack images found in:

(UNIX) <wavedir>/data

(OpenVMS) <wavedir>:[DATA]

(Windows) <wavedir>\data

Where <wavedir> is the main PV-WAVE directory.

```
OPENR, 1, !data_dir + 'pikeselev.dat'
    ; Open the file to read.

pikes = FLTARR(60, 40)
    ; Create the data array for the pikes elevation data.

READF, 1, pikes
    ; Read in the formatted file.

CLOSE, 1
    ; Close the file.

OPENR, 2, !data_dir + 'snowpack.dat'
    ; Open the file to read.

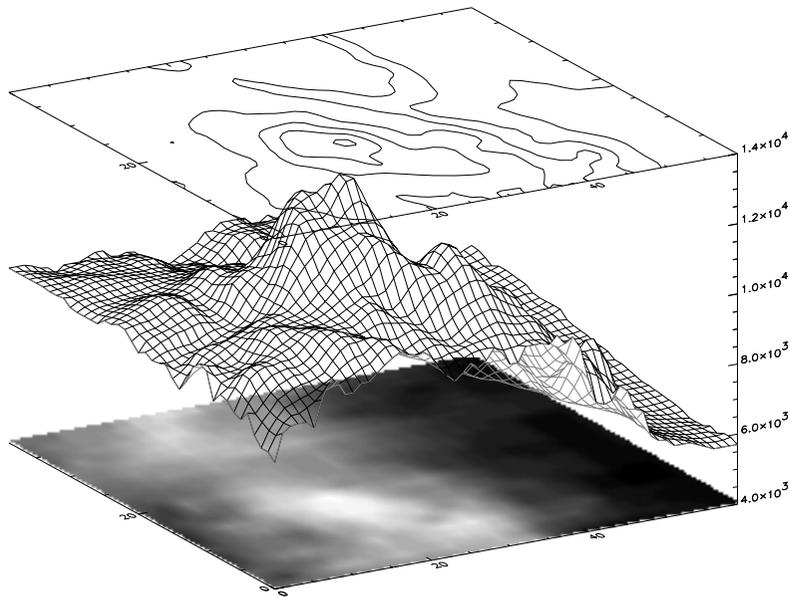
snow = FLTARR(60, 40)
    ; Create the data array for snow pack data.

READF, 2, snow
    ; Read in the formatted file.

CLOSE, 2
    ; Close the file.

LOADCT, 5
    ; Load color table number 5.

SHOW3, SMOOTH(pikes, 3), Bot_image=snow, /Interp
    ; Produce the combined display.
```



**Figure 2-62** Combination image, surface, and contour produced with SHOW3.

## See Also

[CONTOUR](#), [REBIN](#), [SMOOTH](#), [SURFACE](#)

For information on interpolation methods, see .

---

## ***SHOW\_OPTIONS Procedure***

Lists the currently loaded OPI (Option Programming Interface) optional modules and their associated functions and procedures.

### **Usage**

SHOW\_OPTIONS

### **Keywords**

***Functions*** — If nonzero, displays the names of all functions available in the given OPI option.

**License**— If nonzero, displays the license information for the given OPI option.

**Procedures** — If nonzero, displays the names of all procedures available in the given OPI option.

## Discussion

The SHOW\_OPTIONS procedure lists the currently loaded OPI options and the functions and procedures that they contain. OPI options can be loaded explicitly by any PV-WAVE user using the LOAD\_OPTION procedure. These optional modules can be written in C or FORTRAN, and can contain new system functions or other primitives. For detailed information on creating OPI options, see the *PV-WAVE Application Developer's Guide*.

## Example

```
WAVE> SHOW_OPTIONS, /Function, /Procedure
% Option: SAMPLE 1.000000
% Functions:
% PLUS_THREE
% PLUS_TWO
% Procedures:
% ADD_FOUR
% ADD_THREE
% ADD_TWO
```

## See Also

[LOAD\\_OPTION](#), [OPTION\\_IS\\_LOADED](#), [UNLOAD\\_OPTION](#)

---

## **SIGMA Function**

Standard Library function that calculates the standard deviation value of an array. (Optionally, it can also calculate the standard deviation over one dimension of an array as a function of the other dimensions.)

### **Usage**

*result* = SIGMA(*array* [, *npar*, *dim*])

### **Input Parameters**

*array* — The array to be processed. This parameter can be of any data type except string.

*npar* — (optional) The number of parameters. (Default: 0)

*dim* — (optional) The dimension over which to calculate the standard deviation.

### **Returned Value**

*result* — The standard deviation of *array*, or the standard deviation over one dimension of *array* as a function of *dim*.

### **Keywords**

None.

### **Discussion**

The SIGMA function is useful in a variety of data and statistical analyses for estimating thresholds and evaluating the significance of variance.

The number of degrees of freedom in SIGMA is equal to the number of elements in *array* minus the value supplied in the optional *npar*. In other words:

$$\text{degrees of freedom} = \#\_of\_elements\_in\_array - npar$$

The degrees of freedom affects the value of the standard deviation. Specifically, the value of the standard deviation varies as one over the square root of the number of degrees of freedom.

If *dim* is used, then the result of SIGMA is an array with the same dimensions as the input array, except for the dimension specified. Each element in the resulting *array* is the standard deviation of the corresponding vector in the input *array*.

The dimension specified in a SIGMA call must be valid for the array passed; otherwise, the input array is returned as the output array.

If *array* is an array with dimensions of (3,4,5), then the command:

```
std = SIGMA(array, 2, 1)
```

is equivalent to the commands:

```
std = FLTARR(3,5)
FOR j = 0, 4 DO BEGIN
    FOR i = 0, 2 DO BEGIN
        STD(i, j) = SIGMA(array(i, *, j), 2)
    ENDFOR
ENDFOR
```

## Example

```
y = [1., 5., 9., 3., 10., 4.]
std = SIGMA(y, 1)
PRINT, 'The standard deviation = ', std
    3.50238

a = FLTARR(3, 3)
a(*, 0) = [2., 2., 2.]
a(*, 1) = [4., 4., 4.]
a(*, 2) = [6., 6., 6.]
PRINT, a
PRINT, SIGMA(a, 1, 0)
    0.00000    0.00000    0.00000
    ; Print the standard deviation of the column elements.

PRINT, SIGMA(a, 1, 1)
    2.00000    2.00000    2.00000
    ; Print the standard deviation of the row elements.
```

## See Also

[STDEV](#)

---

## **SIN Function**

Returns the sine of the input variable.

### **Usage**

*result* = SIN(*x*)

### **Input Parameters**

*x* — The angle, in radians, that is evaluated.

### **Returned Value**

*result* — The trigonometric sine of *x*.

### **Keywords**

None.

### **Discussion**

If *x* is of double-precision floating-point or complex data type, SIN yields a result of the same type. All other types yield a single-precision floating-point result.

SIN handles complex numbers in the following way:

$$\sin(x) \equiv \text{complex}((\sin(r)\cosh(i), \cos(r)\sinh(i)))$$

where *r* and *i* are the real and imaginary parts of *x*.

If *x* is an array, the result of SIN has the same data type, with each element containing the sine of the corresponding element of *x*.

### **Example**

The following commands produce a dampened sine wave.

```
x = FINDGEN(200)
PLOT, 10000 * SIN(x/5) / EXP(x/100)
```

## See Also

[COS](#), [COSH](#), [SINH](#)

For a list of other transcendental functions, see *Transcendental Mathematical Functions* in Volume 1 of this reference.

---

## SINDGEN Function

Returns a string array with the specified dimensions.

### Usage

*result* = SINDGEN(*dim*<sub>1</sub>, ... , *dim*<sub>*n*</sub>)

### Input Parameters

*dim*<sub>*i*</sub> — The dimensions of the result. This may be any scalar expression and up to eight dimensions may be specified.

### Returned Value

*result* — An initialized string array. If the resulting array is treated as a one-dimensional array, then its initialization is given by the following:

$$array(i) = \text{STRING}(i), \text{ for } i = 0, 1, \dots, \left( \prod_{j=1}^n D_j - 1 \right)$$

### Keywords

None.

### Discussion

Each element of the array is set to the string representation of the value of its one-dimensional subscript, using the default formatting rules of PV-WAVE. These rules are described in .

### Example

This example creates a 4-by-2 string array.

```

a = SINDGEN(4, 2)
    ; Create the string array.
INFO, a
    A      STRING      = Array(4, 2)
PRINT, a
    0      1      2      3
    4      5      6      7

```

## See Also

[BINDGEN](#), [CINDGEN](#), [DINDGEN](#), [FINDGEN](#), [INDGEN](#), [LINDGEN](#)

## ***SINH Function***

Returns the hyperbolic sine of the input variable.

### Usage

*result* = SINH(*x*)

### Input Parameters

*x* — The angle, in radians, that is evaluated.

### Returned Value

*result* — The hyperbolic sine of *x*.

### Keywords

None.

### Discussion

SINH is defined by:

$$\sinh(x) \equiv (e^x - e^{-x})/2$$

If *x* is of double-precision floating-point or of complex data type, SINH yields a result of the same type. All other data types yield a single-precision floating-point result.

If  $x$  is an array, the result of `SINH` has the same dimensions, with each element containing the hyperbolic sine of the corresponding element of  $x$ .

## Example

```
x = [0.3, 0.5, 0.7, 0.9]
PRINT, SINH(x)
      0.304520      0.521095      0.758584      1.02652
```

## See Also

[COS](#), [COSH](#), [SIN](#), [TANH](#)

For a list of other transcendental functions, see [Transcendental Mathematical Functions](#) on page 32 in Volume 1 of this reference.

---

## SIZE Function

Returns a vector containing size and type information for the given expression.

### Usage

```
result = SIZE(expr)
```

### Input Parameters

*expr* — The expression to be evaluated.

### Returned Value

*result* — A vector containing size and type information for *expr*.

### Keywords

*Dimensions* — Returns a longword vector containing the size of each dimension of *expr*.

*Nelements* — Returns the total number of elements in *expr*.

*Ndimensions* — Returns the number of dimensions in *expr*. A zero is returned if *expr* is scalar.

*Type* — Returns the type code of *expr*.

## Discussion

The returned vector is of longword type. It has the following form:

$$[D, S_1, S_2, \dots, S_p, T, E]$$

where  $D$  is the number of dimensions of EXPRESSION (zero if EXPRESSION is scalar or undefined);  $S_i$  is the size of dimension  $i$  (not present if EXPRESSION is scalar or undefined);  $T$  is the type code, encoded as shown in the table below; and  $E$  is the number of elements in EXPRESSION.

Type Code	Data Type
0	Undefined
1	Byte
2	Integer
3	Longword integer
4	Floating point
5	Double precision floating
6	Complex single-precision floating
7	String
8	Structure
9	UPVAR (the variable type pointed to by UPVAR)
10	List
11	Associative array
12	Complex double-precision floating

## See Also

[N\\_ELEMENTS](#), [N\\_PARAMS](#), [N\\_TAGS](#), [SAME](#), [TAG\\_NAMES](#)

---

## **SKIPF Procedure (OpenVMS)**

(OpenVMS Only) Skips records or files on the designated magnetic tape unit.

### **Usage**

SKIPF, *unit*, *files*

SKIPF, *unit*, *records*, *r*

### **Input Parameters**

***unit*** — A number between 0 and 9 specifying the magnetic tape unit to rewind. (Do not confuse this parameter with file logical unit numbers.)

***files*** — The number of files to be skipped. If this number is positive, skipping is in the forward direction. Otherwise, files are skipped backwards.

***records*** — The number of records to be skipped. If this number is positive, skipping is in the forward direction. Otherwise, records are skipped backwards.

***r*** — If this third parameter is present, records are skipped; otherwise, files are skipped. (The value of *r* is never examined; its presence serves only to indicate that records are to be skipped.)

### **Keywords**

None.

### **Discussion**

The number of files or records actually skipped is stored in the system variable !Err. Note that when skipping records, the operation terminates immediately when the end of a file is encountered.

### **See Also**

[TAPRD](#), [TAPWRT](#)

System Variables: [!Err](#)

For more information and sample usage, see .

---

## ***SLICE Function***

Standard Library function that subsets an array along one of its dimensions.

### **Usage**

*result* = SLICE(*array*, *dimension*, *indices*)

### **Input Parameters**

*array* — An array.

*dimension* — An integer ( $\geq 0$ ) designating the dimension to subset.

*indices* — A vector of indices into dimension *dimension*.

### **Returned Value**

*result* — An array of slices perpendicular to dimension *dimension*.

### **Keywords**

None.

### **Examples**

```
a = INDGEN( 6, 5, 2 ) & pm, a, ['', ''], SLICE( a, 0, [1,3,5] )
```

---

## ***SLICE\_VOL Function***

Returns a 2D array containing a slice from a 3D volumetric array.

### **Usage**

*result* = SLICE\_VOL(*volume*, *dim*, *cut\_plane*)

### **Input Parameters**

*volume* — The 3D volume of data to slice.

---

**TIP** For better results, first use VOL\_PAD to preprocess the volume.

---

*dim* — The  $x$  and  $y$  dimensions of the slice to return. Larger values for *dim* increase the slice resolution and execution time. *dim* is typically the largest of the three dimensions of *volume*.

*cut\_plane* — A (3, 2) array defining the slicing plane. The elements in this (3, 2) array are interpreted as follows:

---

cut_plane(0, 0)	The plane's angle of rotation about the $x$ -axis.
cut_plane(1, 0)	The plane's angle of rotation about the $y$ -axis.
cut_plane(2, 0)	Ignored.
cut_plane(0, 1)	The $x$ -coordinate of the center of the plane.
cut_plane(1, 1)	The $y$ -coordinate of the center of the plane.
cut_plane(2, 1)	The $z$ -coordinate of the center of the plane.

---

---

**NOTE** The slicing plane is rotated first about the  $y$ -axis, and then about the  $x$ -axis.

---

## Returned Value

*result* — A 2D array containing a slice from a 3D volumetric array.

## Keywords

*Degrees* — If present and nonzero, *cut\_plane(0:1,0)* is assumed to be in degrees.

## Discussion

SLICE\_VOL extracts a planar oblique slice from a volumetric (3D) array. The slicing plane is defined by the center point ( $X$ ,  $Y$ , and  $Z$ ), and the rotations about the  $y$ - and  $x$ -axes. You can interactively define the slicing plane by calling the VIEWER procedure.

For best results, process volumes with VOL\_PAD before slicing them with SLICE\_VOL.

## Example

For demonstrations of the SLICE\_VOL procedure, use the **Medical Imaging** and **CFD/Aerospace** buttons on the PV-WAVE Demonstration Gallery.

To run the Gallery, enter `wave_gallery` at the `WAVE>` prompt.

## See Also

[VIEWER](#), [VOL\\_PAD](#)

---

## ***SMALL\_INT Function***

Smallest Integer Function. Standard Library function that returns the smallest integer greater than or equal to the passed value. Also known as Ceiling Function.

## Usage

*result* = SMALL\_INT(*x*)

## Input Parameters

*x* — An array (scalar).

## Returned Value

*result* — A long array (scalar) of the same dimensions as *x*: *result*(*i*) is the smallest integer greater than or equal to *x*(*i*).

## Keywords

None.

## Examples

```
PM, SMALL_INT( [-0.5,0,0.5] )
```

```
x=[-2.1,-1.9,0,1.9,2.1]
```

```
PRINT, SMALL_INT(x)
```

```
; -2 -1 0 2 3
```

## See Also

[FIX](#), [GREAT\\_INT](#), [NINT](#)

---

## SMOOTH Function

Smooths an array with a boxcar average of a specified width.

### Usage

```
result = SMOOTH(array, width)
```

### Input Parameters

**array** — The array to be smoothed. The array can be of any number of dimensions.

---

**NOTE** Use the *Intleave* keyword to specify interleaving for 2D and 3D arrays.

---

**width** — The width of the smoothing window. Should be an odd number, smaller than the smallest dimension. (If *width* is even, *width* + 1 is used instead.)

### Returned Value

**result** — A copy of *array* smoothed with a boxcar average of the specified *width*. It has the same type and dimensions as *array*.

### Keywords

**Edge** — A scalar string indicating how edge effects are handled. (Default:

'copy') Valid strings are:

'zero' — Sets the border of the output image to zero.

'copy' — Copies the border of the input image to the output image.  
(Default)

**Intleave** — A scalar string indicating the type of interleaving:

'signal' — Indicates that a mxn array is to be treated as n signals of length m; each signal is smoothed independently.

'image' — Indicates that a mxnxp array is to be treated as p mxn

images; each image is smoothed independently.

## Discussion

The SMOOTH function supports input arrays composed of multiple images as well as input arrays composed of multiple signals. The *Intleave* keyword is used to indicate that the input array is a multi-signal or multi-image array. When the *Intleave* keyword is used, each signal or image in the array is operated on separately and an array of the individual results is returned.

The default behavior for SMOOTH is to operate on the input array as a single piece of data.

SMOOTH is used to selectively average the elements in an array within a moving window of a given *width*. The result smooths out spikes or rapid transitions in the data and is usually called a boxcar average. The window is a box that traverses the input array, element by element. As the window moves through the array, all values within the window are averaged. The average value is then placed at the center of the window in the output array, while the original array is kept intact.

Conceptually, the algorithm used by SMOOTH is shown below for the 1D case ( $n$  is the number of elements in  $A$  and  $w$  is the width of the smoothing window, and  $w/2$  is to be interpreted as the greatest integer less than or equal to  $w/2$ ).

When  $w/2 \leq i \leq n-1-w/2$ , then:

$$R_i = (1/w) \sum_{j=0}^{w-1} A_{i+j-w/2}$$

Otherwise,  $R_i = A_i$ .

The 2D case of an  $n_0 \times n_1$  array is handled similarly:

when  $w/2 \leq i_k \leq n_k-1-w/2$  ( $k=0,1$ ), then:

$$R_{i_0, i_1} = (1/w^2) \sum_{j_0, j_1=0}^{w-1} A_{i_0+j_0-w/2, i_1+j_1-w/2}$$

otherwise

$$R_{i_0, i_1} = A_{i_0, i_1}$$

The extension of these equations to higher dimensional arrays is obvious.

---

**NOTE** Rather than explicitly summing over the entire window each time the window is moved, the sum over each new window is computed as the sum over the old window plus the sum over the leading edge of the new window minus the sum over the trailing edge of the old window. This method is computationally efficient, but due to numerical roundoff it may give slightly different results than the explicit method. Double precision input virtually eliminates these differences.

---

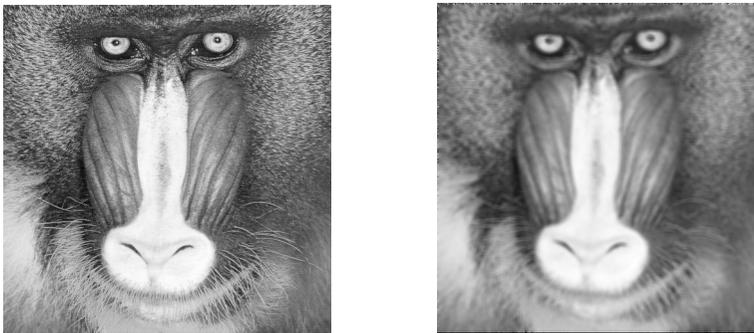
### **Sample Usage**

Typical uses for the SMOOTH function include:

- To remove ripples, spikes, or high frequency noise from a signal or image.
- To blur an image or set of data such that only the general trends in the data can be seen (and are thereby highlighted).
- To isolate the lower spatial frequency components in an image. By subtracting a blurred image from the original image, only the higher spatial frequency components are left. (This is sometimes referred to as unsharp masking.)
- To soften sharp transitions from one color to another in a color table. This helps reduce the banding or contouring artifact evident in color tables with rapid color changes.

### **Example**

Here is what a mandrill image looks like before and after applying the SMOOTH function. For this example, a value of 7 was used for the *width* parameter.



**Figure 2-63** The SMOOTH function has been used to soften the sharp contrasts in this 512-by-512 mandrill image.

### **See Also**

[CONVOL](#), [MEDIAN](#), [ROBERTS](#), [SOBEL](#)

For more information, see the section *Image Smoothing* in Chapter 6 of the *PV-WAVE User's Guide*.

---

## **SOBEL Function**

Performs a Sobel edge enhancement of an image.

### **Usage**

*result* = SOBEL(*image*)

### **Input Parameters**

*image* — The two-dimensional array containing the image to be enhanced.

### **Returned Value**

*result* — A two-dimensional array of integer type containing the image that has been edge-enhanced. It has the same dimensions as *image*.

### **Keywords**

*Col* — Computes the column gradient (horizontal line enhancement). (Default: *Col* = 1)

---

**NOTE** For horizontal line enhancement only, set the keyword *Row* = 0 to disable the vertical line enhancement.

---

*Edge* — A scalar string indicating how edge effects are handled. (Default: 'zero') Valid strings are:

    'zero' — Sets the border of the output image to zero. (Default)

    'copy' — Copies the border of the input image to the output image.

*No\_Clip* — If set, the *result* data type is greater than the *image* data type so that overflow values in *result* are not clipped.

---

**TIP** Use the *No\_Clip* keyword to avoid overflow conditions.

---

**Return** — A scalar string specifying a mathematical function to apply. Valid strings are 'abs', 'phase', and 'value'. The *Return* keyword is used with the *Row* and *Col* keywords per the following table. (Default: 'abs')

	<i>Return</i> = 'abs'	<i>Return</i> = 'phase'	<i>Return</i> = 'value'
<i>Col</i> = 1, <i>Row</i> = 0	ABS(col grad)	Invalid Condition	column gradient
<i>Col</i> = 0, <i>Row</i> = 1	ABS(row grad)	Invalid Condition	row gradient
<i>Col</i> = 1, <i>Row</i> = 1	ABS(row grad)+ ABS(col grad)	ATAN(row grad ÷ col grad), data type is double	Invalid Condition
<i>Col</i> = 0, <i>Row</i> = 0	Invalid Condition	Invalid Condition	Invalid Condition

**Row** — Computes the row gradient (vertical line enhancement). (Default: *Row* = 1)

**NOTE** For vertical line enhancement only, you must disable the horizontal line enhancement by setting the *Col* keyword to 0.

**Same\_Type** — If set, the *result* is the same data type as *image*; otherwise, the *result* image data type is always integer.

**Scale** — If set, each gradient is scaled by the factor 0.25; otherwise, no scaling is performed.

**Zero\_Negatives** — If set, all negative values in the result are set to zero.

## Discussion

The SOBEL function supports multi-layer band interleaved images. When the input array is three-dimensional, it is automatically treated as an array of images,  $array(m, n, p)$ , where  $p$  is the number of  $m$ -by- $n$  images. Each image is then operated on separately and an array of the result images is returned.

SOBEL implements an approximation of the 3-by-3 nonlinear edge enhancement operator:

$$G(j,k) = |X| + |Y|$$

where

$$X = (A_2 + 2A_3 + A_4) - (A_0 + 2A_7 + A_6)$$

$$Y = (A_0 + 2A_1 + A_2) - (A_6 + 2A_5 + A_4)$$

and where the pixels surrounding the neighborhood of the pixel  $F(j, k)$  are numbered as follows:

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ A_7 & F(j, k) & A_3 \\ A_6 & A_5 & A_4 \end{bmatrix}$$

This algorithm is a fast approximation of the SOBEL function, which is actually defined as:

$$G(j, k) = \sqrt{X^2 + Y^2}$$

---

**CAUTION** Because the result image is saved in integer format, large original data values will cause overflow. Overflow occurs when the absolute value of the result is larger than 32,767. Use the *No\_Clip* keyword to avoid overflow.

---

### **Sample Usage**

SOBEL is commonly used to obtain an image that contains only the edges (rapid transitions between light and dark, or from one color to another) that were present in the original image. SOBEL can help enhance features and transitions between areas in an image (for example, a machine part photographed against a white background).

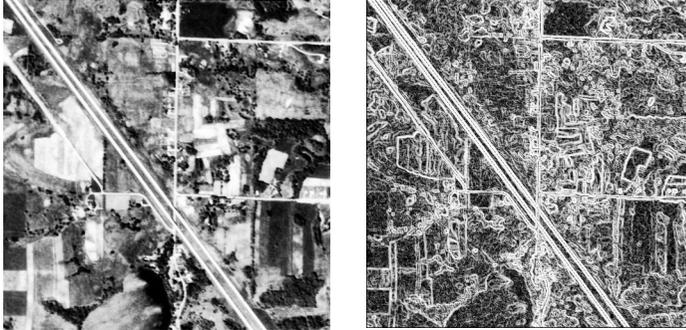
With this information, it is possible to identify and compare features or items in an image with those in another image, usually for verification or detection purposes. SOBEL and other edge-detection algorithms are used extensively for image processing and preprocessing for pattern recognition.

The image returned by SOBEL contains the edges present in the original image, with the brightest edges representing a rapid transition (well-defined features), and darker edges representing smoother transitions (blurred or blended features).

An original image can also be somewhat sharpened by adding or averaging the edge-detected image with the original image.

## Example

Here is what an aerial image looks like before and after applying the SOBEL function.



**Figure 2-64** The SOBEL function has been used with this 512-by-512 aerial image to make the edges stand out with sharp contrast.

## See Also

[CONVOL](#), [ROBERTS](#)

---

## ***SOCKET\_ACCEPT Function***

Waits for a connection on a socket.

### **Usage**

*connection* = SOCKET\_ACCEPT(*socket*)

### **Input Parameters**

*socket* — A socket handle, usually the value returned by SOCKET\_INIT. This handle represents the socket on which the server listens for a client connection.

### **Keywords**

None.

## Returned Value

*connection* — Another socket handle. This handle represents the actual socket connection to the client.

## Discussion

This routine blocks the currently running application until a socket connection is received from the client.

The input parameter, *socket*, must be a value returned from either SOCKET\_INIT or SOCKET\_CONNECT.

SOCKET\_ACCEPT listens on the port represented by the input parameter, *socket*. Immediately upon making a connection, a new socket is assigned to handle the communication flow. This new socket is represented by the returned value *connection*.

## Example

The following simple server program uses SOCKET\_ACCEPT to listen for client connections. This program simply prints a string sent from the client.

```
PRO SERVER
port = 1500
    socket = SOCKET_INIT(port)
    connection = SOCKET_ACCEPT(socket)
    data = BYTARR(15)
    nbytes = SOCKET_READ(connection,data)
    PRINT, 'SERVER received: ', STRING(data)
    SOCKET_CLOSE, connection
    SOCKET_CLOSE, socket
END
```

## See Also

[SOCKET\\_INIT](#)

For more detailed information on using the socket routines, see the *PV-WAVE Application Developer's Guide*.

---

## **SOCKET\_CLOSE Procedure**

Closes a socket connection.

### **Usage**

SOCKET\_CLOSE, *connection*

### **Input Parameters**

*connection* — A socket handle, the value returned by SOCKET\_INIT, SOCKET\_CONNECT, or SOCKET\_ACCEPT.

### **Keywords**

None.

### **Discussion**

The input parameter, *connection*, must be a value returned from either SOCKET\_INIT, SOCKET\_ACCEPT, or SOCKET\_CONNECT.

It is good practice to close any socket handle when you are finished with it. Closing a handle returned from SOCKET\_CONNECT or SOCKET\_ACCEPT terminates that client/server connection (the “session”), while closing a handle returned from SOCKET\_INIT terminates the server.

### **Example**

See the example for [SOCKET\\_ACCEPT](#).

### **See Also**

[SOCKET\\_ACCEPT](#), [SOCKET\\_CONNECT](#), [SOCKET\\_INIT](#)

For more detailed information on using the socket routines, see the *PV-WAVE Application Developer's Guide*.

---

## ***SOCKET\_CONNECT* Function**

Connects to a socket at a given host and port.

### **Usage**

```
connection = SOCKET_CONNECT(host, port)
```

### **Input Parameters**

***host*** — A string specifying the host name or IP address of the server machine (the machine that the client will connect to).

***port*** — A long integer specifying the port number to connect to on the server machine.

### **Keywords**

None.

### **Returned Value**

***connection*** — On success, returns a socket handle for the connection; on failure, returns one of the following values:

- 1 Unable to get host address.
- 2 Could not connect to socket.

### **Discussion**

The *host* parameter can be an IP address number, a DNS name, or any other valid address designation (such as, `localhost`).

It is good practice to close the socket connection when you are finished with it.

### **Example**

The *host* must be a valid IP address. For example:

```
connection=SOCKET_CONNECT('www.vni.com', 80)
    Uses a DNS name for the host.
connection=SOCKET_CONNECT('192.130.240.125', 1500)
```

Uses an IP address number for the host.

```
connection=SOCKET_CONNECT('localhost', 1500)
```

Uses the local loopback address for the host.

```
connection=SOCKET_CONNECT('mymachine', 2100)
```

Uses a local network designation for the host.

## See Also

[SOCKET\\_ACCEPT](#), [SOCKET\\_INIT](#), [SOCKET\\_CLOSE](#)

For more detailed information on using the socket routines, see the *PV-WAVE Application Developer's Guide*.

---

## ***SOCKET\_GETPORT*** Function

Returns the socket port number for the specified socket connection.

### Usage

```
port = SOCKET_GETPORT(socket)
```

### Input Parameters

*socket* — A socket descriptor that was returned from `SOCKET_INIT`.

### Keywords

None.

### Output Parameters

*port* — On success, returns the port number for the socket descriptor; on failure, returns `-1`.

### Discussion

You can call `SOCKET_INIT` with a port number of 0 (zero), which lets your operating system pick an available port. `SOCKET_GETPORT` returns the port picked by the operating system.

## Example

```
s=SOCKET_INIT(0)
PRINT, 'PORT IS: ', SOCKET_GETPORT(s)
```

## See Also

[SOCKET\\_INIT](#)

For more detailed information on using the socket routines, see the *PV-WAVE Application Developer's Guide*.

---

## **SOCKET\_INIT Function**

Binds a socket to a specified port that is designated to listen for client connections.

### Usage

```
socket = SOCKET_INIT(port)
```

### Input Parameters

*port* — The port number on which the server will listen.

### Keywords

None.

### Returned Value

*socket* — On success, returns a socket handle; on failure, returns one of the following values:

- 2 Error creating the socket.
- 3 Error binding the socket.
- 4 Error on the `listen()` command.

## Discussion

You must call this routine in a server program to specify which port to listen on for client connections.

The port number is usually a standard, or previously agreed upon, port that clients use to contact a server. For example, by convention, most Web servers listen for connections on port 80. Most hosts have ports numbered between 1 and 65,535. Typically, ports 0 to 1024 are reserved and only available to someone with administrator or super user privileges.

The returned socket handle is used by other SOCKET\_\* routines to identify a particular socket connection.

It is good practice to close the socket handle when you are finished with it.

## Example

In this simple server program, a socket is bound to port 1500, which is the port that the client program will use to contact the server.

```
PRO SERVER
port = 1500
  socket = SOCKET_INIT(port)
  connection = SOCKET_ACCEPT(socket)
  data = BYTARR(15)
  nbytes = SOCKET_READ(connection,data)
  PRINT, 'SERVER received: ', STRING(data)
  SOCKET_CLOSE, connection
  SOCKET_CLOSE, socket
END
```

## See Also

[SOCKET\\_ACCEPT](#), [SOCKET\\_CLOSE](#)

For more detailed information on using the socket routines, see the *PV-WAVE Application Developer's Guide*.

---

## ***SOCKET\_READ* Function**

Reads data from a socket connection.

### **Usage**

*nbytes* = SOCKET\_READ(*connection*, *data*)

### **Input Parameters**

*connection* — A socket handle (usually returned by either SOCKET\_ACCEPT or SOCKET\_CONNECT).

### **Output Parameters**

*data* — A pre-allocated byte array to read into.

### **Keywords**

None.

### **Returned Value**

*nbytes* — The number of bytes read from the socket.

### **Discussion**

This function can be used in both client and server programs. If used in a client program, it retrieves data sent (using SOCKET\_WRITE) from the server. If used in a server program, it retrieves data sent from the client.

Before calling SOCKET\_READ, you must initialize *data* to be a byte array of sufficient size to hold the data sent from the client. It is up to the programmer to convert the data in the byte array into whatever form the server or client expects to process. PV-WAVE provides several routines for converting data from one type to another, including BYTE, BYTEORDER, COMPLEX, DOUBLE, FLOAT, LONG, and STRING.

## Example

In this example, a client program uses `SOCKET_READ` to retrieve a byte array sent from the server. Note that the `STRING` function is used to convert the byte array to a string before the data is printed.

```
PRO CLIENT
    host = 'localhost'
    port = 1500
    socket = SOCKET_CONNECT(host,port)
    IF socket EQ -1 OR socket EQ -2 THEN BEGIN
        PRINT, 'SOCKET_CONNECT failed with return code: ', socket
        RETURN
    ENDIF
    data = BYTARR(15)
    nbytes = SOCKET_READ(socket,data)
    PRINT, 'CLIENT received: ', STRING(data)
    SOCKET_CLOSE, socket
END
```

## See Also

[SOCKET\\_WRITE](#)

For more detailed information on using the socket routines, see the *PV-WAVE Application Developer's Guide*.

---

## ***SOCKET\_WRITE Procedure***

Writes data to a socket connection.

### **Usage**

`SOCKET_WRITE`, *connection*, *data*

### **Input Parameters**

*connection* — A socket descriptor.

*data* — A byte array of data to write to the socket.

## Keywords

None.

## Discussion

This function can be used in both client and server programs. If used in a client program, it sends data to the server. If used in a server program, it sends data to the client.

All data written with this routine must first be converted to a byte array. For information on converting data to a byte array, refer to documentation on the `BYTE` and `BYTEORDER` functions.

## Example

The following server program uses `SOCKET_WRITE` to send a string to a client. Note that the string is first converted to type byte.

```
PRO SERVER
    port = 1500
    socket = SOCKET_INIT(port)
    connection = SOCKET_ACCEPT(socket)
    data = BYTE('Server String ')
    SOCKET_WRITE, connection, data
    SOCKET_CLOSE, connection
    SOCKET_CLOSE, socket
END
```

## See Also

[SOCKET\\_READ](#)

For more detailed information on using the socket routines, see the *PV-WAVE Application Developer's Guide*.

---

## ***SORT* Function**

Sorts the contents of an array.

### **Usage**

```
result = SORT(array)
```

### **Input Parameters**

*array* — The vector or array to be sorted.

### **Returned Value**

*result* — A vector of subscripts which allow access to the elements of *array* in ascending order.

### **Keywords**

None.

### **Discussion**

String arrays are sorted using the ASCII collating sequence. The result is always a vector of long integer type with the same number of elements as *array*.

The contents of the result are the sorted indices, not the values themselves.

### **Example**

This example uses `SORT` to obtain the vector of subscripts that places the single-precision, floating-point vector *a* into ascending order. To obtain elements of *a* in sorted order, use the vector of subscripts returned by `SORT` as the subscript vector for *a*.

```
a = [4.0, 3.0, 7.0, 1.0, 2.0]
    ; Create an unsorted single-precision, floating-point vector.
b = SORT(a)
    ; Place the vector of sorting subscripts into b.

PRINT, b
      3      4      1      0      2
```

```
PRINT, a(b)
; Print the sorted vector by using b as the subscript vector for a.
1.00000 2.00000 3.00000 4.00000 7.00000
```

## See Also

[AVG](#), [MAX](#), [MIN](#), [MEDIAN](#), [QUERY\\_TABLE](#), [SORTN](#), [UNIQUE](#), [WHERE](#)

---

## SPAWN Procedure (UNIX/OpenVMS)

Spawns a child process to execute a given command.

### Usage

```
SPAWN [, command [, result]]
```

### Input Parameters

*command* — (optional) The name of the command to spawn.

- If present, must be of type string. Under UNIX, *command* can be an array each element is passed to the child process as a separate parameter. Under OpenVMS, *command* is restricted to being a scalar.
- If not present, starts an interactive shell and PV-WAVE execution suspends until the new shell process terminates. This ability is provided primarily for compatibility with the OpenVMS version of PV-WAVE.

---

**TIP** Shells that handle process suspension (e.g., `/bin/csh`) offer a more efficient way to get the same effect.

---

### Output Parameters

*result* — (optional) Indicates where the result is to be directed.

- If present, places the output from the child process into a string array (one line of output per array element).
- If not present, sends the output from the child shell process to the standard output stream.

## Keywords

**Count** — The number of string elements returned if the output is a string variable.

**F77\_Unformatted** — (UNIX only) If set, opens a pipe to the spawned process in a mode to do unformatted FORTRAN F77 input/output.

**Noclisym** — (OpenVMS only) If set, prevents the spawned process from inheriting CLI symbols from its caller. Otherwise, the subprocess inherits all currently defined CLI symbols.

---

**TIP** You may want to specify *Noclisym* to help prevent commands redefined by symbol assignments from affecting the spawned commands.

---

**Nolognam** — (OpenVMS only) If set, prevents the spawned process from inheriting process logical names from its caller. Otherwise, the subprocess inherits all currently defined process logical names.

---

**TIP** You may want to specify *Nolognam* to help prevent commands redefined by logical name assignments from affecting the spawned commands.

---

**Noshell** — (UNIX only) If present and nonzero, specifies that *command* should execute directly as a child process without an intervening shell process. In this case, *command* must be specified as a string array in which the first element is the name of the command to execute and the following parameters are the parameters to be passed to the command.

*Noshell* is useful when performing many spawned operations from a program and speed is a primary concern. Since no shell is present, wildcard characters are not expanded, and other tasks normally performed by the shell do not occur.

**Notify** — (OpenVMS only) If set, broadcasts a message to SYS\$OUTPUT when the subprocess completes or aborts. Otherwise, no message is broadcast. This keyword should not be set unless the *Nowait* keyword is also set.

**Nowait** — (OpenVMS only) If set, causes the calling process to continue executing in parallel with the subprocess. Otherwise, the calling process waits until the subprocess completes.

**Pid** — A named variable for storing the process identification number of the child process.

**Sh** — (UNIX only) If present and nonzero, forces the use of the Bourne shell.

**Unit** — A named variable for storing the number of the file unit.

If present, causes SPAWN to create a child process in the usual manner, but instead of waiting for the specified *command* to finish, SPAWN attaches a bidirectional pipe between the child process and PV-WAVE. From the PV-WAVE session, the pipe appears as a logical file unit. The other end of the pipe is attached to the child process standard input and output.

---

**NOTE** If the *Unit* keyword is used, the *command* parameter must also be present, and *result* is not allowed.

---

## Discussion

Under UNIX, the shell used (if any) is obtained from the SHELL environment variable. Under OpenVMS, the DCL command language interpreter is used.

Once the child process is started, the PV-WAVE session can communicate with it using the normal input/output facilities. After the child process has done its task, the CLOSE procedure is used to delete the process and close the pipe.

Since SPAWN uses GET\_LUN to allocate the file unit, FREE\_LUN should be used to free the unit.

## Example

```
SPAWN, 'ps -a'
```

## See Also

[CALL\\_UNIX](#), [CLOSE](#), [FREE\\_LUN](#), [GET\\_LUN](#),  
[LINKNLOAD](#), [UNIX\\_LISTEN](#), [UNIX\\_REPLY](#)

---

**NOTE** Also see the *PV-WAVE Application Developer's Guide*.

---

---

## ***SORTN* Function**

Standard Library function that sorts an array of n-tuples.

### **Usage**

*result* = SORTN(*a*)

### **Input Parameters**

*a* — An (m,n) array of m n-tuples.

### **Returned Value**

*result* — An array of indices, such that *a(result,\*)* is the sorted version of *a*.

### **Keywords**

None.

### **Examples**

```
a = BYTSCL( RANDOMU(seed,9,2), Top=5 )
```

```
PM, a
```

```
b = SORTN( a )
```

```
PM, b
```

```
PM, a(b,*)
```

### **See Also**

[SORT](#), [UNIQUE](#), [UNIQN](#)

---

## ***SPAWN Procedure (Windows)***

Spawns a child process to execute a given command.

### **Usage**

SPAWN [, *command* [, *result*]]

### **Input Parameters**

*command* — (optional) The name of the command to spawn.

- If present, must be of type string and must be a scalar.
- If not present, starts an interactive shell in a new Command window and PV-WAVE execution suspends until the new Command window is closed.

### **Output Parameters**

*result* — (optional) Indicates where the result is to be directed.

- If present, places the output from the child process into a string array (one line of output per array element).
- If not present, sends the output from the child shell process to the standard output stream. The standard output stream is either the new Command window if one was created or the current Command window.

### **Keywords**

*Console* — If present and nonzero, a new Command window is created and all input and output is directed through the new Command window. This parameter is ignored when the output parameter *result* is specified. The default is to create the new Command window.

---

**NOTE** No keyboard input can be directed to the spawned child process unless a new Command window has been created.

---

*Count* — The number of string elements returned if the output is a string variable.

*Noshell* — If present and nonzero, specifies that *command* should execute directly as a child process without an intervening shell process. In this case, *command* must be specified as a string in which the first element is the name of the command to

execute and the following tokens are the parameters to be passed to the command. The command that is executed with the *Noshell* option cannot be a command that itself creates a shell, such as `dir`.

*Noshell* is useful when performing many spawned operations from a program and speed is a primary concern. Since no shell is present, wildcard characters are not expanded, and other tasks normally performed by the shell do not occur.

***Nowait*** — If set, causes the calling process to continue executing in parallel with the subprocess. Otherwise, the calling process waits until the subprocess completes.

## Example

```
SPAWN, 'dir'
```

## See Also

[CD](#), [LINKLOAD](#)

For background information, see .

---

## ***SPHERE Function***

Defines a spherical object that can be used by the `RENDER` function.

### Usage

```
result = SPHERE()
```

### Parameters

None.

### Returned Value

*result* — A structure that defines an ellipsoidal object.

## Keywords

**Color** — A 256-element double-precision floating-point vector containing the color (intensity) coefficients of the object.

(Default: `Color (*) = 1.0`) For more information, see .

**Decal** — A 2D array of bytes whose elements correspond to indices into the arrays of material properties. For more information, see .

**Kamb** — A 256-element double-precision floating-point vector containing the ambient (flat shaded) coefficients.

(Default: `Kamb (*) = 0.0`) For more information, see .

**Kdiff** — A 256-element double-precision floating-point vector containing the diffuse reflectance coefficients.

(Default: `Kdiff (*) = 1.0`) For more information, see .

**Ktran** — A 256-element double-precision floating-point vector containing the specular transmission coefficients.

(Default: `Ktran (*) = 0.0`) For more information, see .

**Transform** — A 4-by-4 double-precision floating-point array containing the local transformation matrix whose default is the identity matrix. For more information, see .

## Discussion

A SPHERE is used by the RENDER function to render ellipsoid objects, such as for spherical inverse mapping or molecular modeling (atoms). It is centered at the origin, with a radius of 0.5.

You can alter its diameter and orientation with the *Transform* keyword.

## Example

```
ds = 6
checks = BYTARR(ds, ds)
checks(*) = 255
FOR x=0, ds - 1 DO $
    FOR y=0, ds - 1 DO $
        IF ((x mod 2) EQ (y mod 2)) THEN $
            checks(x, y) = 128
    ; Create a 6-by-6 checkerboard image.
two = FLTARR(256)
```

```
two(128) = 0.5 & two(255) = 1.0
; Set checks to be full and one-half intensity.
s = SPHERE(Decal=checks, Color=two)
TV, RENDER(s)
; Render the sphere with the checkerboard decal mapped on.
```

## See Also

[CONE](#), [CYLINDER](#), [MESH](#), [RENDER](#), [VOLUME](#)

For more information, see .

---

## SPLINE Function

Standard Library function that performs a cubic spline interpolation.

### Usage

```
result = SPLINE(x, y, t [, tension])
```

### Input Parameters

*x* — A vector containing the independent coordinates of the dataset. The vector must be monotonic and increasing.

*y* — A vector containing the dependent coordinates of the dataset.

*t* — A vector containing the independent coordinates for which the ordinates are desired. Must be monotonic and increasing.

*tension* — (optional) The amount of tension to be put on the spline curve. The default value is 1.0 (see *Discussion*).

### Returned Value

*result* — A vector containing interpolated ordinate values. In other words, result(i) = value of the function at T(i).

### Keywords

None.

## Discussion

SPLINE performs a cubic spline interpolation of the input vector in order to obtain a vector of interpolated dependent values. The dependent values are calculated at the independent values given by the vector  $t$ .

A tension curve always passes through each known data point. The optional *tension* parameter controls the smoothness of the fitted curve. The higher the tension, the more closely the curve approximates the set of line segments that connect the data points. A lower tension produces a smoother curve that may deviate considerably from the straight-line path between points.

When *tension* is set close to zero (e.g., 0.01), the curve is virtually a cubic spline fit. When *tension* is set to a large value (e.g., >10.0), then the fitted curve will be similar to that obtained from a polynomial interpolation, such as POLY\_FIT or POLYFITW.

The SPLINE function can be useful for those applications where you need to fit the data with a smoother or stiffer curve than that obtained with an interpolating polynomial. Splines also tend to be more stable than polynomials, with less possibility of wild oscillation between the data points.

### Example 1

```
x = FINDGEN(10)
y = RANDOMN(seed, 10)
    ; Create the data.

t = FINDGEN(100)/11.
    ; Create a vector containing the independent points at which
    ; the dependent points are calculated.

PLOT, y
    ; Plot the original data.

PLOT, t, SPLINE(x, y, t), Xstyle=4, Ystyle=4,$
    /Noerase, linestyle=2
    ; Plot using 100 independent points and a default tension of 1.

PLOT, t, SPLINE(x, y, t, 8.), Xstyle=4, $
    Ystyle=4, /Noerase, Linestyle=3
    ; Plot using 100 independent points and a tension of 8.
```

### Example 2

```
x = [2.0, 3.0, 4.0]
    ; X values of the original function.
```

```

y = (x - 3)^2
; Y formed from a quadratic function.
t = FINDGEN(20) / 10. + 2
; Twenty values from 2 to 3.90 for the interpolated points.
z = SPLINE(x, y, t)
; Do the interpolation.

```

## See Also

[CURVEFIT](#), [GAUSSFIT](#), [POLY\\_FIT](#), [POLYFITW](#), [SVDFIT](#)

## SQRT Function

Calculates the square root of the input variable.

### Usage

*result* = SQRT(*x*)

### Input Parameters

*x* — The value or array of values for which the square root is desired.

### Returned Value

*result* — The square root of *x*.

### Keywords

None.

### Discussion

SQRT handles complex numbers (defined by  $c = a + ib$ ), in the following manner:

$$c^{1/2} = \left[ \frac{1}{2}(r+a) \right]^{1/2} \pm i \left[ \frac{1}{2}(r-a) \right]^{1/2}$$

where

$$r = \sqrt{a^2 + b^2}$$

The ambiguous sign is taken to be the same as the sign of  $b$ .

If  $x$  is of double-precision floating-point or complex data type, SQRT yields a result of the same type. All other types yield a single-precision floating-point result.

If  $x$  is an array, the result of SQRT has the same dimensions, with each element containing the square root of the corresponding element of  $x$ .

### Example

```
x = [3, 5, 7, 9]
PRINT, SQRT(x)
      1.73205      2.23607      2.64575      3.00000
```

### See Also

For a list of other transcendental functions, see [Transcendental Mathematical Functions](#) on page 32 in Volume 1 of this reference.

---

## STDEV Function

Standard Library function that computes the standard deviation and (optionally) the mean of the input array.

### Usage

```
result = STDEV(array [, mean])
```

### Input Parameters

*array* — The array to be processed. Can be of any data type except string.

### Output Parameters

*mean* — (optional) The mean of *array*.

## Returned Value

**result** — The standard deviation of *array*. If the *Dimension* keyword is used, then the value of *result* and *mean* will each have the structure of the input array, but with dimension *n* collapsed.

## Keywords

**Dimension** — An integer ( $n \geq 0$ ) indicates that the standard deviation is to be computed over dimension *n* of the input array.

**Variance** — If nonzero, the variance of *array* is returned instead of the standard deviation.

## Discussion

Mean, standard deviation and variance are basic statistical tools used in a variety of applications. They are computed as follows:

$$mean = \frac{\sum_{i=0}^n array_i}{n}$$

$$STD = \sqrt{\frac{\sum_{i=0}^n (array_i - mean)^2}{n - 1}}$$

$$Variance = \frac{\sum_{i=0}^n (array_i - mean)^2}{n - 1}$$

where *n* is the number of elements in *array*.

These equations are implemented as follows:

```
mean = TOTAL(array) / N_ELEMENTS(array)
```

```
std = SQRT(TOTAL((array - mean)^2) / (N_ELEMENTS(array) - 1))
```

The variance is implemented as follows:

```
variance = TOTAL((array - mean)^2) / (N_ELEMENTS(array) - 1)
```

## Example

```
y = [1, 5, 9, 3, 10, 4]
    ; Create a simple array.

std = STDEV(y, arrmean)
    ; Compute the standard deviation and mean array value.

PRINT, std, arrmean
    3.50238    5.33333
    ; Print the results.
```

## See Also

[AVG](#), [SIGMA](#)

---

## STOP Procedure

Stops the execution of a running program and returns control to the interactive mode.

## Usage

STOP [, *expr*<sub>1</sub>, ... , *expr*<sub>n</sub>]

## Input Parameters

*expr*<sub>*i*</sub> — (optional) One or more expressions whose value is printed. If no parameters are present, a brief message describing where the STOP was encountered is printed.

## Keywords

None.

## See Also

[BREAKPOINT](#), [RECALL](#), [RETURN](#)

---

## **STRARR Function**

Returns a string array.

### **Usage**

```
result = STRARR(dim1, ... , dimn)
```

### **Input Parameters**

*dim*<sub>*i*</sub> — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

### **Returned Value**

*result* — A string array.

### **Keywords**

None.

### **Example**

```
r = STRARR(2)
r(0) = 'one'
r(1) = 'two'
PRINT, r
      one  two
```

### **See Also**

[BYTARR](#), [COMPLEXARR](#), [DBLARR](#), [FLTARR](#), [INTARR](#),  
[LONARR](#), [MAKE\\_ARRAY](#), [SINDGEN](#), [STRLEN](#)

---

## ***STRCOMPRESS Function***

Compresses the white space in an input string.

### **Usage**

*result* = STRCOMPRESS(*string*)

### **Input Parameters**

*string* — The string to be compressed.

### **Returned Value**

*result* — A string with all white space (blank spaces and tabs) compressed to a single space or completely removed. If *string* is an array, the result is an array with the same structure—each element contains a compressed copy of the corresponding element of *string*.

### **Keywords**

*Remove\_All* — If nonzero, removes all white space. Otherwise, all white space is compressed to a single space (the default).

### **Discussion**

If not of type string, *string* is converted to string using the default formatting rules of PV-WAVE. (These rules are described in .)

### **Example 1**

In this example, STRCOMPRESS is used to remove the excess white space in a string.

```
s = ' This string   has       extra white space'
      ; Create a string with excess white space characters.
PRINT, STRCOMPRESS(s)
      ; Display the string with all excess white space removed.
This string has extra white space
```

## Example 2

This example uses STRCOMPRESS to remove all white space from each element of a three-element string array.

```
s = strarr(3)
    ; Create a three-element string array.

s(0) = 'a      b  c'
s(1) = '   d    e f'
s(2) = 'g      h   i'
    ; Assign a string to each element of the array.

PRINT, TRANSPOSE(STRCOMPRESS(s, /Remove_All))
abc
def
ghi
    ; Remove all spaces from each element of the array and display
    ; the resulting array. TRANSPOSE is used to display the array as
    ; a column vector.
```

## See Also

[STRLEN](#), [STRPOS](#), [STRPUT](#), [STRTRIM](#)

---

## STRETCH Procedure

Standard Library procedure that linearly expands the range of the color table currently loaded to cover an arbitrary range of pixel values.

### Usage

STRETCH, *low*, *high*

### Input Parameters

***low*** — The lowest pixel value in the selected range; in other words, this is the pixel value that will be displayed with color index 0. (Default: 0)

***high*** — The highest pixel value in the selected range; in other words, this is the pixel value that will be the highest color index available on a particular device (normally 255 on an eight-bit plane device). (Default: !D.N\_Colors - 1)

## Keywords

None.

## Discussion

STRETCH linearly interpolates new Red, Green, and Blue color vectors between *low* and *high*, and then loads them into the image display with LOADCT. The color vectors in the COLORS common block are unchanged.

For example, the command:

```
STRETCH, 100, 150
```

expands the color table so that the pixels in the range of 100 to 150 fill the entire color-intensity range.

To revert to a normal color table, call STRETCH with no parameters.

## Example 1

```
mandril = BYTARR(512, 512)
OPENR, unit, !Data_dir + 'mandril.img', /Get_lun
READU, unit, mandril
FREE_LUN, unit
    ; Read the image.
WINDOW, /Free, Colors=128, XSize=512, YSize=512
TVSCL, mandril
    ; Display the original image using 128 colors.
LOADCT, 4
COLOR_PALETTE
    ; Change the color palette and display it for reference.
!Err = 0
WHILE !Err LE 0 DO BEGIN
    CURSOR, x, y, /Change, /Normal
    range = FIX(y * 128.0 + 5.0)
    STRETCH, 64-range, 64+range
ENDWHILE
    ; Use the position of the cursor to compress or expand the
    ; palette about the middle using the STRETCH function.
```

## Example 2

A simplified version of STRETCH can be written as:

```
PRO STRETCH, lo, hi
    ; Procedure definition.

COMMON colors, r_orig, g_orig, b_orig, r_curr, g_curr, b_curr
    ; Access the color table common block used by
    ; PV-WAVE procedures.

t = BYTSCL(INDGEN(256), Min = lo, Max = hi)
    ; Make a vector of subscripts into the color table arrays.

TVLCT, r_orig(t), g_orig(t), b_orig(t)
    ; Load colors from the original tables transformed by t.

RETURN
END
```

## See Also

[LOADCT](#), [MODIFYCT](#), [TVLCT](#), [WgCtTool](#)

System Variables: [!D.N\\_Colors](#)

For more information about how to access the three color variables in the color table common block, see .

For more information about stretching color tables, see .

---

## **STRING Function**

Converts the input parameters to characters and returns a string expression.

### **Usage**

*result* = STRING(*expr*<sub>1</sub>, ... , *expr*<sub>*n*</sub>)

### **Input Parameters**

*expr*<sub>*i*</sub> — The expressions to be converted to type string.

### **Returned Value**

*result* — The string value for *expr*.

## Keywords

**Format** — Allows the format of the output to be specified in precise detail, using a FORTRAN-style specification. FORTRAN-style formats are described in the *PV-WAVE Programmer's Guide*.

**Print** — If present and nonzero, specifies that any special case processing should be ignored, and that STRING should behave exactly as the PRINT procedure.

## Discussion

STRING can be used to convert one or more expressions to string data type. You can use the *Format* keyword to explicitly specify the format in the converted string. The format specification string used with the *Format* keyword follows this syntax:

*"(q1f1s1f2s2..fnqn)"*

where:

- *q* — is an optional slash (/) record terminator. During output, each record terminator causes the output to move to a new line. During input, each record terminator causes the next line of input to be read.
- *f* — is a FORTRAN-style format string. Some format strings specify how data should be transferred while others control some other function related to how I/O is handled. If you do not specify a format string, the result is formatted automatically.

For more information about the FORTRAN format codes that can be used, refer to the *PV-WAVE Programmer's Guide*.

- *s* — is a comma (,) or a slash (/) that is used to separate input values. The only restriction on these separators is that two commas cannot occur side by side.

---

**NOTE** The quotation marks and parentheses surrounding the format string are required. The quotation marks can be either single or double quotation marks.

---

If the *Format* keyword is not present, PV-WAVE uses its default rules for formatting the output. These rules are described in .

STRING is similar to the PRINT procedure, except that its output is placed in a string rather than being output to the terminal.

## Example 1

The following three commands all yield the string scalar 'ABC':

```
x = STRING([65B,66B,67B])
y = STRING([byte('A'), byte('B'), byte('C')])
z = STRING('A' + 'B' + 'C')
```

In the first expression, the input `x` is a byte vector, and `STRING` converts it into a string scalar. The first parameter is `65B`, which is a notation indicating that the parameter uses a byte data type, but the result is equal to the decimal ASCII code 65.

## Example 2

`STRING` can also be used with a floating-point variable.

```
x = 123.45
result = STRING(x)
PRINT, result
    123.45
```

## Example 3

Here are some examples using the *Format* keyword. In these examples, the *Format* keyword instructs the `STRING` function to convert numbers to a string data type, using an integer format with a maximum field width of 2 characters. In all three cases, the `STRING` command entered is:

```
result = STRING(numbers, Format="(I2)")
```

- Case 1: When `numbers = [36, 9, 72]`, and `result` is computed with the format string shown, the result is:

```
PRINT, result
    36  9 72
```

The values are converted to strings, using a maximum field width of two characters.

- Case 2: When `numbers = [5, 7, 9, 100]`, and `result` is computed with the format string, the result is:

```
PRINT, result
    5  7  9 **
```

The two asterisks indicate that the integer value 100 was too big to be read with an I2 FORTRAN format.

- Case 3: When `numbers = [6.12, 4.507, 4.339]`, and `result` is computed with the format string, the result is:

```
PRINT, result
```

As this example shows, an integer format can be used to output floating point data, but all digits after the decimal point are lost.

## See Also

[PRINT](#), [STRARR](#), [XYOUTS](#)

For more information on string formats, see

For more information on format specification codes, see the *PV-WAVE Programmer's Guide*.

---

## STRJOIN Function

Concatenates all of the elements of a string array into a single scalar string.

### Usage

```
result = STRJOIN(string [, sep])
```

### Input Parameters

*string* — A string array whose elements are to be concatenated.

*sep* — (optional) A scalar string that is used to separate adjoining elements of the input array.

### Returned Value

*result* — A scalar string.

### Keywords

None.

### Example

A string array of directory names is concatenated into a string that can be used for a !Path system variable definition.

```
dirs = ['/bin', '/usr/bin', '/usr/local/bin', $
```

```
    '/etc', '/usr/ucb/bin', '/usr/sbin']  
path = STRJOIN(dirs, ':')  
PRINT, path  
    /bin:/usr/bin:/usr/local/bin:/etc:/usr/ucb/bin:/usr/sbin
```

## See Also

[STRSPLIT](#)

---

## STRLEN Function

Returns the length of the input parameter.

### Usage

```
result = STRLEN(expr)
```

### Input Parameters

*expr* — The expression for which the string length is desired.

If not of type string, *expr* is converted to string using the default formatting rules of PV-WAVE. (These rules are described in .)

### Returned Value

*result* — A long integer containing the string length of *expr*. If *expr* is an array, the result is an array with the same structure, with each element containing the length of the corresponding *expr* element.

### Keywords

None.

### Example

This example uses STRLEN to determine the length of several different strings.

```
a = 'a b c'  
    ; Create a scalar string.  
PRINT, STRLEN(a)
```

```

5
; Display the length of a.

b = 45
; Create an integer scalar variable.

PRINT, STRLEN(b)
8
; Display the length of b converted to string type.

c = STRING(FINDGEN(4), Format = '(f3.1)')
; Create a four-element vector of strings.

PRINT, TRANSPOSE(c)
0.0
1.0
2.0
3.0
; Display the contents of c as a column vector.

PRINT, TRANSPOSE(STRLEN(c))
3
3
3
3
; Display the vector of lengths of elements of c as a column vector.
; This vector is returned by STRLEN.

```

## See Also

[STRMID](#), [STRPOS](#), [STRPUT](#), [STRTRIM](#)

---

## ***STRLOOKUP Function***

Queries, creates, saves, or modifies a string server database.

### **Usage**

*value* = STRLOOKUP(*[name]*)

### **Input Parameters**

***name*** — (optional) Specifies the string name for the database search. The *name* parameter is used to query the string database for its associated value.

## Returned Value

*value* — The returned value depends on the input parameter and/or the use of keywords as shown in the following table.

Value Returned	Parameter or Keywords Used
The value associated with the named string, or a default value if the string isn't found.	<i>name</i> , or <i>name</i> with <i>Default</i> keyword
An integer value of 1 indicating success, or 0 indicating failure.	<i>Add</i> , <i>Load</i> , or <i>Save</i> keywords

## Keywords

**Add** — A string containing a *name: value* pair to merge into the string database for the current session. If the specified string already exists in the database, the *Add* keyword takes precedence. If the string specified with the *Add* keyword doesn't already exist in the database, it is created.

**Default** — (Used only if *name* is specified.) A string containing a default value to be returned, if no match is found in the database for *name*.

**Load** — A string specifying the pathname of a file of strings to be loaded, or merged with the existing string database. Strings merged into the database with the *Load* keyword that match existing strings supercede the existing string definitions in the database.

**Save** — A string specifying the pathname in which to save the strings currently defined in the string database. If the file specified already exists, the contents will be overwritten.

## Discussion

STRLOOKUP provides access to a string database of *name: value* pairs. The string database is created by loading strings from an existing file with the *Load* keyword, or changed by merging strings into the existing database using the *Add* keyword.

String resource files are identified by the application default string (.ads) file extension.

---

**TIP** This function is designed specifically for use in application internationalization, or other customized applications. Internationalization may be achieved by

saving all language-specific messages in one file to be accessed using STRLOOKUP.

---

## Examples

In this example code, STRLOOKUP is used to create the string database by loading a file containing the strings into the string database for the current session.

```
status = STRLOOKUP(Load='/usr/mydir/myapp/myapp.errors')
```

The following example illustrates the syntax for merging a *name: value* pair into the existing string database using the *Add* keyword.

```
status = STRLOOKUP(Add='msg_NoSuchFile: No such file')
```

In this example, STRLOOKUP is used to determine if a specific string exists in the string database, and to obtain the value associated with it if it does. If a matching string name is not found in the string database, the default is set to return the message File OK.

```
value = STRLOOKUP('msg_FileOK', Default='File OK')
```

The following example illustrates how to save all strings currently in the string database to a file.

```
status = STRLOOKUP(Save='/usr/mydir/myapp/myapp.new')
```

## See Also

WoLoadResources, WoLoadStrings in the *PV-WAVE Application Developer's Guide*.

---

## STRLOWCASE Function

Converts a copy of the input string to lowercase letters.

### Usage

```
result = STRLOWCASE(string)
```

### Input Parameters

*string* — The string to be converted.

If not of type string, *string* is converted to string using the default formatting rules of PV-WAVE. (These rules are described in .)

## Returned Value

**result** — A copy of *string* converted to lowercase letters. If *string* is an array, the result is an array with the same structure, with each element containing the substring of the corresponding *string* element.

## Keywords

None.

## Discussion

Only uppercase characters are modified by STRLOWCASE; lowercase and nonalphabetic characters are copied without change.

## Example

This example uses STRLOWCASE to convert uppercase characters to lowercase in several different strings.

```
a = 'A StrInG OF mIXeD CaSe'
    ; Create a string with a mix of uppercase and lowercase characters.
PRINT, STRLOWCASE(a)
    a string of mixed case
    ; Convert the string in a to lowercase and display the result.
b = 45
    ; Create an integer scalar variable.
INFO, STRLOWCASE(b)
    <Expression>      STRING      = '      45'
    ; Examine the result of STRLOWCASE applied to b. Note that b is
    ; converted to a string.
c = STRARR(3)
    ; Create a three-element string vector.
c(0) = 'StrInG 0'
c(1) = 'sTrInG 1'
c(2) = 'StrinG 2'
    ; Assign a string with a mix of uppercase and lowercase characters
    ; to each element of c.
PRINT, TRANSPOSE(STRLOWCASE(c))
```

```
string 0
string 1
string 2
; Display the vector of strings of c after converting them to lowercase.
; Use TRANSPOSE to view the vector as a column.
```

## See Also

[MESSAGE](#), [PRINT](#), [STRING](#), [STRUPCASE](#), [XYOUTS](#)

---

## STRMATCH Function

Matches a specified string to an existing regular expression.

### Usage

```
result = STRMATCH(string, expr [, registers])
```

### Input Parameters

*string* — A string or array of strings to be compared to a regular expression.

*expr* — A regular expression describing the pattern of matching strings for comparison.

### Output Parameters

*registers* — (optional) A string array containing the text of *string*, and matching the regular expression and sub-expressions. The first part of *string* that matches the regular expression is stored in element 0, and any additional matching sub-expressions are stored in elements 1 – 9. If *string* is an array of strings, *registers* only returns the matches for the first element of *string*.

### Returned Value

*result* — A byte value indicating the success or failure of the match.

- 1 Indicates the string matches the regular expression.
- 0 Indicates no match.

If *string* is an array of strings, *result* is an array with the same structure. Each element in the result array contains a value of 1 or 0 (indicating a match, or no match respectively) for the corresponding *string* element.

## Keywords

***Grep*** — If nonzero, the regular expression follows the syntax used by the UNIX command `grep`. Using *Grep* is equivalent to specifying a *Syntax* value of 20.

***Egrep*** — If nonzero, the regular expression follows the syntax used by the UNIX command `egrep`. Using *Egrep* is equivalent to specifying a *Syntax* value of 51.

***Exact*** — If nonzero, the string must match the regular expression exactly.

***Length*** — Specifies a named variable in which to store the number of characters in *string* that match the regular expression. If *Exact* is used with *Length*, the resulting operation is equivalent to `STRLEN(string)`.

***Position*** — If nonzero, specifies a named variable into which the position in *string* where the match occurred is stored. If *Exact* is used with *Position*, the position of the matched string will be zero.

If *string* is an array of strings, the return values of *Length* and *Position* are arrays with the same structure as *string*.

***Syntax*** — The regular expression syntax encoded as bits in a longword. The table in the *Discussion* section provides the syntax details.

## Discussion

A regular expression is a string that uses special characters to represent patterns.

---

**NOTE** STRMATCH uses regular expressions, not wildcard characters, for pattern matching. To use STRMATCH, it is crucial that you understand *regular expressions*. For a detailed discussion of regular expressions, see the chapter *Working with Text* in the *PV-WAVE Programmer's Guide*.

---

By default, STRMATCH uses a regular expression syntax compatible with the UNIX command `awk`. This is equivalent to specifying a *Syntax* value of 35. Keywords are used so STRMATCH uses a syntax compatible with the UNIX commands `grep` or `egrep`; or a completely arbitrary syntax can be specified using *Syntax*. The following table lists information pertinent to *Syntax*.

Bit	Value	Function
0	1	If this bit is set, then ‘ ( ’ and ‘ ) ’ are used for grouping. Otherwise, ‘ \ ( ’ and ‘ \ ) ’ are used for grouping.
1	2	If this bit is set, then ‘   ’ is used as the OR operator. Otherwise, ‘ \   ’ is used as the OR operator.
2	4	If this bit is set, then ‘ \ + ’ and ‘ \ ? ’ are operators. Otherwise, ‘ + ’ and ‘ ? ’ are operators.
3	8	If this bit is set, then ‘   ’ binds tighter than ‘ ^ ’ and ‘ \$ ’. Otherwise, the opposite is true.
4	16	If this bit is set, then newline is treated as an OR operator. Otherwise, newline is treated as a normal character.
5	32	If this bit is set, then certain characters ( ‘ ^ ’, ‘ \$ ’, ‘ * ’, ‘ + ’ and ‘ ? ’) only have special meaning in certain contexts. Otherwise, they always have their special meaning, regardless of the surrounding context.

## Example 1

This example uses STRMATCH to determine if a user’s response to a prompt was valid.

```
x = '' & READ, 'Enter a letter or number: ', x
; Prompt the user for a response.

IF NOT STRMATCH(x, '[a-zA-Z0-9]', /Exact) THEN PRINT, 'Invalid
response.'
; If the response is not a single letter or number, an error message is
; issued.
```

## Example 2

In this example, STRMATCH is used to ensure that names entered by a user are always in “First name, Last name” order.

```
name = 'Smith, Bill J.'

IF STRMATCH(name, '^(.*)', '*(.*)$', reg) THEN name = reg(2) + ' ' +
reg(1)
; If two parts of name are separated by a comma, store the two parts
; in the register and then concatenate them back together to form the
; new name.
```

```
PRINT, name
    Bill J. Smith
```

### **Example 3**

This example demonstrates how to use the STRMATCH function in conjunction with WHERE to perform data subsetting.

```
owners = ['Bill', 'Kathy', 'Janis', 'Ken']
pets = ['cat', 'rabbit', 'fish', 'dog']
    ; The two string arrays associate the owners with their pets.
w = STRMATCH(pets, 'cat|dog')
```

```
PRINT, owners(WHERE(w)), Format='(A)'
    Bill
    Ken
    ; Print the names of any owners of a cat or dog.
```

### **See Also**

[PRINT](#), [STRING](#), [STRSPLIT](#), [STRSUBST](#)

For detailed information on regular expressions, see the chapter *Working with Text* in the *PV-WAVE Programmer's Guide*.

---

## **STRMESSAGE Function**

Returns the text of the error message specified by the input error number.

### **Usage**

```
result = STRMESSAGE(errno)
```

### **Input Parameters**

*errno* — The error number for which the message text is desired.

### **Returned Value**

*result* — The text of the error message specified by *errno*.

## Keywords

None.

## Discussion

This function is especially useful in conjunction with the !Err system variable, which always contains the error number of the last error.

However, you should take care not to make the assumption in your programs that certain error numbers always correspond to certain error messages, as this correspondence may change as PV-WAVE is modified over time.

## Example

The following procedure uses function STRMESSAGE to display the error message associated with an input/output error trapped by procedure ON\_IOERROR.

```
FUNCTION READ_DATA, file_name
    ; Define a function to read, and return a 100-element, floating-point
    ; array.

ON_IOERROR, bad
    ; Declare error label.

OPENR, unit, file_name, /Get_Lun
    ; Open the file and use the Get_Lun keyword to allocate a logical file
    ; unit.

a = FLTARR(100)
    ; Define data array.

READU, unit, a
    ; Read data.

GOTO, DONE
    ; Clean up and return.
    bad:
    ; Exception label.

PRINT, STRMESSAGE(!ERROR)
    ; Print the error message associated with the error number in
    ; !ERROR. Note that STRMESSAGE is used to display the message.

DONE:
FREE_LUN, unit
    ; Close and free the input/output unit.

RETURN, a
```

; Return the result. Variable `a` is undefined if an error occurred.

END

## See Also

[MESSAGE](#), [ON\\_ERROR](#), [ON\\_IOERROR](#), [PRINT](#)

System Variables: [!Err](#), [!Err\\_String](#)

For more information on error handling, see .

---

## STRMID Function

Extracts a substring from a string expression.

### Usage

*result* = STRMID(*expr*, *first\_character*, *length*)

### Input Parameters

*expr* — The expression from which the substring is to be extract-ed.

If not of type string, *expr* is converted to string using the default formatting rules of PV-WAVE. (These rules are described in .)

*first\_character* — The starting position within *expr* at which the substring starts. The first character position is position 0.

*length* — The length of the substring. If there are not enough characters left in the main string to obtain *length* characters, the substring will be truncated.

### Returned Value

*result* — A string of *length* characters taken from *expr*, starting at character position *first\_character*. If *expr* is an array, the result is an array with the same structure, with each element containing the substring of the corresponding *expr* element.

### Keywords

None.

## Example

In this example, `STRPOS` is used to locate the first occurrence of the string *a* within a larger string. Function `STRMID` is then used to extract a substring from that position. Function `STRPOS` is used to locate the second occurrence of the string *a*, and finally, `STRMID` is used to extract another substring from this second position within the larger string.

```
a = 'Extract a substring from a string'
    ; Create a string in the variable a.

POS = STRPOS(a, 'a ')
    ; Locate first occurrence of the string "a ".

PRINT, STRMID(a, pos, 11)
    a substring
    ; Extract 11 characters from a, starting at POS.

POS = STRPOS(a, 'a ', pos + 1)
    ; Search for the second occurrence of the string "a " by searching
    ; from the character position that is one greater than the first
    ; occurrence of that string. Extract 8 characters from a, starting at the
    ; new POS.

PRINT, STRMID(a, pos, 8)
    a string
```

## See Also

[STRLEN](#), [STRPOS](#), [STRPUT](#)

---

## ***STRPOS Function***

Searches for the occurrence of a substring within an object string, and returns its position.

### Usage

```
result = STRPOS(object, search_string [, position])
```

### Input Parameters

*object* — The expression in which to search for the substring.

*search\_string* — The substring to be searched for within *object*.

If *object* or *search\_string* are not of type string, they are converted to string using the default formatting rules of PV-WAVE. (These rules are described in .)

*position* — (optional) The character position at which the search is begun. If *position* is omitted or is less than zero, the search begins at the first character (character position 0).

## Returned Value

*result* — If *search\_string* occurs in *object*, STRPOS returns the character position of the match; otherwise, it returns  $-1$ .

If *object* is an array, *result* is an array of the same structure, and each element contains the position of the substring within *object*.

If *search\_string* is the null string, STRPOS returns the smaller of *position* or one less than the length of *object*.

## Keywords

None.

## Example

See the example for STRMID.

## See Also

[STRLEN](#), [STRLOWCASE](#), [STRMESSAGE](#), [STRPUT](#), [STRUPCASE](#)

---

## **STRPUT Procedure**

Inserts the contents of one string into another.

### **Usage**

STRPUT, *destination*, *source* [, *position*]

### **Input Parameters**

***destination*** — The string into which *source* will be inserted. Must be a named variable of type string. Cannot be an element of an array. If *destination* is an array, *source* is inserted into every element.

***source*** — The string to be inserted into *destination*. Must be scalar.

If not of type string, *source* is converted to string using PV-WAVE's default formatting rules. (These rules are described in .)

***position*** — (optional) The character position at which the insertion is begun. If *position* is omitted or is less than zero, the search begins at the first character (character position 0).

### **Keywords**

None.

### **Discussion**

The *source* string is inserted into the *destination* string, starting at the given *position*. All characters in *destination* that occur either before the starting position, or after the starting position plus the length of *source*, remain unchanged.

### **Example**

This example uses STRPUT to insert a substring into a larger string, starting at position 0 of the destination string. Clipping is then demonstrated by using STRPUT to insert a substring that would otherwise extend the length of the destination string.

```
a = 'Strings are fun'  
    ; Create a string in the variable a.  
STRPUT, a, 'PV-WAVE is '
```

```
    ; Insert the string "PV-WAVE is" into a, starting at position 0.
PRINT, a
    PV-WAVE is fun
    ; Display the result.
STRPUT, a, 'fun to use', 12
    ; Insert a string into a that would extend its length.
PRINT, a
    PV-WAVE is fun
    ; Display the result. Note that the inserted string was clipped so the
    ; length of a did not change.
```

## See Also

[STRLEN](#), [STRLOWCASE](#), [STRMESSAGE](#), [STRPOS](#), [STRUPCASE](#)

---

## ***STRSPLIT*** Function

Splits a string into an array of substrings called tokens.

### Usage

*tokens* = STRSPLIT(*string*, *pattern*)

### Input Parameters

*string* — The scalar string that is scanned and split into tokens.

*pattern* — A scalar string specifying a regular expression. This regular expression is the delimiter used to determine where to split the string.

### Returned Value

*tokens* — A string array of substrings.

### Keywords

None.

## Discussion

---

**NOTE** STRSPLIT uses regular expressions, not wildcard characters, for pattern matching. To use STRSPLIT, it is crucial that you understand *regular expressions*. For a detailed discussion of regular expressions, see the chapter *Working with Text* in the *PV-WAVE Programmer's Guide*.

---

## Example

In this example, a string containing a colon (:) separated list of directories is split into a string array of directories.

```
PATH = '/bin:/usr/bin:/usr/local/bin:' + $
      '/etc:/usr/ucb/bin:/usr/sbin'
dirs = STRSPLIT(path, ':')
PRINT, dirs, Format='(A)'
    /bin
    /usr/bin
    /usr/local/bin
    /etc
    /usr/ucb/bin
    /usr/sbin
```

## See Also

[STRJOIN](#), [STRMATCH](#)

For detailed information on regular expressions, see the chapter *Working with Text* in the *PV-WAVE Programmer's Guide*.

---

## STRSUBST Function

Performs search and replace string substitution.

### Usage

```
result = STRSUBST(expr, pattern, repl)
```

### Input Parameters

*expr* — The original scalar string expression.

*pattern* — A scalar string specifying the regular expression for the search pattern.

*repl* — The replacement scalar string.

### Returned Value

*result* — A scalar string containing the given substitutions.

### Keywords

*Global* — If nonzero, causes the *pattern* to be replaced everywhere in the string with *repl*.

### Discussion

---

**NOTE** STRSUBST uses regular expressions, not wildcard characters, for pattern matching. To use STRSUBST, it is crucial that you understand *regular expressions*. For a detailed discussion of regular expressions, see the chapter *Working with Text* in the *PV-WAVE Programmer's Guide*.

---

By default, only the first occurrence of *pattern* is replaced. Use the *Global* keyword to replace all occurrences of *pattern*.

### Example 1

In this example, a list of .pro files is converted to .cpr files.

```
source = 'foo.pro, bar.pro'
cprfiles = STRSUBST(source, 'pro', 'cpr', $
    /Global)
```

```
    ; The letters "pro" are replaced by "cpr".  
PRINT, cprfiles  
    foo.cpr, bar.cpr
```

## Example 2

Suppose that you wish to change the file extension in the following string from .pp to .dat.

```
str1='$MDATA/ppt/thefile.pp'
```

You might try the following STRSUBST command:

```
str2=STRSUBST(str1, '.pp', '.dat')
```

Unfortunately, this command returns the following string, which is not the desired result:

```
PRINT, str2  
    $MYDATA/.datt/thefile.pp
```

The reason for this is that the *pattern* expression, '.pp', contains the regular expression special character . (dot), which means "match any single character except newline". One way to obtain the correct result is to "escape" this special character with the backslash (\), as follows:

```
str2=STRSUBST(str1, '\\.pp', '.dat')  
PRINT, str2  
    $MYDATA/ppt/thefile.dat
```

---

**NOTE** Two backslashes are needed because in PV-WAVE strings, you need a pair of backslashes to make a single backslash. For more information on this subject, see the chapter *Working with Text* in the *PV-WAVE Programmer's Guide*.

---

## See Also

[STRPUT](#), [STRMATCH](#)

For detailed information on regular expressions, see the chapter *Working with Text* in the *PV-WAVE Programmer's Guide*.

---

## STR\_TO\_DT Function

Converts date and time string data to date/time values.

### Usage

```
result = STR_TO_DT(date_strings [, time_strings])
```

### Input Parameters

*date\_strings* — A string constant or string array that contains date strings.

*time\_strings* — (optional) A string constant or string array that contains time strings.

### Returned Value

*result* — A date/time variable containing the converted date/time data.

### Keywords

*Date\_Fmt* — Specifies the format of the date data in the input variable. Possible values are 1, 2, 3, 4, or 5, as summarized in the following table:

Value	Format Description	Examples for May 1, 1992
1	MM*DD*[YY]YY	05/01/92
2	DD*MM*[YY]YY	01-05-92
3	ddd*[YY]YY	122,1992
4	DD*mmm[mmmmmm]*[YY]YY	01/May/92
5	[YY]YY*MM*DD	1992-05-01

where the asterisk (\*) represents one of the following separators: dash (–), slash (/), comma (,), period (.), or colon (:).

*Time\_Fmt* — Specifies the format of the time portion of the data in the input variable. Possible values are –1 or –2, as summarized in the following table:

Value	Format Description	Examples for 1:30 p.m.
-1	HH*Mn*SS[.SSSS]	13:30:35.25
-2	HHMn	1330

where the asterisk (\*) represents one of the following separators: dash (-), slash (/), comma (,), or colon (:). No separators are allowed between hours and minutes for the -2 format. Both hours and minutes must occupy two spaces.

For a detailed description of the date and time formats, see .

---

**CAUTION** Date and time separators are specified with the !Date\_Separator and !Time\_Separator system variables. The STR\_TO\_DT function only recognizes the standard separators listed above. If any other separator is specified, this function does not work as expected.

---

## Discussion

You can convert just date strings, just time strings, or both. If you do not pass in a date string, the resulting date portion of the date/time structure defaults to the value in the system variable !DT\_Base. If you do not pass in a time string, the time portion of the resulting date/time variable defaults to zero.

## Example 1

```
x = ['3-13-92', '4-20-83', '4-24-64']
    ; Create an array that contains some date strings with the MM DD YY
    ; date format.

y = ['1:10:34', '16:18:30', '5:07:25']
    ; Create an array that contains some time strings with the HH Mn SS
    ; date format.

date1 = STR_TO_DT(x, y, Date_Fmt=1, Time_Fmt=-1)
        ; Use the formats1 and -1 to return date/time data.

DT_PRINT, date1
    3/13/1992 01:10:34
    4/20/1983 16:18:30
    4/24/1964 05:07:25
```

## Example 2

```
date2 = STR_TO_DT('3-13-92', Date_Fmt=1)
PRINT, date2
      { 1992 3 13 0 0 0.00000 87474.000 0 }
```

### See Also

[DT\\_TO\\_STR](#), [JUL\\_TO\\_DT](#), [SEC\\_TO\\_DT](#), [VAR\\_TO\\_DT](#)

For more information on using date/time functions, see .

---

## STRTRIM Function

Removes extra blank spaces from an input string.

### Usage

```
result = STRTRIM(string [, flag])
```

### Input Parameters

*string* — The string that will have leading and/or trailing blanks removed.

If not of type string, *string* is converted to string using the default formatting rules of PV-WAVE. (These rules are described in .)

*flag* — Controls the action of STRTRIM:

- If zero or not present, removes trailing blanks.
- If 1, removes leading blanks.
- If 2, removes both leading and trailing blanks.

### Returned Value

*result* — A copy of *string* with extra (leading and/or trailing) blank spaces removed. If *string* is an array, the result is an array with the same structure—each element contains a trimmed copy of the corresponding element of *string*.

### Keywords

None.

## Example

In this example, STRTRIM is used to trim trailing, leading, then trailing and leading blanks from a string.

```
a = '   A String   '
; Create a string with both leading and trailing blanks.

INFO, a
; Examine the contents of a. Note the presence of both leading and
; trailing blanks.

VARIABLE          STRING      = '   A String   '

b = STRTRIM(a)
; Remove trailing blanks from a.

INFO, b
VARIABLE          STRING      = '   A String   '
; Note that all trailing blanks are removed.

b = STRTRIM(a, 1)
; Remove leading blanks from a.

INFO, b
VARIABLE          STRING      = 'A String   '
; Examine the results. Note that all leading blanks are removed.

b = STRTRIM(a, 2)
; Remove both leading and trailing blanks from a.

INFO, b
VARIABLE          STRING      = 'A String'
; Note that all leading and trailing blanks are removed.
```

## See Also

[STRCOMPRESS](#)

---

## **STRUCTREF Function**

Returns a list of all existing references to a structure.

### **Usage**

```
result = STRUCTREF({structure})
```

### **Input Parameters**

*structure* — The structure name. The name can be specified as {*structure*}, "*structure*", or *x*, where *x* is a variable of type structure.

### **Returned Value**

*result* — A list of the variables, structures, definitions, and common blocks that reference *structure*.

### **Keywords**

None.

### **Discussion**

Use STRUCTREF before you use DELSTRUCT to check if the structure you want to delete is currently referenced by any variables, common blocks, or other structure definitions.

You *cannot* delete a structure that is currently referenced.

If you want to delete a structure that is referenced, you can either:

- Delete the references (variables, structures, etc.) returned by STRUCTREF, and then use the DELSTRUCT procedure.
- Use the DELSTRUCT procedure with the *Rename* keyword. This renames the structure.

If you are trying to free memory, then you must pursue the first option. If you want to delete a structure, and memory is not a concern, then the second option is probably the best choice.

## Example

```
x = {struct1, a:float(0)}  
PRINT, structref(x)  
    <procedure $MAIN$, symbol X>
```

## See Also

[DELFUNC](#), [DELPROC](#), [DELSTRUCT](#), [N\\_TAGS](#), [TAG\\_NAMES](#)

For more information on structures, see .

---

## **STRUPCASE Function**

Converts a copy of the input string to uppercase letters.

### Usage

*result* = STRUPCASE(*string*)

### Input Parameters

*string* — The string to be converted.

If not of type string, *string* is converted to string using the default formatting rules of PV-WAVE. (These rules are described in .)

### Returned Value

*result* — A copy of *string* converted to uppercase letters. If *string* is an array, the result is an array with the same structure, with each element containing the substring of the corresponding *string* element.

### Keywords

None.

### Discussion

Only lowercase characters are modified by STRUPCASE; uppercase and non-alphabetic characters are copied without change.

## Example

This example uses `STRUPCASE` to convert lowercase characters to uppercase in several different strings.

```
a = 'A StRInG oF miXeD caSE'
PRINT, STRUPCASE(a)
    A STRING OF MIXED CASE
    ; Create a string with a mix of uppercase and lowercase characters.
    ; Convert the string in a to uppercase and display the result.

b = 45
INFO, STRUPCASE(b)
    <Expression>      STRING = '      45'
    ; Create an integer scalar variable. Examine the result of
    ; STRUPCASE applied to b. Note that b is converted to a string.

c = STRARR(3)
    ; Create a three-element string vector.

c(0) = 'StrInG 0'
c(1) = 'sTrINg 1'
c(2) = 'StrInG 2'
    ; Assign a string with a mix of uppercase and lowercase characters
    ; to each element of c.

PRINT, TRANSPOSE(STRUPCASE(c))
    STRING 0
    STRING 1
    STRING 2
    ; Display the vector of strings of c after converting it to uppercase.
    ; Use TRANSPOSE to view the vector as a column.
```

## See Also

[MESSAGE](#), [PRINT](#), [STRLOWCASE](#), [STRING](#), [XYOUTS](#)

---

## SUM Function

Sums an array of  $n$  dimensions over one of its dimensions.

### Usage

*result* = SUM (*array*, *dim*)

### Input Parameters

*array* — An array of  $n$  dimensions of any data type except string.

*dim* — The dimension over which *array* is summed. The *dim* parameter must be a number in the range  $0 \leq dim \leq (n-1)$ , where  $n$  is the number of dimensions in *array*.

### Returned Value

*result* — A scalar value equal to the sum of the array elements over the specified dimension.

### Keywords

None.

### Discussion

If *array* is either a single or double-precision complex, or a double-precision floating-point data type, the result is of the same data type. SUM returns a single-precision floating-point value for all other acceptable data types.

### Example

This example illustrates SUM being used for a variety of arrays of different sizes.

```
array1 = FINDGEN(2)
array2 = FINDGEN(2,2)
      ; Create two arrays, array1, a one-dimensional array with two
      ; elements and array2, a two-dimensional array with four elements,
      ; using FINDGEN.

PRINT, array1
      0.00000      1.00000
```

```

PRINT, array2
    0.00000    1.00000
    2.00000    3.00000

PRINT, SUM(array1, 0)
    1.00000

PRINT, SUM(array2, 0)
    1.00000    5.00000

PRINT, SUM(array2, 1)
    2.00000    4.00000
; Print the arrays then sum each array over each of its dimensions
; and print the result.

```

## See Also

[AVG](#), [MAX](#), [MEDIAN](#), [MIN](#), [STDEV](#), [TOTAL](#)

## ***SURFACE Procedure***

Draws the surface of a two-dimensional array with hidden lines removed.

### Usage

`SURFACE, z [, x, y]`

### Input Parameters

*z* — A two-dimensional array containing the values that make up the surface. If *x* and *y* are supplied, the surface is plotted as a function of the X,Y locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of *z*.

*x* — (optional) A vector or two-dimensional array specifying the *x*-coordinates for the surface.

- If *x* is a vector, each element of *x* specifies the *x*-coordinate for a column of *z*. For example, *x*(0) specifies the *x*-coordinate for *z*(0, \*).
- If *x* is a two-dimensional array, each element of *x* specifies the *x*-coordinate of the corresponding point in *z* (*x*<sub>*ij*</sub> specifies the *x*-coordinate for *z*<sub>*ij*</sub>).

*y* — (optional) A vector or two-dimensional array specifying the *y*-coordinates for the surface.

- If  $y$  is a vector, each element of  $y$  specifies the  $y$ -coordinate for a row of  $z$ . For example,  $y(0)$  specifies the  $y$ -coordinate for  $z(*, 0)$ .
- If  $y$  is a two-dimensional array, each element of  $y$  specifies the  $y$ -coordinate of the corresponding point in  $z$  ( $y_{ij}$  specifies the  $y$ -coordinate for  $z_{ij}$ ).

## Keywords

Keywords let you control many aspects of the plot's appearance. SURFACE keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

Ax	Noclip	[XYZ]Gridstyle
Az	Nodata	[XYZ]Margin
Background	Noerase	[XYZ]Minor
Bottom	Normal	[XYZ]Range
Channel	Position	[XYZ]Style
Charsize	Save	[XYZ]Tickformat
Charthick	Skirt	[XYZ]Ticklen
Clip	Subtitle	[XYZ]Tickname
Color	T3d	[XYZ]Ticks
Data	Thick	[XYZ]Tickv
Device	Tickformat	[XYZ]Title
Font	Ticklen	YLabelCenter
Horizontal	Title	ZAxis
Linestyle	Upper_Only	ZValue
Lower_Only	[XYZ]Charsize	

---

## Discussion

Note the following restrictions on the use of SURFACE. If the X,Y grid is not regular or nearly regular, errors in hidden line removal will occur.

If the *T3D* keyword is set, the 3D to 2D transformation matrix contained in !P.T must project the Z axis to a line parallel to the device Y axis, or errors will occur.

## Example

This example displays the surface described by the function

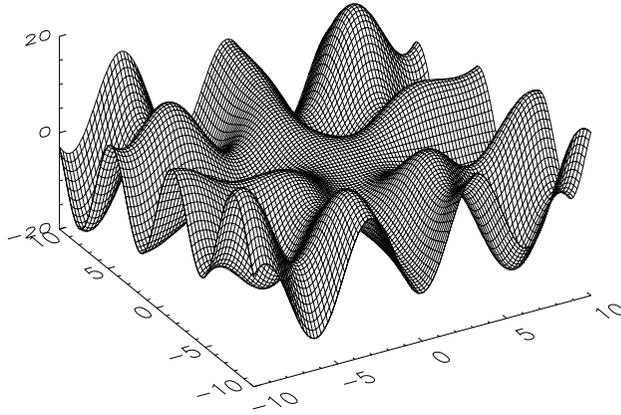
$$f(x, y) = x \sin(y) + y \cos(x)$$

where

$$(x, y) \in \{\mathbb{R}^2 \mid x, y \in [-10, 10]\}$$

```
.RUN
  FUNCTION f, x, y
  RETURN, x * SIN(y) + y * COS(x)
  END
  ; Define the function.
x = FINDGEN(101)/5-10
  ; Create vector of x-coordinates.
y = x
  ; Create vector of y-coordinates.
z = FLTARR(101, 101)
  ; Create an array to hold the function values.
FOR i = 0, 100 DO FOR j = 0, 100 DO z(i, j) = f(x(i), y(j))
  ; Evaluate the function at the given x- and y-coordinates and
  ; place the result in z.
SURFACE, z, x, y, Ax = 50, XCharSize = 2, $
  YCharSize = 2, ZCharSize = 2
  ; Display the surface. The Ax keyword is used to specify the
  ; angle of rotation about the x-axis. The XCharSize, YCharSize,
  ; and ZCharSize keywords are used to enlarge the characters
  ; used to annotate the axes.
XYOUTS, 163, 477, $
  "f(x, y) = x*sin(y) + y*cos(x)", CharSize = 2, /Device
  ; Place a title on the window. Note that the CURSOR procedure
  ; with the Device keyword was used to locate the proper position
  ; for the title.
```

$$f(x, y) = x \cdot \sin(y) + y \cdot \cos(x)$$



**Figure 2-65** Surface with title.

## See Also

[SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE\\_FIT](#), [SURFR](#), [T3D](#), [THREED](#)

System Variables: [!P.T](#)

For more information, see .

---

## ***SURFACE\_FIT*** Function

Standard Library function that determines the polynomial fit to a surface.

### **Usage**

*result* = SURFACE\_FIT(*array*, *degree*)

### **Input Parameters**

*array* — The two-dimensional array of data to which the surface will be fit. The two dimensions do not have to be the same size. The number of data points must be  $\geq (\text{degree} + 1)^2$ .

*degree* — The maximum degree of the fit (in one dimension).

## Returned Value

*result* — A two-dimensional array of values as calculated from the least-squares fit of the data.

## Keywords

None.

## Discussion

SURFACE\_FIT first determines the coefficients of the polynomials in both directions, using the least-squares method of the POLYWARP function. It then uses these coefficients to calculate the data points of the *result* array.

## Example

```
OPENR, 1, !Data_dir + 'pikeselev.dat'  
data = FLTARR(60, 40)  
READF, 1, data  
    ; Read in the data from the pikes elevation file.  
  
CLOSE, 1  
    ; Close the input file.  
  
SURFACE, data  
    ; Show the raw data without fitting.  
  
SURFACE, SURFACE_FIT(data, 1)  
    ; Approximate a linear least-squares fit.  
  
SURFACE, SURFACE_FIT(data, 2)  
    ; Do a second-degree polynomial fit.  
  
SURFACE, SURFACE_FIT(data, 3)  
    ; Do a third-degree polynomial fit.  
  
SURFACE, SURFACE_FIT(data, 4)  
    ; Do a forth-degree polynomial fit.  
  
SURFACE, SURFACE_FIT(data, 5)  
    ; Do a fifth-degree polynomial fit.
```

## See Also

[POLY\\_FIT](#), [POLYWARP](#), [SURFACE](#)

---

## ***SURFR Procedure***

Standard Library procedure that duplicates the rotation, translation, and scaling of the SURFACE procedure.

### **Usage**

SURFR

### **Parameters**

None.

### **Keywords**

*Ax* — The angle of rotation about the *x*-axis in degrees. (Default: 30 degrees)

*Az* — The angle of rotation about the *z*-axis in degrees. (Default: 30 degrees)

### **Discussion**

SURFR should be used for axonometric projections only. The 4-by-4 matrix in the system variable !P.T receives the homogeneous transformation matrix generated by this procedure.

SURFR performs the following steps:

- Translates the unit cube so that the center (.5, .5, .5) is moved to the origin.
- Rotates  $-90$  degrees about the *x*-axis to make the  $+z$ -axis of the data the  $+y$ -axis of the display. The  $+y$  data axis extends from the front of the display to the back.
- Rotates about the *y*-axis *Az* degrees. This rotates the result counterclockwise as seen from above the page.
- Rotates about the *x*-axis *Ax* degrees, tilting the data towards the viewer.
- Translates back to the origin and scales the data so that the data are still contained within the unit cube after the transformation. (This step uses the SCALE3D procedure.)

## Example 1

```
TEK_COLOR
    ; Load a color table.

data = HANNING(50, 50)

SURFR
    ; Create the 3D spatial transformation. By default, the angle of
    ; rotation is defined as 30 degrees around the x- and y-axes.

CONTOUR, data, Nlevels=20, /Follow, /T3D, $
    Charsize=1.5
    ; Create the contour plot in the 3D space.

SURFR, Ax=45, Az=10
    ; Now try it with different angles of rotation.

CONTOUR, data, Nlevels=20, /Follow, /T3D, $
    Charsize=1.5
    ; Create the contour plot in the 3D space.
```

---

**TIP** For an informative display, the above example could be done using the `SHADE_SURF` or `SURFACE` procedures instead of `SURFR`, as shown below. (`SURFR` is not needed in this case because these two procedures themselves set up the 3D view area.)

```
SURFACE, data, /Save, Color=3
CONTOUR, data, Nlevels=20, /Follow, /T3D, Charsize=1.5
```

---

## Example 2

This example displays a cube as a shaded surface and rotates it.

```
xmin = 0 & xmax = 1
xmin = xmin - 1.5 & xmax = xmax+1.5
!x.s = [-xmin,1.0] / (xmax - xmin)
!y.s = !x.s
!z.s = !x.s
    ; Set up scaling for the 3D view.

vert = FLTARR(3, 8)
FOR i = 1, 2 DO vert(0, i) = 1
FOR i = 5, 6 DO vert(0, i) = 1
FOR i = 2, 3 DO vert(1, i) = 1
FOR i = 6, 7 DO vert(1, i) = 1
FOR i = 4, 7 DO vert(2, i) = 1
    ; Create the array of vertices.
```

```

poly = FLTARR(30)
poly(0:4)   = [4, 0, 3, 2, 1]
poly(5:9)   = [4, 1, 2, 6, 5]
poly(10:14) = [4, 5, 6, 7, 4]
poly(15:19) = [4, 4, 7, 3, 0]
poly(20:24) = [4, 3, 7, 6, 2]
poly(25:29) = [4, 0, 1, 5, 4]
           ; Create the connectivity array.

SURFR
img = POLYSHADE(ver, poly, /T3D, /Data)
TV, img
           ; Display the cube with the default rotations.

SURFR, Ax=50, Az=10
img = POLYSHADE(ver, poly, /T3D, /Data)
TV, img
           ; Display the cube rotated 50 degrees around the x-axis and 10
           ; degrees around the z-axis.

```

---

**TIP** This example can be done more simply with the CENTER\_VIEW procedure, which is available in the Advanced Rendering Library. This library is located:

```

(UNIX)      <wavedir>/demo/ar1
(OpenVMS)  <wavedir>:[DEMO.ARL]
(Windows)  <wavedir>\demo\ar1

```

Where <wavedir> is the main PV-WAVE directory. For more information, view the .pro file for this procedure.

---

## See Also

[SCALE3D](#), [SURFACE](#), [T3D](#)

System Variables: [!P.T](#)

---

## SVBKSB Procedure

Uses “back substitution” to solve the set of simultaneous linear equations  $A\mathbf{x} = \mathbf{b}$ , given the  $u$ ,  $w$ , and  $v$  arrays created by the SVD procedure from the matrix  $a$ .

### Usage

SVBKSB,  $u$ ,  $w$ ,  $v$ ,  $b$ ,  $x$

### Input Parameters

$u$  — The  $m$ -by- $n$  column matrix of the decomposition of  $a$ , as returned by SVD.

$w$  — The vector of singular values, as returned by SVD.

$v$  — The  $n$ -by- $n$  orthogonal matrix of the decomposition of  $A$ , as returned by SVD.

$b$  — The vector containing the right-hand side of the equation.

### Output Parameters

$x$  — A variable containing the result.

### Keywords

None.

### Discussion

Since no input quantities are destroyed, SVBKSB may be called numerous times with different  $b$  vectors.

### Example

Here’s a typical use of SVD and SVBKSB to solve the system  $A\mathbf{x} = \mathbf{b}$ :

```
SVD, A, w, u, v
; Decompose square matrix A.

small = WHERE(w LT MAX(w) * 1.0e-6, count)
; Get subscripts of singular values, using a given threshold.

IF count NE 0 THEN w(small) = 0.0
; Zero singular values less than the threshold. See the Numerical
; Recipes book (referred to in the See Also section) for details.
```

```
SVBKSB, u, w, v, b, x
    ; The vector x now contains the solution.
PRINT, TOTAL (ABS (A # x - b) )
    ; The residual,  $\mathbf{Ax} - \mathbf{b}$  should be near 0.
```

## See Also

### [SVD](#)

SVBKSB is based on the routine of the same name in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988. It is used by permission.

---

## SVD Procedure

Performs a singular value decomposition on a matrix.

### Usage

```
SVD, a, W [, u [, v]]
```

### Input Parameters

*a* — The two-dimensional matrix to be decomposed. It has *m* rows and *n* columns, and *m* must be greater than or equal to *n*.

### Output Parameters

*W* — An *n*-element vector of “singular values” equal to the diagonal elements of *w*:  $w_{i,j} = W_j$ ,  $w_{i,j} = 0$  for  $i \neq j$ .

*u* — (optional) The *m*-by-*n* column matrix of the decomposition of *a*.

*v* — (optional) The *n*-by-*n* orthogonal matrix of the decomposition of *a*.

### Keywords

None.

## Discussion

SVD performs a singular value decomposition on a matrix  $a$ . It is useful for solving linear least-squares equations. SVD is able to deal with matrices that are singular or very close to singular.

The SVD procedure function transforms an  $m$ -by- $n$  matrix  $a$  to the product of an  $m$ -by- $n$  column orthogonal matrix  $u$ , an  $n$ -by- $n$  diagonal matrix  $w$ , and the transpose of an  $n$ -by- $n$  orthogonal matrix  $v$ . In other words,  $u$ ,  $w$ , and  $v$  are matrices that are calculated by SVD.

By definition, the product of these three matrices ( $u$ ,  $w$ ,  $v$ ) is equal to  $a$ :

$$a = uwv^t$$

where the superscript  $t$  denotes the matrix transpose.

In a linear system of equations,

$$Ax = b$$

where there are at least as many equations as unknowns, the least-squares solution vector  $x$  is given by:

$$x = v \cdot [\text{diag}(1/w_i)](u^t \cdot b)$$

## Example

A PV-WAVE function that returns  $x$ , given  $A$  and  $b$  is:

```
Function SVD_SOLVE, A, b
    ; Return the vector x, the solution of the system of equations Ax = b.
    ; A is an (m, n) matrix, where m ≥ n.

SVD, A, w, u, v
    ; Call SVD to decompose A.

n = N_ELEMENTS(w)
    ; Make the diagonal matrix Wi,i = 1/wi.

wp = FLTARR(n,n)

FOR i=0, n-1 DO IF w(i) NE 0 THEN wp(i,i) = 1./w(i)

RETURN, v # wp # (TRANPOSE(u) # b)
    ; Calculate x from equation and return.

END
```

## See Also

### [SVBKSB](#)

The SVD function is based on the subroutine SVDCMP in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988, and is used by permission.

PV-WAVE's SVD is a translation of the ALGOL procedure SVD described in *Linear Algebra, Vol. II of The Handbook for Automatic Computation*, by Wilkinson and Reinsch, Springer-Verlag, New York, NY, 1971.

For more information on SVD, also consult *Computer Models for Mathematical Computations*, by Forsythe, Malcom, and Moler, Prentice Hall, Englewood Cliffs, NJ, 1977.

---

## SVDFIT Function

Standard Library function that uses the singular value decomposition method of least-squares curve fitting to fit a polynomial function to data.

### Usage

*result* = SVDFIT(*x*, *y*, *m*)

### Input Parameters

*x* — A vector containing the independent *x*-coordinates of the data.

*y* — A vector containing the dependent *y*-coordinates of the data. This vector should have the same number of elements as *x*.

*m* — The number of coefficients in the fitted function. For polynomials, *m* is one more than the degree of the polynomial.

### Returned Value

*result* — A vector containing the coefficients of the polynomial equation which best approximates the data. It has a length of *m*.

## Keywords

**Chisq** — The sum of the squared errors, multiplied by the weights, if *Weight* is specified.

**Covar** — The covariance matrix of the *m* coefficients.

**Funct** — The name of a user-supplied basis function with *m* coefficients (see *User-Supplied Basis Function*).

**Singular** — The number of singular values (i.e., the number of values that are inconsistent with the other data) encountered in evaluating the fit. Should be 0; if not, the computed polynomials probably do not accurately reflect the data.

**Variance** — The estimated variance (sigma squared) for each of the *m* coefficients.

**Weight** — A vector of weighting factors for determining the weighting of the least-squares fit. Must have the same number of elements as *x*.

**Yfit** — A vector containing the calculated *y* values of the fitted function.

## Discussion

You must define any keywords to be returned by SVDFIT before calling it. The value or structure of the variable doesn't matter, since the variable will be redefined dynamically in the function call. For example:

```
yf = 1
    ; Define the output variable yf.
c = SVDFIT(x, y, m, Yfit=yf)
    ; Do the SVD fit and return the Yfit vector in the variable yf.
```

### **User-Supplied Basis Function**

If *Funct* is not supplied, a polynomial function is used as the basis function. This function is of the form:

$$result(i, j) = x(i)^j$$

The function is called with two parameters in the following fashion:

$$result = FUNCT(x, m)$$

where *x* and *m* are as defined above.

The file containing *Funct* should reside in the current working directory or in the search path defined by the system variable !Path.

If *Func* does not have the file extension `.pro`, it should be compiled before it is called in SVDFIT.

See the Standard Library function `cosines.pro`, which defines a basis function of the form:

$$result(i, j) = COS(j * x(i))$$

Weighting is useful when you want to correct for potential errors in the data you are fitting to a curve. The weighting factor, *Weight*, adjusts the parameters of the curve so that the error at each point of the curve is minimized. For more information, see the section [Weighting Factor on page 180](#) under the CURVEFIT function in Volume 1 of this reference.

## See Also

[COSINES](#), [FUNCT](#), [REGRESS](#)

For more examples of basis functions, see *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988.

---

## SYSTIME Function

Returns the current system time as either a string or as the number of seconds elapsed since January 1, 1970.

### Usage

*result* = SYSTIME(*param*)

### Input Parameters

*param* — If present and nonzero, causes the number of seconds elapsed since January 1, 1970 to be returned as a double-precision floating-point value. Otherwise, a scalar string containing the current date/time in standard system format is returned.

### Returned Value

*result* — Either a value representing the number of seconds since January 1, 1970 or a string containing the time in the standard system time format.

## Keywords

None.

## Example

```
t1 = SYSTIME(1)
    ; Calculate number of seconds elapsed since January 1, 1970.
t2 = SYSTIME(0)
    ; Calculate the current date/time in standard system format.
PRINT, t1, t2
      6.9248465e+08 Wed Dec 11 13:48:33 1991
```

## See Also

For information on other ways that PV-WAVE can handle dates and times, see the *PV-WAVE User's Guide*.

---

## T3D Procedure

Standard Library procedure that accumulates one or more sequences of translation, scaling, rotation, perspective, or oblique transformations and stores the result in the system variable !P.T.

### Usage

T3D

### Parameters

None.

### Keywords

**Reset** — A scalar which, if nonzero, resets !P.T (the transformation matrix) back to the default identity matrix.

**Translate** — A three-element vector containing the specified translations in the  $x$ ,  $y$ , and  $z$  directions.

**Scale** — A three-element vector containing the specified scaling factors in the  $x$ ,  $y$ , and  $z$  directions.

**Rotate** — A three-element vector, in units of degrees, containing the specified rotations about the  $x$ -,  $y$ -, and  $z$ -axes. Rotations are performed in the order of X, Y, and then Z.

**Perspective** — A scalar ( $p$ ) indicating the  $z$  distance to the center of the projection. Objects are projected onto the XY plane at  $Z=0$ , and the “eye” is located at point  $(0, 0, p)$ .

**Oblique** — A two-element vector containing the oblique projection parameters. Points are projected onto the XY plane at  $Z=0$  as follows:

$$X' = X + Z(d * \cos(a))$$

$$Y' = Y + Z(d * \sin(a))$$

where  $Oblique(0) = d$  and  $Oblique(1) = a$

**XYexch** — If nonzero, exchanges the  $x$ - and  $y$ -axes.

**XZexch** — If nonzero, exchanges the  $x$ - and  $z$ -axes.

**YZexch** — If nonzero, exchanges the  $y$ - and  $z$ -axes.

## Discussion

All parameters are entered in the form of keywords. The transformation specified by each keyword is performed in the order of its description above. For example, if both *Translate* and *Scale* are specified, the translation is done first, even if *Scale* is the first keyword in the procedure call.

The 4-by-4 transformation matrix, !P.T, is updated by this procedure, but the system variable !P.T3d is not set. This means that for the transformations to take effect, you must set !P.T3d equal to 1, or use the *T3d* keyword, in any plotting procedure that will make use of this transformation. Since all the graphic routines use the !P.T matrix for output, the T3D procedure can be used to effect graphic output.

---

**CAUTION** It is possible to create a transformation matrix with T3D that will not work correctly with the SURFACE or SHADE\_SURF procedures. The only T3D transformations allowed with these procedures are those that end up with the z-axis placed vertically on the display screen.

---

This procedure follows the example given in *Fundamentals of Interactive Computer Graphics*, by Foley and Van Dam, Addison Wesley Publishing Company, Reading, MA, 1982.

---

**NOTE** The matrix notation used in the procedure is reversed from the normal PV-WAVE sense in order to conform to this reference. Moreover, a right-handed system is used, meaning that positive rotations are counterclockwise when looking from a positive axis to the origin.

---

### Example 1

```
T3D, /Reset, Rotate=[30,0,0], Perspective=-1
      ; Reset transformation, rotate 30 degrees about the x-axis, and then
      ; do a perspective transformation with the center of the projection at
      ; (0, 0, -1).
```

### Example 2

Transformations may also be cascaded. For example:

```
T3D, /Reset, Translate=[-.5,-.5,0], Rotate=[0,0,45]
      ; Reset, translate the center (.5, .5, 0) to the origin, and rotate
      ; 45 degrees counterclockwise about the z-axis.
```

```
T3D, Translate=[.5,.5,0]
; Move the origin back to the center of the viewport.
```

### Example 3

```
b = FINDGEN(37)*10
y = SIN(b*!Dtor)/EXP(b/200)*4
sz = SIZE(y)
j = REFORM(y, sz(1), 1)
; Create the data, and reform it into a 2D array so that it can be used
; with the SURFACE procedure.

SURFACE, j, /Nodata, /Save, YMargin=[0, 0], $
ZStyle=1, ZRange=[0, 1]
; Draw a 3D axis and set up a 3D screen.

T3D, /YZexch
; Exchange the y- and z-axes so that plots are stacked one in front of
; the other along the usual y-axis.

TEK_COLOR
PLOT, b, y, /T3d, ZValue=1.0, YMargin=[0, 0], /Noerase
; Create a line plot stacked on the 3D axis.

POLYFILL, b, y, /T3d, Color=6, Z=1.0
; Fill under the curve.

PLOT, b, y, /T3d, ZValue=0.5, YMargin=[0, 0], /Noerase
POLYFILL, b, y, /T3d, Color=8, Z=0.5
; Create and fill a second curve.

PLOT, b, y, /T3d, ZValue=0.0, YMargin=[0, 0], /Noerase
POLYFILL, b, y, /T3d, Color=7, Z=0.0
; Create and fill a third curve.

T3D, /Reset
```

### See Also

[SURFR](#)

System Variables: [!P.T](#), [!P.T3d](#)

For more information and examples, see .

---

## ***TAG\_NAMES* Function**

Returns a string array containing the names of the tags in a structure expression.

### **Usage**

*result* = TAG\_NAMES(*expr*)

### **Input Parameters**

*expr* — The expression for which the tag names will be returned. Must be of structure type.

### **Returned Value**

*result* — The string array containing the names of the tags. If *expr* is a structure containing nested structures, only the names of tags in the outermost structure are returned (as TAG\_NAMES does not search for tags recursively).

### **Keywords**

None.

### **Example**

This example uses TAG\_NAMES to display the tag names of a structure and one of its fields, which is also a structure.

```
a = {struc1, t1: 0.0D, t2: {struct2, $
    t2_t1: INTARR(3), t2_t2: 0.0, t2_t3: 0L}, $
    t3: FLTARR(12), t4: 0L}
    ; Create a structure containing four fields, the second of which is also a structure.

PRINT, TAG_NAMES(a)
    T1 T2 T3 T4
    ; Display tag names of a.

PRINT, TAG_NAMES(a.t2)
    T2_T1 T2_T2 T2_T3
    ; Display tag names of the structure in the second field of a.
```

### **See Also**

[DELSTRUCT](#), [N\\_TAGS](#), [STRUCTREF](#)

---

## TAN Function

Returns the tangent of the input variable.

### Usage

```
result = TAN(x)
```

### Input Parameters

*x* — The angle, in radians, that is evaluated. Cannot be a complex data type.

### Returned Value

*result* — The tangent of *x*.

### Keywords

None.

### Discussion

If *x* is of double-precision floating-point, TAN yields a result of the same data type. All other data types, except complex, yield a single-precision floating-point result.

If *x* is an array, the result of TAN has the same dimensions, with each element containing the tangent of the corresponding element of *x*.

### Example

```
x = [-60, -30, 0, 30, 60]
PRINT, TAN(x * !Dtor)
      -1.73205 -0.577350 0.00000 0.577350 1.73205
```

### See Also

[ATAN](#), [COS](#), [SIN](#), [TANH](#)

For a list of other transcendental functions, see *Transcendental Mathematical Functions* in Volume 1 of this reference.

---

## TANH Function

Returns the hyperbolic tangent of the input variable.

### Usage

*result* = TANH(*x*)

### Input Parameters

*x* — The angle, in radians, that is evaluated.

### Returned Value

*result* — The hyperbolic tangent of *x*.

### Keywords

None.

### Discussion

TANH is defined by:

$$\tanh(x) \equiv \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

If *x* is of double-precision floating-point or of complex data type, TANH yields a result of the same type. All other data types yield a single-precision floating-point result.

If *x* is an array, the result of TANH has the same dimensions, with each element containing the hyperbolic tangent of the corresponding element of *x*.

### Example

```
x = [0.3, 0.5, 0.7, 0.9]
```

```
PRINT, TANH(x)
```

```
0.291313  0.462117  0.604368  0.716298
```

### See Also

[ATAN](#), [COSH](#), [SINH](#), [TAN](#)

For a list of other transcendental functions, see *Transcendental Mathematical Functions* in Volume 1 of this reference.

---

## TAPRD Procedure (OpenVMS)

Reads the next record on the selected tape unit into the specified array.

### Usage

TAPRD, *array*, *unit* [, *byte\_reverse*]

### Input Parameters

***array*** — A named variable into which the data should be read. The length of *array* and the records on the tape may range from 14 to 65,235 bytes.

- If *array* is larger than the tape record, the last portion of *array* is not changed.
- If *array* is shorter than the tape record, a data overrun error occurs.

***unit*** — A number between 0 and 9 specifying the magnetic tape unit to rewind. (Do not confuse this parameter with file logical unit numbers.)

***byte\_reverse*** — (optional) If present, causes the even and odd numbered bytes to be swapped after reading, regardless of the type of data or variables. This facilitates reading tapes written on IBM machines.

### Keywords

None.

### Discussion

No data or format conversion, with the exception of optional byte reversal, is performed by TAPRD. The input array must be defined with the desired type and dimensions.

If the read is successful, the system variable !Err is set to the number of bytes read.

### See Also

[SKIPF](#), [TAPWRT](#)

For more information, see the section *Accessing Magnetic Tape* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

---

## TAPWRT Procedure (OpenVMS)

Writes data from the input array to the selected tape unit.

### Usage

TAPWRT, *array*, *unit* [, *byte\_reverse*]

### Input Parameters

***array*** — The variable from which the data should be output. May be an expression. The length of *array* and the records on the tape may range from 14 to 65,235 bytes.

***unit*** — A number between 0 and 9 specifying the magnetic tape unit to rewind. (Do not confuse this parameter with file logical unit numbers.)

***byte\_reverse*** — (optional) If present, causes the even and odd numbered bytes to be swapped on output, regardless of the type of data or variables. This facilitates writing tapes compatible with IBM machines.

### Keywords

None.

### Discussion

One physical record containing the same number of bytes as *array* is written each time TAPWRT is called.

### See Also

[SKIPF](#), [TAPRD](#)

For more information, see the section *Accessing Magnetic Tape* in Chapter 8 of the *PV-WAVE Programmer's Guide*.

---

## **TEK\_COLOR Procedure**

Standard Library procedure that loads a color table, which contains 32 distinct colors and is similar to the default Tektronix 4115 color table, into the display.

### **Usage**

TEK\_COLOR

### **Parameters**

None.

### **Keywords**

None.

### **Discussion**

TEK\_COLOR loads the first 32 elements of the color table with the Tektronix 4115 default color map. This creates a useful color table if you desire distinctive colors.

### **Example 1**

```
b = FINDGEN(37)
x = b * 10
y = SIN(x * !Dtor)
    ; Create an array containing the values for a sine function from 0 to
    ; 360 degrees.
PLOT, x, y, XRange=[0,360], XStyle=1, YStyle=1
    ; Plot data and set the range to be exactly 0 to 360.
COLOR_PALETTE
    ; Put up a window containing a display of the current color table and
    ; its associated color indices.
TEK_COLOR
    ; Load a predefined color table that contains 32 distinct colors.
POLYFILL, x, y, Color=6
POLYFILL, x, y/2, Color=3
POLYFILL, x, y/6, Color=4
    ; Fill in areas under the curve with different colors.
z = COS(x * !Dtor)
    ; Create an array containing the values for a COS function from 0 to 360 degrees.
```

```
OPLOT, x, z/8, Linestyle=2, Color=5
; Plot the cosine data on top of the sine data.
```

## Example 2

This example creates a contour plot of Pike's Peak, with the area in between the contour lines filled with a solid color.

```
OPENR, 1, !Data_dir + 'pikeselev.dat'
pikes = FLTARR(60, 40)
READF, 1, pikes
; Read in the data file.

CLOSE, 1

c_pikes = FLTARR(62, 42)
c_pikes(1, 1) = pikes
; Close any open contours.

TEK_COLOR
; Load a color table.

CONTOUR2, c_pikes, $
    Levels=[5,6,7,8,9,10,11,12,13,14,15]*1000,$
    XStyle=1, YStyle=1, /Fill
; Produce a filled contour plot.

pikes = REBIN(pikes, 600, 400)
; Enlarge the original dataset for better display.

TV, BYTSCl(pikes, Top=10)
; Display the plot as an image similar to the filled contour plot. (Displaying it
; as an image allows access to image processing and analysis
; routines, such as DEFROI, HISTOGRAM, and PROFILES.)
```

## See Also

[C\\_EDIT](#), [COLOR\\_EDIT](#), [COLOR\\_PALETTE](#), [LOADCT](#),  
[PALETTE](#), [TVLCT](#)

---

## **TENSOR Functions**

Compute the generalized tensor product of two arrays.

### **Usage**

$c = \text{Tensor\_ADD}( a, b )$

$c = \text{Tensor\_DIV}( a, b )$

$c = \text{Tensor\_EQ}( a, b )$

$c = \text{Tensor\_EXP}( a, b )$

$c = \text{Tensor\_GE}( a, b )$

$c = \text{Tensor\_GT}( a, b )$

$c = \text{Tensor\_LE}( a, b )$

$c = \text{Tensor\_LT}( a, b )$

$c = \text{Tensor\_MAX}( a, b )$

$c = \text{Tensor\_MIN}( a, b )$

$c = \text{Tensor\_MOD}( a, b )$

$c = \text{Tensor\_MUL}( a, b )$

$c = \text{Tensor\_NE}( a, b )$

$c = \text{Tensor\_SUB}( a, b )$

### **Input Parameters**

$a$  — An array.

$b$  — An array.

### **Keywords**

None.

### **Returned Value**

$c$  — The generalized tensor product of  $a$  and  $b$ . For logical operators (EQ, NE, etc.), a byte type result is returned. Otherwise, the result type depends on the type(s) of the input parameters.

If  $a$  is an  $m$  dimensional array of dimension lengths  $a_1, \dots, a_m$  and if  $b$  is an  $n$  dimensional array of dimension lengths  $b_1, \dots, b_n$  then  $c$  is a  $m+n$  dimensional array with dimension lengths  $a_1, \dots, a_m, b_1, \dots, b_n$ .

Each element of  $c$  is computed as:

$$c(i_1, \dots, i_m, j_1, \dots, j_n) = a(i_1, \dots, i_m) \% b(j_1, \dots, j_n)$$

where  $\%$  symbolizes the operator associated with the selected function:

Function	PV-WAVE Operator
TENSOR_ADD	+
TENSOR_DIV	/
TENSOR_EQ	EQ
TENSOR_EXP	^
TENSOR_GE	GE
TENSOR_GT	GT
TENSOR_LE	LE
TENSOR_LT	LT
TENSOR_MAX	>
TENSOR_MIN	<
TENSOR_MOD	MOD
TENSOR_MUL	*
TENSOR_NE	NE
TENSOR_SUB	-

## Discussion

The TENSOR functions are useful when it is necessary to apply a binary operator to all combinations of elements of one array with elements of a second array.

---

**NOTE** The combined dimensions of the two input parameters cannot exceed eight (8). If it does, an error is printed. If input parameters are not of the same type, one will be converted according to the rules used by PV-WAVE binary operators.

---

## Example

Refer to the Standard Library routine WHEREIN, where TENSOR\_EQ is used to perform intersections and other related set operations.

(UNIX) <vni\_dir>/wave/lib/std/wherein.pro

(Windows) <VNI\_DIR>\wave\lib\std\wherein.pro

(OpenVMS) <VNI\_DIR>:[WAVE.LIB.STD]WHEREIN.PRO

## See Also

[WHEREIN](#)

---

## **THREED Procedure**

Standard Library procedure that plots a two-dimensional array as a pseudo three-dimensional plot on the currently selected graphics device.

### **Usage**

THREED, *array* [, *space*]

### **Input Parameters**

*array* — The two-dimensional array to plot.

*space* — The spacing between lines of the plot:

- If *space* is omitted, spacing will be set to  $(\text{MAX}(A) - \text{MIN}(A)) / \text{ROWS}$ .
- If *space* is negative, no hidden lines will be removed.

### **Keywords**

*Title* — A string containing the main plot title.

*XTitle* — A string containing the title of the *x*-axis.

*YTitle* — A string containing the title of the *y*-axis.

### **Discussion**

The orientation of the data plotted by THREED is fixed.

### **Example**

```
OPENR, 1, !Data_dir + 'pikeselev.dat'  
data = FLTARR(60, 40)  
READF, 1, data  
CLOSE, 1  
THREED, data  
SURFACE, data
```

## See Also

[SURFACE](#), [T3D](#)

---

## TODAY Function

Returns a date/time variable containing the current system date and time.

### Usage

```
result = TODAY()
```

### Parameters

None.

### Returned Value

*result* — A date/time variable containing the current system date and time.

### Keywords

None.

### Example

```
dttoday = TODAY()
PRINT, dttoday
      {1992 4 29 14 47 55.0000 87521.617 0}
DT_PRINT, dttoday
      04/29/1992 14:47:55
```

## See Also

[DTGEN](#)

For more information, see Chapter 8, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

---

## TOTAL Function

Sums the elements of an input array.

### Usage

*result* = TOTAL(*array*)

### Input Parameters

*array* — The array that is totalled. Can be of any data type except string.

### Returned Value

*result* — A scalar value equal to the sum of all the elements of *array*. If the *Dimension* keyword is used, then *result* will have the structure of the input array, but with the specified dimensions collapsed.

### Keywords

*Dimension* — A scalar or array of integers ( $n \geq 0$ ) that specifies the dimension(s) over which to operate.

### Discussion

If *array* is of double-precision floating-point or complex data type, the result is of the same type. If *array* is any other data type, TOTAL returns single-precision floating-point.

The *Dimension* keyword lets you sum the elements across one or more dimensions of the input array. The following examples best illustrate the use of *Dimension*.

---

**NOTE** If the input is of type byte, integer, or long, TOTAL performs its summation in long arithmetic. This is faster than performing the summation in floating-point arithmetic. To sum an array of longs with floating-point arithmetic, you must first convert the long values to double precision.

---

## Example 1

In this example, TOTAL is used to compute the sums of all elements in various rows and columns of a 3-by-2 integer array.

```
a = INDGEN(3, 2)
    ; Create a 3-by-2 integer array. Each element has a value equal to its ; one-dimensional
    ; subscript.

PRINT, a
    0     1     2
    3     4     5

PRINT, TOTAL(a(*, 0))
    3.00000
    ; Display the sum of the elements in the first row.

PRINT, TOTAL(a(1, *))
    5.00000
    ; Display the sum of the elements in the second column.

PRINT, TOTAL(a)
    15.0000
    ; Display the sum of all elements in the array.

PRINT, TOTAL(a, Dim=0)
    3.0000
    12.0000
    ; Display the sum of elements across dimension 0.

PRINT, TOTAL(a, Dim=1)
    3.00000     5.00000     7.00000
    ; Display the sum of elements across dimension 1.

PRINT, TOTAL(a, Dim=[0,1])
    4.00000
    ; Display the sum of elements on the diagonal.
```

## Example 2

```
a = INDGEN( 4, 4 )    &    PM, a
    0     4     8     12
    1     5     9     13
    2     6    10     14
    3     7    11     15

PM, TOTAL( a, d=0 )
    6.00000     22.0000     38.0000     54.0000
```

```

PM, TOTAL( a, d=1 )
    24.0000
    28.0000
    32.0000
    36.0000
PM, TOTAL( a, d=[0,1] ) ; trace
    30.0000

```

```

a = INDGEN( 2, 2, 4 )    &    PM, a
    0      2
    1      3

    4      6
    5      7

    8     10
    9     11

    12     14
    13     15

```

```

PM, TOTAL( a, d=2 )
    24.0000    32.0000
    28.0000    36.0000

```

```

PM, TOTAL( a, d=[0,1] )
    3.00000

    11.0000

    19.0000

    27.0000

```

```

a = INTARR( 5, 10, 5, 10, 5 )
INFO, TOTAL( a, d=1 )

```

```
<Expression>    FLOAT    = Array(5, 1, 5, 10, 5)
INFO, TOTAL( a, d=[0,2] )
<Expression>    FLOAT    = Array(1, 10, 1, 10, 5)
```

## See Also

[AVG](#), [MAX](#), [MEDIAN](#), [MIN](#), [SIZE](#)

---

## TQLI Procedure

Uses the QL algorithm with implicit shifts to determine the eigenvalues and eigenvectors of a real, symmetric, tridiagonal matrix.

### Usage

TQLI,  $d$ ,  $e$ ,  $z$

### Input Parameters

$d$  — An  $n$ -element vector. On input, it contains the diagonal elements of the matrix.

$e$  — An  $n$ -element vector. On input, it contains the off-diagonal elements of the matrix.  $e_0$  is arbitrary.

### Output Parameters

$d$  — An  $n$ -element vector. On output, it contains the eigenvalues.

$e$  — An  $n$ -element vector. On output, it is destroyed.

$z$  — A matrix containing the  $n$  eigenvectors. The eigenvectors are stored by rows; for example,  $z(*, 0)$  contains the first eigenvector.

### Keywords

None.

### Discussion

The TRED2 procedure may be used to reduce a real symmetric matrix to the tridiagonal form that is suitable for input to TQLI.

If the eigenvectors of a tridiagonal matrix are desired,  $z$  should be input as an identity matrix. If the eigenvectors of a matrix that has been reduced by TRED2 are desired,  $z$  should be input as the matrix  $Q$  output by TRED2.

TQLI is based on the routine of the same name found in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988. It is used by permission.

However, because the order of subscripts in PV-WAVE are reversed in comparison to those in *Numerical Recipes*, the  $z$  matrix is transposed from the result described in that book.

## Example

To determine the eigenvalues and eigenvectors of a real, symmetric matrix  $A$ :

```
asave = A
      ; Save original matrix, as TRED2 destroys its input.
TRED2, A, d, e
      ; Reduce matrix A to tridiagonal form.
TQLI, d, e, A
      ; Obtain n eigenvalues in d, and n eigenvectors in A(*, i).
```

To verify the operation of these routines, use the definition of eigenvalues and eigenvectors. Matrix  $A$  is said to have an eigenvector  $x$  and corresponding eigenvalue  $\lambda$  if:

$$A \cdot x = \lambda x$$

This is demonstrated by the following code fragment:

```
FOR i=0, N_ELEMENTS(d)-1 DO BEGIN
      ; For each eigenvector/value:
PRINT, 'Eigenvalue', i, '=', d(i)
      ; Print the eigenvalue.
PRINT, 'Eigenvector = ', Aa(*,i)
      ; Print the computed eigenvector.
PRINT, 'A x / lambda = ', asave # A(*,i)/d(i)
      ; Print the eigenvector again. This row vector should be equal to the
      ; eigenvector printed above.
ENDFOR
```

## See Also

[TRED2](#)

---

## **TRANSPOSE Function**

Transposes the input array.

### **Usage**

*result* = TRANSPOSE(*array*)

### **Input Parameters**

*array* — The array to be transposed. In previous versions of PV-WAVE, the input array had to be one or two dimensions. The input array can have any number of dimensions.

### **Returned Value**

*result* — The transposed array.

### **Keywords**

*Dimension* — A vector of two integers ( $n \geq 0$ ) designating the dimensions to transpose. Default: [0,1]

### **Discussion**

TRANSPOSE provides a convenient way to convert a row vector into a column vector, or the reverse. For example, it can be used to change a two-dimensional array into a one-dimensional vector, or to change an  $m$ -by- $n$  array into an  $n$ -by- $m$  array.

### **Example 1**

For example, executing the statements:

```
a = INDGEN(10)
    ; Create a 10-element row vector, a.
b = TRANSPOSE(a)
```

```

; Create a 10-element column vector, b using the TRANSPOSE function
INFO, a, b
a      INT      = Array(10)
b      INT      = Array(1, 10)

```

## Example 2

TRANSPOSE can also be used to shift the elements of a square array around the diagonal. For example, suppose you have the following array:

$$\begin{bmatrix} 1 & 2 & 2 \\ 3 & 1 & 2 \\ 3 & 3 & 1 \end{bmatrix}$$

Applying TRANSPOSE to this array will yield the following result:

$$\begin{bmatrix} 1 & 3 & 3 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \end{bmatrix}$$

## Example 3

Here is what an aerial image looks like before and after applying TRANSPOSE.



**Figure 2-66** TRANSPOSE has been used with this 512-by-512 aerial image to flip it diagonally (rotate it and create a mirror image).

## Example 4

```

a = INDGEN( 2, 4, 3 )      &      PM, a
0      2      4      6
1      3      5      7

```

```
8      10      12      14
9      11      13      15
```

```
16     18     20     22
17     19     21     23
```

```
PM, TRANSPOSE( a, d=[0,2] )
```

```
0      2      4      6
8      10     12     14
16     18     20     22
```

```
1      3      5      7
9      11     13     15
17     19     21     23
```

## See Also

[INVERT](#), [ROT](#), [ROTATE](#)

---

## TRED2 Procedure

Reduces a real, symmetric matrix to tridiagonal form, using Householder's method.

### Usage

```
TRED2, a [, d [, e]]
```

### Input Parameters

*a* — An *n*-by-*n* real, symmetric matrix.

## Output Parameters

$a$  — This input parameter is replaced, on output, by the orthogonal matrix  $Q$ , effecting the transformation. The TQLI procedure uses this result to find the eigenvectors of the matrix  $A$ .

$d$  — (optional) An  $n$ -element vector containing the diagonal elements of the tridiagonal matrix.

$e$  — (optional) An  $n$ -element vector containing the off-diagonal elements of the tridiagonal matrix.

## Keywords

None.

## See Also

### [TQLI](#)

TRED2 is based on a routine of the same name in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988. It is used by permission.

---

## **TRIDAG Procedure**

Solves tridiagonal systems of linear equations.

## Usage

TRIDAG,  $a, b, c, r, u$

## Input Parameters

$a$  — An  $n$ -element vector containing  $n - 1$  subdiagonal elements. Element  $a_0$  is ignored.

$b$  — An  $n$ -element vector containing  $n$  diagonal elements.

$c$  — An  $n$ -element vector containing  $n - 1$  superdiagonal elements. Element  $c_{n-1}$  is ignored.

$r$  — An  $n$ -element vector containing the right-hand side of the equation  $\mathbf{A}^T \mathbf{u} = \mathbf{r}$ .

## Output Parameters

$u$  — An  $n$ -element floating-point vector containing the solution for tridiagonal systems of linear equations.

## Keywords

None.

## Discussion

The input vectors  $a$ ,  $b$ ,  $c$ , and  $r$  are not modified by TRIDAG. They contain, respectively, the subdiagonal, diagonal, and superdiagonal elements of  $\mathbf{A}$ , and the right-hand side of the equation:

$$\mathbf{A}^T \mathbf{u} = \mathbf{r}$$

The solution is stored in the variable  $u$ .

---

**NOTE** Because PV-WAVE subscripts are in column-row order, the above equation is written as  $\mathbf{A}^T \mathbf{u} = \mathbf{r}$ , rather than as  $\mathbf{A} \mathbf{u} = \mathbf{r}$ .

---

## See Also

[LUBKSB](#), [LUDCMP](#), [MPROVE](#), [SVBKS](#)

TRIDAG is based on a routine of the same name in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988, and is used by permission.

---

## **TRNLOG Function (OpenVMS)**

Searches the OpenVMS name tables for a specified logical name and returns the equivalence string(s) in a variable.

### **Usage**

*result* = TRNLOG(*logname*, *value*)

### **Input Parameters**

*logname* — A scalar string containing the name of the logical to be translated.

### **Output Parameters**

*value* — A variable into which the equivalence string is placed. If *logname* has more than one equivalence string, the first one is used unless the *Full\_Translation* keyword is also used.

### **Returned Value**

*result* — The OpenVMS status code associated with the translation as a longword value:

- An odd value (the least significant bit is set) indicates success.
- An even value indicates failure.

### **Keywords**

*Acmode* — An integer specifying the access mode to be used in the translation. Valid access mode values are as follows:

- 0 Kernel
- 1 Executive
- 2 Supervisor
- 3 User

If you use the *Acmode* keyword, all names at access modes less privileged than the specified mode are ignored.

If you don't use *Acmode*, the translation proceeds without regard to access mode. However, the search proceeds from the outermost (User) to the innermost (Kernel) mode. Thus, if two logical names with the same name but different access modes exist in the same table, the name with the outermost access mode is used.

**Full\_Translation** — If present and nonzero, turns the *value* parameter into a string array containing all of the equivalence strings.

If *Full\_Translation* is not present, *value* only receives the first equivalence string as a scalar value, when translating a multi-valued logical name.

For example, under recent versions of OpenVMS, the SYS\$SYSROOT logical can have multiple values. To see these values from within PV-WAVE:

```
ret = TRNLOG('SYS$SYSROOT', trans, /Full, /Issue_Error)
      ; Translate the logical.

PRINT, trans
      ; View the equivalence strings.
```

**Issue\_Error** — If present and nonzero, causes TRNLOG to issue an error message if the translation fails.

**Result\_Acmode** — If present, specifies a named variable into which the access mode value of the translated logical will be placed. (The access mode values are summarized in the *Acmode* keyword description.)

**Result\_Table** — If present, specifies a named variable into which the name of the logical table containing the translated logical will be placed, as a scalar string.

**Table** — A scalar string giving the name of the logical table in which the search for *logname* will occur. If *Table* is not specified, the standard OpenVMS logical tables are searched until a match is found, starting with LNM\$PROCESS\_TABLE and ending with LNM\$SYSTEM\_TABLE.

## See Also

[DELETE\\_SYMBOL](#), [DELLOG](#), [GET\\_SYMBOL](#), [SETLOG](#),  
[SET\\_SYMBOL](#)

---

## TV Procedure

Displays images without scaling the intensity.

### Usage

TV, *image* [, *x*, *y* [, *channel* ]]

TV, *image* [, *position*]

### Input Parameters

*image* — A vector or two-dimensional matrix to be displayed as an image.

If *image* is not already of byte type, it is converted prior to use, although the conversion may distort the data contained in the image.

---

**NOTE** To insure data integrity, use the TVSCL procedure instead of TV. TVSCL does a byte scaling before displaying the image.

---

*x*, *y* — (optional) The lower-left *x*- and *y*-coordinates of the displayed image.

*channel* — (optional) The memory channel to be written. If not specified, it is assumed to be zero. This parameter is ignored on display systems that have only one memory channel.

*position* — (optional) A number specifying the position of the image. Positions run from the left of the window to the right, and from the top of the window to the bottom (see the *Discussion* section for details).

### Keywords

*Centimeters* — If present, specifies that all position values (the *x y* parameters and the *XSize* and *YSize* keywords) are in centimeters from the origin. This is useful when dealing with devices that do not provide a direct relationship between image pixels and the size of the resulting image, such as PostScript printers.

If *Centimeters* is not present, position values are taken to be in device coordinates.

*Channel* — The memory channel to be written. This keyword is identical to the *channel* input parameter; only one needs to be used.

**Data** — If present, specifies that all position and size values are in data coordinates. This is useful when drawing an image over an existing plot, since the plot establishes the data scaling.

**Device** — If present, specifies that all position and size values are in device coordinates. This is the default.

**Inches** — If present, specifies that all position and size values are in inches from the origin. This is useful when dealing with devices that do not provide a direct relationship between image pixels and the size of the resulting image, such as PostScript printers.

**Normal** — If present, specifies that all position and size values are in normalized coordinates in the range 0.0 to 1.0. This is useful when you want to draw an image in a device-independent manner.

**Order** — If specified, overrides the current setting of the !Order system variable for the current image only. If nonzero, *Order* causes the image to be drawn from the top-down, instead of from the bottom-up (the default).

**True** — (UNIX/OpenVMS Only) If present and nonzero, indicates that a true-color (24-bit) image is to be displayed and specifies the index of the dimension over which color is interleaved:

- 1 Displays pixel-interleaved images of dimensions  $(3, m, n)$ .
- 2 Displays row-interleaved images of dimensions  $(m, 3, n)$ .
- 3 Displays image-interleaved images of dimensions  $(m, n, 3)$ . (Image interleaving is also known as band interleaving.)

---

**NOTE** To use *True*, the *image* parameter must have three dimensions, one of which is equal to 3.

---

**XSize** — The width of the resulting image. This keyword is intended for devices with scalable pixel size (such as PostScript), and is ignored by pixel-based devices that are unable to change the size of their pixels.

**YSize** — The height of the resulting image, with the same limitations as *XSize*.

**Z** — The  $z$  position. The value of  $z$  is of use only if the three-dimensional transformation is in effect via the *T3d* keyword.

## Discussion

---

**Windows USERS** Because Windows NT reserves 20 out of the available 256 colors, you might achieve better results displaying color images with the TVSCL procedure. TVSCL automatically scales the color intensities to the full range of available colors.

---

If *position* is used instead of the *x* and *y* parameters, the position of the image is calculated based on the largest grid of images of this size that will fit on the display, numbered from left to right and top to bottom. Specifically, the position is calculated as explained below.

The starting *x*-coordinate position is defined as:

$$x = x_{dim} \bullet position_{modulo N_x}$$

The starting *y*-coordinate position is defined as:

$$y = YSize - y_{dim} \bullet int(1 + (position/N_x))$$

where

*YSize* is the height of the display or window,

*x<sub>dim</sub>* and *y<sub>dim</sub>* are the dimensions of the array,

and the images across the display surface are defined as:

$$N_x = \frac{Xsize}{Ysize}$$

For example, when displaying 128-by-128 images on a 512-by-512 display, the position numbers run from 0 to 15 as follows:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

## Example 1

```
mandril = BYTARR(512, 512)
OPENR, unit, !Data_dir + 'mandril.img', /Get_lun
READU, unit, mandril
FREE_LUN, unit
    ; Read the image.
WINDOW, XSize=512, YSize=512
TV, mandril
    ; Display the image.
TV, mandril, /Order
    ; Display the image from bottom-up instead of top-down; this inverts
    ; the image.
small_img = CONGRID(mandril, 100, 100)
    ; Interpolate a smaller image of size 100-by-100 pixels.
CONTOUR, small_img
TV, small_img, 0, 0, /Data
    ; Create a contour plot of the data and overlay the image at the origin,
    ; specifying the image position in data coordinates.
FOR i=0, 24 DO TV, small_img, i
    ; Tile the entire window with images, using the position parameter
    ; rather than specifying X and Y locations.
```

## Example 2

This example uses TV in conjunction with the REBIN and CONGRID procedures to enlarge small images.

```
data = BYTSCL(DIST(60))
    ; Create the data.
TV, data
    ; Display the data.
LOADCT, 5
    ; Load a color table.
enlarge = REBIN(data, 480, 480)
    ; Enlarge the original data.
TV, enlarge
    ; Display the area enlarged with REBIN.
enlarge = REBIN(data, 480, 480, /Sample)
```

```

TV, enlarge
    ; Redisplay the area enlarged with REBIN using the nearest neighbor
    ; method for the sampling.

enlarge = CONGRID(data, 480, 480)

TV, enlarge
    ; Display the area enlarged with CONGRID.

enlarge = CONGRID(data, 480, 480, /Interp)

TV, enlarge
    ; Redisplay the area enlarged with CONGRID using the bilinear
    ; interpolation method for the sampling.

```

## See Also

[BYTSCL](#), [TVCRS](#), [TVLCT](#), [TVRD](#), [TVSCL](#), [TVSIZE](#)

System Variables: [!Order](#)

For more information, see .

---

## ***TVCRS Procedure***

Manipulates the cursor within a displayed image, allowing it to be enabled and disabled, as well as positioned.

### **Usage**

TVCRS [, *on\_off*]

TVCRS [, *x*, *y*]

### **Input Parameters**

*on\_off* — (optional) Specifies whether the cursor should be on or off. If present and nonzero, enables the cursor. If zero or not specified, disables the cursor.

*x* — (optional) The column to which the cursor will be positioned.

*y* — (optional) The row to which the cursor will be positioned.

## Keywords

**Centimeters** — If present, specifies that all position values are in centimeters from the origin.

**Data** — If present and nonzero, specifies that the cursor position is in data coordinates.

**Device** — If present and nonzero, specifies that the cursor position is in device coordinates.

**Hide** — If present and nonzero, causes a disabled cursor to always be blanked out.

By default, disabling the cursor works differently for window systems than for other devices. For window systems, the cursor is restored to the standard cursor used for non-PV-WAVE windows (and remains visible), while for other devices it is completely blanked out.

**Inches** — If present, specifies that all position values are in inches from the origin.

**Normal** — If present and nonzero, specifies that the cursor position is in normalized coordinates.

**T3d** — If present, indicates that the generalized transformation matrix in !P.T is to be used. (For a description of !P.T, see the *Save* plotting keyword in [Chapter 3, Graphics and Plotting Keywords](#).)

If not present, the user-supplied coordinates are simply scaled to screen coordinates.

**Z** — The  $z$  position. The value of  $z$  is of use only if the three-dimensional transformation is in effect via the *T3d* keyword.

## Discussion

Normally, the cursor is disabled and is not visible. Using TVCRS with one parameter allows the cursor to be enabled or disabled, while using TVCRS with two parameters enables the cursor and places it on pixel location (X, Y).

## Example

To enable the image display cursor and position it at device coordinate, use the following command:

```
TVCRS, 100, 100
```

To enable the image display cursor and position it at data coordinate, use the following command:

```
TVCRS, 0.5, 3.2, /Data
```

To disable and hide the image display cursor, use the following command:

```
TVCRS, /Hide_Cursor
```

To enable the image display cursor but not set the cursor position, use the following command:

```
TVCRS, 1
```

## See Also

[CURSOR](#), [TVRD](#)

For more information, see .

---

## TVLCT Procedure

Loads the display color translation tables from the specified variables.

### Usage

```
TVLCT,  $v_1$ ,  $v_2$ ,  $v_3$  [, start]
```

### Input Parameters

$v_1$ ,  $v_2$ ,  $v_3$  — Contain the three values to be used for the specified color system (HLS, HSV, or RGB, as detailed in the *Discussion* section below). May be either scalar or vector expressions.

*start* — (optional) An integer value specifying the starting point in the color translation table into which the color intensities are loaded.

If *start* is not specified, a value of zero is used, causing the tables to be loaded starting at the first element of the translation tables.

### Keywords

*CMY* — Indicates that the parameters specify color using the CMY (cyan, magenta, yellow) color system.

*Get* — If set to 1, returns the actual RGB values loaded into the device when either the *Hls* or *Hsv* keywords are present. (The *Get* keyword has no effect when loading RGB tables.)

For example, the statements:

```
TVLCT, H, S, V, /Hsv  
TVLCT, R, G, B, /Get
```

load a color table based on the HSV system, and store the equivalent RGB values into the H, S, and V vectors.

**Hls** — Indicates that the parameters specify color using the HLS color system.

**Hsv** — Indicates that the parameters specify color using the HSV color system.

## Discussion

Color tables may be based on the following color systems: RGB (red, green, blue; the default), CMY (cyan, magenta, yellow), HLS (hue, lightness, saturation), and HSV (hue, saturation, value).

The meaning and type for the  $v_1$ ,  $v_2$ , and  $v_3$  parameters are dependent upon the color system selected, as described below. If no color-system keywords are present, the RGB color system is used.

- **CMY** — Parameters contain the cyan, magenta, and yellow values. All parameters are interpreted as integers in the range of 0 to 255 (255 being full intensity). May be scalars or may contain up to 256 elements.
- **HLS** — Parameters contain the hue, lightness, and saturation values. All parameters are floating-point. Hue is expressed in degrees and is reduced modulo 360.  $v_2$  and  $v_3$  may range from 0 to 1.0.
- **HSV** — Parameters contain the hue, saturation, and value (similar to intensity) values. All parameters are floating-point. Hue is expressed in degrees. The saturation and value may range from 0 to 1.0.
- **RGB** — Parameters contain the red, green, and blue values. Values are interpreted as integers in the range of 0 to 255 (255 being full intensity). May be scalars or may contain up to 256 elements.

## See Also

[COLOR\\_EDIT](#), [LOADCT](#), [MODIFYCT](#), [STRETCH](#),  
[TEK\\_COLOR](#), [WgCtTool](#)

For more information, including a comparison of TVLCT and LOADCT, see the section *Experimenting with Different Color Tables* in Chapter 11 of the *PV-WAVE User's Guide*.

---

## TVRD Function

Returns the contents of the specified rectangular portion of a displayed image.

### Usage

*result* = TVRD(*x*<sub>0</sub>, *y*<sub>0</sub>, *n*<sub>*x*</sub>, *n*<sub>*y*</sub> [, *channel* ])

### Input Parameters

*x*<sub>0</sub> — The starting column of data to read.

*y*<sub>0</sub> — The starting row of data to read.

*n*<sub>*x*</sub> — The number of columns to read.

*n*<sub>*y*</sub> — The number of rows to read.

*channel* — (optional) The memory channel to be read. If not specified, it is assumed to be zero. This parameter is ignored on display systems that only have one memory channel.

### Returned Value

*result* — Byte array of dimensions *n*<sub>*x*</sub>-by-*n*<sub>*y*</sub>.

---

**UNIX and OpenVMS USERS** If the display is a 24-bit display, and both the *channel* parameter and *True* keyword are absent, the maximum RGB value in each pixel is returned.

---

### Keywords

**Channel** — The memory channel to be read. This keyword is identical to the *channel* input parameter; only one needs to be specified.

**Order** — If specified, overrides the current setting of the !Order system variable for the current image only. If nonzero, *Order* causes the image to be drawn from the top-down, instead of from the bottom-up (the default).

**True** — (UNIX/OpenVMS Only) If present and nonzero, indicates that a true-color (24-bit) image is to be read and specifies the index of the dimension over which color is interleaved:

- 1 Displays an array that is pixel-interleaved and has dimensions of (3, *n*<sub>*x*</sub>, *n*<sub>*y*</sub>).
- 2 Displays a line-interleaved array of dimensions (*n*<sub>*x*</sub>, 3, *n*<sub>*y*</sub>).
- 3 Displays an image-interleaved array of dimensions (*n*<sub>*x*</sub>, *n*<sub>*y*</sub>, 3).

## Example

For an example of TVRD, see the [REBIN](#) function.

## See Also

[TV](#), [TVSCL](#), [ZOOM](#)

System Variables: [!Order](#)

For more information, see .

---

## TVSCL Procedure

Scales the intensity values of an input image into the range of the image display, usually from 0 to 235 on Windows and 0 to 255 on X Windows systems, and outputs the data to the image display at the specified location.

## Usage

TVSCL, *image* [, *x*, *y* [, *channel* ]]

TVSCL, *image* [, *position*]

## Input Parameters

*image* — A vector or two-dimensional matrix to be displayed as an image. If *image* is not already of byte type, it is converted prior to use.

*x*, *y* — (optional) The lower-left *x*- and *y*-coordinates of the displayed image.

*channel* — (optional) The memory channel to be written. If not specified, it is assumed to be zero. This parameter is ignored on display systems that have only one memory channel.

*position* — (optional) Number specifying the position of the image. Positions run from the left of the window to the right, and from the top of the window to the bottom (see *Discussion* for details).

## Keywords

*Bottom* — Specifies the lower bound when scaling the intensity values.

**Centimeters** — If present, specifies that all position values (the *x y* parameters and the *XSize* and *YSize* keywords) are in centimeters from the origin. This is useful when dealing with devices that do not provide a direct relationship between image pixels and the size of the resulting image, such as PostScript printers.

If *Centimeters* is not present, position values are taken to be in device coordinates.

**Channel** — The memory channel to be written. This keyword is identical to the *channel* input parameter; only one needs to be used.

**Data** — If present, specifies that all position and size values are in data coordinates. This is useful when drawing an image over an existing plot, since the plot establishes the data scaling.

**Device** — If present, specifies that all position and size values are in device coordinates. This is the default.

**Inches** — If present, specifies that all position and size values are in inches from the origin. This is useful when dealing with devices that do not provide a direct relationship between image pixels and the size of the resulting image, such as PostScript printers.

**Normal** — If present, specifies that all position and size values are in normalized coordinates in the range 0.0 to 1.0. This is useful when you want to draw an image in a device-independent manner.

**Order** — If specified, overrides the current setting of the *!Order* system variable for the current image only. If nonzero, *Order* causes the image to be drawn from the top-down, instead of from the bottom-up (the default).

**Top** — Specifies the upper bound when scaling the intensity values.

**True** — (UNIX/OpenVMS Only) If present and nonzero, indicates that a true-color image is to be displaced and specifies the index of the dimension over which color is interleaved:

- 1 Displays pixel-interleaved images of dimensions  $(3, m, n)$ .
- 2 Displays row-interleaved images of dimensions  $(m, 3, n)$ .
- 3 Displays image-interleaved images of dimensions  $(m, n, 3)$ . (Image interleaving is also known as band interleaving.)

---

**NOTE** To use *True*, the *image* parameter must have three dimensions, one of which is equal to 3.

---

*XSize* — The width of the resulting image. This keyword is intended for use by devices with scalable pixel size (such as PostScript), and is ignored by pixel-based devices that are unable to change the size of their pixels.

*YSize* — The height of the resulting image, with the same limitations as *XSize*.

## Discussion

If *position* is used instead of the *x* and *y* parameters, the position of the image is calculated based on the largest grid of images of this size that will fit on the display, numbered from left to right and top to bottom. Specifically, the position is calculated as explained below.

The starting *x*-coordinate position is defined as:

$$x = x_{dim} \bullet position_{modulo N_x}$$

The starting *y*-coordinate position is defined as:

$$y = YSize - y_{dim} \bullet int(1 + (position/N_x))$$

where

*YSize* is the height of the display or window,  $x_{dim}$  and  $y_{dim}$  are the dimensions of the array,

and the images across the display surface are defined as:

$$N_x = \frac{Xsize}{Ysize}$$

For example, when displaying 128-by-128 images on a 512-by-512 display, the position numbers run from 0 to 15 as follows:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

## Example

This example uses TVSCL and TV to exhibit the difference between images whose intensity values are scaled into the full range of the image display and images that have not been scaled.

```
OPENR, unit, FILEPATH('aerial_demo.img', $
    Subdir='data'), /Get_Lun
    ; Open the file containing the image.

img = BYTARR(512, 512)
    ; Create an array large enough to hold the image.

READU, unit, img
    ; Read the image data.

FREE_LUN, unit
    ; Close the file and free the file unit number.

WINDOW, 0, XSize = 1024, YSize = 512
    ; Create a window large enough to hold two 512-by-512 images.

TV, img
    ; Display the original image in the left half of the window.

TVSCL, img, 1
    ; Use TVSCL to display the scaled image in the right half of the window.
```



**Figure 2-67** Unscaled image (left); scaled image (right).

## See Also

[BYTSCL](#), [TV](#), [TVCRS](#), [TVLCT](#), [TVRD](#), [TVSIZE](#)

System Variables: [!Order](#)

For more information, see .

---

## TVSIZE Procedure

Displays or prints images at the current or specified size and device resolution.

### Usage

TVSIZE, *image* [, *x*, *y* [, *channel* ]]

TVSIZE, *image* [, *position*]

### Input Parameters

*image* — A 2D array containing the image data.

*x*, *y* — (optional) The lower-left position of the displayed image.

*channel* — (optional) The memory channel to be written. If not specified, it is assumed to be zero. This parameter is ignored on display systems that have only one memory channel.

*position* — (optional) A number specifying the position of the image. Positions run from the left of the window to the right, and from the top of the window to the bottom.

### Keywords

**Bottom** — Specifies that the image intensity values are to be scaled (as with the TVSCL procedure). The value specified is the lower bound of the intensity values.

**Centimeters** — If present and non-zero, specifies that the *XSize* and *YSize* keyword parameters are in centimeters. This is the default.

**Channel** — The memory channel to be written. This keyword is identical to the channel input parameter; only one needs to be used.

**Data** — If present and non-zero, specifies the (*x*, *y*) position parameters are in data coordinates.

**Device** — If present and non-zero, specifies the (*x*, *y*) position parameters are in device coordinates.

**Inches** — If present and non-zero, specifies that the *XSize* and *YSize* keyword parameters are in inches. (Default: centimeters)

**Normal** — If present and non-zero, specifies the (*x*, *y*) position parameters are in normal coordinates.

**Order** — If specified, overrides the current setting of the !Order system variable for the current image only. If nonzero, *Order* causes the image to be drawn from the top-down, instead of from the bottom-up (the default).

**Resize\_Device** — If present and non-zero, resizes the output device to match the size of the image. This keyword is ignored when the current device is a screen device, such as X or WIN32. See the *Discussion* section for more information.

**Top** — Specifies that the image intensity values are to be scaled (as with the TVSCL procedure). The value specified is the upper bound of the intensity values.

**Screen\_Dpi** — Specifies the resolution of the screen in pixels per inch. Default: 100 dpi.

**XSize** — Specifies the width of the displayed (or printed) image. The default is to display it with the same width it would have if displayed on the screen.

**YSize** — Specifies the height of the displayed (or printed) image. The default is to display it with the same height it would have if displayed on the screen.

**Z** — The *z* position. The value of *z* is of use only if the 3D transformation is in effect via the *T3d* keyword.

## Discussion

TVSIZE lets you easily control the size of a displayed or printed PV-WAVE image.

It is usually the case that the resolution of an output device, such as a laser printer, does not match the resolution of your computer screen. Because of this difference in resolution, you may notice that printed images look smaller than images that appear on your screen. TVSIZE detects these differences in display resolution and automatically adjusts the size of the image, if necessary.

By default, a screen resolution of 100 dots per inch (dpi) is assumed for your system. (If you wish, you can specify your screen resolution precisely using the *Screen\_Dpi* keyword.) The screen dpi is compared to the value of the system variables !D.X\_Px\_Cm and !D.Y\_Px\_Cm. (These system variables hold the (*x*, *y*) resolution in number of pixels per centimeter for the currently active output device.) If there is a difference of greater than 20% between the screen resolution and the output device resolution, TVSIZE resizes the image accordingly. If the output device does not have scalable pixels, the CONGRID routine is used to expand the number of pixels to match the given size. If the device supports scalable pixels it simply receives the requested image size.

If you have annotated an image with text or lines or other graphics objects, you may notice that the printed output does not appear the way it appears on your screen.

Specifically, the annotations may appear to be moved away from the image. If this is the case, use the *Resize\_Device* keyword. This keyword explicitly resizes the output device to match the image size, resulting in output that closely matches the graphics as they appear on your screen.

---

**TIP** TVSIZE assumes that your screen resolution is 100 dots per inch (dpi). To improve the ability of TVSIZE to estimate the size of the image on the screen and correctly size the image for printing, do the following:

1. At a point in the application where the screen device is active save the screen resolution (in dpi):

```
scr_dpi = [!D.X_Px_Cm, !D.Y_Px_Cm] * 2.54
```

2. When displaying the image, use the command:

```
TVSIZE, image, x, y, Screen_Dpi=scr_dpi
```

Since the values of the system variables !D.X\_Px\_Cm and !D.Y\_Px\_Cm change to match the current device do not use them directly in the TVSIZE command.

---

**NOTE** The system variables !D.X\_Px\_Cm and !D.Y\_Px\_Cm may not exactly reflect the resolution of the screen. These values are reported to PV-WAVE and cannot be verified.

---

## Examples

To print an image at the same physical size displayed:

```
TVSIZE, image, 0, 0
```

To print an image so the output is 5 inches wide and 7 inches tall:

```
TVSIZE, image, 0, 0, XSize=5, YSize=7, /Inches
```

## See Also

[TV](#), [TVSCL](#), [CONGRID](#)

---

## UNIQN Function

Finds the unique  $n$ -tuples from a set of  $n$ -tuples.

### Usage

$b = \text{UNIQN}( a )$

### Input Parameters

$a$  — An  $m$ -by- $n$  array of  $m$   $n$ -tuples.

### Returned Value

$b$  — A  $p$ -by- $n$  array of the  $p$  unique  $n$ -tuples in  $a$ .

### Example

```
a = [ [0,1,1,0,1], [0,1,1,1,0] ]    &    PM, a
      0      0
      1      1
      1      1
      0      1
      1      0
PM, UNIQN( a )
      0      0
      0      1
      1      0
      1      1
```

### See Also

[SORTN](#), [UNIQUE](#)

---

## **UNIQUE Function**

Returns a vector (one-dimensional array) containing the unique elements from another vector.

### **Usage**

```
result = UNIQUE(vec)
```

### **Input Parameters**

*vec* — A vector containing duplicate values.

### **Returned Value**

*result* — A new vector containing the unique elements of the original vector.

### **Keywords**

None.

### **Discussion**

This function works on any vector (one-dimensional array) variable.

### **Example 1**

For a very simple example, suppose you have a vector defined as:

```
vec = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5]
```

The following command produces a new vector containing only the unique elements of *vec*:

```
result = UNIQUE(vec)
PRINT, result
      1  2  3  4  5
```

### **Example 2**

With `UNIQUE`, you can determine the unique values in any column of a table.

For example, a table called `phone_data` contains information on phone calls made during a three-day period. This table contains eight columns of phone information: the date, time, caller's initials, phone extension, area code of call, number of call, duration, and cost. (For more information on the structure of this table, see the *PV-WAVE User's Guide*.)

To obtain a list of the dates on which calls were made.

```
dates = UNIQUE(phone_data.DATE)
```

---

**NOTE** Tables are represented as an array of structures. In this command, `phone_data.DATE` represents the `DATE` field of the structure called `phone_data`.

---

The result is a one-dimensional variable called `dates` that contains a list of the dates on which calls were made:

```
PRINT, dates
      901002  901003  901004
```

### Example 3

In some applications, it may be possible to generate the text for menu buttons from the unique elements in a table column. For example, the following commands display a menu of dates. The menu selection is then passed into the `QUERY_TABLE` function where the total cost of calls is calculated for that date.

```
unique_date = UNIQUE(phone_data.DATE)
              ; Find the unique dates in the phone_data table.

date_pick = unique_date $
            (TVMENU(STRING(unique_date)))
            ; Display a menu of unique dates. The menu selection returns
            ; the selected date to the variable date_pick. Note that the
            ; parameter passed to TVMENU must be type string.

total_cost = QUERY_TABLE(phone_data, $
                          'DATE, SUM(COST) Where DATE = ' + $
                          'date_pick Group By DATE')
            ; Find the total cost of calls made on the selected date.
```

### See Also

[BUILD\\_TABLE](#), [GROUP\\_BY](#), [ORDER\\_BY](#), [QUERY\\_TABLE](#), [UNIQN](#)

---

## UNIX\_LISTEN Function (UNIX Only)

Standard Library function that allows PV-WAVE to be called by external routines written in C.

### Usage

*result* = UNIX\_LISTEN ( )

### Parameters

None.

### Returned Value

*result* — The number of parameters returned to UNIX\_LISTEN.

### Keywords

**Procedure** — A string that is set by the client and retrieved by UNIX\_LISTEN. Its intended use is to control program flow within the server (PV-WAVE).

**Program** — An integer used as an identifier for the external routines. If an external routine (or client) calls `call_wave`, UNIX\_LISTEN checks if the value of *Program* matches the value of the program parameter in `call_wave` sent by the client. The client's program identifier is set by the program parameter in the `call_wave` function call.

**User** — A string that is set by the client and retrieved by UNIX\_LISTEN. Its intended use is for controlling access to the server, as an authentication mechanism.

### Discussion

UNIX\_LISTEN waits until an external routine calls the function `call_wave`, and then returns the number of parameters passed to UNIX\_LISTEN.

Parameters are passed into PV-WAVE through the common block UT\_COMMON. A maximum of thirty parameters can be passed to PV-WAVE.

The common block UT\_COMMON is included in the server routine with the command @UT\_COMMON. The first of the thirty available parameters is `ut_param0`, the second is `ut_param1`, and the thirtieth parameter is `ut_param29`.

As well as being returned by `UNIX_LISTEN`, the number of parameters is also contained in the variable `ut_num_params` in the `UT_COMMON` common block.

## See Also

[CALL\\_UNIX](#), [LINKNLOAD](#), [SPAWN \(UNIX/OpenVMS\)](#), [UNIX\\_REPLY](#)

For more information and an example, see the *PV-WAVE Application Developer's Guide*.

---

## ***UNIX\_REPLY Function (UNIX Only)***

Standard Library function that allows PV-WAVE to return a value or values that it has calculated to an external routine written in C.

### Usage

```
result = UNIX_REPLY(reply)
```

### Input Parameters

*reply* — A variable of any data type, except of type structure, representing the result of an operation that PV-WAVE, as a server, has performed.

### Returned Value

*result* — A number indicating the status of the reply operation. A return value of -1 indicates an error. Errors can also be trapped by the `ON_IOERROR` routine.

### Keywords

*Return\_Params* — If present and nonzero, causes `UNIX_REPLY` to return the modified parameters to the client. The number of parameters to be sent back is the same as the number that came in with `UNIX_LISTEN`. This number is tracked internally by PV-WAVE.

## See Also

[CALL\\_UNIX](#), [LINKNLOAD](#), [ON\\_IOERROR](#), [SORTN](#), [SPAWN \(UNIX/OpenVMS\)](#), [UNIX\\_LISTEN](#)

For more information and an example, see the *PV-WAVE Application Developer's Guide*.

---

## **UNLOAD\_OPTION Procedure**

Explicitly unloads an Option Programming Interface (OPI) optional module.

### **Usage**

UNLOAD\_OPTION, *option\_name*

### **Input Parameters**

*option\_name* — A string containing the name of the option to be unloaded.

### **Keywords**

None.

### **Discussion**

The UNLOAD\_OPTION procedure explicitly unloads a previously loaded OPI option. All resources used for the OPI option are freed. OPI options can be loaded explicitly by any PV-WAVE user using the LOAD\_OPTION procedure. These optional modules can be written in C or FORTRAN, and can contain new system functions or other primitives. For detailed information on creating OPI options, see the *PV-WAVE Application Developer's Guide*.

### **Example**

```
WAVE> UNLOAD_OPTION, 'SAMPLE'
```

### **See Also**

[LOAD\\_OPTION](#), [OPTION\\_IS\\_LOADED](#), [SHOW\\_OPTIONS](#)

---

## UPVAR Procedure

Accesses a variable that is not on the current program level.

### Usage

UPVAR, *name*, *local*

### Input Parameters

*name* — A string containing the name of a variable that is on the program level specified by the *Level* keyword.

### Output Parameters

*local* — The name of the local variable that you want to bind to the variable *name*.

### Keywords

*Add* — If nonzero, creates a new variable on the \$MAIN\$ program level if the variable with the name “*name*” doesn’t already exist.

*Level* — An integer (n) specifying the program level on which to find the variable *name*. (Default: program level 0, which is \$MAIN\$)

If  $n \geq 0$ , the program level is counted from \$MAIN\$ (level 0) to the current procedure (absolute).

If  $n < 0$ , the program level is counted from the current procedure to the \$MAIN\$ level (relative).

### Discussion

---

**NOTE** Creating two or more local bindings to the same variable can result in unpredictable behavior and is not recommended or supported.

---

---

**NOTE** If you bind a local variable to a variable on the same program level, UPVAR cannot distinguish between the two, and you may receive unpredictable results.

---

You can use the INFO procedure with the *Upvar* keyword to determine whether a variable exists on a specific program level and, if it exists, determine its name. The name returned by INFO and the *Upvar* keyword can be used directly in the UPVAR procedure (see *Example 1*).

## Example 1

A variable name is obtained with the INFO command and its *Upvar* keyword. The variable name is then used in the UPVAR command.

```
INFO, var, Upvar = name
    ; Obtain the name of the variable "var", which is on the $MAIN$ program level.
UPVAR, name, local
    ; Bind the variable "var" from the $MAIN$ program level.
```

## Example 2

UPVAR is useful in VDA Tool development, such as when you need to pass variables from one program level to another without using Common Blocks. For example, the following procedure is a callback for a TM\_CONVERT method. This procedure replots a variable whenever a window has been resized. To find out which variable to plot, it first lists the variables associated with a specific instance of a VDA Tool, then uses UPVAR to pass the variables into the local procedure. (Assume that for this VDA Tool, variables were created on the \$MAIN\$ program level, which is where UPVAR gets variables by default.)

```
PRO ConvertPlotTool, tool_name
    IF !D.NAME EQ 'X' THEN BEGIN
        plot_var = TmEnumerateVars(tool_name)
        UPVAR, plot_var(0), local
        PLOT, local, Xstyle=4, Ystyle=4, /Nodata, /Noerase
    END
END
```

## See Also

[ADDVAR](#), [DELVAR](#), [INFO](#)

---

## **USERSYM Procedure**

Lets you create a custom symbol for marking plotted points.

### **Usage**

USERSYM,  $x$  [,  $y$ ]

### **Input Parameters**

$x$ ,  $y$  — Vectors containing the  $x$  and  $y$  vertices of the symbol to be created. In the case of vector-drawn symbols, these vertices are connected, in order, with vectors forming the symbol.

If only  $x$  is specified, it must be a  $(2, n)$  array of vertices, with element  $(0, i)$  containing the  $x$ -coordinate of the vertex, and element  $(1, i)$  containing the  $y$ -coordinate.

### **Keywords**

**Color** — An integer specifying the color used to draw the symbols, or used to fill the polygon. (Default: the line color)

**Fill** — A flag which, if set, fills the polygon defined by the vertices. If *Fill* is not set, lines are drawn connecting the vertices.

**Thick** — The thickness of the lines drawn in constructing the symbol. (Default: 1.0)

### **Discussion**

The  $x$  and  $y$  parameters are offsets from the data point, in units of approximately the size of a character. In other words, a value of 1 corresponds to the actual  $x$  or  $y$  size of a character, while a value of .5 is equal to one-half the character size.

Symbols may be drawn with vectors or may be filled. Symbols may be of any size and may have up to 50 vertices. To use a user-defined symbol, set the value of the *Psym* plotting keyword or the !Psym system variable to +8 or -8.

## Example

This example uses `USERSYM` to define a plotting symbol that resembles a house. The `Fill` keyword is used to fill the interior of the polygon defining the house. Six random points are then plotted using the house symbol to mark the data points.

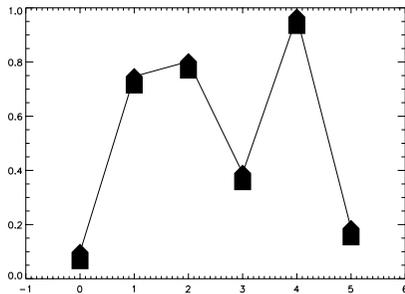
The `Psym` keyword, in the call to `PLOT`, is given a value of `-8` so that the data points are connected by lines. Also, the `Symsize` keyword is used with a value of `6` to increase the size of the plotting symbols. This example uses `PV-WAVE:IMSL` Statistics procedures `RANDOMOPT` and `RANDOM`.

```
x = [0, -0.5, -0.5, 0.5, 0.5, 0]
y = [0.5, 0, -1, -1, 0, 0.5]
      ; Define the x- and y-coordinates of the vertices.

USERSYM, x, y, /Fill
      ; Create the filled symbol.

RANDOMOPT, Set = 12542
      ; Generate six random points.

pts = RANDOM(6)
Range = [-0.1, 5.1]
      ; Plot the points, using the new user-defined symbol.
```



**Figure 2-68** Scattered data plot with user-defined markers for data points.

## See Also

System Variables: [!P.Psym](#)

Plotting Keywords: [Psym](#) and [Symsize](#)

---

## ***USGS\_NAMES Function***

Queries a database containing names, FIPS codes, and longitude/latitude values for cities and towns in the United States.

### **Usage**

```
result = USGS_NAMES( [name] )
```

### **Input Parameters**

***name*** — (optional) A string parameter containing the name of a city or town. All names that begin with this string are returned. If you omit the *name* parameter, then the entire database is returned.

### **Keywords**

***County*** — Specifies either a string containing the county name or the county FIPS code. By default, no county is specified.

***State*** — Specifies either a string containing the two-letter state abbreviation or an integer specifying the state FIPS code. By default, no state is specified.

### **Returned Value**

***result*** — An unnamed structure array containing the city or town name, state FIPS code, county FIPS code, longitude, and latitude.

### **Discussion**

The search string arguments are not case sensitive. A search for city names BOULDER and boulder produce identical results.

The result of this function is an unnamed structure array. If *state* and *county* are specified as strings, then the result is given in the form:

```
{, name:'', state:0, county:0, lon:0.0, lat:0.0}
```

where the *state* and *county* tags' fields are FIPS numbers (integers).

If *State* and *County* are specified as FIPS numbers (integers), the result is given in the form:

```
{, name:'', state:'', county:'', lon:0.0, lat:0.0}
```

where the `state` tag field is a string containing the two-letter state abbreviation and the `county` tag field is a string containing the name of the county.

The `county` tag field will only be included in the output if the *County* keyword is specified.

If no matches are found, a structure is returned with all fields set to zero or an empty string, as appropriate.

## Example 1

Find the longitude and latitude of Boulder, Colorado:

```
boulder = USGS_NAMES('boulder', State='CO')
PRINT, boulder.lon, boulder.lat
      -105.270      40.0150
```

## Example 2

Find the FIPS codes for Colorado and Larimer county:

```
codes = USGS_NAMES(State='CO', County='Larimer')
PRINT, codes(0).state, codes(0).county
      8      69
```

## Example 3

Find the two-letter state abbreviation and the county name given the FIPS codes:

```
result = USGS_NAMES(State=8, County=69)
PRINT, result(0).state, ', ', result(0).county
      CO, Larimer
```

## Example 4

Return all the towns in the database named “Lincoln”:

```
result = USGS_NAMES('Lincoln')
FOR i=0,N_ELEMENTS(result)-1 DO PRINT, result(i)
  { Lincoln  5  143  -94.4233  35.9494}
  ; This is a one-line sample of the output. The complete output
  ; is not shown for this example.
```

## Example 5

Return all the towns in Larimer county, Colorado:

```
result = USGS_NAMES(State='CO', $
                County='Larimer')

FOR i=0,N_ELEMENTS(result)-1 DO PRINT, result(i)
  { Berthoud  8  69  -105.081  40.3083}
  { Buckeye  8  69  -105.094  40.8272}
  { Drake    8  69  -105.340  40.4319}
    ; This is a three-line sample of the output. The complete output
    ; is not shown for this example.
```

## See Also

[MAP](#)

---

## VAR\_MATCH Function

Standard Library function that scans for PV-WAVE variables that match the given criteria.

### Usage

```
vars = VAR_MATCH()
```

### Input Parameters

None.

### Returned Value

*vars* — An array of variable names that match the criteria given by the keywords. If no matches are found a scalar NULL string is returned.

### Keywords

*Count* — If set, returns the number of variables that matched the criteria.

*Dimensions* — A string or string array specifying the exact dimensions of the variable(s) that you wish to have returned. If an array, returns variables matching any o\*f the dimensions. The string must have the following pattern:

'NxNx...xN' or  
['NxNx...xN', ..., 'NxNx...xN']

For example:

Dimensions = '256x256x3', or

Dimensions = ['256x256x3', '3x512', '3x640']

If you set *Dimensions* equal to N, it will return all 1D arrays with N elements. Similarly, *Dimensions* = '0' will return all scalar variables (much like specifying *NDimensions* = 0)

**Level** — Specifies the program level for which variables are to be considered.

If *Level* ≥ 0, levels are counted up from \$MAIN\$ to the current level.

If *Level* ≤ 0, levels are counted down from the current level.

If this keyword is not specified, \$MAIN\$ is assumed.

**Names** — Specifies patterns to be matched. The patterns are used to find and display variable names. If this keyword is not specified, all variables are considered. As with INFO, the asterisk (\*) and question mark (?) may be used to match any string or a single character, respectively.

**NDimensions** — A scalar or array specifying the number of dimensions to be matched.

**Type** — A scalar or array specifying the type(s) of variables to be matched, according to the following table:

Type Codes

Type Code	Data Type
0	Undefined
1	Byte
2	Integer
3	Longword integer
4	Floating point
5	Double precision floating
6	Complex single-precision floating
7	String

## Type Codes (Continued)

Type Code	Data Type
8	Structure
9	(not used)
10	List
11	Associative array
12	Complex double-precision floating

## Discussion

The filtering precedence of the VAR\_MATCH function is *Level*, *Names*, *Type*, *NDimensions*, and *Dimensions*.

## Example

The following line of code finds the names of any byte, integer, or long arrays with dimensions (3, 256).

```
var_list = VAR_MATCH(Dimensions = '3x256', Type = [1, 2, 3])
```

## See Also

[INFO](#), [SIZE](#), [UPVAR](#)

---

## VAR\_TO\_DT Function

Converts scalar or array values representing dates and times to date/time variables.

## Usage

```
result = VAR_TO_DT(yyyy, mm, dd, hh, mn, ss)
```

## Input Parameters

*yyyy* — A scalar or array containing year numbers.

*mm* — A scalar or array containing month numbers (1 – 12).

*dd* — A scalar or array containing day numbers (1 – 31).

**hh** — A scalar or array containing hour numbers (0 – 23). If zero or not specified, *hh* is 0 hours.

**mn** — A scalar or array containing minute numbers (0 – 59). If zero or not specified, *mn* is 0 minutes.

**ss** — A scalar or array containing second numbers (0.0000 – 59.9999). If zero or not specified, *ss* is 0.0 seconds.

## Returned Value

**result** — A date/time variable containing the converted values.

## Keywords

None.

## Discussion

Use this function to create date/time variables from time stamp data that does not conform to a format used by the `STR_TO_DT` function. For example, you can read the date and time data into atomic variables representing each of the date and time elements (i.e., year, month, day, etc.). Then the variables can be converted to date/time variables using `VAR_TO_DT`.

If the year, month, and day values are all zero, then the value of the `!DT_Base` system variable is used for the date portion of the resulting date/time variable.

The parameters can be arrays of any numeric values, but all parameters must have the same dimension; that is, the parameters must be either all scalars or all arrays of the same size. Also you can only omit parameters from the end of the parameter list.

## Example 1

This example illustrates how to convert a single date value into date/time variable.

```
z = VAR_TO_DT(1992, 11, 22, 12, 30)
    ; Note that seconds have been omitted. This command creates a
    ; date/time variable containing November 22, 1992 at 12:30.
PRINT, z
    { 1992 11 22 12 30 0.00000 87728.521 0}
DT_PRINT, z
    11/22/1992    12:30:00
```

## Example 2

This example illustrates how to return a date/time variable for an array containing values representing dates.

```
years = [1992,1993]
months = [3,4]
days = [17,18]
    ; Create arrays that contain date/time information for two days.
y = VAR_TO_DT(years, months, days)
    ; Convert the date/time arrays to a date/time structure variable.

DT_PRINT, y
    03/17/1992
    04/18/1993
```

### See Also

[DT\\_TO\\_VAR](#), [SEC\\_TO\\_DT](#), [JUL\\_TO\\_DT](#), [STR\\_TO\\_DT](#)

For more information, see the *PV-WAVE User's Guide*.

---

## **VECTOR\_FIELD3 Procedure**

Plots a 3D vector field from three arrays.

### Usage

VECTOR\_FIELD3, *vx*, *vy*, *vz*, *n\_points*

### Input Parameters

*vx* — A 3D array containing the *x*-component of the vector field, or an *n*-element vector containing the *x*-component of the vector field.

*vy* — A 3D array containing the *y*-component of the vector field, or an *n*-element vector containing the *y*-component of the vector field.

*vz* — A 3D array containing the *z*-component of the vector field, or an *n*-element vector containing the *z*-component of the vector field.

---

**NOTE** The arrays *vx*, *vy*, and *vz* must be the same size.

---

***n\_points*** — If *vx*, *vy*, and *vz* are all 3D arrays and *n\_points* is a (3, *n*) array, then *n\_points* is used to specify where the vectors are plotted.

If *vx*, *vy*, and *vz* have the dimensions (*i*, *j*, *k*), then:

*n\_points*(0,\*) should range between 0 and *i* – 1

*n\_points*(1,\*) should range between 0 and *j* – 1

*n\_points*(2,\*) should range between 0 and *k* – 1

If *vx*, *vy*, and *vz* are all 3D arrays and *n\_points* is a single positive value *n*, then *n* vectors are plotted with random starting locations.

If *vx*, *vy*, and *vz* are all 3D arrays and *n\_points* is zero or negative, then one vector is plotted for each element in *vx*.

If *vx*, *vy*, and *vz* are *n*-element vectors and *n\_points* is a (3, *n*) array, then the starting locations for each vector are taken from *n\_points*.

## Keywords

***Axis\_Color*** — The color to use when plotting the axis. To suppress the axis, set *Axis\_Color* to –1.

***Mark\_Color*** — The color index to use when plotting the markers.

***Mark\_Size*** — The marker size.

***Mark\_Symbol*** — A number ranging from 1 to 7 defining the marker symbol to use. The default is 3 (a period). Markers are plotted at the tail of each vector. For a list of symbols, see the description of !Psym in [Chapter 4, System Variables](#).

***Max\_Color*** — The highest color index to use when plotting.

***Min\_Color*** — The lowest color index to use when plotting. The color of each vector ranges from *Min\_Color* to *Max\_Color*.

***Max\_Length*** — A scalar value for the maximum plotted length of each vector. Each vector ranges from zero to *Max\_Length*.

***Thick*** — The line thickness (in pixels) to use when plotting vectors.

***Vec\_Color*** — A 3D array (with the same dimensions as *vx*) containing the data for the vector colors.

A vector plotted where *Vec\_Color* is at its maximum has the color *Max\_Color*, while a vector plotted at a location where *Vec\_Color* is minimum has the color *Min\_Color*.

If *Vec\_Color* is not supplied, then the color of each vector is determined by its length.

## Discussion

VECTOR\_FIELD3 plots a 3D velocity vector field from volumetric or 3D data.

## Examples

```
PRO vec_demo1
    ; This program displays a 3D vector field using x, y, z data.

winx = 800
winy = 600
    ; Specify the window size.

v_num = 1000
    ; Specify the number of vectors.

xvec = FLTARR(v_num)
yvec = FLTARR(v_num)
zvec = FLTARR(v_num)
points = FLTARR(3, v_num)
    ; Create the arrays for the vectors and their starting points.

FOR k=0, 9 DO BEGIN
    FOR j=0, 9 DO BEGIN
        FOR i=0, 9 DO BEGIN
            ind = i + (j * 10) + (k * 10 * 10)
            xvec(ind) = COS(!PI * FLOAT(i)/10.0)
            yvec(ind) = SIN(!PI * FLOAT(j)/10.0)
            zvec(ind) = SIN(!PI * FLOAT(k)/10.0)$
                + COS(!PI * FLOAT(i)/10.0)
            points(*, ind) = [i, j, k]
        ENDFOR
    ENDFOR
ENDFOR
    ; Create the data for the vectors and their starting points.

T3D, /Reset
T3D, Translate=[-0.5, -0.5, -0.5]
T3D, Scale=[0.5, 0.5, 0.5]
T3D, Rotate=[0.0, 0.0, -30.0]
T3D, Rotate=[-60.0, 0.0, 0.0]
T3D, Translate=[0.5, 0.5, 0.5]
    ; Set up the transformation matrix for the view.
```

```

WINDOW, 0, XSize=winx, YSize=winy, $
      XPos=256, YPos=128, Colors=128, $
      Title='3-D Velocity Vector Field'
LOADCT, 4
      ; Set up the viewing window and load the color table.

VECTOR_FIELD3, xvec, yvec, zvec, points, $
      Max_Length=0.5, Min_Color=32, $
      Max_Color=127, Axis_Color=100, $
      Mark_Symbol=3, Mark_Color=127, $
      Mark_Size=0.5, Thick=2
      ; Plot the vector field with the vector directions defined by xvec,
      ; yvec, and zvec, and the vector starting points defined by
      ; points.

END

```

For other examples, see the `vec_demo2` and `vol_demo1` demonstration programs in:

```

(UNIX)      <wavedir>/demo/ar1
(OpenVMS)   <wavedir>:[DEMO.ARL]
(Windows)   <wavedir>\demo\ar1

```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[VOL\\_MARKER](#)

## VEL Procedure

Standard Library procedure that draws a graph of a velocity field with arrows pointing in the direction of the field. The length of an arrow is proportional to the strength of the field at that point.

### Usage

VEL, *u*, *v*

### Input Parameters

*u* — The *x*-component of the velocity field at each point. This parameter must be a two-dimensional array.

$v$  — The y-component of the velocity field at each point. This parameter must have the same dimensions as  $u$ .

## Keywords

**Length** — The length of each arrow segment, expressed as a fraction of the longest arrow divided by *Nsteps*. *Length* is used to calculate the proportional length of each arrow segment. (Default: 0.1)

**Nsteps** — The number of segments in each arrow. (Default: 10)

**Nvecs** — The number of arrows to draw. (Default: 200)

**Xmax** — The aspect ratio (the x-axis size as a fraction of the y-axis size). (Default: 1.0)

## Discussion

VEL selects *Nvecs* random points within the boundary of the ( $u$ ,  $v$ ) arrays. At each point, the field is bilinearly interpolated from the ( $u$ ,  $v$ ) arrays and a vector of the correct proportional length is drawn.

The field is recalculated at the endpoint of this vector and a new vector is iteratively drawn, until an arrow with *Nsteps* number of segments is drawn.

An arrowhead is drawn at the end of this arrow, and the procedure moves on to another random point to initiate the loop again. The graph is plotted with the title “Velocity Field”.

---

**CAUTION** Extra care must be taken if you run the PLOT\_FIELD and VEL procedures in the same PV-WAVE session. Each procedure calls a routine named ARROWS, but the ARROWS routines are slightly different. If you get an error in the ARROWS routine when you are using VEL, recompile VEL (by typing .RUN VEL), and then try again.

---

## Examples

```
u = FLTARR (21, 21)
```

```
v = FLTARR (21, 21)
```

```
    ; Create the arrays.
```

```
.RUN
```

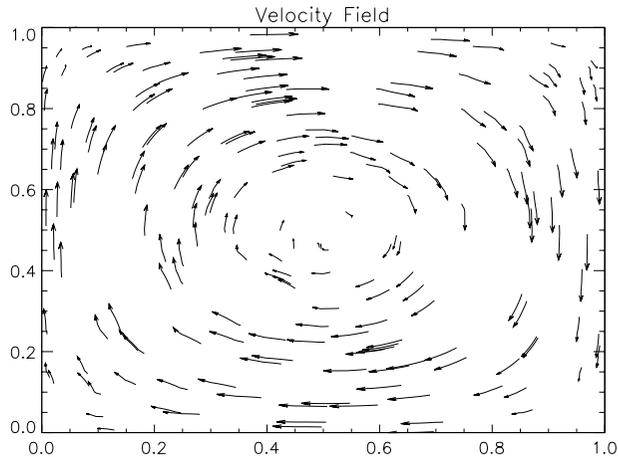
```
    ; After you type .RUN, the WAVE prompt changes to a dash (–) to
```

```
    ; indicate that you may enter a complete program unit.
```

```

FOR j = 0, 20 DO BEGIN
FOR i = 0, 20 DO BEGIN
x = 0.05 * FLOAT(i)
z = 0.05 * FLOAT(j)
u(i, j) = -SIN(!Pi*x) * COS(!Pi*z)
v(i, j) = COS(!Pi*x) * SIN(!Pi*z)
ENDFOR
ENDFOR
END
; This procedure stuffs values into the arrays; the last END exits the
; programming mode, compiles and executes the procedure, and
; then returns you to the WAVE> prompt.
VEL, u, v
; Display the velocity field with default values ().

```

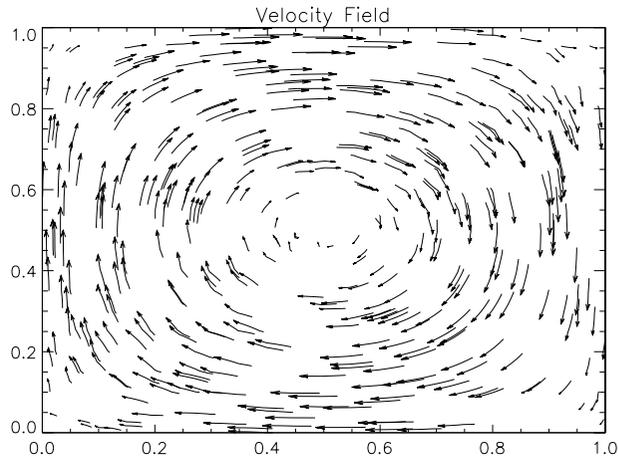


**Figure 2-69** Velocity field displayed with default values.

```

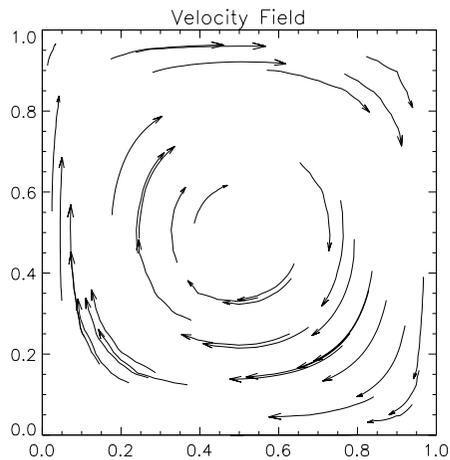
VEL, u, v, Nvecs=400
; Display the velocity field using 400 arrows ().

```



**Figure 2-70** Velocity field displayed with 400 arrows.

```
VEL, u, v, Nvecs=40, Xmax=.7, Length=.4, Nsteps=20
; Display the velocity field with individual modifications
; ().
```



**Figure 2-71** Velocity field displayed using various keywords.

## See Also

[PLOT\\_FIELD](#), [VELOVECT](#)

---

## VELOVECT Procedure

Standard Library procedure that draws a two-dimensional velocity field plot, with each directed arrow indicating the magnitude and direction of the field.

### Usage

VELOVECT,  $u$ ,  $v$  [,  $x$ ,  $y$ ]

### Input Parameters

$u$  — The  $x$  component of the two-dimensional field. This parameter must be a two-dimensional array.

$v$  — The  $y$  component of the two-dimensional field. This parameter must be a two-dimensional array of the same size as  $u$ .

$x$  — (optional) The abscissa values. This parameter must be a vector whose size equals the first dimension of  $u$  and  $v$ .

$y$  — (optional) The ordinate values. This parameter must be a vector whose size equals the second dimension of  $u$  and  $v$ .

### Keywords

**Dots** — If present and nonzero, places a dot at the position of the missing data. Otherwise, nothing is drawn for missing points. *Dots* is only valid if the *Missing* keyword is also specified.

**Length** — A length factor. The default value is 1.0, which makes the longest ( $u$ ,  $v$ ) vector have a length equal to the length of a single cell.

**Missing** — A two-dimensional array with the same size as the  $u$  and  $v$  arrays. It is used to specify that specific points have missing data.

If the magnitude of the vector at  $(i, j)$  is less than the corresponding value in *Missing*, then the data is considered to be valid. Otherwise, the data is considered to be missing.

Thus, one way to set up a *Missing* array is to initialize all elements to some large value:

```
missing_array = FLTARR(n, m) + 1.0E30
```

Then, if point  $(i, j)$  is a missing point, set the corresponding element to a negative value:

```
missing_array(i, j) = -missing_array(i, j)
```

## Discussion

VELOVECT draws a two-dimensional velocity field plot. The arrows indicate the magnitude and the direction of the field.

If missing values are present, you can use the *Missing* keyword to specify that they be ignored during the plotting, or the *Dots* keyword to specify that they be marked with a dot.

The system variables `![XY].Title` and `!P.Title` may be used to title the axes and the main plot.

## Examples

```
u = FLTARR(21, 21)
```

```
v = FLTARR(21, 21)
```

```
    ; Create the arrays.
```

```
.RUN
```

```
    ; After you type .RUN, the WAVE prompt changes to a dash (-) to
```

```
    ; indicate that you may enter a complete program unit.
```

```
FOR j = 0, 20 DO BEGIN
```

```
FOR i = 0, 20 DO BEGIN
```

```
x = 0.05 * FLOAT(i)
```

```
z = 0.05 * FLOAT(j)
```

```
u(i, j) = -SIN(!Pi*x) * COS(!Pi*z)
```

```
v(i, j) = COS(!Pi*x) * SIN(!Pi*z)
```

```
ENDFOR
```

```
ENDFOR
```

```
END
```

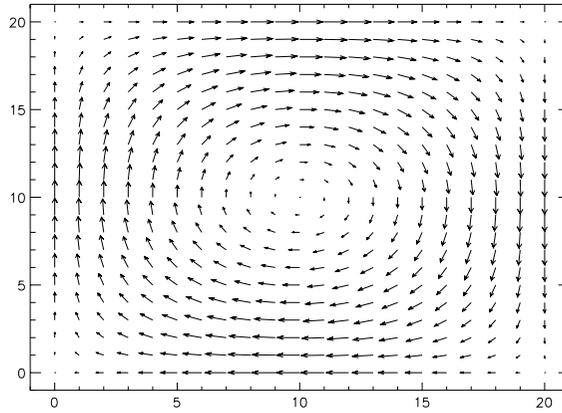
```
    ; This procedure stuffs values into the arrays; the last END exits the
```

```
    ; programming mode, compiles and executes the procedure, and
```

```
    ; then returns you to the WAVE prompt.
```

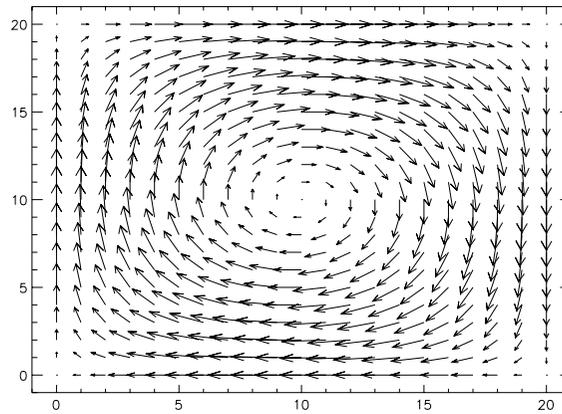
```
VELOVECT, u, v
```

```
    ; Display the velocity field with default values ().
```



**Figure 2-72** Velocity field displayed using default values.

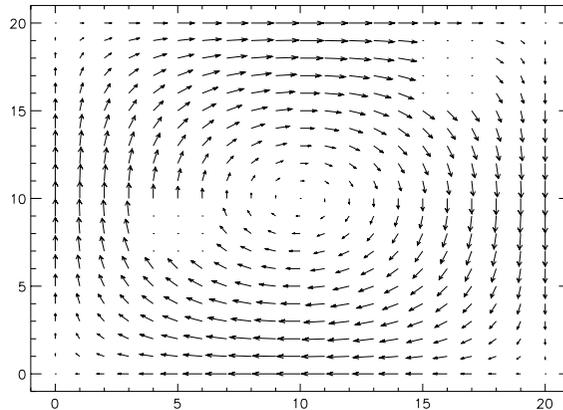
```
VELOVECT, u, v, Length=2
; Display the velocity field using arrows twice the length of a single
; cell ().
```



**Figure 2-73** Velocity field displayed using arrows twice the length of a single cell.

```
missing = FLTARR(21, 21) + 1.e30
missing(4:6, 7:9) = -1.e30
missing(15:17, 16:19) = -1.e30
VELOVECT, u, v, Missing=missing, /Dots
```

; Display the velocity field that contains missing data ().



**Figure 2-74** Velocity field that contains missing data.

## See Also

[PLOT\\_FIELD](#), [VEL](#)

---

## **VIEWER Procedure**

Lets users interactively define a 3D view, a slicing plane, and multiple cut-away volumes for volume rendering. (Creates a View Control and a View Orientation window in which to make these definitions.)

### **Usage**

`VIEWER`, *win\_num*, *xsize*, *ysize*, *size\_fac*, *xpos*, *ypos*, *colors*, *retain*, *xdim*, *ydim*, *zdim*

### **Input Parameters**

*win\_num* — The number of the graphics window to use for the View Orientation window. Since this window is left on the screen when VIEWER returns, it is up to the calling program to delete it when desired. (The View Control window is automatically deleted when VIEWER returns.)

*xsize*, *ysize* — The X and Y dimensions, respectively, of the graphics window that VIEWER is setting up the view for.

*size\_fac* — A factor controlling the size of the View Orientation window. The X dimension of the View Orientation window is (*size\_fac* \* *xsize*) and the Y dimension is (*size\_fac* \* *ysize*). Typically, *size\_fac* should be in the range 0.5 to 1.0.

*xpos, ypos* — The location (*x* and *y* positions, respectively) for the View Orientation and View Control windows.

If the View Orientation window is near the bottom of the screen, then the View Control window is created above the View Orientation window. Otherwise, the View Control window is created below the View Orientation window.

*colors* — The value (number of colors to allocate) to use as the *Colors* keyword value in the WINDOW procedure call. A typical value for an 8-bit color system is 128 or 256.

*retain* — The value (flag) to use as the *Retain* keyword value in the WINDOW procedure call. Typically this is the value 1 or 2.

For more information on *colors* and *retain*, see the description of the WINDOW procedure in the *PV-WAVE Reference*.

*xdim, ydim, zdim* — The size (first, second, and third dimension, respectively) of the array containing the data that is displayed using the view specified by VIEWER.

For example, if a 20-by-30-by-40 array is to be displayed using SHADE\_VOLUME and POLYSHADE, then *xdim* should be 20, *ydim* should be 30, and *zdim* should be 40.

If a 20-by-30 array is to be displayed using the SURFACE or CONTOUR procedures, then *xdim* should be 20, *ydim* should be 30, and *zdim* should be (MAX (array) + 1 . 0).

## Keywords

*Ax* — On input, if this keyword is omitted, or if the variable passed to *Ax* is undefined, then no controls for setting the view rotation about the *x*-axis are displayed.

If a valid value is passed to *Ax*, then this value is the initial (default) view rotation about the *x*-axis.

*Ay* — The view rotation about the *y*-axis (similar to *Ax*).

*Az* — The view rotation about the *z*-axis (similar to *Ax*).

On output, the *Ax*, *Ay*, *Az* input keywords each return a single floating-point value containing the *x*, *y*, and *z* rotation, respectively, that was selected.

The calling program does not need to do anything with the returned values  $A_x$ ,  $A_y$ , and  $A_z$ , since VIEWER automatically calls CENTER\_VIEW to set the system view transformation !P.T, as well as !P.T3D, !X.S, !Y.S, and !Z.S. The calling program may, however, use these returned values in subsequent calls to VIEWER to let users “pick up where they left off.”

**Bg\_Color** — The window background color. (Default: 0)

**Bot\_Color** — The bottom shadow color for buttons. (Default: 0)

**Cut\_Plane** — On input, if this keyword is omitted, or if the variable passed to *Cut\_Plane* is undefined, then no controls for setting the slicing plane are displayed.

For the cut-away controls to be displayed, a previously-defined variable must be passed to *Cut\_Plane*. If this variable is a valid (3, 2) integer array, then it is assumed to contain the initial (default) slicing plane.

If *Cut\_Plane* is a valid array, then its contents are interpreted as shown in the following table:

---

<i>Cut_Plane</i> (0, 0)	The plane’s angle of rotation about the $x$ -axis.
<i>Cut_Plane</i> (1, 0)	The plane’s angle of rotation about the $y$ -axis.
<i>Cut_Plane</i> (2, 0)	Ignored.
<i>Cut_Plane</i> (0, 1)	The $x$ -coordinate of the center of the plane.
<i>Cut_Plane</i> (1, 1)	The $y$ -coordinate of the center of the plane.
<i>Cut_Plane</i> (2, 1)	The $z$ -coordinate of the center of the plane.

---

**NOTE** The slicing plane is rotated about the  $y$ -axis first, then about the  $x$ -axis. If the variable passed to *Cut\_Plane* is defined but is not a valid (3, 2) array, then controls to set the slicing plane are displayed, the initial (default) slicing plane is defined with no rotation about the  $x$ - or  $y$ -axes, and the center point of the plane is at ( $x_{dim}/2$ ,  $y_{dim}/2$ ,  $z_{dim}/2$ ).

---

On output, the variable passed to *Cut\_Plane* is always a valid (3, 2) single-precision floating-point array in the form previously described.

You can use the array returned from *Cut\_Plane* as input to the SLICE\_VOL procedure to extract the slice, or in subsequent calls to VIEWER.

**Cut\_Vol** — On input, if this keyword is omitted, or if the variable passed to *Cut\_Vol* is undefined, then no controls for setting the cut-away volume(s) are displayed.

For the cut-away controls to be displayed, a previously-defined variable must be passed to *Cut\_Vol*. If this variable is a valid (6,  $n$ ) integer array, then it is assumed

to contain the initial (default) cutting volumes. This (6, *n*) array contains the subscript ranges in the display array that are to be cut away.

For example, if a (6, 2) array called *ca* is passed in, then two initial cutting volumes are defined. In this case, the contents of *ca* are interpreted as follows:

---

<i>ca</i> (0, 0)	<i>x</i> dimension of the first cut-away.
<i>ca</i> (1, 0)	<i>y</i> dimension of the first cut-away.
<i>ca</i> (2, 0)	<i>z</i> dimension of the first cut-away.
<i>ca</i> (3, 0)	<i>x</i> position of the first cut-away.
<i>ca</i> (4, 0)	<i>y</i> position of the first cut-away.
<i>ca</i> (5, 0)	<i>z</i> position of the first cut-away.
<i>ca</i> (0:5, 1)	Defines second cut-away (similar to <i>ca</i> (0:5, 0)).

---

If the variable passed to *Cut\_Vol* is defined, but is not a valid (6, *n*) array, then no initial cut-aways are defined, although the controls to define cut-aways are displayed.

On output, the variable passed to *Cut\_Vol* is always a valid (6, *n*) integer array of the form previously described. If you do not define any cut-aways, then this variable is returned as a (6, 1) array containing all zeroes.

Typically, after the cut-away information is returned to the calling program, the array of data to be displayed is modified by the calling program using the cut-away information. This modification usually involves setting portions of the display array to zero before using the display array with other commands, such as *RENDER*, *SHADE\_VOLUME*, *VOL\_REND*, and *VECTOR\_FIELD3*.

For example, if the display array is a 20-by-30-by-40 array (*xdim*=20, *ydim*=30, *zdim*=40) and the variable returned from *Cut\_Vol* is a (6, 1) array containing the values:

```
[19, 10, 13, 0, 6, 24]
```

then the portions of the display array *da* to set to zero are as follows:

```
da(0:(0+19), 6:(6+10), 24:(24+13)) = 0
```

You may use the array returned from *Cut\_Vol* in subsequent calls to *VIEWER*.

***Fg\_Color*** — The foreground color. Used for buttons and for the current cut-away volume. The default is !P.Color.

***HL\_Color*** — The color to use when highlighting buttons and for drawing the slicing plane. The default is !P.Color.

***Out\_Mode*** — If present and nonzero, then users are not allowed to exit VIEWER until they have specified a view in which all the vertices of the view cube lie completely within the View Orientation window.

---

**NOTE** Setting *Out\_Mode* to 1 ensures that VIEWER always sets up a view that is compatible with POLYSHADE. (POLYSHADE will not work properly if one or more polygon vertices lie outside the window.)

---

***Persp*** — On input, if this keyword is omitted, or if the variable passed to *Persp* is undefined, then no controls for setting the view perspective projection distance are displayed.

If a valid value is passed to *Persp*, then this value is used as the initial (default) setting for the perspective projection, and controls for setting the perspective are displayed.

On output, this input keyword returns the perspective projection distance that was selected.

The calling program does not need to do anything with the returned values for *Zoom* or *Persp*, since VIEWER automatically sets the view. However, you may use the returned value(s) in subsequent calls to VIEWER.

***Top\_Color*** — The top shadow color for buttons. Also used to draw the view cube. The default is !P.Color.

***Zoom*** — On input, if this keyword is omitted, or if the variable passed to *Zoom* is not a valid scalar or three-element vector, then no controls for setting the *Zoom* factor(s) are displayed.

If the variable passed to *Zoom* is a single value, then this value is used as the initial (default) setting for the zoom factor and a single control is provided for zooming the view equally in all three dimensions ( $x$ ,  $y$ , and  $z$ ).

If the variable passed to *Zoom* is a three-element vector, then these three values are the initial (default) values for the zoom factors and three controls are provided for zooming the view independently in the  $x$ ,  $y$ , and  $z$  directions.

On output, this input keyword returns the *Zoom* value(s) that was selected.

## Discussion

VIEWER returns the view parameters, slicing plane parameters, and cut-away volume coordinates to the calling program.

VIEWER automatically defines the view by calling the procedure CENTER\_VIEW, but it is up to the calling program to decide what to do with any

slicing plane or cut-away information returned to it. It is also up to the calling program to perform the rendering.

---

**NOTE** This procedure sets the system variables !P.T, !P.T3D, !X.S, !Y.S, and !Z.S, overriding any values you may have previously set. (These system variables are described in [Chapter 4, System Variables.](#))

---

Due to the large amount of code in this procedure, the following command must be entered (once per PV-WAVE session) before any procedure that calls VIEWER can be run:

```
WAVE> .size 32766 32766
```

### **Interactive Usage**

When VIEWER is called, a View Control and a View Orientation window are created that let users interactively define 3D viewing parameters. These windows are shown in [and](#) .



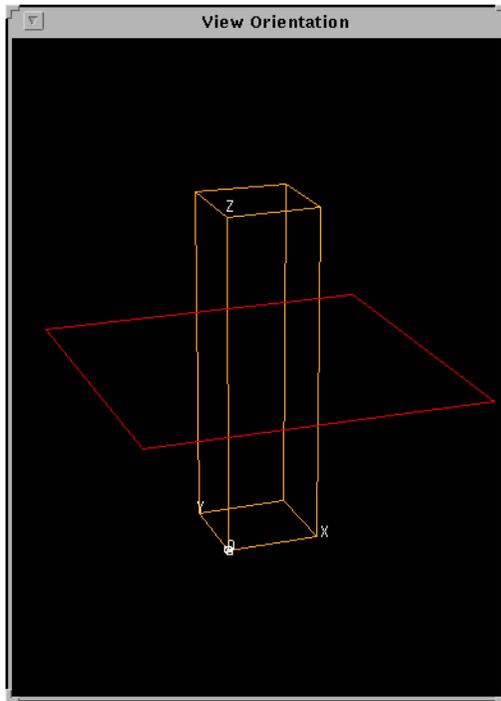
**Figure 2-75** The View Control window contains buttons used for setting the view, slicing plane, and cut-away volume parameters.

The controls (buttons) provided for the user depend on the keywords supplied to VIEWER. For example, if you do not wish the user to be able to set the y rotation parameter, then do not use the *Ay* keyword.

---

**TIP** It is advisable to use only the *Ax*, *Az*, and *Zoom* keywords when setting up a view for SURFACE or SHADE\_SURF, since these procedures are limited in the type of viewing transformation they can utilize. (Other commands, such as PLOTS, POLYFILL, POLY\_PLOT, and POLYSHADE, are compatible with most view transformations; therefore, you may freely use the *Ay* and *Persp* keywords when setting up the view for these routines.)

---



**Figure 2-76** The View Orientation window displays a cube representing the current state of the parameters set in the View Control window.

The available buttons in the View Control window are for X rotation, Y rotation, Z rotation, Perspective, and Zoom. Depending on how VIEWER is called, the *Zoom* keyword may consist of a single control, in which case the view is zoomed equally in the *x*, *y*, and *z* dimensions. There may also be three zoom controls provided, in which case the zoom factors for the *x*, *y*, and *z* dimensions may be set independently. There is also a **NONE** button provided for turning off the perspective projection.

---

**NOTE** If the zoom factor is large, or the perspective parameter is small, then the cube display in the View Orientation window may be erroneous. To cure the problem, reduce the zoom or increase the perspective (or set the perspective to **NONE**).

---

All of the parameters can be changed by clicking on the plus [+] or minus [-] button for that parameter. The [+] button increases the value of that parameter and the [-] button decreases it. For rapid (coarse) control, use the left mouse button. For fine

control, use the middle mouse button. For extra-fine control, use the right mouse button.

---

**Windows USERS** If you have a two-button mouse, use <Alt> in combination with the left mouse button.

---

If the *Cut\_Plane* keyword is supplied to VIEWER, then controls are provided for setting the slicing plane. The rotation and position of the slicing plane can be set. The center of the slicing plane can not be located outside the original volume. The slicing plane is visible in the View Orientation window.

If the *Cut\_Vol* keyword is supplied to VIEWER, then controls are provided for setting the cut-away volume(s). The *x* size, *y* size, *z* size, *x* position, *y* position, and *z* position for each cut-away volume can be set. The size of the cut-away can not be larger than the original volume. No part of any cut-away volume may be positioned outside the original volume.

Three additional buttons, **CLEAR**, **NEXT**, and **NEW**, are also provided in the View Control window:

- To create a new cut-away, click on the **NEW** button. This creates a small new cut-away near the coordinate origin. This cut-away is visible in the View Orientation window. You can then set the size and position of this cut-away by using the [+] and [-] buttons.
- If multiple cut-aways have been defined, the **NEXT** button allows users to toggle between them to modify their size and position.
- The **CLEAR** button removes all the cut-away volumes.

When you have finished setting the parameters, click the **DONE** button to set the view and return the parameters to the calling program.

If the *Out\_Mode* keyword has been set, then clicking the **DONE** button does not cause a return to the calling program unless users have specified a view in which all the vertices of the view cube lie completely within the View Orientation window. In this case, users then need to change the view parameters (rotation, zoom, and/or perspective) before they could exit.

When VIEWER returns to the calling program, the View Control window is deleted, but the View Orientation window is left on the screen.

The **RESET** button can be used to reset all the parameters to their initial state.

## Examples

For demonstrations of the VIEWER procedure, use the **4-D Data**, **Medical Imaging**, **Oil/Gas Exploration**, and **CFD/Aerospace** buttons on the PV-WAVE Demonstration Gallery. To run the Gallery, enter `wave_gallery` at the `WAVE>` prompt.

## See Also

[CENTER\\_VIEW](#)

---

## ***VOL\_MARKER Procedure***

Displays colored markers scattered throughout a volume.

### Usage

`VOL_MARKER, vol, n_points`

### Input Parameters

*vol* — A 3D volume of data to plot markers in.

*n\_points* — The number of markers to plot.

### Keywords

*Axis\_Color* — The color to use when plotting the axis. To suppress the axis, set *Axis\_Color* to -1.

*Copy* — If specified, preserves the input variable, *vol*. If this keyword is not specified, the input variable is altered during processing.

*Mark\_Size* — The maximum marker size.

*Mark\_Symbol* — A number specifying the marker symbol to use. The number should be between 1 and 7. The default is 2 (an asterisk). For a list of symbols, see the description of !Psym in [Chapter 4, System Variables](#).

*Mark\_Thick* — The maximum line thickness to use when plotting markers. Typically, this is an integer between 1 and 7. (A thickness of 1.0 is normal, 2.0 is twice as wide, and so on.)

## Discussion

VOL\_MARKER plots a polymarker field from volumetric data. The color of each marker displayed by VOL\_MARKER is determined by the value of the volumetric data at the point where the marker is plotted.

Unless the *Copy* keyword is specified, the original input variable is altered.

## Examples

```
PRO vol_demo2
    ; This program displays an MRI scan of a human head using three
    ; different display techniques.

volx = 115
voly = 75
volz = 105
    ; Specify the size of the volumes.

winx = 512
winy = 512
    ; Specify the window size.

head = BYTARR(volx, voly, volz)
OPENR, 1, !Data_Dir + 'man_head.dat'
READU, 1, head
CLOSE, 1
    ; Read in the volumetric data.

band = 5
head = VOL_PAD(head, band)
head = SMOOTH(head, band)
    ; Pad the volume with zeroes and smooth it.

CENTER_VIEW, Xr=[0, 124], Yr=[0, 84], $
    Zr=[0, 114], Az=60.0, Ax=(-60.0), $
    Winx=512, Winy=512, Zoom=0.9
WINDOW, 0, XSize=winx, YSize=winx, $
    XPos=16, YPos=384, Colors=128
LOADCT, 9
    ; Set up the viewing window and load the color table.

VOL_MARKER, head, 6000, Axis_Color=100, $
    Mark_Symbol=2, Mark_Size=2, Mark_Thick=2
    ; Render the data using a 3D polymarker field.

WINDOW, 1, XSize=winx, YSize=winx, $
    XPos=496, YPos=324, Colors=128
    ; Create a second window for plotting.
```

```

SET_SHADING, Light=[-1.0, 1.0, 0.5], $
    /Gouraud, /Reject
    ; Change the direction of the light source for shading.

SHADE_VOLUME, head, 18, vertex_list, polygon_list, /Low
    ; Compute the 3D contour surface as a list of polygons.

TVSCL, POLYSHADE(vertex_list, polygon_list,$
    XSize=winx, YSize=winy, /Data, /T3d)
    ; Display the polygon list with light source shading.

WINDOW, 2, XSize=winx, YSize=winy, $
    XPos=256, YPos=48, Colors=128
    ; Create another window for plotting.

head = VOL_TRANS(head, 128, !P.T)
    ; Transform the volumetric data to the current view.

TVSCL, VOL_REND(head, winx, winy, Depth_Q=0.4)
    ; Display a translucent image of the data.

END

```

## See Also

[VECTOR\\_FIELD3](#)

---

## ***VOL\_PAD Function***

Returns a 3D volume of data padded on all six sides with zeroes.

### **Usage**

*result* = VOL\_PAD(*volume*, *pad\_width*)

### **Input Parameters**

*volume* — On input, volume contains the 3D volume of data to pad.

*pad\_width* — The width of the padding around the volume. The size of the resulting volume increases by  $(2 * \textit{pad\_width})$  in all three dimensions.

### **Returned Value**

*result* — The padded *volume* data.

## Keywords

None.

## Discussion

For best results, process volumes with VOL\_PAD before transforming them with VOL\_TRANS or slicing them with SLICE\_VOL. For more information, see .

## Examples

See the Examples section in the description of the [VOL\\_MARKER](#) routine.

For other examples, see the `vol_demo3`, `vol_demo4`, and `grid_demo4` demonstration programs in:

(UNIX) `<wavedir>/demo/arl`

(OpenVMS) `<wavedir>:[DEMO.ARL]`

(Windows) `<wavedir>\demo\arl`

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[VOL\\_TRANS](#), [PADIT](#)

---

## ***VOL\_REND Function***

Renders volumetric data in a translucent manner.

### **Usage**

*result* = VOL\_REND(*volume*, *imgx*, *imgy*)

### **Input Parameters**

*volume* — A 3D array containing volumetric data. *volume* is normally scaled into the range {0 ... 255}.

*imgx* — The *x* dimension of the image to return.

*imgy* — The *y* dimension of the image to return.

## Returned Value

*result* — An 8-bit image of the volumetric data.

## Keywords

*Depth\_q* — A scalar depth queuing factor. The *Depth\_q* keyword values should be between 0.0 and 1.0:

- A factor of 1.0 causes voxels in the back to be just as bright as the voxels in the front.
- A factor of 0.5 causes voxels in the back to be half as bright as those in front.

*Opaque* — A 3D array (with the same dimensions as volume) containing the translucency values for each voxel. *Opaque* is normally scaled into the range {0 ... 255}, where 0 is clear and 255 is completely opaque. (Default: 0)

## Discussion

If no keywords are used, the final intensity value produced at a given pixel with VOL\_RENDER is the brightest value along the Z dimension of the volume. This default behavior can be enhanced, however, by using the *Opaque* and *Depth\_q* keywords.

- If *Opaque* is set to 0, the resulting value will be unaffected; if *Opaque* is set to 255, then values behind that position in the opaque array will be completely blocked.
- If *Depth\_q* is set to a value less than 1.0, a bright spot in the back will be dimmed in proportion to the distance it is from the viewpoint. (A bright spot in the front will remain bright.)

Typically, you would first process a volume of data with VOL\_PAD and VOL\_TRANS, and then render it with VOL\_RENDER.

## Examples

```
PRO vol_demo3
    ; This program displays 3D fluid data using two display techniques.

volx = 17
voly = 17
volz = 59
    ; Specify the size of the volumes.

winx = 512
```

```

winy = 512
    ; Specify the window size.

flow_axial = FLTARR(volx, voly, volz)
OPENR, 1, !Data_Dir + 'cfd_axial.dat', /Xdr
READU, 1, flow_axial
CLOSE, 1
flow_axial = VOL_PAD(flow_axial, 1)
    ; Read in the data and pad with zeroes.

CENTER_VIEW, Xr=[0.0, 18.0], $
    Yr=[0.0, 18.0], Zr=[0.0, 60.0], $
    Az=210.0, Ay=120.0, Ax=0.0, Winx=512, Winy=512, Zoom=0.85
    ; Set up the view.

SET_SHADING, Light=[-1.0, 1.0, 0.5], /Gouraud, /Reject
    ; Change the direction of the light source for shading.

SHADE_VOLUME, flow_axial, 110, vertex_list, polygon_list, /Low
    ; Compute the 3D contour surface as a list of polygons.

WINDOW, 1, XSize=winx, YSize=winx, XPos=16, YPos=256, Colors=128
LOADCT, 3
    ; Set up the viewing window and load the color table.

img1 = POLYSHADE(vertex_list, polygon_list, $
    XSize=winx, YSize=winy, /Data, /T3d)
TVSCL, img1
    ; Construct the shaded surface representation of the data as a list of
    ; polygons and display it.

WINDOW, 2, XSize=winx, YSize=winx, $
    XPos=496, YPos=324, Colors=128
    ; Create a new window for plotting.

vol_dim = MAX([volx, voly, volz])
flow_axial = BYTSCL(flow_axial)
    ; Scale the data into the range of bytes 0 – 255.

vol2 = VOL_TRANS(flow_axial, vol_dim, !P.T)
    ; Transform the volume of data into the current view.

img2 = VOL_REND(vol2, winx, winy, Depth_q=0.4)
    ; Render the data as a translucent image.

TVSCL, img2
    ; Display the image.

END

```

For other examples, see the demonstration programs `vol_demo2` and `vol_demo4` in:

(UNIX) <wavedir>/demo/arl  
(OpenVMS) <wavedir>:[DEMO.ARL]  
(Windows) <wavedir>\demo\arl

Where <wavedir> is the main PV-WAVE directory.

## See Also

[VOL\\_TRANS](#)

---

## ***VOL\_TRANS* Function**

Standard Library routine that returns a 3D volume of data transformed by a 4-by-4 matrix.

### **Usage**

*result* = VOL\_TRANS(*volume*, *dim*, *trans*)

### **Input Parameters**

*volume* — The 3D volume of data to transform.

*dim* — A scalar value specifying the *x*, *y*, and *z* dimensions of the transformed volume to return. Normally, *dim* is the largest of the three dimensions of the original volume. Generally, the original volume should “fit” inside the transformed volume.

*trans* — The 4-by-4 transformation matrix to use for the transformation. *trans* is often the system viewing transformation matrix !P.T. (For more information, see the section *Geometric Transformations* in Chapter 6 of the *PV-WAVE User’s Guide*.)

### **Returned Value**

*result* — A 3D volume of data transformed by a 4-by-4 matrix.

### **Keywords**

None.

### **Discussion**

The returned volume is scaled into the range of bytes. For best results, the volume to transform should first be processed using the VOL\_PAD function. For more

information, see the section *Volume Manipulation* in Chapter 7 of the *PV-WAVE User's Guide*.

## Examples

See the Examples sections in the description of the [VOL\\_MARKER](#) and [VOL\\_RENDER](#) routines.

For another example, see the `vol_demo4` demonstration program in:

```
(UNIX)      <wavedir>/demo/ar1  
(OpenVMS)  <wavedir>:[DEMO.ARL]  
(Windows)  <wavedir>\demo\ar1
```

Where `<wavedir>` is the main PV-WAVE directory.

## See Also

[VOL\\_PAD](#), [VOL\\_RENDER](#)

---

## VOLUME Function

Defines the volumetric data that can be used by the RENDER function.

### Usage

```
result = VOLUME(voxels)
```

### Input Parameters

*voxels* — A 3D byte array containing voxel data.

### Returned Value

*result* — A structure that defines a volumetric object.

### Keywords

*Color* — A 256-element double-precision floating-point vector containing the color (intensity) coefficients of the object.

(Default: `Color(*) = 1.0`) For more information, see .

*Kamb* — A 256-element double-precision floating-point vector containing the ambient (flat shaded) coefficients.

(Default: `Kamb=FINDDGEN(256)/255`) For more information, see .

**Kdiff** — A 256-element double-precision floating-point vector containing the diffuse reflectance coefficients.

(Default: `Kdiff(*)=0.0`) For more information, see .

**Ktran** — A 256-element double-precision floating-point vector containing the specular transmission coefficients.

(Default: `Ktran(*)=0.0`) For more information, see the section *Transmission Component* in Chapter 7 of the *PV-WAVE User's Guide*.

**Transform** — A 4-by-4 double-precision floating-point array containing the local transformation matrix whose default is the identity matrix. For more information, see the section *Setting Object and View Transformations* in Chapter 7 of the *PV-WAVE User's Guide*.

## Discussion

A VOLUME is used by the RENDER function to render volumetric data. You must specify a 3D array of bytes that represent this data in the call to VOLUME. Each byte in the voxel array corresponds to an index into the material properties associated with the volume.

For example, the material properties used for shading the point  $(x, y, z)$  in some data are `Color(voxels(x, y, z))`, `Kamb(voxels(x, y, z))`, etc. The surface normal at  $(x, y, z)$  is calculated using a 3D Sobel gradient operator on the actual voxel values. The default orientation of a volume is an origin-centered unit cube. For more information, see the section *Defining Object Material Properties* in Chapter 7 of the *PV-WAVE User's Guide*.

If the voxels are not cubic, you may adjust the scaling of the dimensions with the *Transform* keyword by using a matrix generated by the T3D procedure with the *Scale* keyword.

Volumetric data is applicable to any voxel processing domain, such as for the visualization of astronomical, geological, and medical data.

## Example

```
voxels = BYTARR(16, 16, 16)
voxels(*) = 255
diffuse = FLTARR(256)
T3D, /Reset, Rotate=[15, 30, 45]
cube = VOLUME(voxels, Transform=!P.T, $
             Kdiff=diffuse, Kamb=FLTARR(256))
TV, RENDER(cube)
```

## See Also

[CONE](#), [CYLINDER](#), [MESH](#), [RENDER](#), [SHADE\\_VOLUME](#), [SPHERE](#)

For more information, see the section *Ray-tracing* in Chapter 7 of the *PV-WAVE User's Guide*.

---

## VRML\_AXIS Procedure

Adds an axis to a VRML world.

### Usage

VRML\_AXIS, *origin* [, *length*, *range*]

### Input Parameters

***origin*** — A 3-element array of coordinates specifying the axis origin.

***length*** — (optional) The length of the axis. (Default: 1)

***range*** — (optional) A 2-element array specifying the coordinate values defining the axis. (Default: [0, 1])

### Keywords

#### ***Object Properties***

The following keywords describe or define the VRML axis object.

***Radius*** — Specifies the radial length of all objects of the axis. (Default: 0.01 of the *length* parameter)

***Title*** — A string specifying the axis title.

***Transform*** — A 4-by-4 matrix for rotating an axis.

---

**TIP** If you need an axis pointing in some direction other than *x*, *y*, or *z*, then use the *Transform* keyword to rotate one of these three axes in the desired direction.

---

***X*** — Adds an *x*-axis (the default).

***Y*** — Adds a *y*-axis.

**Z** — Adds a z-axis.

### ***Material Properties***

The following properties, when set, are applied to all objects of an axis (cylinders, cones, cubes, spheres, and text), as applicable.

***AmbientColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***DiffuseColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***EmissiveColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***Shininess*** — Scalar shininess factor.

***SpecularColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is taken as grayscale.

***Transparency*** — Scalar transparency factor, in the range of 0 to 255.

### ***Font Family Settings***

These three font attributes are mutually exclusive.

***Serif*** — Serif font (the default).

***Sans*** — Sans-serif font.

***Typewriter*** — Monospaced font.

### ***Font Attributes***

Either or both of these keywords may be used. If neither is set, then the typeface is “normal.”

***Bold*** — Boldface type.

***Italic*** — Italic type.

## **Discussion**

The VRML\_AXIS procedure has no direct correlation with a VRML predefined type. This procedure produces a combination of a cylinder (the axis body), a cone (the axis arrowhead), and text (the axis range and titles).

## Example

```
VRML_OPEN, 'vrm1_axis.wrl'  
    ; Start the VRML file.  
  
VRML_AXIS, Title = 'Default Axis'  
    ; Create a default axis first.  
  
VRML_AXIS, [1, 0, 0], 3, [-100, 200], /Y,$  
    Radius = .05, Title = 'Testing Y -Sans - Italic Axis', $  
    /Sans, /Italic  
    ; Make a Y axis, starting at a different origin.  
  
VRML_AXIS, [0, 0, 0], 3, /Z, $  
    Title = 'Testing Z Axis', $  
    DiffuseColor = [127, 255, 191], $  
    AmbientColor = [55, 70, 60], $  
    EmissiveColor = [200, 100, 50], $  
    Transparency = 120, $  
    Specular = [10, 10, 10], /Bold, /TypeWriter  
    ; Make a Z axis, with a material list.  
  
VRML_CLOSE
```

## See Also

[VRML\\_CONE](#), [VRML\\_CYLINDER](#), [VRML\\_OPEN](#), [VRML\\_SURFACE](#),  
[VRML\\_TEXT](#)

For a discussion of VRML primitives and color parameter definitions, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 3.

---

## VRML\_CAMERA Procedure

Positions a VRML camera for rendering a VRML view.

### Usage

VRML\_CAMERA, *position*

### Input Parameters

*position* — A 3-element array specifying the camera position coordinates.

## Keywords

*FocalLength* — Camera focal length. (Default: 5)

*LookAt* — A 3-element array ( $x, y, z$ ) of a target location used to calculate the camera orientation. (Default: [0, 0, 0])

*ViewAngle* — The viewing angle (field of view) in degrees. (Default: 45)

## Discussion

A viewpoint, or camera, is a predefined viewing position and orientation in a VRML world. You can specify the location and viewing direction of the viewpoint using the VRML\_CAMERA procedure.

## See Also

[VRML\\_LIGHT](#), [VRML\\_OPEN](#), [VRML\\_SPOTLIGHT](#)

For a discussion of VRML cameras, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 22.

---

## **VRML\_CLOSE Procedure**

Closes the VRML file.

### Usage

VRML\_CLOSE

### Input Parameters

None.

### Keywords

None.

## Discussion.

The VRML\_CLOSE procedure closes a file that was opened using the VRML\_OPEN procedure; it also resets the internal structure that supports the VRML functions.

## See Also

[VRML\\_OPEN](#)

---

## VRML\_CONE Procedure

Creates a VRML cone.

## Usage

VRML\_CONE

## Input Parameters

None.

## Keywords

### ***Object Properties***

The following keywords describe or define the VRML cone object.

***Center*** — A 3-element array specifying the true center of the cone object. (Default: [0, 0, 0])

***Height*** — Specifies the cone height (Default: 2.0)

***Orientation*** — A 3-element array specifying the orientation of the cone (in the direction of the apex). (Default: cone axis orientation along y, with the apex at +y)

***Radius*** — Specifies the radius of the cone base from the y-axis. (Default: 1.0)

***Transform*** — A 4-by-4 matrix containing the transformation to be applied to the cone object; similar to !P.T.

## **Material Properties**

The following properties, when set, are applied to the object.

**AmbientColor** — A 3-element array of RGB color with each element ranging between 0 to 255. If the value is scalar, then the color is interpreted as grayscale.

**DiffuseColor** — A 3-element array of RGB color with each element ranging between 0 to 255. If the value is scalar, then the color is interpreted as grayscale.

**EmissiveColor** — A 3-element array of RGB color with each element ranging between 0 to 255. If the value is scalar, then the color is interpreted as grayscale.

**Shininess** — Scalar shininess factor, in the range from 0 to 255.

**SpecularColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**Texture\_Image** — A texture image to apply to the object. The image is wrapped completely around the object. The texture image values and transparency values lie in the range of 0 to 255.

There are four options:

(*w, h*) or (1, *w, h*) — Grayscale image

(2, *w, h*) — Grayscale in (0, \*, \*) plus transparency in (1, \*, \*)

(3, *w, h*) — True color image (red, green, blue)

(4, *w, h*) — True color plus transparency in (3, \*, \*)

**Transparency** — Scalar transparency factor, in the range of 0 to 255.

## **Discussion**

The VRML\_CONE procedure supports the cone node in VRML. The default cone has a base radius of 1.0 and a height of 2.0 such that it extends +1.0 and -1.0 in each direction from the object center.

## **Example**

The 'wavedir.dat' file used in this example is found in the following directory:

(UNIX) <wavedir>/demo/web/vrml

(OpenVMS) <wavedir>:[DEMO.WEB.VRML]

(Windows) <wavedir>\demo\web\vrml

where <wavedir> is the main PV-WAVE directory.

```
VRML_OPEN, 'vrml_cone.wrl'  
x = [0, 1, 2, 3, 4]  
y = [0, 1, 0, 1, 0]  
z = [4, 3, 2, 1, 0]  
RESTORE, 'wavelogo.dat' ; img, r, g, b  
sz = SIZE(img)  
col = TRANSPOSE([[r], [g], [b]])  
img24 = REFORM(col(*, img), 3, sz(1), sz(2))  
    ; Create texture.  
VRML_CONE  
    ; Create the default cone.  
VRML_CONE,Center = [0, 3, 0], Radius = .4, $  
    Height = 4.0, Texture = img24  
    ; Create the textured cone.  
VRML_CONE,Center = [2, 0, 0], $  
    Radius = 1.4, $  
    Height = 1.2, $  
    DiffuseColor = [127, 255, 191], $  
    AmbientColor = [55, 70, 60], $  
    SpecularColor = [0, 100, 0]  
    ; Create a colorful cone.  
VRML_CLOSE
```

## See Also

[VRML\\_CUBE](#), [VRML\\_CYLINDER](#),  
[VRML\\_OPEN](#), [VRML\\_SPHERE](#), [VRML\\_TEXT](#)

For a discussion of cones and other VRML primitives, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 3.

---

## **VRML\_CUBE Procedure**

Positions a VRML cube in the world.

### **Usage**

VRML\_CUBE

### **Input Parameters**

None.

### **Keywords**

#### ***Object Properties***

The following keywords describe or define the VRML cube object.

***Center*** — A 3-element array specifying the true center of the cube object. (Default: [0, 0, 0])

***Rotation*** — A 3-element array of values in degrees, specifying the rotation about the *x*, *y*, and *z* axes, respectively.

***Transform*** — A 4-by-4 matrix containing the transformation to be applied to the cube object.

***Widths*** — A scalar value applied to each of the three directions, or a 3-element array of values specifying the width in each direction (*x*, *y*, *z*) individually. (Default: 2.0)

#### ***Material Properties***

The following properties, when set, are applied to the object.

***AmbientColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***DiffuseColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***EmissiveColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***Shininess*** — Scalar shininess factor, in the range of 0 to 255.

***SpecularColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***Texture\_Image*** — A texture image to apply to the object. The image is wrapped completely around the object. The texture image values and transparency values lie in the range of 0 to 255.

There are four options:

(*w, h*) or (1, *w, h*) — Grayscale image

(2, *w, h*) — Grayscale in (0, \*, \*) plus transparency in (1, \*, \*)

(3, *w, h*) — True color image (red, green, blue)

(4, *w, h*) — True color plus transparency in (3, \*, \*)

***Transparency*** — A scalar transparency factor, in the range of 0 to 255.

## Discussion

The VRML\_CUBE procedure produces a 2-by-2-by-2 cube centered around a defined origin. This routine supports the cube node in VRML.

## Example

The 'wavelogo.dat' file used in this example is found in the following directory:

(UNIX) <wavedir>/demo/web/vrml

(OpenVMS) <wavedir>:[DEMO.WEB.VRML]

(Windows) <wavedir>\demo\web\vrml

where <wavedir> is the main PV-WAVE directory.

```
RESTORE, 'wavelogo.dat'
    ; Restore variables for img, r, g, b.

sz = SIZE(img)
col = TRANSPOSE([[r], [g], [b]])
img24 = REFORM(col(*, img), 3, sz(1), sz(2))
VRML_CUBE, Center = [-5, -3, 5], Width = 3, Texture = img24
    ; A cube with the wave logo on it.

VRML_CUBE, Center = [4, -2, -2], $
    Rotation = [0, 0, 0], $
    DiffuseColor = [127, 255, 191], $
```

```
AmbientColor = [55, 70, 60], $
SpecularColor = [0, 100, 0]
VRML_CUBE, Center = [4, -2, -2], $
Rotation = [45, 0, 0], $
DiffuseColor = [127, 255, 191], $
AmbientColor = [55, 70, 60], $
SpecularColor = [0, 100, 0]
VRML_CUBE, Center = [4, -2, -2], $
Rotation = [0, 45, 0], $
DiffuseColor = [127, 255, 191], $
AmbientColor = [55, 70, 60], $
SpecularColor = [0, 100, 0]
VRML_CUBE, Center = [4, -2, -2], $
Rotation = [0, 0, 45], $
DiffuseColor = [127, 255, 191], $
AmbientColor = [55, 70, 60], $
SpecularColor = [0, 100, 0]
; An Escher-like cube.
```

## See Also

[VRML\\_CONE](#), [VRML\\_CYLINDER](#), [VRML\\_OPEN](#),  
[VRML\\_SPHERE](#), [VRML\\_TEXT](#)

For a discussion of cubes and other VRML primitives, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 3.

---

## **VRML\_CYLINDER Procedure**

Positions a VRML cylinder in the world.

### **Usage**

VRML\_CYLINDER

### **Input Parameters**

None.

## Keywords

### ***Object Properties***

The following keywords describe or define the VRML cylinder object.

***Bottom*** — Displays the bottom of the cylinder only. (Default: set)

---

**NOTE** The default setting for each of the *Bottom*, *Sides*, and *Top* keywords is set, unless one or more of those keywords is explicitly specified. See the *Discussion* for more information.

---

***Center*** — A 3-element array specifying the true center of the cylinder object. (Default: [0, 0, 0])

***Height*** — Specifies the cylinder height. (Default: 2.0)

***Orientation*** — A 3-element vector specifying the orientation of the cylinder. (Default: along the y-axis)

***Radius*** — Specifies the cylinder radius from the axis. (Default: 1.0)

***Sides*** — Displays sides of the cylinder. (Default: set)

***Top*** — Displays the top of the cylinder. (Default: set)

***Transform*** — A 4-by-4 matrix containing the transformation to be applied to the object.

### ***Material Properties***

The following properties, when set, are applied to the object.

***AmbientColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***DiffuseColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***EmissiveColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***Shininess*** — Scalar shininess factor, in the range of 0 to 255.

***SpecularColor*** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

***Texture\_Image*** — A texture image to apply to the object. There are four options:

(*w, h*) or (1, *w, h*) — Grayscale image

(2, *w, h*) — Grayscale in (0, \*, \*) plus transparency in (1, \*, \*)

(3, *w, h*) — True color image (red, green, blue)

(4, *w, h*) — True color plus transparency in (3, \*, \*)

The image is wrapped completely around the object. The texture image values and transparency values lie in the range of 0 to 255.

**Transparency** — Scalar transparency factor, in the range of 0 to 255.

## Discussion

The VRML\_CYLINDER procedure supports the cylinder node in VRML.

The *Bottom*, *Sides* and *Top* keywords all refer to the viewed (solid) surfaces of the cylinder. If none of the keywords is explicitly specified, the cylinder appears as a closed solid object. If any of these keywords are specified, the walls corresponding to the keywords are the only solid walls displayed; the non-specified surfaces do not appear. For example, If *Sides* is specified, then neither the top nor the bottom of the cylinder is displayed, and the cylinder appears to be a hollow tube.

## Example

```
VRML_OPEN, 'vrlm_cylldr.wld'
VRML_CYLINDER, Center = [10, 11.5, -10], $
  Orientation = [0, 1, 0], Radius = 0.4, $
  Height = 0.5, DiffuseColor = [160, 160, 0]
  ; The cylinder appears as a solid object in this example, because
  ; none of the surface viewing keywords (Bottom, Sides, or Top) was
  ; specified.
VRML_CLOSE
```

## See Also

[VRML\\_CONE](#), [VRML\\_CUBE](#), [VRML\\_OPEN](#), [VRML\\_SPHERE](#),  
[VRML\\_TEXT](#)

For a discussion of cylinders and other VRML primitives, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 3.

---

## **VRML\_LIGHT Procedure**

Sets up the light source for a VRML world.

### **Usage**

VRML\_LIGHT, *position*

### **Input Parameters**

*position* — A 3-element array specifying the positioning of the light source.

### **Keywords**

*Color* — A 3-element array of RGB values, in the range of 0 to 255. (Default: [255, 255, 255], the color white)

*Intensity* — A number normalized from 0 to 1, where 1 is full intensity. (Default: 1)

### **Discussion**

VRML supports three node types to control lighting. The VRML\_LIGHT procedure implements one of the types, the PointLight, which emanates light radially in all directions.

---

**NOTE** See the VRML\_SPOTLIGHT procedure for another implementation of the VRML node types to control lighting.

---

### **Example**

```
VRML_OPEN, 'vrm1_light.wrl'  
...  
VRML_LIGHT, [-1, -1, 0], Color = [255, 0, 0], Intensity = 1.0  
VRML_CLOSE
```

### **See Also**

[VRML\\_CAMERA](#), [VRML\\_SPOTLIGHT](#)

For a discussion of VRML lighting, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 19.

---

## **VRML\_LINE Procedure**

Creates a VRML polyline object.

### **Usage**

VRML\_LINE,  $x$ ,  $y$ [,  $z$ ]

### **Input Parameters**

$x$ ,  $y$  — Each parameter is a 1D array containing  $npoint$  elements specifying the polyline coordinates.

$z$  — (optional) A 1D array containing  $npoint$  elements, specifying a  $z$ -axis polyline coordinate. (Default: 0, a line on the  $x$ - $y$  plane)

### **Keywords**

#### ***Object Properties***

The following keywords describe or define the VRML polyline object.

***Transform*** — A 4-by-4 matrix containing the transformation to be applied to the polyline object.

#### ***Material Properties***

The following properties, when set, are applied to the line.

***AmbientColor*** — A  $(3, n)$  array of RGB color, in the range of 0 and 255. If *AmbientColor* is an  $n$ -element array, then the color is interpreted as grayscale.

***DiffuseColor*** — A  $(3, n)$  array of RGB color, in the range of 0 and 255. If *DiffuseColor* is an  $n$ -element array, then the color is interpreted as grayscale.

***EmissiveColor*** — A  $(3, n)$  array of RGB color, in the range of 0 and 255. If *EmissiveColor* is an  $n$ -element array, then the color is interpreted as grayscale.

***MaterialIndices*** — An array of indices into the material property arrays in the range of 0 to  $n - 1$ , relating each vertex of the polyline to the set of material properties (*AmbientColor*, *DiffuseColor*, *EmissiveColor*, *SpecularColor*, *Transparency*,

*Shininess*). There should be *NPOINT* elements in this array (one per vertex in the polyline). (Default:  $n = npoint$ , where the set of material properties relates one-to-one to its corresponding vertices)

*Shininess* — An  $n$ -element array of shininess, in the range of 0 to 255.

*SpecularColor* — A  $(3, n)$  array of RGB color, in the range of 0 to 255. If *SpecularColor* is an  $n$ -element array, then the color is interpreted as grayscale.

*Transparency* — An  $n$ -element array of transparency, in the range of 0 to 255.

## Discussion

The VRML\_LINE procedure draws a polyline using an IndexedLineSet node.

## Example

```
VRML_OPEN, 'vrml_line.wrl'  
x = FINDGEN(10)/9  
y = RANDOMU(s, 10)  
VRML_LINE, x, y  
VRML_CLOSE
```

## See Also

[PLOTS](#), [VRML\\_OPEN](#), [VRML\\_POLY](#)

For a discussion of VRML polylines, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 14.

---

## **VRML\_OPEN Procedure**

Opens a VRML file and writes out header information consistent with VRML formatting.

### **Usage**

VRML\_OPEN [, *filename*]

### **Input Parameters**

*filename* — (optional) A string specifying the file in which to write the VRML formatting. (Default: 'wave.wrl')

### **Keywords**

*CGI* — Writes out http content-type header for a common gateway interface (CGI) script. The format is of the following form:

Content-type: x-world/x-vrml

*SceneInfo* — A scalar or array of strings containing comments about the scene.

*StdOut* — Directs the output to standard out, rather than to a file.

*Title* — A scalar string specifying the title of the created VRML world.

*Version* — A scalar string. If the VRML version is not “1.0 ascii”, then this keyword indicates it: e.g.: `Version = '1.1 qf8b'`.

### **Discussion**

The VRML\_OPEN procedure opens a .wrl file and writes the required header information to the file. VRML\_OPEN must precede all other VRML routines, and the file must be closed using the VRML\_CLOSE procedure.

### **Example**

```
VRML_OPEN, 'test_vrml.wrl', Title = 'Test VRML'  
...  
VRML_CLOSE
```

## See Also

[VRML\\_CLOSE](#)

For a discussion of VRML, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996.

---

## VRML\_POLY Procedure

Constructs a surface or a solid described by a set of polygons.

### Usage

VRML\_POLY, *vlist*, *plist*

### Input Parameters

*vlist* — Vertex list (3, *NVERT*) of *x*, *y*, *z* coordinates for *NVERT* vertices of the polygons.

*plist* — Polygon list describing *NPOLY* polygons. (See the POLYSHADE function for more information about this parameter.)

### Keywords

#### **Object Properties**

The following keywords describe or define the VRML polygon object.

***Solid*** — If set, indicates that the polygon encloses a solid.

---

**TIP** Browser viewing efficiency is enhanced by using the *Solid* keyword, because the browser knows to look only at the specified inside or outside of the polygon shape instead of both.

---

***Transform*** — A 4-by-4 matrix containing the transformation to be applied to the object.

#### **Material Properties**

The following properties, when set, are applied to the polygons.

**AmbientColor** — A (3,  $n$ ) array of RGB color, in the range of 0 and 255. If *AmbientColor* is an  $n$ -element array, then the color is interpreted as grayscale.

**DiffuseColor** — A (3,  $n$ ) array of RGB color, in the range of 0 and 255. If *DiffuseColor* is an  $n$ -element array, then the color is interpreted as grayscale.

**EmissiveColor** — A (3,  $n$ ) array of RGB color, in the range of 0 and 255. If *EmissiveColor* is an  $n$ -element array, then the color is interpreted as grayscale.

**MaterialIndices** — An array of indices into the material property arrays in the range of 0 to  $n - 1$ , relating each polygon face to one of the set of material properties (*AmbientColor*, *DiffuseColor*, *EmissiveColor*, *Shininess*, *SpecularColor*, *Transparency*). There should be *NPOLY* elements in this array (one per polygon). (Default:  $n = NPOLY$ , where the set of material properties relates one-to-one to its corresponding polygon)

**Shininess** — An  $n$ -element array of shininess, in the range of 0 to 255.

**SpecularColor** — A (3,  $n$ ) array of RGB color, in the range of 0 to 255. If *SpecularColor* is an  $n$ -element array, then the color is interpreted as grayscale.

**Transparency** — An  $n$ -element array of transparency, in the range of 0 to 255.

**VertexColor** — If set, *MaterialIndices* are per-vertex rather than per-polygon, so that the material properties describe each vertex, rather than describing each polygon.

## Discussion

The `VRML_POLY` procedure creates a VRML node, based on the `PV-WAVE` variables for vertex list and polygon list.

This procedure allows for any polygonal shape to be drawn in a VRML world, with input parameters tailored to fit in with other `PV-WAVE` routines.

## Example

```
z = HANNING(14,8) * 10.  
POLY_SURF, z, vlist, plist  
    ; Build a vertex/polygon set from surface data.  
VRML_OPEN, 'vrml_poly.wrl'  
    ; Open a VRML file.  
VRML_LIGHT, [7, 4, 20]  
VRML_CAMERA, [0, -0, 20], LookAt = [7, 4, 0]  
    ; Set up a light source and an initial viewpoint.
```

```

r = BYTSCL(z(*))
g = BYTSCL(DIST(14, 8))
g = g(*)
b = BYTSCL(INDGEN(14, 8))
b = b(*)
    ; Make up some RGB color values -- one color for each
    ; element of z (thus, one color per polygon).
rgb = TRANSPOSE([[r], [g], [b]])
    ; Put the colors into a (3, NPOLY) array.
VRML_POLY, vlist, plist, EmissiveColor = rgb
    ; Write the surface to the VRML file.
VRML_CLOSE
    ; Close the VRML file.

```

## See Also

[MESH](#), [POLY\\_MERGE](#), [POLY\\_PLOT](#), [POLY\\_SURF](#),  
[POLYSHADE](#), [VRML\\_LINE](#), [VRML\\_OPEN](#), [VRML\\_SURFACE](#)

For a discussion of VRML polygons, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 14.

## ***VRML\_SPHERE Procedure***

Creates a sphere in a VRML world.

### **Usage**

VRML\_SPHERE

### **Input Parameters**

None.

### **Keywords**

#### ***Object Properties***

The following keywords describe or define the VRML sphere object.

**Center** — A 3-element array specifying the center of the sphere object. (Default: [0, 0, 0])

**Radius** — Specifies the sphere radius from the center. (Default: 1.0)

**Transform** — A 4-by-4 matrix containing the transformation to be applied to the object.

### **Material Properties**

The following properties, when set, are applied to the object.

**AmbientColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**DiffuseColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**EmissiveColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**Shininess** — A scalar shininess factor, in the range of 0 to 255.

**SpecularColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**Texture\_Image** — A texture image to apply to the object. The texture image is wrapped completely around object (CLAMPed). Image values and transparencies lie in range of 0 to 255.

There are four options:

(*w*, *h*) or (1, *w*, *h*) — Grayscale image

(2, *w*, *h*) — Grayscale in (0, \*, \*) plus transparency in (1, \*, \*)

(3, *w*, *h*) — True color image (red, green, blue)

(4, *w*, *h*) — True color plus transparency in (3, \*, \*)

**Transparency** — A scalar transparency factor, in the range of 0 to 255.

## **Discussion**

The VRML\_SPHERE procedure supports the sphere node in VRML.

## **Example**

The 'wavelogo.dat' file used in this example is found in the following directory:

(UNIX) <wavedir>/demo/web/vrml  
(OpenVMS) <wavedir>:[DEMO.WEB.VRML]  
(Windows) <wavedir>\demo\web\vrml

where <wavedir> is the main PV-WAVE directory.

```
VRML_OPEN, 'vrml_sphere.wrl'  
RESTORE, 'wavelogo.dat' ; img, r, g, b  
sz = SIZE(img)  
col = TRANSPOSE([[r], [g], [b]])  
texture = REFORM(col(*, img), 3, sz(1), sz(2))  
    ; Create the texture, using the PV-WAVE logo.  
VRML_SPHERE  
    ; Create a base sphere.  
VRML_SPHERE, Center = [-2, -1, 0], Radius = .5, Texture = texture  
    ; Create a textured sphere.  
VRML_SPHERE, Center = [2, 0, 2], Radius = 1.5, $  
    Shininess = 202, Transparency = 170  
    ; Create a somewhat transparent, reflective sphere.  
VRML_CLOSE
```

## See Also

[POLY\\_SPHERE](#), [SPHERE](#), [VRML\\_CUBE](#),  
[VRML\\_CYLINDER](#), [VRML\\_OPEN](#), [VRML\\_TEXT](#)

For a discussion of spheres and other VRML primitives, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 3.

---

## VRML\_SPOTLIGHT Procedure

Creates a VRML spotlight.

### Usage

VRML\_SPOTLIGHT, *position*

### Input Parameters

*position* — A 3-element array specifying the positioning of the spotlight.

## Keywords

**Angle** — Specifies the angle of the light cone in degrees. (Default: 45)

**Color** — A 3-element array of RGB values, in the range of 0 to 255. (Default: [255, 255, 255], the color white)

**Direction** — The orientation of the light cone. (Default: [0, 0, -1])

**Intensity** — A number normalized from 0 to 1, where 1 is full intensity. (Default: 1)

**Rate** — The exponential drop-off rate for light intensity from the axis of the light cone. (Default: 0)

## Discussion

VRML\_SPOTLIGHT implements one of the three types of lighting supported by VRML. (VRML\_LIGHT is another.) A spotlight emanates light inside of a cone shape.

## Example

```
VRML_OPEN, 'vrm1_spotlight.wrl'  
...  
VRML_SPOTLIGHT, [-1, -1, 0], $  
    Color = [255, 0, 0], $  
VRML_CLOSE
```

## See Also

[VRML\\_CAMERA](#), [VRML\\_LIGHT](#), [VRML\\_OPEN](#)

For a discussion of VRML lighting, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 19.

---

## **VRML\_SURFACE Procedure**

Creates a VRML surface plot based on PV-WAVE variables.

### **Usage**

VRML\_SURFACE, z [, x, y]

### **Input Parameters**

*z* — A 2D array containing the values that make up the surface. If *x* and *y* are supplied, the surface is plotted as a function of the *x*, *y* locations. (See the SURFACE and CONTOUR routines for more information.)

*x* — (optional) A 1D or 2D array specifying the *x*-coordinates of the data.

*y* — (optional) A 1D or 2D array specifying the *y*-coordinates of the data.

### **Keywords**

*Light* — A 3-element array of [*x*, *y*, *z*] position for a light source, in normal coordinates (0 to 1 is the normalized range).

*Title* — A string specifying the plot title.

*Transform* — A 4-by-4 matrix containing the transformation to apply to the object.

*XTitle* — A scalar string specifying the *x*-axis title.

*YTitle* — A scalar string specifying the *y*-axis title.

*ZTitle* — A scalar string specifying the *z*-axis title.

### **Discussion**

The VRML\_SURFACE procedure uses PV-WAVE- type variables as input to create a meshed surface in a VRML world.

---

**NOTE** The *x*, *y*, *z* data are scaled into a unit cube, so that the axes for the graphic fall within the range of 0 to 1 for all three axes. Use *Transform* to move the cube around, stretch it, and so on.

---

## Example

```
x = FINDGEN(21) - 10.0
x = x # REPLICATE(1, 21)
y = TRANSPOSE(x)
    ; Create 2D arrays of x- and y-coordinates.
z = x * SIN(y) + y * COS(x)
    ; Evaluate a function of x and y for the surface.
VRML_OPEN, 'vrml_surf.wrl'
    ; Open the VRML file.
VRML_CAMERA, [1, -3, 1], $
    LookAt=[.5, .5, .5]
    ; Set an initial viewpoint.
VRML_SURFACE, z, x, y, ZTitle = 'f(x, y)', $
    Title = 'f(x, y) = x*sin(y) + y*cos(x)'
    ; Write the surface to the VRML file.
VRML_CLOSE
    ; Close the VRML file.
```

## See Also

[CONTOUR](#), [POLY\\_SURF](#), [SHADE\\_SURF](#), [SURFACE](#), [VRML\\_AXIS](#),  
[VRML\\_OPEN](#), [VRML\\_POLY](#)

---

## VRML\_ TEXT Procedure

Creates a VRML text object in an open VRML file.

### Usage

VRML\_TEXT, *text*

### Input Parameters

*text* — A string, or array of strings containing the VRML object text.

## Keywords

### **Object Properties**

The following keywords describe or define the VRML text object.

**Center** — A 3-element array with coordinate position of the text object centered around designated coordinates  $[x, y, z]$ .

**FontSize** — Specifies the point size of the font. (Default: 10)

**Left** — If set, left-justify the text about the *Center*-designated coordinates  $[x, y, z]$ .

---

**NOTE** The *Left* and *Right* keywords are mutually exclusive.

---

**Right** — If set, right-justify the text about the *Center*-designated coordinates  $[x, y, z]$ .

**Rotation** — A 3-element array specifying rotation angles in degrees around  $x$ ,  $y$  and  $z$ . (Default: orientation is horizontal, with text parallel to the  $x$ - $y$  plane, reading from  $-x$  to  $+x$ )

**Transform** — A 4-by-4 matrix containing the transformation to apply to the object. Applied after *Center* and *Rotation*.

### **Material Properties**

The following properties, when set, are applied to the text object.

**AmbientColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**DiffuseColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**EmissiveColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**Shininess** — A scalar shininess factor, in the range of 0 to 255.

**SpecularColor** — A 3-element array of RGB color with each element ranging between 0 and 255. If the value is scalar, then the color is interpreted as grayscale.

**Texture\_Image** — A texture image to apply to the object. The image is wrapped completely around the object. The texture image values and transparency values lie in the range of 0 to 255.

There are four options:

$(w, h)$  or  $(1, w, h)$  — Grayscale image

$(2, w, h)$  — Grayscale in  $(0, *, *)$  plus transparency in  $(1, *, *)$

$(3, w, h)$  — True color image (red, green, blue)

$(4, w, h)$  — True color plus transparency in  $(3, *, *)$

**Transparency** — A scalar transparency factor, in the range of 0 to 255.

### **Font Family Settings**

These three font attributes are mutually exclusive.

**Serif** — Serif font (the default).

**Sans** — Sans-serif font.

**Typewriter** — Monospaced font.

### **Font Attributes**

Either or both of these keywords may be used. If neither is set, then the typeface is “normal.”

**Bold** — Boldface type.

**Italic** — Italic type.

## **Discussion**

The VRML\_TEXT procedure uses the VRML text and font features, which provide for 3-D text shapes.

If the *text* parameter is an array of strings, each string is placed on a separate line in the VRML world. The lines of text objects are centered about the baseline position of the first element string, and the lines are separated based on the point size of the font used.

## **Example**

```
VRML_OPEN, 'vrm1_text.wrl'  
VRML_TEXT, ['Visual Numerics', 'PV-WAVE'], $  
    Center = [0, 5, 0], $  
    EmissiveColor = [0, 0, 255], $  
    /Bold, /Serif, FontSize = 12  
VRML_CLOSE
```

## See Also

[VRML\\_CONE](#), [VRML\\_CUBE](#), [VRML\\_CYLINDER](#), [VRML\\_LINE](#),  
[VRML\\_OPEN](#), [VRML\\_SPHERE](#), [XYOUTS](#)

For a discussion of VRML text and definitions of attributes, see *The VRML Sourcebook*, by Andrea L. Ames, et al., John Wiley & Sons, Inc., 1996, Chapter 4.

---

## ***vtkADDATTRIBUTE Procedure***

Collects point attributes for VTK datasets.

### **Usage**

`vtkADDATTRIBUTE, attributes`

### **Input Parameters**

***attributes*** — A list variable containing all of the attributes for a dataset. Passing an undefined variable on the first call to creates the initial list. Using the variable on subsequent calls will add elements to the list. You should never have to create or modify the contents of this variable manually.

### **Keywords**

***Name*** — A scalar string specifying a name for this attribute. The default name is the attribute name in lower case.

***Lookup\_table\_name*** — Only used with scalar attributes. A scalar string specifying the name of the lookup table to be associated with a scalar attribute. The default table is “default.”

One and only one of the following keywords can be used to add an attribute of the selected type:

***Scalars*** — A vector of floating point numbers containing scalar values for each entry in points.

***Lookup\_table*** — An array of floating point numbers of size  $(3, m)$  containing normalized RGB values.

***Vectors*** — An array of floating point numbers of size  $(3, n)$ , where  $n$  equals the number of *Points*, containing the  $x$ ,  $y$ , and  $z$  components for each vector.

**Normals** — An array of floating point numbers of size  $(3, n)$ , where  $n$  equals the number of points, containing the  $x$ ,  $y$ , and  $z$  components for each normal, where the  $x$ ,  $y$ , and  $z$  values are normalized to a unit length of 1.

**Color\_scalars** — An array of floating point numbers of size  $(m, n)$ , where  $n$  equals the number of *Points* and  $m$  is the number of values per color scalar. Values are between 0.0 and 1.0.

**Texture\_coordinates** — An array of floating point numbers of size  $(m, n)$ , where  $m$  is 1, 2, or 3 and  $n$  equals the number of points.

**Tensors** — An array of floating point numbers of size  $(3, 3, n)$ , where  $n$  equals the number of points.

## Discussion

This procedure allows a set of attributes to be collected and passed to one of the dataset creation routines: `vtkPOLYDATA`, `vtkSTRUCTUREDPOINTS`, `vtkSTRUCTUREDGRID`, `vtkRECTILINEARGRID`, or `vtkUNSTRUCTUREDGRID`. Datasets can have one or more attributes associated with their points, and even more than one attribute of the same type, with the *Name* assigned to the attribute used to distinguish them. For *Scalars*, *Normals*, *Color\_scalars*, *Texture\_coordinates*, and *Tensors*, the number of supplied attributes must equal the number of points in the dataset to which they will be assigned.

---

## ***vtkAXES Procedure***

Creates a set of axes.

### **Usage**

`vtkAXES`

### **Input Parameters**

None.

### **Keywords**

***Charsize*** — A floating point scalar or three-element array, the size of the text for tickmark labels and axes labels. (Default:  $0.4 * Lengths$ )

**Format** — A FORTRAN style format string to use for the tick mark labels. (Default: ' (%10.2) ')

**Name** — Specify a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

**Position** — An array of three floating point numbers in data coordinates describing the origin for the axis. (Default: [0, 0, 0])

**Lengths** — An array of three floating point numbers describing the length of the  $x$ ,  $y$ , and  $z$  axes, respectively, specified in data coordinates. (Default: [1, 1, 1])

**Labels** — If non-zero, any tick marks drawn are labeled with default values.

**LOD** — If nonzero, the tickmarks are created as level-of-detail actors to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

**Sigfig** — An integer, the number of significant figures to use for the tick mark labels. (Default: 2).

**TextColor** — The color to use for text used for [XYZ]Title. See [vtkWINDOW](#) (page 1065) for possible ways to specify the color. (Default: ' white ')

**Tickscale** — A float, a scaling value passed to `vtkSCATTER` for the tick mark glyphs. (Default: 0.33)

**Ticksymbol** — An integer, passed to `vtkSCATTER` to set the glyph to use for the tick marks. (Default: 0, a sphere)

Other keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

[XYZ]Tickname

[XYZ]Ticks

[XYZ]Tickv

[XYZ]Title

## Discussion

This procedure creates three axes displayed as lines in the  $x$ ,  $y$ , and  $z$  direction, with an optional label at the top of each axis.

## Example

```
vtkAxes, lengths = [1,1.5,2]
```

## See Also

[AXIS](#)

---

## ***vtkCAMERA Procedure***

Changes the camera's parameters.

### **Usage**

vtkCAMERA

### **Input Parameters**

None.

### **Keywords**

***Name*** — Specify a name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

***Position*** — An array of three floating point numbers describing the  $x$ ,  $y$ , and  $z$  position for the camera in data coordinates.

***FocalPoint*** — An array of three floating point numbers describing the  $x$ ,  $y$ , and  $z$  position for the camera's focal point in data coordinates.

***ClippingRange*** — An array of two floating point numbers describing the distance from the camera to the front and back clipping planes (in data coordinates).

***ViewUp*** — An array of three floating point numbers describing a vector that represents the up direction for the view.

***Distance*** — A floating point number describing the distance from the focal point to the camera (which will modify the *FocalPoint* value) in data coordinates.

***ViewAngle*** — A floating point number that sets the view angle of the camera in degrees.

***Azimuth*** — A scalar value describing the angle in degrees to rotate the camera about the view up vector centered at the focal point. This moves the camera from side to side.

**Elevation** — A scalar value describing the angle in degrees to rotate the camera about the cross product of the direction of projection and the view up vector centered on the focal point. This moves the camera up and down.

**Roll** — A scalar value describing the angle in degrees to rotate the camera about the direction of projection. This rolls the camera about the direction of projection.

## Discussion

A default camera is created for each `vtkWINDOW` with these properties: position and focal point such that all objects are visible, with the camera centered on the entire scene; view up along the Y axis; view angle set to 30 degrees; and a clipping range set to 0.1, 1000.0.

---

**NOTE** `vtkSURFACE`, `vtkSCATTER` and `vtkPOLYSHADE` change this default and set the up vector to be along the  $z$  axis.

---

## Example

```
pyramid_list = [[0,0,0],[0,1,0],[1,0,0],[1,1,0],[.5,.5,1]]
vertex_list=[3,0,2,4,3,2,3,4,3,3,1,4,3,1,0,4,4,0,1,3,2]
    vtkPolyshade, pyramid_list, vertex_list
    vtkCamera, Azimuth=25, Elevation=45, ViewAngle=120
```

---

## ***vtkCLOSE Procedure***

Closes the VTK process.

### **Usage**

`vtkCLOSE`

### **Input Parameters**

None.

### **Keywords**

None.

## Discussion

This procedure closes all VTK windows and shuts down the Tcl/Tk-spawned process. It should be called before exiting PV-WAVE.

## Example

A standard close call.

```
vtkCLOSE
```

## See Also

[vtkINIT](#), [vtkWINDOW](#), [vtkERASE](#), [vtkWDELETE](#)

---

## ***vtkCOLORBAR Procedure***

Adds a color bar legend to a VTK scene using the current PV-WAVE color table.

## Usage

```
vtkCOLORBAR
```

## Keywords

***Vertical*** — If set, the color bar is aligned vertically. Default alignment is horizontal.

***Title*** — The title of the legend. (Default: none)

***Position*** — A three-element array, the position of the lower left corner of the color bar. (Default: [0,0,0])

***NumLabels*** — The number of labels to draw. (Default: 5)

***Width*** — The width of the legend in device coordinates. (Default: 0.8, or 0.15 with */Vertical*)

***Height*** — The height of the legend in device coordinates. (Default: 0.15, or 0.9 with */Vertical*)

***CRange*** — A two-element vector, the range of colors (Default: [0,255])

***LRange*** — A two-element vector, the label range (Default: *CRange*)

*Sigfig* — An integer, the number of significant figures to use for the labels.  
(Default: 3).

*Name* — Specifies a name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

*NoShadow* — If set, labels are drawn without shadows.

---

## ***vtkCOMMAND Procedure***

Sends Tcl and VTK commands to the Tcl process.

### **Usage**

`vtkCOMMAND, command`

### **Input Parameters**

*command* — A string representing the VTK command to invoke.

### **Keywords**

*Result* — A string or string array containing any results from the execution of the command in the Tcl shell.

### **Discussion**

The Basic interface to send raw Tcl or VTK commands to the spawned Tcl process.

### **Example**

Use `vtkCOMMAND` to set the background blue.

```
vtkwindow, 1
vtkcommand, 'renderer1 SetBackground 0.0 0.0 1.0'
vtkrenderwindow
```

### **See Also**

[vtkRENDERWINDOW](#)

---

## ***vtkERASE Procedure***

Erases the contents of the current VTK window.

### **Usage**

```
vtkERASE [, background_color]
```

### **Input Parameters**

*background\_color* — (optional) The background color to be used for the window, specified as a 24-bit color. See [vtkWINDOW](#) (page 1065) for possible ways to specify the color.

### **Keywords**

None.

### **Discussion**

This procedure works like ERASE for PV-WAVE windows. It removes all actors, cameras, and lights from the current window.

### **Example**

This example shows vtkERASE removing the axes from the window.

```
vtkwindow, 1  
vtkaxes  
vtkerase
```

### **See Also**

[ERASE](#), [VtkCLOSE](#), [vtkWINDOW](#)

---

## ***vtkGRID Procedure***

Adds 3D grid lines to a VTK scene.

### **Usage**

`vtkGRID [, Number=n]`

### **Keywords**

***Number*** — A scalar or a three element array, the number of segments with grid lines in each (*x*, *y*, *z*) direction. (Default: [1,1,1], a box)

***Lengths*** — A scalar or a three-element array, the extent of the grid. (Default: [1,1,1])

***Position*** — A three-element array, the position of the origin of the grid. (Default: [0,0,0])

***Color*** — The color to use for the polylines (passed to `vtkPLOTS`). See `vtkWINDOW` for possible ways to specify the color. (Default: 'white')

***Thick*** — A float, the thickness of the grid lines (passed to `vtkPLOTS`). (Default: 1.0)

***Name*** — A string, the name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

***UseAxes*** — If nonzero, the most recently created `vtkAXES` scale is used to define the `Lengths` array, which then does not need to be defined explicitly.

***LOD*** — If nonzero, use level-of-detail actors for the grid lines (passed to `vtkPLOTS`).

### **Example**

```
vtkSURFACE, DIST(20)
```

```
vtkGRID, /useAxes, Thick=6
```

```
vtkGRID, Number=[2,2,6], Color='red', Thick=4, /useAxes
```

---

## ***vtkHEDGEHOG Procedure***

Creates a HedgeHog (vector) plot.

### **Usage**

*vtkHEDGEHOG*, *points*, *vectors*, *scalars*

### **Input Parameters**

*points* — A 3, *n* array of points (location of the lines).

*vectors* — A 3, *n* array of vectors (orientation and length of the lines).

*scalars* — (optional) An *n*-element array of scalars (colors of the lines).

### **Keywords**

*Scalefactor* — A float, a scaling factor to control the size of the oriented lines of the HedgeHog object (Default: 1.0).

*SRange* — A two-element integer array, the scalar range (Default: [0,255]).

*LOD* — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

*NoRotate* — Does not perform any camera rotations. Used when a previous call to *vtkSURFACE*, *vtkSCATTER* or *vtkPOLYSHADE* has already set the camera angle.

*NoAxes* — If set, no axes are created.

*NoErase* — If nonzero, prevents the window from being erased to the background color before drawing the new scene. If not set, then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

These keywords are passed to *vtkAXES*:

Charsize	TextColor	[XYZ]Title	[XYZ]Ticks
[XYZ]Tickv	[XYZ]Tickn	TickScale	TickSymbol
Labels	Sigfig	Format	

See [vtkAXES](#) for descriptions.

## Example

This example plots a Hanning surface and its normals.

```
LOADCT, 2
n = 50
x = REBIN(INDGEN(n), n, n)
y = TRANSPOSE(x)
z = (n/2) * (HANNING(n, n))
norm = NORMALS(JACOBIAN(LIST(x, y, z)))
;
points = FLTARR(3, n*n, /NoZero)
points(0, *) = x(*)
points(1, *) = y(*)
points(2, *) = z(*)
vectors = FLTARR(3, n*n, /NoZero)
vectors(0, *) = (norm(0)) (*) * SQRT(z(*))
vectors(1, *) = (norm(1)) (*) * SQRT(z(*))
vectors(2, *) = (norm(2)) (*) * SQRT(z(*))
scalars = z(*)
;
vtkHedgeHog, points, vectors, scalars, /NoAxes, scalef=0.75
vtkSurface, REFORM(points(2, *), n, n), $
  Shades=REBIN([0.8, 0.8, 0.8, .75], 4, n, n), $
  /NoErase, /NoRotate, /NoAxes
```

## See Also

[vtkAXES](#)

---

## ***vtkINIT Procedure***

Initializes the VTK system.

### **Usage**

vtkINIT

### **Input Parameters**

None.

### **Keywords**

***File*** — If set, a temporary file is used to communicate data sets to VTK instead of a socket connection. For very large data sets with many floating-point values, this method is considerably faster; however, read/write permissions are required. If set to one (*/File*), any data set greater than 1024 bytes is written to file. If set to a value (*File=fbytes*), data sets larger than *fbytes* are written to file; smaller data sets are sent via sockets. Setting */File* is equivalent to setting *File=1024*. This keyword affects only data: commands are always sent by the socket connection.

***Noshell*** — If set, the keyword is passed along to the SPAWN procedure that initiates the VTK Tcl process. This keyword is required when calling VTK routines from a JWAVE wrapper and should not be used otherwise.

***Path*** — Used in conjunction with the *File* keyword, a string indicating the file path to the directory where the temporary file(s) are to be created.

***Print*** — If present and nonzero, causes the output from the spawned Tcl/Tk shell to be sent back to PV-WAVE and displayed in the console. This keyword is useful for debugging low-level VTK calls.

***Timeout*** — A floating point scalar specifying a time interval in seconds which vtkINIT will wait before giving up on establishing a socket connection to the spawned Tcl shell. (Default: 20)

### **Discussion**

This procedure must be performed before any other VTK commands. It causes a Tcl/Tk shell to be spawned and sets up communication with it. It also initialized various internal VTK parameters. The following routines will automatically call

vtkINIT if it has not already been called: vtkWINDOW, vtkPOLYSHADE, vtkSURFACE, and vtkSCATTER.

## Example

This example uses vtkINIT to initialize VTK with a timeout of 10 seconds.

```
vtkINIT, timeout=10
```

## See Also

[VtkCLOSE](#)

---

## vtkLIGHT Procedure

Adds a light to a VTK window.

## Usage

```
vtkLIGHT
```

## Input Parameters

None.

## Keywords

**Name** — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

**Color** — The color for the light. See [vtkWINDOW](#) (page 1065) for possible ways to specify the color. (Default: ' white ' )

**Position** — An array of three floating point numbers describing the  $x$ ,  $y$ , and  $z$  position for the light. The default behavior is to have the light follow the camera position.

**FocalPoint** — An array of three floating point numbers describing the  $x$ ,  $y$ , and  $z$  position for the light's focal point.

**Intensity** — A float value between 0.0 and 1.0 specifying the intensity of the light.

***DirectionAngle*** — An array of two floating point numbers that set the position and focal point of a light based on elevation and azimuth. The light is moved to shine from the given angle. Angles are given in degrees.

## Discussion

A white light, which follows the camera position, is created by default for a VTK Window.

---

**NOTE** Other light parameters not supported in this wrapper can be set using the assigned *Name* and `vtkCOMMAND`.

---

## Example

```
pyramid_list = [[0,0,0],[0,1,0],[1,0,0],[1,1,0],[.5,.5,1]]
vertex_list=[3,0,2,4,3,2,3,4,3,3,1,4,3,1,0,4,4,0,1,3,2]
vtkPolyshade, pyramid_list, vertex_list
    vtkLight, Color='0000FF'XL, Position=[10,10,10]
    vtkLight, Color='00FF00'XL, Position=[10,10,-10]
    vtkLight, Color='FF0000'XL, Position=[-10,-10,-10]
```

---

## ***vtkPLOTS Procedure***

Adds a polyline.

### Usage

`vtkPLOTS`, *points*

### Input Parameters

*points* — An array of floating point numbers of size  $(3, n)$  where  $n$  is the number of points. `points(0, *)` is taken as an  $x$  value, `points(1, *)` is taken as a  $y$  value, and `points(2, *)` is taken as a  $z$  value.

## Keywords

**Name** — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

**Color** — The color to use for the polyline. See [vtkWINDOW](#) (page 1065) for possible ways to specify the color. (Default: 'white')

**Thick** — A float describing the thickness of the polyline. The default is 1.0, which is scaled to correspond to a radius of 0.001 in data coordinates.

**LOD** — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

**Nolines** — If nonzero, causes a cloud of points to be displayed rather than a polyline.

## Discussion

This procedure is similar to the `PLOTS` procedure for `PV-WAVE` windows. Polylines are drawn as connected cylinders.

## Example

```
z = findgen(1000)/999
x = z*sin(50*z)
y = z*cos(50*z)
vtkplots, transpose([[x],[y],[z]]), thick=5, color='blue'
vtkwindow,2
vtkplots, transpose([[x],[y],[z]]), color='red',/nolines
```

## See Also

[PLOTS](#), [vtkSCATTER](#)

---

## ***vtkPOLYDATA Procedure***

Passes vertex/polygon lists, lines, points, and triangles to VTK.

### **Usage**

`vtkPOLYDATA`, *points*

### **Input Parameters**

*points* — A two-dimensional array of floating point numbers of size  $(3, n)$  describing  $x$ ,  $y$ , and  $z$  points.

### **Keywords**

*Restore* — An associative array containing all of the data and attributes for a poly-data dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

*Name* — Specifies a name to be used to create this data source. This name can be used in calls to `vtkCOMMAND`.

*Filename* — File to store data using standard VTK ASCII format.

*Save* — Returns the data in the specified variable stored in an associative array. Data is *not* sent to VTK if this parameter is specified.

*Polygons* — A vector of integers describing polygons, organized as vertex count followed by indices into *points*, repeated for all polygons. This is the same format as a polygon list used in [POLYSHADE](#).

*Vertices* — A vector of integers describing vertices, organized as vertex count followed by indices into *points*, repeated for all vertices.

*Lines* — A vector of integers describing polylines, organized as vertex count followed by indices into *points*, repeated for all polylines.

*Triangle\_Strips* — A vector of integers describing triangle strips, organized as vertex count followed by indices into *points*, repeated for all triangle strips.

*Attributes* — A list created using `vtkADDATTRIBUTE` containing one or more attributes associated with the points in the dataset.

## Discussion

Contains points and polygons (like the polygon vertex list in PV-WAVE) as well as vertices, lines, and triangle strips. See the VTK documentation, which can be downloaded from <http://public.kitware.com>, for more details on the data and attributes for the PolyData dataset format.

---

## vtkPOLYSHADE Procedure

Renders a polygon object.

### Usage

vtkPOLYSHADE, *vertices*, *polygons*

### Input Parameters

**Vertices** — A (3,  $n$ ) array containing the  $x$ -,  $y$ -, and  $z$ -coordinates of each vertex in data coordinates.

**Polygons** — An integer or longword array containing the indices of the vertices of each polygon. The vertices of each polygon should be listed either clockwise or counterclockwise order when observed from outside the surface. The vertex description of each polygon is a vector of the form [ $n$ ,  $i_0$ ,  $i_1$ , ... ,  $i_{n-1}$ ], and the array polygons is the concatenation of the lists of each polygon.

### Keywords

**Name** — Specify a name to be used to create this object. If an undefined variable is used or no name specified then a random name is used. This name can be used in calls to vtkCOMMAND to modify this object.

**Color** — An array expression, of the same dimensions as the number of vertices passes (the value “ $n$ ” above), containing the color index at each vertex. Alternately an expression describing one color to be used for the entire polygonal surface or wireframe. If this keyword is omitted, a white surface is displayed.

To specify a single color, see [vtkWINDOW](#) (page 1065) for possible ways to specify the color. If a two-dimensional array of colors is specified (for an image overlay) the shades variable can be in any of these formats:

FIX( $n$ )	A one-dimensional array of short integers or bytes specifying an index into the current PV-WAVE color table for each point. The RGB color for each point is obtained from the corresponding entry in the current PV-WAVE color table.
------------	---

LONG( $n$ )	A one-dimensional array of long integers specifying the 24-bit color at each point.
FLOAT(3, $n$ )	A floating point array of size (3, $n$ ) containing the normalized values specifying the red, green, and blue components of the color at each point.
FLOAT(4, $n$ )	A floating point array of size (4, $n$ ) containing the normalized values specifying the red, green, blue, and alpha components of the color at each vertex. The alpha component is the transparency where 0.0 is completely transparent and 1.0 is opaque.

**Wireframe** — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

**LOD** — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

**NoAxes** — If set, no axes are created.

**NoRotate** — Does not perform any camera rotations. Used when a previous call to vtkSURFACE, vtkSCATTER or vtkPOLYSHADE has already set the camera angle.

**NoErase** — If nonzero prevents the window from being erased to the background color before drawing the new scene. If not set then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

These keywords are passed to vtkAXES:

[XYZ]Ticks	[XYZ]Tickv	[XYZ]Tickn	TickScale
TickSymbol	Labels	Sigfig	Format

Other keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

[Ax](#)

[Az](#)

## Discussion

This procedure is similar to the POLYSHADE procedure for PV-WAVE windows. Wireframes can be produced as well as surfaces shaded in one color or overlaid with an image. Transparency is also supported.

## Example

```
pyramid_list = [[0,0,0],[0,1,0],[1,0,0],[1,1,0],[.5,.5,1]]  
vertex_list=[3,0,2,4,3,2,3,4,3,3,1,4,3,1,0,4,4,0,1,3,2]  
vtkPolyshade, pyramid_list, vertex_list, color='blue'
```

## See Also

[AXIS](#), [POLYSHADE](#)

---

## ***vtkPPMREAD Function***

Reads a PPM file.

### Usage

```
image = vtkPPMREAD (filename)
```

### Input Parameters

*filename* — File path of PPM file.

### Returned Value

*image* — An array (3, *width*, *height*) of bytes containing the 24-bit image.

### Keywords

None.

### Discussion

This function is used to read the rudimentary PPM binary files created by VTK and containing images stored as RGB values. The images can be displayed with TV, `image, True=1` or converted to an 8-bit image using `ipcolor_24_8`.

## Example

Example of reading the file `wave.ppm` and storing an image.

```
Image=vtkppmread('wave.ppm')
```

## See Also

[vtkPPMWRITE](#), [vtkTVRD](#)

---

## ***vtkPPMWRITE Procedure***

Writes the contents of a VTK window to a PPM file.

### **Usage**

```
vtkPPMWRITE [, window_index]
```

### **Input Parameters**

*window\_index* — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

### **Keywords**

*Filename* — File path to store PPM file. (Default: 'wave.ppm')

### **Discussion**

This procedure saves a snapshot of the selected window as a PPM file. It is important to make sure the VTK window fully visible when this routine is called because an obscuring portion of another window will be captured as part of the image. This is a limitation of VTK.

### **Example 1**

Writing a PPM file.

```
vtkwindow, 1  
vtkaxes  
vtkPPMWRITE, 1
```

### **Example 2**

Writing the PPM file to the file name of PV.ppm.

```
vtkwindow, 2, background='blue'
```

```
vtkPPMWRITE, 2, filename='PV.ppm'
```

## See Also

[vtkPPMREAD](#), [vtkTVRD](#)

---

## ***vtkRECTILINEARGRID Procedure***

Passes data describing a rectilinear grid to VTK.

### **Usage**

`vtkRECTILINEARGRID, Dimensions`

### **Input Parameters**

***Dimensions*** — A 3-element vector of integers describing dimensions in *x*, *y*, and *z*. Use 1 for the third dimension if only a two-dimensional array is described.

### **Keywords**

***Restore*** — An associative array containing all of the data and attributes for a poly-data dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

***Name*** — Specifies a name to be used to create this data source. This name can be used in calls to `vtkCOMMAND`.

***Filename*** — File to store data using standard VTK ASCII format.

***Save*** — Returns the data in the specified variable stored in an associative array. Data is *not* sent to VTK if this parameter is specified.

***X\_coordinates*** — A vector of floating point numbers of the same length as the first dimension in *Dimensions* describing monotonically increasing coordinate values. Increasing integers starting with 0 are used as a default.

***Y\_coordinates*** — A vector of floating point numbers of the same length as the second dimension in *Dimensions* describing monotonically increasing coordinate values. Increasing integers starting with 0 are used as a default.

*Z\_coordinates* — A vector of floating point numbers of the same length as the third dimension in *Dimensions* describing monotonically increasing coordinate values. Increasing integers starting with 0 are used as a default.

*Attributes* — A list created using `vtkADDATTRIBUTE` containing one or more attributes associated with the points in the dataset.

## Discussion

This procedure creates a dataset with a regular topology and semiregular geometry aligned along the *x*, *y*, and *z* axes. See the VTK documentation, which can be downloaded from <http://public.kitware.com>, for more details on the data and attributes for the RectilinearGrid dataset format.

---

## ***vtkRENDERWINDOW Procedure***

Renders a VTK window.

### Usage

`vtkRENDERWINDOW` [, *window\_index*]

### Input Parameters

*window\_index* — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

### Keywords

None.

### Discussion

Call this procedure after all objects (lights, cameras, surfaces, polygon meshes, etc.) have been added to the window. This routine starts the rendering process and creates the initial rendered scene. You need to call this procedure only if you used the *Norender* keyword with `vtkWINDOW`, or if you are making low-level calls to VTK using `vtkCOMMAND`.

## Example 1

This example shows the essence of `vtkRENDERWINDOW`.

```
vtkwindow, 1, /norender
vtkaxes
vtkrenderwindow
```

## Example 2

A more complicated example:

```
vtkwindow, 2, /norender
V=[[0,0,0],[1,0,0],[1,1,0],[0,1,0]]
p=[4,0,1,2,3]
vtkpolyshade, v, p
vtkaxes
vtktext, 'This is VTK', charsize=[1.0,1.0,1.0], color='blue'
vtkrenderwindow, 2
```

## See Also

[vtkWINDOW](#), [vtkCOMMAND](#)

---

## ***vtkSCATTER Procedure***

Renders 3D points.

### **Usage**

`vtkSCATTER`, *points*

### **Input Parameters**

*points* — A float array of size  $(3, n)$  describing  $x$ ,  $y$ , and  $z$  points in data coordinates.

## Keywords

**Name** — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

**Symbol** — A scalar integer describing the type of marker or glyph to be displayed for each point. Supported values are 0-4 where:

0 = Sphere

1 = Cube

2 = Cone

3 = Cylinder

4 = Earth

**Color** — An array expression of the same dimensions as the number of points (the value  $n$  above), containing the color index at each point. Alternately an expression describing one color to be used for all points. If this keyword is omitted, white points are displayed.

To specify a single color, see [vtkWINDOW](#) (page 1065). If a vector of colors is specified, the color variable can be in any of these formats:

<code>FIX(<math>n</math>)</code>	A one-dimensional array of short integers or bytes specifying an index into the current PV-WAVE color table for each point. The RGB color for each point is obtained from the corresponding entry in the current PV-WAVE color table.
<code>LONG(<math>n</math>)</code>	A one-dimensional array of long integers specifying the 24-bit color at each point.
<code>FLOAT(3, <math>n</math>)</code>	A floating point array of size (3, $n$ ) containing the normalized values specifying the red, green, and blue components of the color at each point.

**Scale** — A float value specifying the scaling factor for the size of each glyph. (Default: 1.0)

**LOD** — If nonzero, the glyphs are created as level-of-detail actors to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.



```
vtkSCATTER, data, Color=c, Xtitle='X Values', Ytitle='Y $  
Values', $  
Ztitle='Z Values', TextColor=16, ax=10, az=100, Scale=.6  
; A better look at the data.
```

## See Also

[AXIS](#) [vtkPLOTS](#)

---

## ***vtkSLICEVOL Procedure***

Creates a sliced 3D volume at specific  $x$ ,  $y$ ,  $z$  locations.

### Usage

`vtkSLICEVOL, v, [sx=sx, sy=sy, sz=sz, xc=xc, yc=yc, zc=zc]`

### Input Parameters

$v$  — A 3D array, the volume to slice.

### Keywords

$sx$  — A 1D array, the  $x$  coordinate(s) at which to slice the volume.

$sy$  — A 1D array, the  $y$  coordinate(s) at which to slice the volume.

$sz$  — A 1D array, the  $z$  coordinate(s) at which to slice the volume.

$xc$  — A 1D array with the same number of elements as the first dimension of  $v$ , the  $x$  coordinates of the volume.

$yc$  — A 1D array with the same number of elements as the second dimension of  $v$ , the  $y$  coordinates of the volume.

$zc$  — A 1D array with the same number of elements as the third dimension of  $v$ , the  $z$  coordinates of the volume.

*Interp* — If set, the shading is interpolated (passed to RESAMP).

*Dim* — An integer, the number of vertices on each side of each plane. (Default: 25)

**Name** — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

**Wireframe** — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

**LOD** — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

**NoRotate** — Does not perform any camera rotations. Used when a previous call to `vtkSURFACE`, `vtkSCATTER` or `vtkPOLYSHADE` has already set the camera angle.

**NoErase** — If nonzero, prevents the window from being erased to the background color before drawing the new scene. If not set, then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

Other keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

[Ax](#)

[Az](#)

## Discussion

If no slices are requested through the `sx`, `sy`, and `sz` keywords, the volume is sliced at the midpoints of each index. If `xc`, `yc`, or `zc` are not provided, indices into `v` are used.

## Example

```
x = genvect(-5,5,.25)
y = genvect(-4,4,.2)
z = genvect(-3,3,.15)
v = sqrt(TENSOR_ADD(TENSOR_ADD(x^2,0.3*y^2),1.5*z^2))
vtkSliceVol, v, sx=[-4,.4,2.6], sy=-.15, sz=[-3,1], $
  xc=x, yc=y, zc=z, dim=15
```

## See Also

[SLICE](#)

---

## ***vtkSTRUCTUREDGRID Procedure***

Passes data describing a structured grid to VTK.

### **Usage**

`vtkSTRUCTUREDGRID`, *dimensions*, *points*

### **Input Parameters**

*dimensions* — A 3-element vector of integers describing dimensions in  $x$ ,  $y$ , and  $z$ . Use “1” for the third dimension if only a two-dimensional array is described.

*points* — A two-dimensional array of floating point numbers of size  $(3, n)$  describing  $x$ ,  $y$ , and  $z$  points.

### **Keywords**

*Restore* — An associative array containing all of the data and attributes for a poly-data dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

*Name* — Specifies a name to be used to create this data source. This name can be used in calls to `vtkCOMMAND`.

*Filename* — File to store data using standard VTK ASCII format.

*Save* — Returns the data in the specified variable stored in an associative array. Data is *not* sent to VTK if this parameter is specified.

*Attributes* — A list created using `vtkADDATTRIBUTE` containing one or more attributes associated with the points in the dataset.

### **Discussion**

1, 2, or 3D point data on a topological grid, where the actual points are specified as  $x$ ,  $y$ , and  $z$  values in Cartesian coordinates. See the VTK documentation, which can be downloaded from <http://public.kitware.com>, for more details on the data and attributes for the StructuredGrid dataset format.

---

## ***vtkSTRUCTUREDPOINTS Procedure***

Passes data describing structured points to VTK.

### **Usage**

`vtkSTRUCTUREDPOINTS`, *dimensions*

### **Input Parameters**

*dimensions* — A 3-element vector of integers describing dimensions in *x*, *y*, and *z*. Use 1 for the third dimension if only a two-dimensional array is described.

### **Keywords**

*Restore* — An associative array containing all of the data and attributes for a poly-data dataset, usually created using the *Save* keyword. If this parameter is passed then only the keywords *Name* and *Filename* can be used.

*Name* — Specifies a name to be used to create this data source. This name can be used in calls to `vtkCOMMAND`.

*Filename* — File to store data using standard VTK ASCII format.

*Save* — Returns the data in the specified variable stored in an associative array. Data is NOT sent to VTK if this parameter is specified.

*Origin* — A 3-element vector of floating point numbers containing the *x*, *y*, and *z* origin point for the data.

*Spacing* — A 3-element vector of floating point numbers containing the spacing (width, height, length) of the cubical cells that compose the data set.

*Attributes* — A list created using `vtkADDATTRIBUTE` containing one or more attributes associated with the points in the dataset.

### **Discussion**

Definition of a 1, 2, or 3D arrays (describing lines, grids and voxels), their origin, and spacing. See the VTK documentation, which can be downloaded from <http://public.kitware.com>, for more details on the data and attributes for the Structured-Points dataset format.

---

## ***vtkSURFACE Procedure***

Renders a surface.

### **Usage**

`vtkSURFACE, z [,x] [,y]`

### **Input Parameters**

**z** — A two-dimensional array containing the values that describe the surface. If *x* and *y* are supplied, the surface is plotted as a function of the *x* and *y* locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of *z*.

**x** — (optional) A vector or two-dimensional array specifying the *x*-coordinates for the surface.

If *x* is a vector, each element of *x* specifies the *x*-coordinate for a column of *z*. For example, *x*(0) specifies the *x*-coordinate for *z*(0, \*).

If *x* is a two-dimensional array, each element of *x* specifies the *x*-coordinate of the corresponding point in *z* (*xij* specifies the *x*-coordinate for *zij*).

**y** — (optional) A vector or two-dimensional array specifying the *y*-coordinates for the surface.

If *y* is a vector, each element of *y* specifies the *y* coordinate for a row of *z*. For example, *y*(0) specifies the *y*-coordinate for *z* (\*, 0).

If *y* is a two-dimensional array, each element of *y* specifies the *y*-coordinate of the corresponding point in *z* (*yij* specifies the *y*-coordinate for *zij*).

### **Keywords**

**Name** — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

**Shades** — An array expression of the same dimensions as *z*, containing the color index at each point. Alternately, an expression describing one color to be used for the entire surface or wireframe. If this keyword is omitted, a white surface is displayed.

To specify a single color, see [vtkWINDOW](#) (page 1065). If a two-dimensional array of colors is specified (for an image overlay), the shades variable can be in any of these formats:

**Fix(x, y)** — A two-dimensional array of short integers or bytes specifying an index into the current PV-WAVE color table for each point. The RGB color for each point is obtained from the corresponding entry in the current PV-WAVE color table.

**Long(x, y)** — A two-dimensional array of long integers specifying the 24-bit color.

**Float(3, x, y)** — A floating point array of size (3, x, y) containing the normalized values specifying the red, green, and blue components of the color at each point.

**Float(4,x)** — A floating point array of size (4, x, y) containing the normalized values specifying the red, green, blue, and alpha components of the color at each point. The alpha component is the transparency where 0.0 is completely transparent and 1.0 is opaque.

**Wireframe** — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

**LOD** — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

**NoRotate** — Does not perform any camera rotations. Used when a previous call to `vtkSURFACE`, `vtkSCATTER` or `vtkPOLYSHADE` has already set the camera angle.

**NoAxes** — If present and non zero, then no *x*, *y*, or *z* axes will be drawn.

**NoErase** — If nonzero, prevents the window from being erased to the background color before drawing the new scene. If not set, then all lights, cameras, and objects are removed from the scene before objects for the new scene are added.

**TextColor** — The color to use for text used for the axes titles. See [vtkWINDOW](#) for possible ways to specify the color. (Default: 'white')

These keywords are passed to [vtkAXES](#):

[XYZ]Ticks	[XYZ]Tickv	[XYZ]Tickn	TickScale
TickSymbol	Labels	Sigfig	Format

Other keywords are listed below. For a description of each keyword, see [Chapter 3, Graphics and Plotting Keywords](#).

[Ax](#)

[Az](#)

[Charsize](#)

[\[XYZ\]Title\[](#)

[\[XYZ\]Range](#)

## Discussion

This procedure is similar to the SURFACE and SHADE\_SURF procedures for PV-WAVE windows. Wireframes can be produced as well as surfaces shaded in one color or overlaid with an image. Transparency is also supported.

## Example

This example demonstrates a surface created in a regular PV-WAVE window compared to one using vtkSURFACE.

```
pikes=FLTARR(60,40)
s=DC_READ_FREE(!Data_Dir+'pikeselev.dat',pikes)
    ; Read in the data values for elevation.
snow=FLTARR(60,40)
s=DC_READ_FREE(!Data_Dir+'snowpack.dat',snow)
    ; Read in the data for snowpack
loadct,5
    ; Load a color table.
surface,pikes
    ; Create a wiremesh surface using a regular PV-WAVE window.
vtksurface,pikes/250,/wireframe
    ; Create a wiremesh surface using the VTK toolkit. Notice that the toolkit doesn't scale
    ; the data for you so in order to make sense out of the resulting graphic you need to
    ; scale the data yourself, in this example it was done by dividing by 250.
shade_surf,pikes,shades=bytsc1(snow)
    ; Create a shaded surface using a regular PV-WAVE window.
vtksurface,pikes/250,shades=bytsc1(snow)
    ; Create a shaded surface using the VTK toolkit. As above, the user does the scaling
    ; of the data.
```

## See Also

[AXIS](#), [vtkAXES](#)

---

## ***vtkSURFGEN Procedure***

Generates a 3D surface from sampled points assumed to lie on a surface.

### **Usage**

`vtkSURFGEN`, *points*

### **Input Parameters**

*points* — A 3,  $n$  array of points that lie on a surface.

### **Keywords**

***Reverse*** — By default, the normals of the computed surface are inward facing. If outward normals are required, set this keyword.

***Neighbors*** — An integer, the number of neighbors each points has. Use a larger value if the spread of points is not even. (Default: 20)

***Spacing*** — A float, the spacing of the 3D sampling grid. If not set, the VTK class makes a reasonable guess.

***Filename*** — An ASCII VTK file to create containing the dataset generated by the VTK filter.

***Data*** — (Output). A returned associative array containing two keys: `data("POINTS")` is a 3,  $n$  float array containing the points generated by the VTK filter and `data("VERTICES")` is an  $n+1$  element long array containing topology information for the generated dataset. A filename must be defined to have data returned by this keyword.

***Name*** — A string, the name to be used to create this object. If an undefined variable is used or no name is specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

***Color*** — The color to use for the generated surface. See `vtkWINDOW` for possible ways to specify the color. (Default: 'white')

***Wireframe*** — When present and nonzero, a wire-frame mesh is drawn rather than a shaded surface.

***LOD*** — If nonzero, a level-of-detail actor is created to aid in keeping a high frame-rate during frequent render requests due to user mouse interaction. If set to a value greater than 1, the number of points to use in the random cloud.

This routine also accepts these keywords to control the initial camera and the axes (see [vtkSURFACE](#) and [vtkAXES](#)):

[XYZ]Range	Ax	Az	NoRotate
NoAxes	NoErase	Charsize	TextColor
[XYZ]Title	[XYZ]Ticks	[XYZ]Tickv	[XYZ]Tickn
Tickscale	Ticksymbol	Labels	Format

Sigfig

## Example

```
@math_startup
s = TRANSPOSE(random(100, /Sphere, Parameter=3))
vtkSURFGEN, s, /NoAxes
vtkSCATTER, s, /NoAxes, Symb=1, Color='red', Scale=0.5, $
/NoRotate, /NoErase
```

## See Also

[vtkAXES](#), [vtkSURFACE](#)

---

## ***vtkTEXT Procedure***

Adds a text string.

### **Usage**

vtkTEXT, *string*

### **Input Parameters**

*string* — The scalar string containing the text that is to be output to the display surface. If not of string type, it is converted prior to use.

## Keywords

**Name** — Specifies a name to be used to create this object. If an undefined variable is used or no name specified, then a random name is used. This name can be used in calls to `vtkCOMMAND` to modify this object.

**Position** — An array of three floating point numbers specifying the  $x$ ,  $y$ , and  $z$  position for the beginning of text. (Default: [0,0,0])

**Color** — The color to use for text. See [vtkWINDOW](#) (page 1065) for possible ways to specify the color. (Default: 'white')

**Follow** — If nonzero, forces the text to always be facing the camera.

**Orientation** — An array of three floating point numbers specifying the  $x$ ,  $y$ , and  $z$  rotations of text in degrees. The actual rotations are performed in this order:  $z$  then  $x$  and finally  $y$ . This keyword has no effect if *Follow* is specified.

Keyword [Charsize](#) is also supported. For a description, see [Chapter 3, Graphics and Plotting Keywords](#).

## Discussion

This procedure is similar to the XYOUTS procedure for PV-WAVE windows.

## Example

```
vtkText, "This is vtkText", color="red", charsize=10
```

---

## ***vtkTVRD Function***

Returns the contents of a VTK window as a bitmapped image.

## Usage

```
image = vtkTVRD([window_index])
```

## Input Parameters

***window\_index*** — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

## Keywords

*Filename* — File path to store a temporary PPM file. Default is “wave.ppm.”

## Returned Value

*image* — An array (3, *width*, *height*) of bytes containing the 24-bit image.

## Discussion

This function works like TVRD for PV-WAVE windows. It uses vtkPPMWRITE and vtkPPMREAD to save and then read the contents of a window. The temporary file created is deleted when done.

## Example

```
vtkWINDOW, 7
vtkAXES
tv, vtkTVRD(7), /TRUE
```

## See Also

[TVRD](#), [vtkPPMWRITE](#), [vtkPPMREAD](#)

---

## ***vtkUNSTRUCTUREDGRID Procedure***

Passes data describing an unstructured grid to VTK.

### Usage

vtkUNSTRUCTUREDGRID, *points*, *cells*, *cell\_types*

### Input Parameters

*points* — A two-dimensional array of floating point numbers of size (3, *n*) describing *x*, *y*, and *z* points.

*cells* — A Vector of integers describing cells, organized as vertex count followed by indices into Points, repeated for all cells.

*Cell\_types* — A Vector of integers describing the cell type for each cell. Valid types are values between 1-12.

## Keywords

*Restore* — An associative array containing all of the data and attributes for a poly-data dataset, usually created using the *Save* keyword. If this parameter is passed, then only the keywords *Name* and *Filename* can be used.

*Name* — Specifies a name to be used to create this data source. This name can be used in calls to `vtkCOMMAND`.

*Filename* — File to store data using standard VTK ASCII format.

*Save* — Returns the data in the specified variable stored in an associative array. Data is NOT sent to VTK if this parameter is specified.

*Attributes* — A list created using `vtkADDATTRIBUTE` containing one or more attributes associated with the points in the dataset.

## Discussion

Arbitrary combinations of twelve (12) cell types, ranging from points, lines, polygons to voxels. See the VTK documentation, which can be downloaded from <http://public.kitware.com>, for more details on the data and attributes for the `UnStructuredGrid` dataset format.

---

## ***vtkWDELETE Procedure***

Closes a VTK window without shutting down the Tcl process.

### Usage

`vtkWDELETE` [, *window\_index*]

### Input Parameters

*window\_index* — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

## Keywords

*All* — If nonzero, causes all VTK windows to be closed.

## Discussion

This procedure works like `WDELETE` for `PV-WAVE` windows. It closes an individual VTK window but does not shut down the Tcl process. Use `vtkCLOSE` to close all windows and shut down the spawned Tcl process.

## Example 1

Deleting a window.

```
vtkwindow, 1  
vtkwdelete
```

## Example 2

```
vtkwindow, 1  
vtkwindow, 2  
vtkwindow, 3  
vtkwdelete, /all
```

## See Also

[vtkCLOSE](#), [vtkWINDOW](#)

---

## ***vtkWINDOW Procedure***

Creates a VTK window.

## Usage

```
vtkWINDOW [window_index]
```

## Input Parameters

*window\_index* — (optional) An integer specifying the index of the newly created window.

If *window\_index* is omitted, 0 is used as the index of the new window.

If the value of *window\_index* specifies an existing window, the existing window is deleted and a new window is created.

## Keywords

**Free** — If nonzero, creates a window using an unused window index. This keyword can be used instead of specifying the *window\_index* parameter.

**NoRender** — If nonzero, prevents individual objects (vtkLIGHT, vtkAXES, vtkPOLYSHADE, etc.) from being rendered as they are added to the window. Specifying *NoRender* can speed up the initial display of a scene if you have multiple objects in it. If specified, you must manually call vtkRENDERWINDOW after you have added all of your objects to the window.

**Background** — Background color for the window. The color can be specified in any of the following ways (the color 'red' is used here as an example):

'red'	See the file <code>&lt;vni&gt;/vtk-3_2/lib/vtkcolornames.pro</code> for a complete list of supported color names, where <code>&lt;vni&gt;</code> is the path to the PV-WAVE installation.
'FF0000'XL	A long integer hexadecimal value specifying the 24-bit color.
[1.0, 0.0, 0.0]	A three-element vector of normalized floating point values specifying the red, green, and blue components of the color.
2	If a short byte or short integer value is passed, the RGB color is obtained from the corresponding entry in the current PV-WAVE color table. In this case, when TEK_COLOR has been called, color index 2 is red.

**/NoInteract** — If nonzero, indicates that you do not wish to provide the standard set of mouse controls for viewing the 3D scene. The resulting scene can be manipulated only by programmatically setting the positional parameters for objects or cameras.

**XPos, YPos** — The *x* and *y* positions of the lower-left corner of the new window, specified in device coordinates.

If no position is specified, a position of (0,0) is used.

**XSize** — The width of the window, in pixels. (Default: 400)

*YSize* — The height of the window, in pixels. (Default: 400)

## Discussion

This procedure is similar to the `PV-WAVE WINDOW` command. It allows the created window to have built-in interaction associated with it.

## Example 1

This example shows how to bring up a VTK window.

```
vtkWINDOW, 1
```

## Example 2

This example shows how to bring up a VTK window with a blue background and with the mouse controls disabled. *windownum* is the number of the free window.

```
vtkWINDOW, windownum, /Free, background='blue', /nointeract
```

## See Also

[vtkRENDERWINDOW](#), [vtkCLOSE](#), [vtkWDELETE](#), [vtkERASE](#), [vtkWSET](#)

---

## ***vtkWRITEVRML Procedure***

Creates a Virtual Reality Modeling Language file (VRML .wrl file) from a scene in a VTK window.

## Usage

```
vtkWRITEVRML, filename [, WindowID=id, Speed=s]
```

## Input Parameters

*Filename* — A string, the file to write (should end in “.wrl”).

## Keywords

*Windowid* — An integer, the VTK window to use as the source. (Default: the currently active VTK window)

*Speed* — A float, the navigation speed. (Default: 4.0)

## Discussion

To view a VRML .wrl file in a browser, you need a plug-in. Consult the FAQ at <http://www.vrml.org> for current information and to obtain a plug-in.

## See Also

[VRML Routines](#)

---

## ***vtkWSET Procedure***

Sets the active VTK window.

### Usage

`vtkWSET` [, *window\_index*]

### Input Parameters

*window\_index* — (optional) An integer specifying the index of an existing VTK window. If omitted, the current window is used.

### Keywords

None.

### Discussion

This procedure works like WSET for PV-WAVE windows, which is used to select the current, or "active" window to be used by the VTK routines.

### Example

Setting the window to the first window opened.

```
vtkwindow, 1
```

```
vtkwindow, 2
```

```
vtkwindow, 3
```

```
vtkaxes  
vtkwset, 1  
vtkaxes
```

## **See Also**

[vtkWINDOW](#), [WSET](#)

---

## ***WAIT Procedure***

Suspends execution of a PV-WAVE program for a specified period.

### **Usage**

WAIT, *seconds*

### **Input Parameters**

*seconds* — The duration of the wait, in seconds.

### **Keywords**

None.

### **Discussion**

WAIT is useful in interactive programs that repetitively read cursor positions.

---

**NOTE** Because of other activity on the system, the duration of program suspension may be longer than requested.

---

### **See Also**

[HAK](#), [MESSAGE](#), [TVCRS](#)

---

## ***WCOPY Function (Windows)***

Copies the contents of a graphics window onto the Clipboard.

### **Usage**

*status* = WCOPY( [*window\_index*] )

### **Input Parameters**

*window\_index* — (optional) The index of the window to copy to the clipboard. If not specified, the current window is assumed.

### **Output Parameters**

None.

## Returned Value

*status* — The value returned by WCOPY; expected values are:

- < 0 Indicates an error.
- 0 Indicates a successful copy.

## Keywords

None.

## Discussion

You can copy graphics to the Clipboard in two ways:

- The WCOPY function
- The **Copy to Clipboard** option on the graphics window Control menu

## Example

This example demonstrates a simple method for resizing graphics using WCOPY and WPASTE.

```
WINDOW, 2,  
SHADE_SURF, DIST(40)  
; Display some graphics in the window.
```

— Now, resize the window using the mouse. —

---

**NOTE** The graphics inside the window are not resized.

---

```
status = WCOPY(2)  
; Copy the graphics to the Clipboard.  
status = WPASTE(2)  
; Paste the plot back into the window. The graphics are redrawn to fit  
; in the resized window.
```

## See Also

[WINDOW](#), [WPASTE](#)

---

**Windows USERS** The graphics window Control menu includes a command that copies graphics to the Clipboard. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## **WDELETE Procedure**

Deletes the specified window.

### **Usage**

WDELETE [, *window\_index*]

### **Input Parameters**

*window\_index* — (optional) The window index of the window to be deleted. If not specified, deletes the current window, and the system variable !D.Window is set to the index of the first open window (or to -1 if no other windows are open).

### **Keywords**

*All* — If present and nonzero, deletes all open windows.

*X\_No\_Close* — (UNIX/OpenVMS Only) If present and nonzero, allows the window to be deleted for PV-WAVE, but remain active for the X Window System server. Otherwise, the window is deleted for both systems.

### **See Also**

[ERASE](#), [WINDOW](#), [WSET](#), [WSHOW](#)

---

**Windows USERS** The graphics window Control menu provides a function for closing a window. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## **WEOF Procedure**

(OpenVMS Only) Writes an end-of-file mark on the designated unit at the current position.

### **Usage**

WEOF, *unit*

### **Input Parameters**

*unit* — An integer between 0 and 9 specifying the magnetic tape unit on which the end-of-file mark will be written. (Do not confuse this parameter with file logical unit numbers.)

### **Keywords**

None.

### **Discussion**

To use WEOF, you must mount the tape as a foreign volume. The end-of-file mark is also sometimes called a tape mark.

### **See Also**

[SKIPF](#), [TAPRD](#), [TAPWRT](#)

For more information, see in Chapter 8 of the *PV-WAVE Programmer's Guide*.

---

## **WgAnimateTool Procedure**

Creates a window for animating a sequence of images.

### **Usage**

WgAnimateTool, *image\_data* [, *parent* [, *shell* ]]

### **Input Parameters**

*image\_data* — A 3D array of images. The dimensions of the array are (*m*, *n*, *n\_frames*), where (*m,n*) is the size of an individual image, and *n\_frames* is the total number of frames in the sequence.

**parent** — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgAnimateTool runs on its own (i.e., in its own event loop).

## Output Parameters

**shell** — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

## Keywords

**Cmap** — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

**Delay** — The minimum elapsed time between displayed images, specified in milliseconds (floating point).

**Dims** — A three-element vector specifying the size of the images to be read from the file, and the number of images to read. The elements of the vector are [*m*, *n*, *n\_frames*], where (*m,n*) is the size of an individual image, and *n\_frames* is the total number of frames in the sequence.

**Do\_tvsc1** — Indicates whether TVSCL or TV should be used to scale the image values to the current color table.

- 1 Specifies use of TVSCL.
- 0 Indicates that TV is used instead (no scaling).

**File** — A string containing the name of the file from which the image data is read. When using the *File* keyword, the *Dims* keyword must also be supplied. If present, the input variable *image\_data* is ignored.

**Order** — The order in which the image is drawn. If present and nonzero, the image is inverted. In other words, the image is drawn from bottom to top instead of from top to bottom.

**Pixmap** — Indicates whether pixmaps should be used for the animation.

- 1 Specifies the use of pixmaps.
- 0 Specifies the data is stored in a variable.

Pixmaps dramatically improve the speed of the animation, but require more memory than when data stored in variables is used.

**Position** — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the AnimateTool window (long integer). The elements of the vector

are  $[x, y]$ , where  $x$  (horizontal) and  $y$  (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

**Title** — A string containing the title that appears in the header of the AnimateTool window. Default value is “Animate Tool”.

### ***Color/Font Keywords***

For additional information on the color and font keywords, see [Setting Colors and Fonts](#) in the *PV-WAVE Application Developer's Guide*.

**Background** — Specifies the background color name.

**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

## **Discussion**

WgAnimateTool is an interactive window that lets you use the mouse to control the pace and direction of an animated series of images.

Using the WgAnimateTool window is similar in many ways to using the WgMovieTool window, but WgAnimateTool is intended to be used as a stand-alone utility widget, while WgMovieTool is designed so that it can be included inside larger layout widgets.

### ***Input Data Requirements***

The animation can use data from either pixmaps or a variable. Although the animation will run faster using pixmaps, it does require more memory to store the data as a pixmap than as a variable.

When reading the data from a file, the file containing the data must be a binary file containing the images in sequential order.

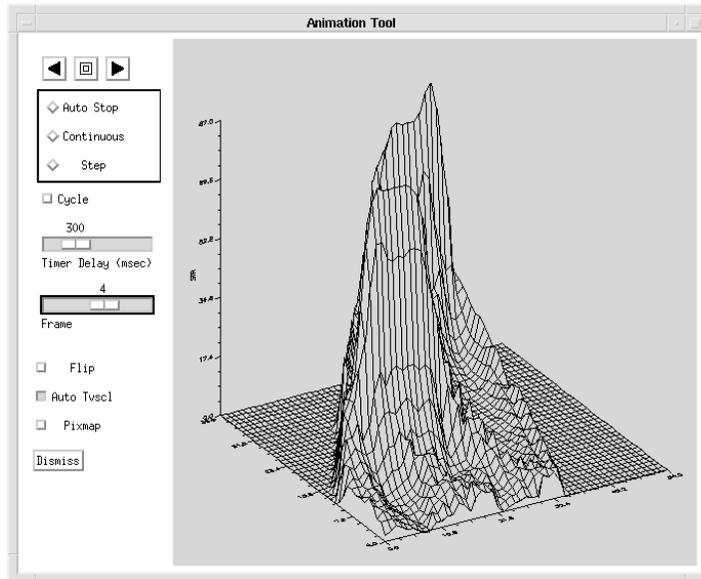
### ***Event Handling***

You can use the AnimateTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the AnimateTool widget. The AnimateTool widget handles its own event loop by calling WwLoop.

- **Stand-alone widget in its own window created by another application** — The `AnimateTool` widget has its own Main window, but the application (not the `AnimateTool` widget) handles the event loop by calling `WwLoop`.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.



**Figure 2-77** `WgAnimateTool` creates an interactive window that lets you use the mouse to control the orientation, pace, and direction of an animated series of images.

## Contents of the Window

The `AnimateTool` window has two main parts — the display area and the control area.

### *AnimateTool* Display Area

The display area is the largest area of the window; it is where the animation sequence takes place.

### *AnimateTool* Control Area

Use the following controls to operate the `AnimateTool` window:

- **Reverse button** — Cycle through the images from last to first.

- **Stop button** — Freeze display at the current image.
- **Forward button** — Cycle through the images from first to last.
- **Mode** — Select **Step** to have the sequence proceed only when you click on one of the arrow buttons. Select **Continuous** to have `AnimateTool` move through the images in a non-stop manner, pausing only when you click the **Stop** button. Select **Auto Stop** to have `AnimateTool` stop sequencing when it reaches either end of the data.
- **Cycle** — When enabled, the animation sequences repeatedly through the data, alternating between forward and reverse. When disabled, the animation returns to the first (or last) image in `image_data` after every image in the sequence has been displayed.
- **Timer delay** — Decrease or increase the rate of display. The number shown above the slider is the number of milliseconds delay between contiguous images in the animation sequence.
- **Frame** — The ordinal number of the currently displayed image is displayed above this slider. If you wish, use the left mouse button to drag the slider and display a different image.
- **Flip** — When enabled, the first data value is used to draw the pixel in the upper-left corner of the image. When disabled, the first data value is used to draw the pixel in the lower-left corner of the image.
- **Auto Tvscl** — By determining whether the TV or TVSCL is used to draw the images, controls whether the input image data is automatically scaled to use the full range of available colors. Selecting this option may increase the contrast of the displayed images. **Auto Tvscl** does not affect the actual data; it only affects the display of the data.
- **Pixmap** — When enabled, the data is stored in pixmaps; when disabled, the data is taken directly from the variable.
- **Dismiss** — Destroy the `AnimateTool` window and erase it from the screen.

## Example

Enter the commands shown below into a file, and compile the procedure with the `.RUN` command. If the variable `parent` is defined, `WgAnimateTool` is created as a child of `parent`; otherwise, `WgAnimateTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgAnimateTool` window, close it by clicking the **Dismiss** button.

```
PRO Sample_wganimatetool, parent, tool_shell
```

```

heart = BYTARR(256, 256, 15)
    ; Define a variable to hold the data.
IF !Version.platform EQ 'VMS' THEN $
    OPENR, u, GETENV('WAVE_DIR')+
    '[data]heartbeat.dat', /Get_lun $

ENDIF ELSE BEGIN
OPENR, u, !Dir+'/data/heartbeat.dat', /Get_lun
ENDELSE
READU, u, heart
    ; Read the file that of images showing a beating human heart.

CLOSE, u
FREE_LUN, u
    ; Close the file and free the LUN.

IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgAnimateTool, heart, parent, tool_shell, $
    /Do_tvsc1, /Pixmap
    ; Create WgAnimateTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
    ; output parameter "tool_shell".

ENDIF ELSE BEGIN
WgAnimateTool, heart, /Do_tvsc1, /Pixmap
    ; Create WgAnimateTool and display it as its own Main window. In
    ; other words, the WgAnimateTool window runs on its own (i.e., in its
    ; own event loop).

ENDELSE
END

```

## See Also

[MOVIE](#), [WgMovieTool](#)

For more information about how to transfer image data to variables, refer to in Chapter 8 of the *PV-WAVE Programmer's Guide*.

For more information about pixmaps, refer to [Appendix B, Output Devices and Window Systems](#).

For more information about color table indices, refer to in Chapter 11 of the *PV-WAVE User's Guide*.

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program

based the PV-WAVE Widget Toolbox, refer to , in the *PV-WAVE Application Developer's Guide*.

---

## **WgCbarTool Procedure**

Creates a simple color bar that can be used to view and interactively shift a color table.

### **Usage**

WgCbarTool [*parent* [, *shell* [, *windowid* [, *movedCallback*], [, *range*]]]]

### **Input Parameters**

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgCbarTool runs on its own (i.e., in its own event loop).

*movedCallback* — (optional) A string containing the name of the callback routine that is executed when the color bar is shifted to the left or right.

*range* — (optional) The range of colors to be displayed in the color bar. The default is to display the entire range of colors, as defined in the system variable !D.Table\_Size.

### **Output Parameters**

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

*windowid* — (optional) The window ID of the graphics window. (For information on window IDs, see the description for the [WINDOW](#) procedure.)

### **Keywords**

*Cmap* — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

*Horizontal* — If present and nonzero, the color bar is oriented horizontally (see ). (Default: vertical orientation)

*Popup* — If present and nonzero, the color bar widget is displayed in its own Main window.

*Position* — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the CbarTool window (long integer). The elements of the vector are

[*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are either: 1) if *parent* is present and the *Popup* keyword is not specified or has a value of zero, measured from the upper-left corner of a layout (container) widget or 2) if the *Popup* keyword is present and nonzero (regardless of whether *parent* is present and/or nonzero), measured from the upper-left corner of the screen.

**Size** — A two-element vector specifying the width and height of the color bar (long integer). If not specified, the default size of the color bar is 30-by-256 pixels.

**Title** — A string containing the title that appears in the header of the CbarTool window. Default value is “Color Bar”.

**Vertical** — If present and nonzero, the color bar is oriented vertically. This is the default orientation.

### **Color/Font Keywords**

For additional information on the color and font keywords, see

**Background** — Specifies the background color name.

**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

### **Attachment Keywords**

For additional information on attachment keywords, see

**Bottom** — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the color bar widget is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the color bar widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the color bar widget is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the color bar widget is attached to the left side of the parent widget.

**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the color bar widget is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the color bar widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, `Top=wid`), then the top of the color bar widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, `/TOP`), then the top of the color bar widget is attached to the top of the parent widget.

## Discussion

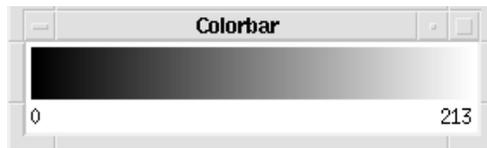
The color bar displays the colors of the current color table and “rotates” them.

To rotate the color table using the color bar, press and drag the left mouse button inside the color bar. As you “slide” colors into different color table indices, the colors that “scroll off” the end of the color table are added to the opposite end.

---

**NOTE** Use the window manager menu of the window frame to dismiss the Cbar-Tool window from the screen.

---



**Figure 2-78** WgCbarTool creates an array of colors that match the colors in the current color table. This color bar widget has been created using the *Horizontal* option; the default is for the color bar to be displayed in a vertical orientation.

## Event Handling

You can use the color bar widget in one of the following three ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the color bar widget. The color bar widget handles its own event loop by calling `WwLoop`.
- **Stand-alone widget in its own window created by another application** — If the *Popup* keyword is present and nonzero, the color bar widget has its own Main window. The application (not the color bar widget) handles the event loop by calling `WwLoop`.
- **Combined with other widgets in a layout widget** — Another application combines the color bar widget with other widgets inside a layout widget. The application (not the color bar widget) handles the event loop by calling `WwLoop`.

When the input parameter *parent* is present and nonzero, the widget operates in either the second or third mode listed, depending on the value of the *Popup* keyword. The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

You can use the output parameter *windowid* to keep track of the PV-WAVE window ID that is assigned to the color bar. You can create multiple instances of the color bar; each one will be assigned a different PV-WAVE window ID.

## The Colors Common Block

This procedure modifies the PV-WAVE internal color table, as well as the color table variables in the `Colors` common block.

Most color table procedures maintain the current color table in a common block called `Colors`, defined as follows:

```
COMMON Colors, r_orig, g_orig, b_orig, r_curr, g_curr, b_curr
```

The variables are integer vectors of length equal to the number of color indices. Your program can access these variables by declaring the common block. The modifications you make to the color table by interacting with the `CbarTool` widget are stored in `r_curr`, `g_curr`, and `b_curr`.

## Example

Enter the commands shown below into a file, and compile the procedure with the `.RUN` command. If the variable `parent` is defined, `WgCbarTool` is created as a child of `parent`; otherwise, `WgCbarTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgCbarTool` window, close it using the window manager menu.

```
PRO Sample_wgcbartool, parent, tool_shell
IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgCbarTool, parent, tool_shell, /Horizontal, /Popup
    ; Create WgCbarTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
    ; output parameter "tool_shell".
ENDIF ELSE BEGIN
WgCbarTool, /Horizontal
    ; Create a horizontally oriented WgCbarTool and display it as its own
    ; Main window (see ). In other words, the WgCbarTool
    ; window runs on its own (i.e., in its own event loop).
ENDELSE
```

END

## See Also

[WgCeditTool](#)

For more information about color table indices, refer to in Chapter 11 of the *PV-WAVE User's Guide*.

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

---

## ***WgCeditTool Procedure***

Creates a full-featured set of menus and widgets enclosed in a window; this window allows you to edit the values in PV-WAVE color tables in many different ways.

### **Usage**

`WgCeditTool [ , parent [ , shell ]]`

### **Input Parameters**

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, `WgCeditTool` runs on its own (i.e., in its own event loop).

### **Output Parameters**

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

### **Keywords**

*Cmap* — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

*Position* — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the `CeditTool` window (long integer). The elements of the vector are [*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

*Title* — A string containing the title that appears in the header of the CeditTool window. Default value is “Color Editor”.

## Color/Font Keywords

For additional information on the color and font keywords, see

*Background* — Specifies the background color name.

*Basecolor* — Specifies the base color.

*Font* — Specifies the name of the font used for text.

*Foreground* — Specifies the foreground color name.

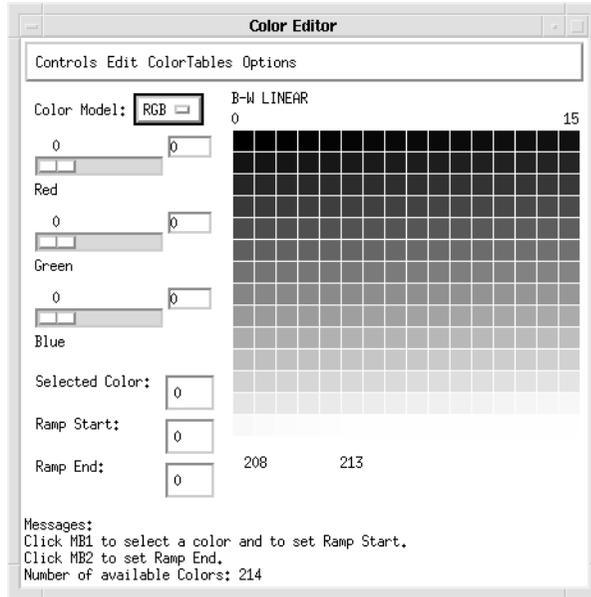
## Discussion

Using the features of this window, you can edit individual color table values, or you can change ranges of values. (Range of colors can be selected and linearly interpolated.) You can use either sliders or a color wheel to adjust the current color.

You use a palette of color cells to select the current color — point to the color that interests you and click the left mouse button to select it. In this sense, the CeditTool window is like an artist’s palette, where basic colors are mixed to produce different colors.

You can use either the RGB, HLS, or the HSV color system to create a new color table, and you can view the result as a color bar or as a set of three intensity graphs. You can store color tables you have modified as custom color tables, and they will be available for your later use.

In some ways, the WgCeditTool window is similar to the WgCtTool window, in that it provides easy ways to interactively modify color tables. For example, with just a few clicks, you can use the WgCeditTool window to edit individual colors in the color table. Or you can use the **Options** menu to open several other utility widgets. But if you do not need all the options of the WgCeditTool window, or if you just need a quick way to stretch or reverse a color table, then use the WgCtTool window, instead.



**Figure 2-79** The WgCeditTool window lets you use the mouse to create a new color table based on either the HLS, HSV, or RGB color systems.

## What is a Color Table?

On workstations that support color, the specific colors that are used depend on the current *color table*. A color table maps data values into colors. Many workstations support thousands of different colors, but only some number, usually 256, can be displayed at any one time.

By default, WgCeditTool uses as many colors in the color table as are currently available in the workstation's colormap. This number depends on the colors that have already been allocated by other applications running on that workstation.

### ***The Colors Common Block***

This procedure modifies the PV-WAVE internal color table, as well as the color table variables in the COLORS common block.

Most color table procedures maintain the current color table in a common block called Colors, defined as follows:

```
COMMON Colors, r_orig, g_orig, b_orig, r_curr, g_curr, b_curr
```

The variables are integer vectors of length equal to the number of color indices. Your program can access these variables by declaring the common block. The modifications you make to the color table by interacting with the CeditTool widget are stored in `r_curr`, `g_curr`, and `b_curr`.

### ***Custom Color Table File***

The names of custom color tables that have been saved are stored in a file named `.wg_colors`. This file is placed in your home directory, which in UNIX is defined by `$HOME`, and in OpenVMS is defined by `SYS$LOGIN`.

---

**CAUTION** Do not attempt to edit the `.wg_colors` file — doing so could lead to unpredictable results.

---

## **Contents of the Window**

The CeditTool window has three main parts — the color palette area, the control area, and the message area. The options provided by the menu bar are discussed in a later section.

### ***CeditTool Color Palette Area***

An array of cells, one for each color index from the colormap. A message near the bottom of the window informs you how many colors were available to PV-WAVE, and thus how many color cells could be displayed.

### ***CeditTool Control Area***

The window's control area contains sliders, a menu, and text fields that are used to manipulate colors:

- **Color Model (menu)** — Choose a color system from the menu. By default, the CeditTool uses the RGB color system. For more information about color systems, refer to in Chapter 11 of the *PV-WAVE User's Guide*.
- **Color Cell Text Fields and Sliders** — These controls allow you to modify the three basic components of any color in the color table. The labels on the text fields and sliders change to coincide with the current color system. First, select a color to modify by clicking on one of the color cells in the display area, or by entering its index number in the **Selected Color** text field. To modify the selected color, either enter a new color value in one of the text fields, or use the slider to change the value. If you enter a new color value into a text field, press `<Return>` to apply the new value to the slider and the color cell that corre-

sponds to the selected color. If you use the sliders, the change is applied immediately as the slider moves.

- **Selected Color** — The index number of the currently selected color in the color table. Once a color is selected, other sliders and text fields in the tool control area can be used to modify its composition. Select a color by clicking the left mouse button on a color cell in the palette of cells, or by entering the color's index number in this text field and pressing <Return>.
- **Ramp Start and End** — Two text fields where you can enter color table indices between which the CeditTool will perform a linear color interpolation; this is an easy way to edit the current color table. Either enter the beginning and ending color indices in the **Ramp Start** and **Ramp End** text fields or use the mouse to select the ramp's starting and ending colors. (To use the mouse, click the left mouse button on the starting color cell and the middle mouse button on the ending color cell.) Select **Edit=>Ramp** on the CeditTool window. The affected cells in the displayed color table are updated immediately, although no permanent change is made until you save the color table as a custom color table.

## CeditTool Menu Bar

The CeditTool menu bar consists of four menu buttons located near the top of the window:

### **Controls Menu**

- **Save Custom** — Save the current set of color indices using the same custom color table name as before.
- **Save As Custom** — Save the current set of color indices as a new custom color table with a unique name.
- **Rename Custom** — Rename a custom color table. You must select the custom color table from the Custom Color Table option menu (select **ColorTables=>Custom**) before you can rename it.
- **Delete Custom** — Delete one of the custom color tables from the list. For details about where this list of color tables is stored, refer to [Custom Color Table File on page 1086](#).
- **Exit** — Destroy the CeditTool window and all its children.

### ***Edit Menu***

- **Ramp** — Create a color ramp between the start ramp color and the end ramp color. See the description of **Ramp Start** and **Ramp End** on the previous page for more information about choosing start and end ramp colors.
- **Restore** — Restore the original color map for the selected color table.

### ***ColorTables Menu***

- **System** — Display a list of system color tables. System color tables are provided with PV-WAVE and cannot be modified.
- **Custom** — Display a list of custom color tables. These are color tables that you have created using **Controls=>Save Custom** or **Controls=>Save As Custom** (from the **Controls** menu).

### ***Options Menu***

- **Color Wheel** — Select the HLS or HSV components for a particular color by interacting with a dial and a slider; the color wheel is shown in .

The characteristics you can adjust in the dial are: 1) hue (azimuth of mouse from the center), and 2) saturation (distance from the center).

The characteristics you can adjust in the vertical slider are value (in the HSV color system) or lightness (in the HLS color system).

---

**TIP** The farther you click from the center of the wheel (with the left mouse button), the more saturated the color (for that particular hue).

---

- **Intensity Graphs** — The value of each of the three color system parameters (HLS or HSV) is plotted versus pixel value in a separate line graph. This results in three line graphs showing the current values of the three parameters for the current color table, as shown in .
- **Color Bar** — Display a vertical color bar; this color bar displays every one of the colors in the current color table; a full-range color bar is shown in .

The color bar feature lets you display the colors of the current color table and “rotate” them. To rotate the color table using the color bar, press and drag the left mouse button up or down in the color bar.

If you have any other CeditTool options displayed on your screen, like the color wheel, you will see immediate effects as the color table shifts in relation to the mouse’s movement.

---

**NOTE** As you “slide” colors into different color table indices, the colors that “scroll off” the end of the table are added to the opposite end.

---

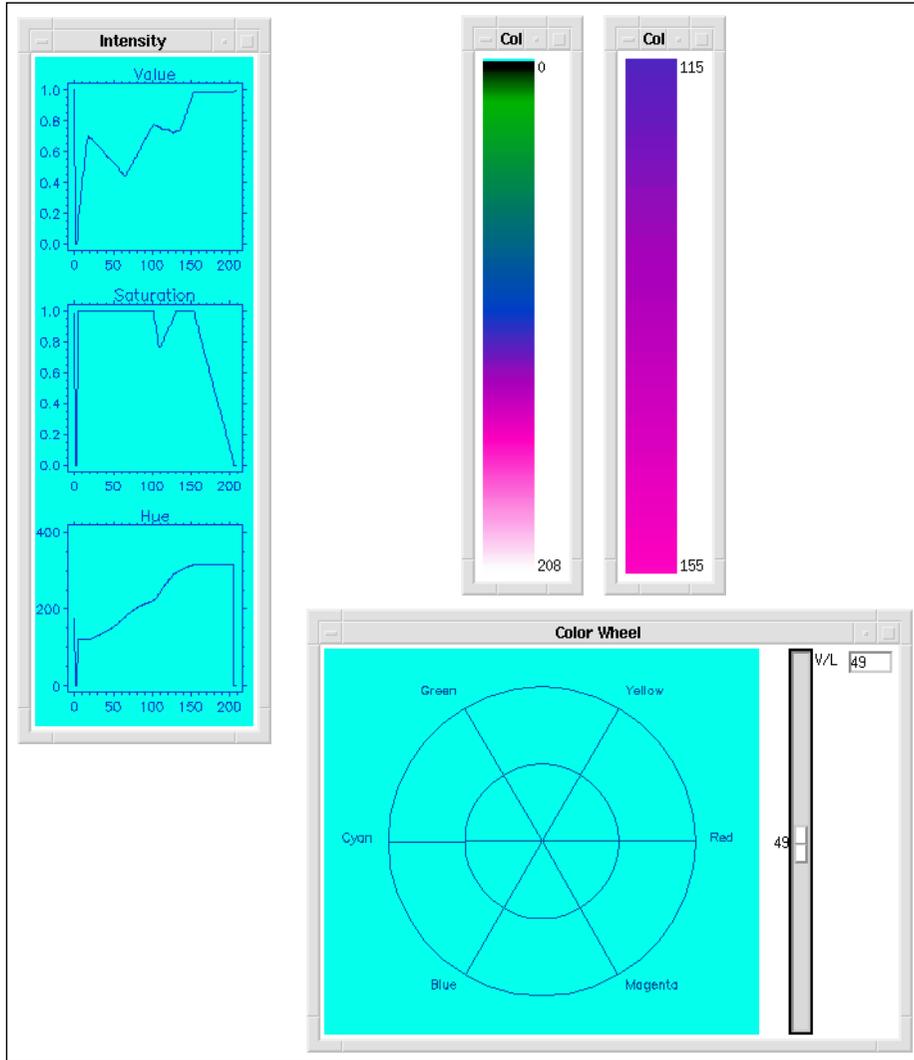
- **Color Bar Range** — Display a vertical color bar displaying only the range of selected colors; a partial-range color bar is shown in . The range of selected colors is the same as the range chosen for the linear interpolation in the text fields **Ramp Start** and **Ramp End**.

## Event Handling

You can use the CeditTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the CeditTool widget. The CeditTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The CeditTool widget has its own Main window, but the application (not the CeditTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.



**Figure 2-80** Utility widgets accessible via WgCeditTool's Options menu. The four utility widgets are (starting from the upper left, proceeding clockwise): 1) intensity graphs, 2) a color bar showing the entire range of the color table, 3) a color bar showing only a selected range of the color table (range modifiable by user), and 4) a color wheel and slider that can be used to interactively modify the current color. For more information about any of these utility widgets, refer to the *Options Menu* section.

## Example

Enter the commands shown in this example into a file, and compile the procedure with the `.RUN` command. If the variable `parent` is defined, `WgCeditTool` is created as a child of `parent`; otherwise, `WgCeditTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgCeditTool` window, close it by selecting **Controls=>Exit**.

```
PRO Sample_wgcedittool, parent, tool_shell
IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgCeditTool, parent, tool_shell
    ; Create WgCeditTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
    ; output parameter "tool_shell".

ENDIF ELSE BEGIN
WgCeditTool
    ; Create WgCeditTool and display it as its own Main window. In other
    ; words, the WgCeditTool window runs on its own (i.e., in its own
    ; event loop).

ENDELSE
END
```

## See Also

[COLOR\\_EDIT](#), [COLOR\\_PALETTE](#), [WgCfTool](#)

For more information about color table indices, refer to in Chapter 11 of the *PV-WAVE User's Guide*.

For more information about color systems, refer to in Chapter 11 of the *PV-WAVE User's Guide*.

For more information about how to write an application program based on WAVE Widgets, refer to .

For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

---

## **WgCtTool Procedure**

Creates a simple widget that can be used interactively to modify a PV-WAVE color table.

### **Usage**

WgCtTool [, *parent* [, *shell* ]]

### **Input Parameters**

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgCtTool runs on its own (i.e., in its own event loop).

### **Output Parameters**

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

### **Keywords**

*Cmap* — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

*Position* — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the CtTool window (long integer). The elements of the vector are [*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

*Title* — A string containing the title that appears in the header of the window. Default value is “Color Table Tool”.

### **Color/Font Keywords**

For additional information on the color and font keywords, see .

*Background* — Specifies the background color name.

*Basecolor* — Specifies the base color.

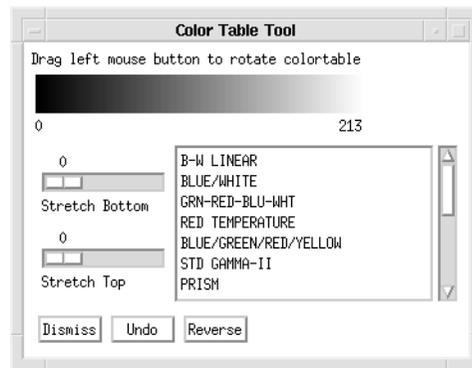
*Font* — Specifies the name of the font used for text.

*Foreground* — Specifies the foreground color name.

## Discussion

The WgCtTool window () allows you to interactively modify system color tables by stretching, rotating, and reversing them.

In some ways, the WgCtTool window is similar to the WgCeditTool window, in that it provides easy ways to interactively modify color tables. For example, with a single click, you can reverse the color table using the WgCtTool window. But if you need to edit individual colors in the color table, or save your changes for future use, then use the WgCeditTool window, instead.



**Figure 2-81** The WgCtTool window lets you interactively modify system color tables by stretching, rotating, and reversing them.

## The Colors Common Block

This procedure modifies the PV-WAVE internal color table, as well as the color table variables in the COLORS common block.

Most color table procedures maintain the current color table in a common block called Colors, defined as follows:

```
COMMON Colors, r_orig, g_orig, b_orig, $
           r_curr, g_curr, b_curr
```

The variables are integer vectors of length equal to the number of color indices. Your program can access these variables by declaring the common block. The modifications you make to the color table by interacting with the CtTool widget are stored in r\_curr, g\_curr, and b\_curr.

## Contents of the Window

The CtTool window has three main parts — the color bar area, the control area, and the system color table list.

### ***CtTool Color Bar Area***

The built-in color bar lets you display the colors of the current color table and “rotate” them. To rotate the color table using the color bar, press and drag the left mouse button to the left or to the right in the color bar. As you “slide” colors into different color table indices, the colors that “scroll off” the end of the color table are added to the opposite end.

### ***CtTool Control Area***

Use the following controls to operate the CtTool window:

- **Stretch Bottom** — Select the lower limit for the stretched color table.
- **Stretch Top** — Select the upper limit for the stretched color table.
- **Undo** — Discard color table modifications you have made using the CtTool window, and return to the color table that CtTool was using when it was first created.
- **Reverse** — Swap the color table (as currently defined), end for end.
- **Dismiss** — Destroy the CtTool window and erase it from the screen.

The CtTool window uses the STRETCH procedure to linearly expand a range of color table indices. The **Stretch Bottom** number is used for the first parameter to the STRETCH command, and the **Stretch Top** number is subtracted from the number of colors available in the color table to determine the second parameter to the STRETCH command. For more information about the STRETCH command, refer the description for [STRETCH](#).

### ***System Color Table List***

This area lists the 16 system color tables that are provided as part of PV-WAVE. Use the left mouse button to select one. The color table you select from the list will immediately be displayed in the color bar area.

Because system color tables are “read-only”, no system color table will be permanently altered by any changes you make with the CtTool window. For this reason, the changes you make with CtTool are temporary and can be overwritten by any other PV-WAVE routine writing to the `Colors` common block.

---

**NOTE** To save color table changes for later use, you can use another utility widget, [WgCeditTool](#).

---

## Event Handling

You can use the CtTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the CtTool widget. The CtTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The CtTool widget has its own Main window, but the application (not the CtTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

## Example

Enter the commands shown below into a file, and compile the procedure with the .RUN command. If the variable *parent* is defined, WgCtTool is created as a child of *parent*; otherwise, WgCtTool runs on its own (i.e., in its own event loop).

When you are finished interacting with the WgCtTool window, close it by clicking the **Dismiss** button.

```
PRO Sample_wgcttool, parent, tool_shell
IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgCtTool, parent, tool_shell
    ; Create WgCtTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
    ; output parameter "tool_shell".
ENDIF ELSE BEGIN
WgCtTool
    ; Create WgCtTool and display it as its own Main window. In other
    ; words, the WgCtTool window runs on its own (i.e., in its own event
    ; loop).
ENDELSE
END
```

## See Also

[COLOR\\_EDIT](#), [COLOR\\_PALETTE](#), [WgCeditTool](#)

For more information about color table indices, refer to in Chapter 11 of the *PV-WAVE User's Guide*.

For more information about color systems, refer to in Chapter 11 of the *PV-WAVE User's Guide*.

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

---

## **WgIsoSurfTool Procedure**

Creates a window with a built-in set of controls; these controls allow you to easily view and modify an iso-surface taken from a three-dimensional block of data.

### **Usage**

WgIsoSurfTool, *surface\_data* [, *parent* [, *shell* ]]

### **Input Parameters**

*surface\_data* — A 3D variable containing volumetric surface data. The dimensions of *surface\_data* define the size of the cube that you see when the WgIsoSurfTool window is first opened.

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgIsoSurfTool runs on its own (i.e., in its own event loop).

### **Output Parameters**

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

### **Keywords**

*Cmap* — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

*Cube\_color* — The color in which the rotatable cube is drawn; a positive integer in the range (0...255). The number that is provided for *cube\_color* is used as an index into the current color table.

*Position* — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the IsoSurfTool window (long integer). The elements of the vector

are  $[x, y]$ , where  $x$  (horizontal) and  $y$  (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

**Thresh\_range** — The range of values to display in the threshold slider. The default is to use the entire range of values defined by *surface\_data*, or in other words, the range:

*(min\_surface\_data...max\_surface\_data)*

**Thresh\_value** — The initial position of the threshold slider. By default, the movable portion of the slider is positioned in the middle of the threshold range.

**Title** — A string containing the title that appears in the header of the IsoSurfTool window. Default value is “Isosurface Tool”.

**View\_persp** — The initial perspective distance. If *View\_persp* is zero, then the initial setting on the **Perspective Distance** slider is 0.5, but the **Perspective** pushbutton is initially deselected. If *View\_persp* is nonzero, then the initial setting on the **Perspective Distance** slider is the value specified by *View\_persp* and the **Perspective** pushbutton is selected.

---

**NOTE** Using no perspective to draw the cube is equivalent to having the eyepoint an infinite distance away from the cube, and will produce a cube with sides that all appear to be parallel to one another.

---

**View\_rot** — A three-element vector specifying the initial view rotation, in degrees. If not supplied, the initial rotation of the cube and the iso-surface is  $[30, 30, 0]$  — in other words, 30 degrees rotation around the  $x$ -axis, 30 degrees rotation around the  $y$ -axis, and 0 degrees rotation around the  $z$ -axis.

**View\_zoom** — The initial zoom factor. (Default: 0.5)

### **Color/Font Keywords**

For additional information on the color and font keywords, see .

**Background** — Specifies the background color name.

**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

## Discussion

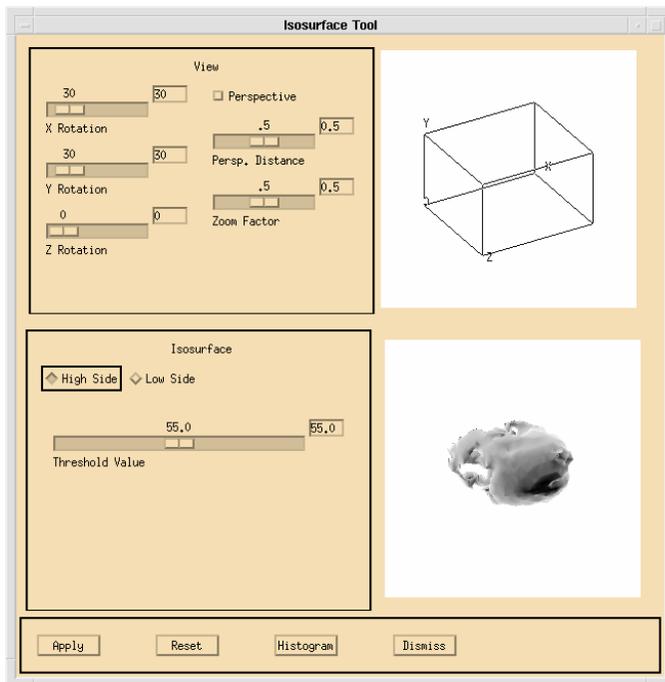
WgIsoSurfTool is an interactive window that lets you use the mouse to view and modify iso-surfaces; you can view any iso-surface contained within the three-dimensional input dataset, *surface\_data*.

If the zoom factor is large, or the perspective parameter is small, then the cube display in the View Orientation window may be erroneous. To cure the problem, reduce the zoom or increase the perspective (or disable the perspective entirely).

---

**TIP** Some iso-surfaces can be time-consuming to render, especially those that compose many polygons. PV-WAVE will display its “wait cursor” while it is performing the calculations necessary to display the iso-surface.

---



**Figure 2-82** WgIsoSurfTool creates an interactive window that lets you use the mouse to easily view and modify an iso-surface taken from a three-dimensional block of data. An iso-surface is a pseudo-surface of constant density within a volumetric data set.

## Event Handling

You can use the IsoSurfTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the IsoSurfTool widget. The IsoSurfTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The IsoSurfTool widget has its own Main window, but the application (not the IsoSurfTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

## Contents of the Window

The IsoSurfTool window has two main parts — the display area and the control area.

### *IsoSurfTool Display Area*

The display area is towards the right of the window; it is divided into an upper and lower region:

- **View orientation area** — The cube establishes a frame of reference for viewing the iso-surface.
- **Iso-surface area** — This region contains an image displaying an iso-surface; the iso-surface shows similarities in whatever property the data is measuring.

---

**NOTE** The orientation of the iso-surface is copied from the orientation of the cube.

---

### *IsoSurfTool Control Area*

Use the following controls to operate the IsoSurfTool window:

- **X, Y, and Z Rotation** — These controls allow you to rotate the cube (counterclockwise around the desired axis) a specified number of degrees. The current rotation is shown in the text field to the right of the slider. To modify the rotation, either enter a new value in one of the text fields, or use the left mouse button to drag one of the sliders. If you enter a new value into a text field, press <Return> to apply the new value to the slider and the surface. If you use the sliders, the change is applied immediately as the slider moves.
- **Perspective** — If enabled, the cube is drawn with perspective; if disabled, the cube is drawn without perspective.

- **Perspective Distance** — Controls the amount of perspective used when drawing the cube; in other words, how close the eyepoint is to the cube. The closer the eyepoint gets to the cube, the greater the amount of perspective exaggeration that is used to draw the cube.
- **Zoom Factor** — Controls the amount of magnification used to draw the cube.
- **High side/low side** — If **High Side** is enabled, only data values above the threshold value are highlighted on the surface; if **Low Side** is enabled, only data values below the threshold value are highlighted.

For more information about **High Side/Low Side**, refer to the description of the *Low* keyword for the [SHADE\\_VOLUME](#) procedure.

- **Threshold Value** — Only data values above or below this value are highlighted on the surface, depending on whether **High Side** or **Low Side** is enabled.
- **Apply** — Redraw the iso-surface with the specified viewing parameters.
- **Reset** — Return to the default viewing parameters and the initial threshold value.
- **Histogram** — Draw a histogram of *surface\_data* in the iso-surface display area. Viewing this histogram can be useful in determining where to place the threshold value.
- **Dismiss** — Destroy the IsoSurfTool window and erase it from the screen.

---

**NOTE** When you click **Apply**, the graphics you see being redrawn in the iso-surface portion of the display area are being sent to this window via the Z-buffer virtual graphics device. This reduces the time required to redraw the iso-surface by about fourfold. For more information on the Z-buffer graphics device, refer to [Appendix B, Output Devices and Window Systems](#).

---

## Example

Enter the commands shown below into a file, and compile the procedure with the `.RUN` command. If the variable `parent` is defined, `WgIsoSurfTool` is created as a child of `parent`; otherwise, `WgIsoSurfTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgIsoSurfTool` window, close it by clicking the **Dismiss** button.

```
PRO Sample_wgisurftool, parent, tool_shell
head = BYTARR(115, 75, 105)
    ; Define a variable to hold the data.
```

```

IF !Version.platform EQ 'VMS' THEN $
    OPENR, u, GETENV('WAVE_DIR')+
    '[data]man_head.dat', /Get_lun $
ELSE $
OPENR, u, '$WAVE_DIR/data/man_head.dat', /Get_lun
READU, u, head
    ; Read the man_head.dat file that contains three-dimensional volumetric data.
CLOSE, u
FREE_LUN, u
    ; Close the file and free the LUN.
reduced_head = REBIN(head, 23, 15, 21)
    ; Sample the data so fewer data points are passed in to
    ; WgIsoSurfTool. Doing this improves performance significantly,
    ; because WgIsoSurfTool has fewer polygons to process.
IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgIsoSurfTool, reduced_head, parent, tool_shell
    ; Create WgIsoSurfTool as a child of the widget known as
    ; "parent". The window of the newly created widget is returned
    ; via the optional output parameter "tool_shell".
ENDIF ELSE BEGIN
WgIsoSurfTool, reduced_head
    ; Create WgIsoSurfTool and display it as its own Main window.
    ; In other words, the WgIsoSurfTool window runs on its own (i.e., in
    ; its own event loop).
ENDELSE
END

```

## See Also

[RENDER](#), [SHADE\\_VOLUME](#)

For information about drawing iso-surfaces using voxel data, see , in the *PV-WAVE User's Guide*. This chapter includes a number of examples showing iso-surfaces that have been drawn using the `RENDER` function.

For more information about how to write an application program based on `WAVE` Widgets, refer to . For more information about how to write an application program based on the `PV-WAVE` Widget Toolbox, refer to .

---

## WgMovieTool Procedure

Creates a window that cycles through a sequence of images.

### Usage

WgMovieTool, *image\_data* [, *parent* [, *shell* [, *windowid* [, *rate*]]]]

### Input Parameters

*image\_data* — A 3D array of images (byte). The dimensions of the array are (*m*, *n*, *n\_frames*), where (*m,n*) is the size of an individual image, and *n\_frames* is the total number of frames in the sequence.

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgMovieTool runs on its own (i.e., in its own event loop).

*rate* — (optional) Minimum cycling speed, specified in frames per second; the exact cycling speed varies depending on system load. (This number corresponds to the numeral “1” underneath the speed slider.) By default, under “optimum” conditions, the rate is 30 frames per second (if *Pixmap* is present and nonzero) or 3 frames per second (if *Pixmap* is not supplied or is equal to zero).

### Output Parameters

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

*windowid* — (optional) The window ID of the PV-WAVE graphics window. (For information on window IDs, see the description for the [WINDOW](#) procedure.)

### Keywords

*Cmap* — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

*Dims* — A three-element vector specifying the size of the images to be read from the file, and the number of images to read. The elements of the vector are [*m*, *n*, *n\_frames*], where (*m,n*) is the size of an individual image, and *n\_frames* is the total number of frames in the sequence.

*Do\_tvsc1* — Indicates that TVSCL should be used to scale the image values to the current color table.

- 1 Specifies TVSCL is used to scale the image.

0 Specifies TV is used instead (no scaling).

**File** — A string containing the name of the file from which the image data is read. When using the *File* keyword, the *Dims* keyword must also be supplied. If present, the input variable *image\_data* is ignored.

**Maximum** — The maximum cycling speed, specified in frames per second; the exact cycling speed varies depending on system load. (This number corresponds to the label to the right of the speed slider.) If *Maximum* is not specified, the maximum rate is 100 frames per second (if *Do\_tvscf* is not supplied or is equal to zero) or 10 frames per second (if *Do\_tvscf* is present and nonzero).

**Order** — The order in which the image is drawn. If present and nonzero, the image is inverted. In other words, the image is drawn from bottom to top instead of from top to bottom.

**Pixmap** — Indicates that pixmaps should be used for the animation.

1 Specifies pixmaps are used.

0 Specifies the data is stored in a variable.

Using pixmaps dramatically improves the speed of the animation, but requires more memory from the X Window System's X server than when data stored in variables is used.

**Popup** — If present and nonzero, the MovieTool widget is displayed in its own Main window.

**Position** — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the MovieTool window (long integer). The elements of the vector are [*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are either: 1) if *parent* is present and the *Popup* keyword is not specified or has a value of zero, measured from the upper-left corner of a layout (container) widget or 2) if the *Popup* keyword is present and nonzero (regardless of whether *parent* is present and/or nonzero), measured from the upper-left corner of the screen.

**Size** — A two-element vector specifying the width and height of the display area (long integer). If not specified, the default size of the display area is *m*-by-*n* pixels, where *m* and *n* are defined by *image\_data*.

**Title** — A string containing the title that appears in the header of the MovieTool window. Default value is "Movie Tool".

**View** — A two-element vector specifying the width and height of the viewport onto the display area (long integer). If not specified, the default size of the viewport is  $m$ -by- $n$  pixels, where  $m$  and  $n$  are defined by *image\_data*.

### **Color/Font Keywords**

For additional information on the color and font keywords, see .

**Background** — Specifies the background color name.

**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

### **Attachment Keywords**

For additional information on attachment keywords, see *Form Layout: Attachments* in the *PV-WAVE Application Developer's Guide*.

**Bottom** — If a widget ID is specified (for example, `Bottom=wid`), then the bottom of the color bar widget is attached to the top of the specified widget. If no widget ID is specified (for example, `/Bottom`), then the bottom of the movie widget is attached to the bottom of the parent widget.

**Left** — If a widget ID is specified (for example, `Left=wid`), then the left side of the movie widget is attached to the right side of the specified widget. If no widget ID is specified (for example, `/Left`), then the left side of the movie widget is attached to the left side of the parent widget.

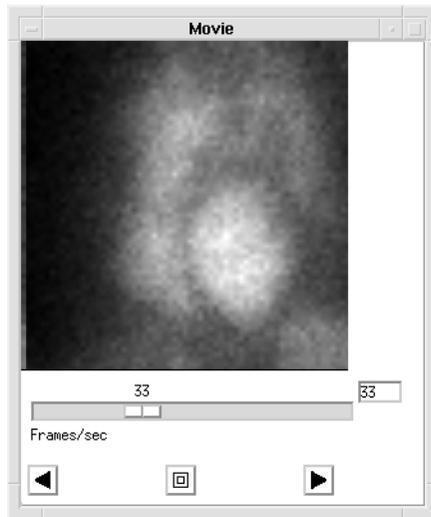
**Right** — If a widget ID is specified (for example, `Right=wid`), then the right side of the movie widget is attached to the left side of the specified widget. If no widget ID is specified (for example, `/Right`), then the right side of the movie widget is attached to the right side of the parent widget.

**Top** — If a widget ID is specified (for example, `Top=wid`), then the top of the movie widget is attached to the bottom of the specified widget. If no widget ID is specified (for example, `/Top`), then the top of the movie widget is attached to the top of the parent widget.

## **Discussion**

`WgMovieTool ()` is an interactive window that lets you use the mouse to control the pace and direction of an animated series of images.

Using the `WgMovieTool` window is similar in many ways to using the `WgAnimateTool` window, but `WgMovieTool` is designed so that it can be included inside larger layout widgets, while `WgAnimateTool` is intended to be used as a stand-alone utility widget. Be sure to familiarize yourself with the attachment keywords if you plan to place `WgMovieTool` inside a larger layout widget.



**Figure 2-83** `WgMovieTool` creates an interactive window that lets you use the mouse to control the pace and direction of an animated series of images.

`MovieTool` provides a graphical user interface (GUI) to a Standard Library procedure, `movie.pro`. Unlike the blocking behavior that you encounter with `movie.pro`, you can interact with other windows while `MovieTool` is open and running.

## Event Handling

You can use the `MovieTool` widget in one of the following three ways:

- **From the `WAVE>` prompt** — Enter the procedure name at the `WAVE>` prompt to display the `MovieTool` widget. The `MovieTool` widget handles its own event loop by calling `WwLoop`.
- **Stand-alone widget in its own window created by another application** — If the `Popup` keyword is present and nonzero, the `MovieTool` widget has its own Main window. The application (not the `MovieTool` widget) handles the event loop by calling `WwLoop`.

- **Combined with other widgets in a layout widget** — Another application combines the MovieTool widget with other widgets inside a layout widget. The application (not the MovieTool widget) handles the event loop by calling WwLoop.

When the input parameter *parent* is present and nonzero, the widget operates in either the second or third mode listed, depending on the value of the *Popup* keyword. The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

You can use the output parameter *windowid* to keep track of the PV-WAVE window ID that is assigned to the MovieTool. You can create multiple instances of the MovieTool; each one will be assigned a different PV-WAVE window ID.

## Contents of the Window

The MovieTool window has two main parts — the display area and the control area.

### ***MovieTool Display Area***

The display area is the largest area of the window; it is where the animation sequence takes place.

### ***MovieTool Control Area***

Use the following controls to operate the MovieTool window:

- **Frames/sec slider** — Decrease or increase the rate of display. The number shown to the left of the slider is the minimum cycling speed for the animation sequence; this number is controlled with the *Rate* keyword. The number shown to the right of the slider is the maximum cycling speed for the animation sequence; this number is controlled with the *Maximum* keyword.
- **Left-pointing arrow** — Cycle through the images from last to first.
- **Stop button** — Freeze display at the current image.
- **Right-pointing arrow** — Cycle through the images from first to last.

---

**NOTE** Use the window manager menu of the window frame to dismiss the MovieTool window from the screen.

---

## Example

Enter the commands shown below into a file, and compile the procedure with the `.RUN` command. If the variable `parent` is defined, `WgMovieTool` is created as a child of `parent`; otherwise, `WgMovieTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgMovieTool` window, close it using the window manager menu.

```
PRO Sample_wgmovietool, parent, tool_shell
heart = BYTARR(256, 256, 15)
    ; Define a variable to hold the data.
IF !Version.platform EQ 'VMS' THEN $
    OPENR, u, GETENV('WAVE_DIR')+
    '[data]heartbeat.dat', /Get_lun $
ENDIF ELSE BEGIN
OPENR, u, !Dir+'/data/heartbeat.dat', /Get_lun
ENDELSE
READU, u, heart
    ; Read the heartbeat.dat file that contains images showing a
    ; beating human heart.
CLOSE, u
FREE_LUN, u
    ; Close the file and free the LUN.
IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgMovieTool, heart, parent, tool_shell, $
    /Do_tvsc1, /Pixmap, /Popup
    ; Create WgMovieTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
    ; output parameter "tool_shell".
ENDIF ELSE BEGIN
WgMovieTool, heart, /Do_tvsc1, /Pixmap
    ; Create WgMovieTool and display it as its own Main window. In other
    ; words, the WgMovieTool window runs on its own (i.e., in its own
    ; event loop).
ENDELSE
END
```

## See Also

[MOVIE](#), [WgAnimateTool](#)

For more information about how to transfer image data to variables, refer to in Chapter 8 of the *PV-WAVE Programmer's Guide*.

For more information about pixmaps, refer to [Appendix B, Output Devices and Window Systems](#).

For more information about color table indices, refer to in Chapter 11 of the *PV-WAVE User's Guide*.

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

---

## **WgOrbit Procedure**

Creates an interactive window for viewing objects.

### **Usage**

WgOrbit, *vertices*, *polygons*, *parent*, *shell*

### **Input Parameters**

*vertices* — A (3,n) array of points on the surfaces of the objects.

*polygons* — A vector defining polygons which describe the surfaces: it is a concatenation of vectors of the form [*m*, *i*<sub>1</sub>, ..., *i*<sub>*m*</sub>] where *m* is the number of vertices defining a polygon and where *vertices* (\*, *i*<sub>1</sub>), ..., *vertices* (\*, *i*<sub>*m*</sub>) are those vertices arranged in counter-clockwise order when viewed from outside the object.

*parent* — (optional) The widget ID of the parent widget.

### **Output Parameters**

*shell* — (optional) The ID of the newly created widget.

### **Keywords**

*position* — A two-element vector positioning the widget's upper-left corner (measured in pixels from the upper-left corner of the screen).

*shades* — A vector specifying the color for each vertex.

*size* — Two-element vector specifying window size. The default is [500,500].

*title* — A string specifying the title for the widget.

*wid* — (output) The window ID of the graphics window.

## Examples

```
POLY_SPHERE, 1, 10, 10, v, p  
v(1:2,*) = [ 2*v(1,*), 3*v(2,*) ]  
WgOrbit, v, p
```

---

## WgSimageTool Procedure

Creates two windows: 1) a scrolling image window and 2) an optional smaller window that shows a reduced view of the entire image.

### Usage

WgSimageTool, *image\_data* [, *parent* [, *shell* ]]

### Input Parameters

*image\_data* — A 2D byte variable containing image data for a single image.

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgSimageTool runs on its own (i.e., in its own event loop).

### Output Parameters

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

### Keywords

*Do\_tvsc1* — Indicates that TVSCL should be used to scale the image data before it is displayed.

- 1 Specifies TVSCL is used to scale the image.
- 0 Specifies TV is used to display the data (not scaled).

*Dsize* — A two-element vector specifying the width and height of the image to be displayed in the scrolling area (long integer). Default is for *Dsize* to be the same as *image\_data* in both directions. If *Dsize* is larger than *Wsize*, you will be able to use scroll bars to move the image around in the display window. An error occurs if *Dsize* is smaller than the size of *image\_data* in either direction.

---

**NOTE** When *Dsize* is greater than *Wsize*, and scroll bars are placed on the right and at the bottom of the scrolled image window, then 12 pixels are subtracted in each direction (from the displayed image area) to allow room for the scroll bars.

---

***Noreduce*** — If present and nonzero, the reduced size window that is used to display the entire image is not displayed. Only the larger of the two windows is displayed.

***Order*** — The order in which the image is drawn. If present and nonzero, the image is inverted. In other words, the image is drawn from bottom to top instead of from top to bottom.

***Position*** — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the *SimageTool* window (long integer). The elements of the vector are [*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

***Reduced\_size*** — A two-element vector specifying the width and height of the image display area of the reduced size window that displays the entire image (long integer). The reduced size image is computed using CONGRID; if *Reduced\_size* is not specified, the default size of the reduced image is 256-by-256. This keyword is ignored if *Noreduce* is present and nonzero.

***Rtitle*** — A string containing the title that appears in the header of the small window that is used to display the entire image. Default value is “Reduced Image”.

***Title*** — A string containing the title that appears in the header of the window that is used to display the scrolled image. Default value is “Full Size Image”.

***Wsize*** — A two-element vector specifying the width and height of the image display area of the scrolled image window (long integer). Default display area size is either: 1) 512-by-512, or 2) the size of *image\_data*, whichever is less.

### ***Color/Font Keywords***

For additional information on the color and font keywords, see .

***Background*** — Specifies the background color name.

***Basecolor*** — Specifies the base color.

***Font*** — Specifies the name of the font used for text.

***Foreground*** — Specifies the foreground color name.

## Discussion

WgSimageTool () provides a convenient way to view images, either full-size or reduced-size. Scroll bars are provided for viewing large images when they are viewed at full size.

## Event Handling

You can use the SimageTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the SimageTool widget. The SimageTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The SimageTool widget has its own Main window, but the application (not the SimageTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

## Contents of the Window

The larger of the two SimageTool windows has two main parts — the display area and the control area.

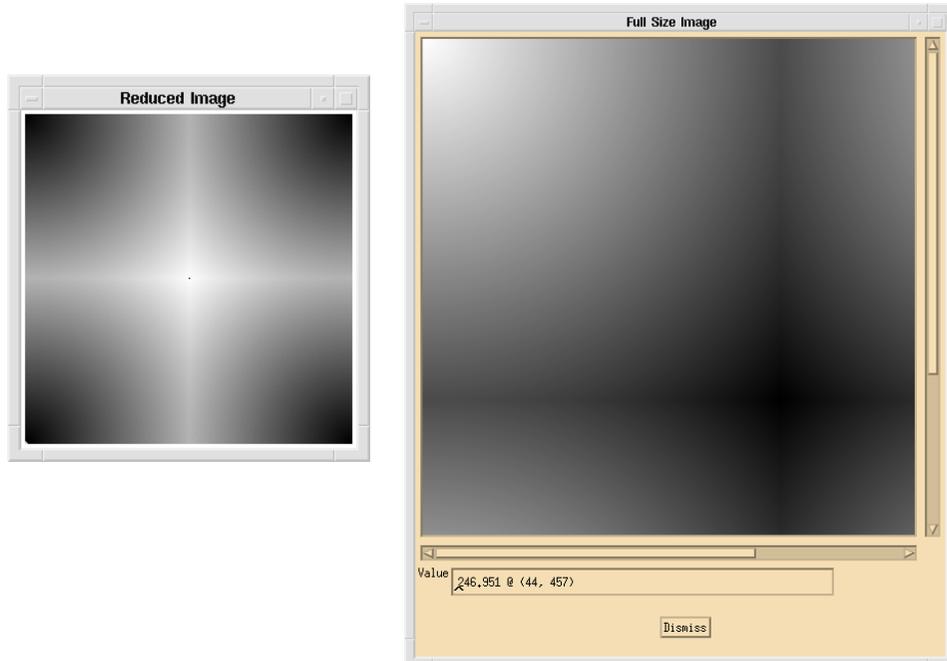
### ***SimageTool Display Area***

The display area is the largest area of the window; it is where the full-size image is displayed.

### ***SimageTool Control Area***

Use the following controls to operate the SimageTool window:

- **Value** — If the pointer is within the boundaries of the image, its location is echoed in this area, along with the value of the image data directly underneath the pointer.
- **Dismiss** — Destroy the SimageTool window(s) and erase it (them) from the screen.



**Figure 2-84** By default, WgSimageTool displays two windows. One window displays the image at full-size, and adds scroll bars to the right and bottom for viewing other portions of the image. The other window displays the image at a reduced size (256-by-256 by default) and does not offer the option of scroll bars. Use the *Noreduce* keyword to suppress the reduced-size window and display only the larger window.

## Example

Enter the commands shown below into a file, and compile the procedure with the `.RUN` command. If the variable `parent` is defined, WgSimageTool is created as a child of `parent`; otherwise, WgSimageTool runs on its own (i.e., in its own event loop).

When you are finished interacting with the WgSimageTool window, close it by clicking on the **Dismiss** button.

```

PRO Sample_wgsimagetool, parent, tool_shell
x = DIST(7000)
    ; Create a "dummy" variable to use as data for WgSimageTool.

IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgSimageTool, x, parent, tool_shell, /Do_tvsc1
    ; Create WgSimageTool as a child of the widget known as "parent".

```

```

; The window of the newly created widget is returned via the optional
; output parameter "tool_shell".

ENDIF ELSE BEGIN

WgSimageTool, x, /Do_tvsc1
; Create WgSimageTool and display it as its own Main window. In
; other words, the WgSimageTool window runs on its own (i.e., in its
; own event loop).

ENDELSE
END

```

## See Also

[TV](#), [TVSCL](#)

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

## WgSliceTool Procedure

Creates a window with a built-in set of controls; these controls allow you to easily select and view “slices” from a three-dimensional block of data.

### Usage

WgSliceTool, *block\_data* [, *parent* [, *last\_slice* [, *shell* ]]]

### Input Parameters

***block\_data*** — A 3D variable containing volumetric data. The dimensions of *block\_data* define the size of the cube that you see when the SliceTool window is first opened.

***parent*** — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgSliceTool runs on its own (i.e., in its own event loop).

### Output Parameters

***last\_slice*** — (optional) The two-dimensional set of values describing the most recent slice that was displayed. The data type of *last\_slice* is the same as the data type for *block\_data*.

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

## Keywords

*Cmap* — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

*Cube\_color* — The color in which the rotatable cube is drawn; a positive integer in the range (0...255). The number that is provided for *cube\_color* is used as an index into the current color table.

*Position* — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the SliceTool window (long integer). The elements of the vector are [*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

*Slice\_color* — The color in which the slice is drawn; a positive integer in the range (0...255). The number that is provided for *slice\_color* is used as an index into the current color table.

*Slice\_rot* — A three-element vector specifying the XYZ slicing plane rotation, in degrees. If not supplied, the initial rotation is [0, 0, 0] — in other words, no rotation.

*Slice\_trans* — A three-element vector specifying the initial slicing plane translation, in degrees. If not supplied, the initial translation is [0, 0, 0] — in other words, no translation.

*Title* — A string containing the title that appears in the header of the SliceTool window. Default value is “Slicer Tool”.

*View\_persp* — The initial perspective distance. If *View\_persp* is zero, then the initial setting on the **Perspective Distance** slider is 0.5, but the **Perspective** pushbutton is initially deselected. If *View\_persp* is nonzero, then the initial setting on the **Perspective Distance** slider is the value specified by *View\_persp* and the **Perspective** pushbutton is selected.

---

**NOTE** Using no perspective to draw the cube is equivalent to having the eyepoint an infinite distance away from the cube, and will produce a cube with sides that all appear to be parallel to one another.

---

*View\_rot* — A three-element vector specifying the initial view rotation, in degrees. If not supplied, the initial rotation of the cube is [30, 30, 0] — in other words, 30

degrees rotation around the *x*-axis, 30 degrees rotation around the *y*-axis, and 0 degrees rotation around the *z*-axis.

***View\_zoom*** — The initial zoom factor. (Default: 0.5)

### ***Color/Font Keywords***

For additional information on the color and font keywords, see .

***Background*** — Specifies the background color name.

***Basecolor*** — Specifies the base color.

***Font*** — Specifies the name of the font used for text.

***Foreground*** — Specifies the foreground color name.

## **Discussion**

WgSliceTool () is an interactive window that lets you use the mouse to view and modify the location of a slice that bisects a volume of data; you can position the slice anywhere within the three-dimensional input dataset, `surface_data`.

If the zoom factor is large, or the perspective parameter is small, then the cube display in the View orientation area may be erroneous. To cure the problem, reduce the zoom or increase the perspective (or disable the perspective entirely).

To select a slice, adjust the sliders that control perspective, zoom, and orientation of the slice in relationship to the cube. When you are satisfied with your selections, click the **Apply** button.

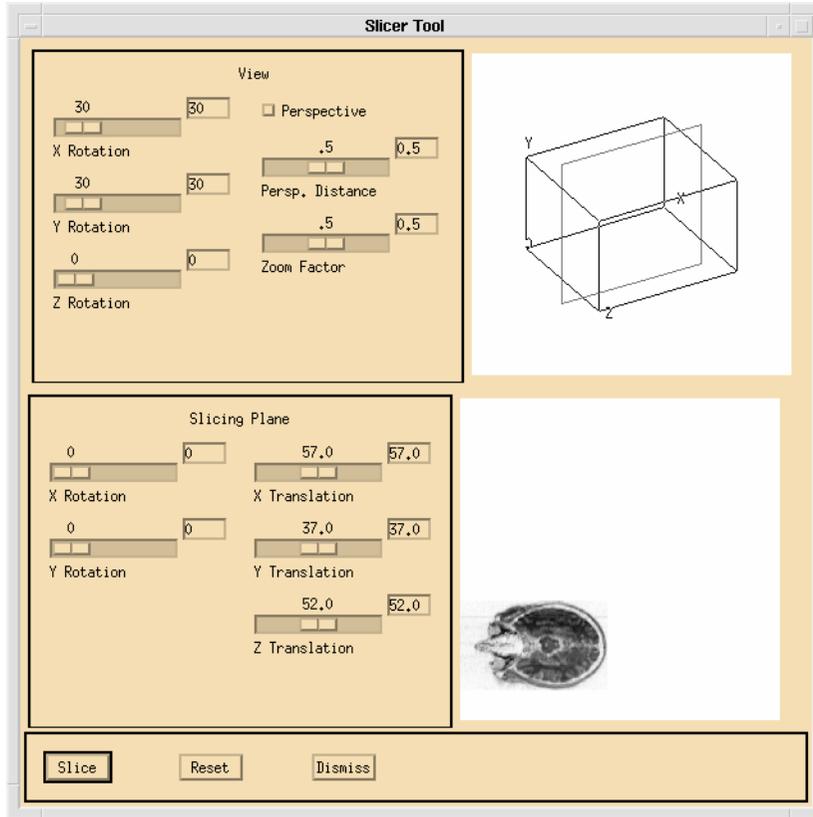
Slices of data are displayed “head-on”. In other words, the data slice is not projected into 3D space — the data displayed is the actual 2D slice that you selected.

## **Event Handling**

You can use the SliceTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the SliceTool widget. The SliceTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The SliceTool widget has its own Main window, but the application (not the SliceTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.



**Figure 2-85** WgSliceTool is an interactive window that lets you use the mouse to view and modify the location of a slice that bisects a volume of data; your selections in the lower left part of the window affect the position of the slice.

### ***SliceTool Display Area***

The display area is towards the right of the window; it is divided into an upper and lower region:

- **View orientation area** — The cube establishes a frame of reference for positioning the slice of data.
- **Data slice area** — This region contains an image displaying the actual slice of data that you have selected for viewing.

The orientation of the cube provides a frame of reference for selecting and understanding the slice of data that you have chosen to view.

## ***SliceTool Control Area***

Use the following controls to operate the SliceTool window:

- **X, Y, and Z Rotation** — These controls allow you to rotate the cube (counterclockwise around the desired axis) a specified number of degrees. The current rotation is shown in the text field to the right of the slider. To modify the rotation, either enter a new value in one of the text fields, or use the left mouse button to drag one of the sliders. If you enter a new value into a text field, press <Return> to apply the new value to the slider and the surface. If you use the sliders, the change is applied immediately as the slider moves.
- **Perspective** — If enabled, the cube is drawn with perspective; if disabled, the cube is drawn without perspective.
- **Perspective Distance** — Controls the amount of perspective used when drawing the cube; in other words, how close the eyepoint is to the cube. The closer the eyepoint gets to the cube, the greater the amount of perspective exaggeration that is used to draw the cube.
- **Zoom Factor** — Controls the amount of magnification used to draw the cube.
- **Slice** — Redraw the cube and the data slice with the specified viewing parameters.
- **Reset** — Return to the default viewing parameters and the initial orientation and positioning of the slice.
- **Dismiss** — Destroy the SliceTool window and erase it from the screen.

## **Example**

Enter the commands shown below into a file, and compile the procedure with the .RUN command. If the variable `parent` is defined, `WgSliceTool` is created as a child of `parent`; otherwise, `WgSliceTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgSliceTool` window, close it by clicking on the **Dismiss** button.

```
PRO Sample_wgslicetool, parent, tool_shell
head = BYTARR(115, 75, 105)
    ; Define a variable to hold the data.
IF !Version.platform EQ 'VMS' THEN $
    OPENR, u, GETENV('WAVE_DIR')+
    '[data]man_head.dat', /Get_lun $
ELSE $
    OPENR, u, '$WAVE_DIR/data/man_head.dat', $
```

```

        /Get_lun
READU, u, head
    ; Read the man_head.dat file that contains three-dimensional
    ; volumetric data.

CLOSE, u
FREE_LUN, u
    ; Close the file and free the LUN.

IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgSliceTool, head, slice, parent, tool_shell
    ; Create WgSliceTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
    ; output parameter "tool_shell".

ENDIF ELSE BEGIN

WgSliceTool, head, slice
    ; Create WgSliceTool and display it as its own Main window. In other
    ; words, the WgSliceTool window runs on its own (i.e., in its own event loop).

ENDELSE
END

```

## See Also

[SLICE\\_VOL](#), [VIEWER](#)

For information about other approaches to slicing volumes, see , in the *PV-WAVE User's Guide*. This chapter includes an example showing how to render selected slices from a large amount of volume data.

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .



specified y variables. If not specified, the default size of the display area is 512-by-512 pixels.

**Title** — A string containing the title that appears in the header of the window. Default value is “Stripchart Tool”.

### ***Color/Font Keywords***

For additional information on the color and font keywords, see .

**Background** — Specifies the background color name.

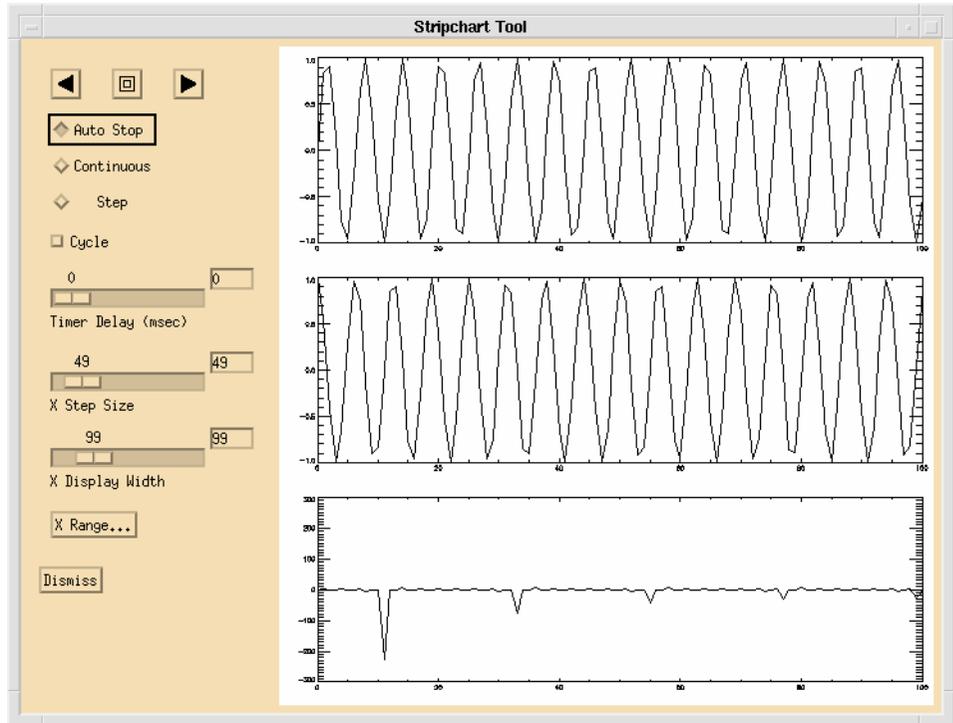
**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

### **Discussion**

The WgStripTool window () allows you to view data in moving strip charts; you can adjust the display width in the strip charts to create the effect of zooming in and out. You can also adjust the rate and the manner with which the strip charts progress through the data.



**Figure 2-86** The WgStripTool window lets you interactively view data in moving strip charts; up to ten strip charts can be simultaneously displayed by the WgStripTool window. You can adjust the display width in the strip charts to create the effect of zooming in and out. You can also adjust the rate and the manner with which the strip charts progress through the data.

## Event Handling

You can use the WgStripTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the WgStripTool widget. The WgStripTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The WgStripTool widget has its own Main window, but the application (not the WgStripTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

## Contents of the Window

The StripTool window has two main parts — the display area and the control area.

### ***StripTool Display Area***

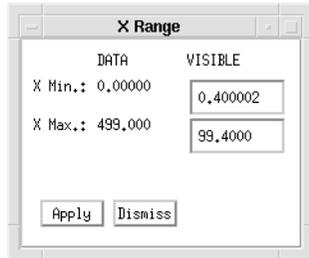
The display area is the largest area of the window; it is where the data strip charts are displayed. The number of strip charts you see in this area is determined by the number of *y* variables that were included in the call that invoked the WgStripTool window.

### ***StripTool Control Area***

Use the following controls to operate the WgStripTool window:

- **Reverse button** — Cycle through the data in a reverse direction.
- **Stop button** — Freeze strip chart display at current location.
- **Forward button** — Cycle through the data in a forward direction.
- **Mode** — Select **Auto Stop** to have WgStripTool stop whenever it reaches the beginning or the end of the data in the *y* variable(s). Select **Continuous** to have WgStripTool move through the data in a non-stop manner, pausing only when you click the **Stop** button. Select **Step** to have the strip chart proceed only when you click on one of the arrow buttons.
- **Timer Delay** — Decrease or increase the rate of display. The number shown to the right of the slider is the number of milliseconds delay between contiguous frames of the strip chart.
- **X Step Size** — The amount by which the strip charts are moved forward or back when the strip charts are updated.
- **X Display Width** — The range of data displayed in the horizontal direction. A small value for display width has the effect of “zooming in” to view a narrow range of data; a large value for display width has the effect of “zooming out” to view a broader range of data.
- **X Range** — A more precise way to specify **X Display Width**. Click this button to display another dialog box containing text fields where you can type precise values for the minimum and maximum values you wish to have displayed along the *x*-axis.  
  
This other dialog box is shown in .
- **Dismiss** — Destroy the WgStripTool window and erase it from the screen.

The selections you make in the control area affect all displayed strip charts. This way, you can be assured that the data in one strip chart is in “sync” with the data in any other strip chart.



**Figure 2-87** Use this dialog box to precisely control the range of the x-axis in the WgStripTool window. This dialog box is displayed if you click the X Range button in the WgStripTool control area.

## Example

Enter the commands shown below into a file, and compile the procedure with the .RUN command. If the variable `parent` is defined, WgStripTool is created as a child of `parent`; otherwise, WgStripTool runs on its own (i.e., in its own event loop).

When you are finished interacting with the WgStripTool window, close it by clicking on the **Dismiss** button.

```

PRO Sample_wgstriptool, parent, tool_shell
x = indgen(500)
y1 = sin(x)
y2 = cos(x)
y3 = tan(x)
    ; Create an independent variable and three dependent variables
    ; describing sinusoidal curves to use as data for WgStripTool.

IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgStripTool, x, y1, y2, y3, $
    0, 0, 0, 0, 0, 0, 0, parent, tool_shell
    ; Create WgStripTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the
    ; optional output parameter "tool_shell".

ENDIF ELSE BEGIN

WgStripTool, x, y1, y2, y3
    ; Create WgStripTool and display it as its own Main window. In other

```

```
; words, the WgStripTool window runs on its own (i.e., in its own event
; loop). Notice how the zeroes are not needed as placeholders, since
; the optional parameters "parent" and "tool_shell" are not being used
; in this procedure call.
```

```
ENDELSE
```

```
END
```

## See Also

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

---

## WgSurfaceTool Procedure

Creates a surface window with a built-in set of controls: these controls allow you to interactively modify surface parameters and view the result of those modifications.

### Usage

WgSurfaceTool, *surface\_data* [, *parent* [, *shell* ]]

### Input Parameters

*surface\_data* — A 2D variable containing surface data.

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgSurfaceTool runs on its own (i.e., in its own event loop).

### Output Parameters

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

### Keywords

*Auto\_congrid* — If present and nonzero, the *z* variable is resized to 50-by-50 (with CONGRID) prior to drawing the surface. This enables PV-WAVE to draw the surface much faster, although there is a chance that some surface detail is lost.

*Auto\_redraw* — If present and nonzero, the surface is automatically redrawn any time one of the surface parameters is adjusted.

**Cmap** — The index of the color table to load when the widget is created; a positive integer in the range (0...15).

**Elevation** — If present and nonzero, the surface is drawn using simple elevation shading. *Elevation* and *Gouraud* are mutually exclusive.

**Gouraud** — If present and nonzero, the surface is drawn using Gouraud shading. *Gouraud* and *Elevation* are mutually exclusive.

**Lines** — If present and nonzero, lines are drawn instead of a grid.

**Lower** — If present and nonzero, only the lower portion of the grid is drawn. *Lower* and *Upper* are mutually exclusive keywords.

**Nogrid** — If present and nonzero, no grid is drawn on the surface.

**Position** — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the SurfaceTool window (long integer). The elements of the vector are [*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

**Skirt** — If present and nonzero, a skirt is drawn connecting the surface to the *x*- and *y*-axes. Refer to for an example of a surface with and without a skirt.

**Title** — A string containing the title that appears in the header of the SurfaceTool window. Default value is “Surface Tool”.

**Upper** — If present and nonzero, only the upper portion of the grid is drawn. *Upper* and *Lower* are mutually exclusive keywords.

**Xrot** — The initial counter-clockwise rotation of the surface around the *x*-axis, measured in degrees.

**Zrot** — The initial counter-clockwise rotation of the surface around the *z*-axis, measured in degrees.

### **Color/Font Keywords**

For additional information on the color and font keywords, see .

**Background** — Specifies the background color name.

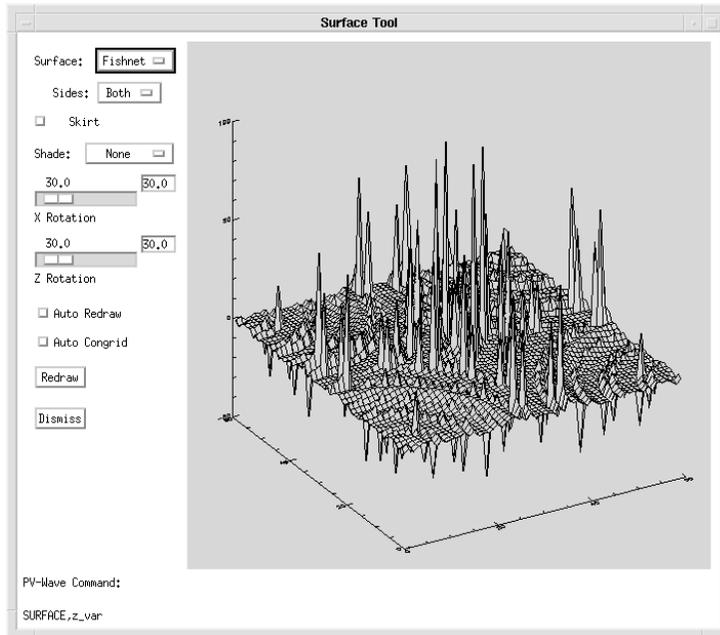
**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

## Discussion

WgSurfaceTool () is an interactive window that lets you use the mouse to control the orientation and appearance of a displayed surface.



**Figure 2-88** WgSurfaceTool creates an interactive window that lets you use the mouse to control the orientation and appearance of a displayed surface.

## Event Handling

You can use the SurfaceTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the SurfaceTool widget. The SurfaceTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The SurfaceTool widget has its own Main window, but the application (not the SurfaceTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

## Contents of the Window

The SurfaceTool window has three main parts — the display area, the control area, and the message area.

### ***SurfaceTool Display Area***

The display area is the largest area of the window; it is where the surface is displayed.

### ***SurfaceTool Control Area***

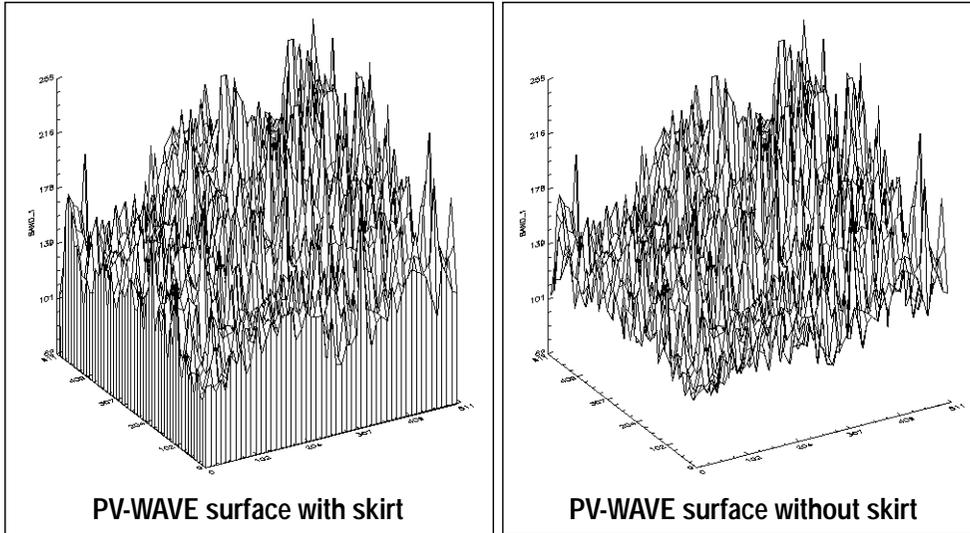
Use the following controls to operate the SurfaceTool window:

- **Surface (menu)** — From the menu, choose the method by which the surface is drawn. The choices are: **Fishnet** (mesh), **Lines** (lines in one direction only), and **None** (no lines appear superimposed on a shaded surface).
- **Sides (menu)** — From the menu, choose the representation of the top and bottom sides of the surface. The choices are: **Both** (lines on both top and bottom), **Upper** (lines on upper surface only), and **Lower** (lines on lower surface only).
- **Skirt** — Controls whether a skirt is added to the surface. A skirt helps establish a frame of reference between the surface and the  $x$ -,  $y$ -, and  $z$ -axes. Refer to for an example of a surface with and without a skirt.
- **Shade (menu)** — From the menu, select the algorithm by which the surface is shaded. The choices are: **None** (no shading), **Gouraud**, and **Elevation**.
- **X and Z Rotation** — These controls allow you to rotate the surface (counterclockwise around the  $x$ - or  $z$ -axis) a specified number of degrees. The current rotation is shown in the text field to the right of the slider. To modify the rotation, either enter a new value in one of the text fields, or use either slider. If you enter a new value into a text field, press <Return> to apply the new value to the slider and the surface. If you use the sliders, the change is applied immediately as the slider moves.
- **Auto Redraw** — Controls whether the contents of the display area are redrawn every time a modification is made to one of the controls in the SurfaceTool control area.
- **Auto Congrid** — The surface is resized to 50-by-50 (with CONGRID) prior to drawing the surface, for faster display of large datasets. The default is for **Auto Congrid** to be enabled. **Auto Congrid** does not affect the actual data; it only affects the display of the data.
- **Redraw** — Causes the contents of the SurfaceTool display area to be redrawn.

- **Dismiss** — Destroy the SurfaceTool window and erase it from the screen.

### **SurfaceTool Message Area**

The message area displays the PV-WAVE command that is being used to display the surface.



**Figure 2-89** A surface with and without a skirt.

### **Example**

Enter the commands shown below into a file, and compile the procedure with the .RUN command. If the variable `parent` is defined, `WgSurfaceTool` is created as a child of `parent`; otherwise, `WgSurfaceTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgSurfaceTool` window, close it by clicking on the **Dismiss** button.

```
PRO Sample_wgsurfacetool, parent, tool_shell
x = DIST(75)
    ; Create a "dummy" variable to view as view as a surface using WgSurfaceTool.

IF N_ELEMENTS(parent) NE 0 THEN BEGIN
WgSurfaceTool, x, parent, tool_shell
    ; Create WgSurfaceTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
```

```
        ; output parameter "tool_shell".
ENDIF ELSE BEGIN
WgSurfaceTool, x
    ; Create WgSurfaceTool and display it as its own Main window.
    ; In other words, the WgSurfaceTool window runs on its own (i.e., in
    ; its own event loop).
ENDELSE
END
```

## See Also

[SHADE\\_SURF](#), [SURFACE](#)

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

---

## **WgTextTool Procedure**

Creates a scrolling window for viewing text from a file or character string.

### **Usage**

WgTextTool [, *parent* [, *shell* ]]

### **Input Parameters**

*parent* — (optional) The widget or shell ID of the parent widget (long). If *parent* is not specified, WgTextTool runs on its own (i.e., in its own event loop).

---

**NOTE** Either the *File* or the *Text* keyword must be supplied to identify the text that WgTextTool will display.

---

### **Output Parameters**

*shell* — (optional) The ID of the newly created widget. If the procedure fails, zero (0) is returned.

### **Keywords**

*Cols* — The number of text columns to display when the window is first created.

*File* — The name of the file to display. The *File* and *Text* keywords are mutually exclusive, and one of them must be used.

*Position* — A two-element vector specifying the *x*- and *y*-coordinates of the upper-left corner of the TextTool window (long integer). The elements of the vector are [*x*, *y*], where *x* (horizontal) and *y* (vertical) are specified in pixels. These coordinates are measured from the upper-left corner of the screen.

*Rows* — The number of text rows to display when the window is first created.

*Text* — A string containing the text that will be displayed. The *Text* and *File* keywords are mutually exclusive, and one of them must be used.

*Title* — A string containing the title that appears in the header of the TextTool window. If not specified, either the name of the file is used for the title, or when TextTool is displaying a string, the value of the title defaults to “Scrolling Window”.

## Color/Font Keywords

For additional information on the color and font keywords, see .

**Background** — Specifies the background color name.

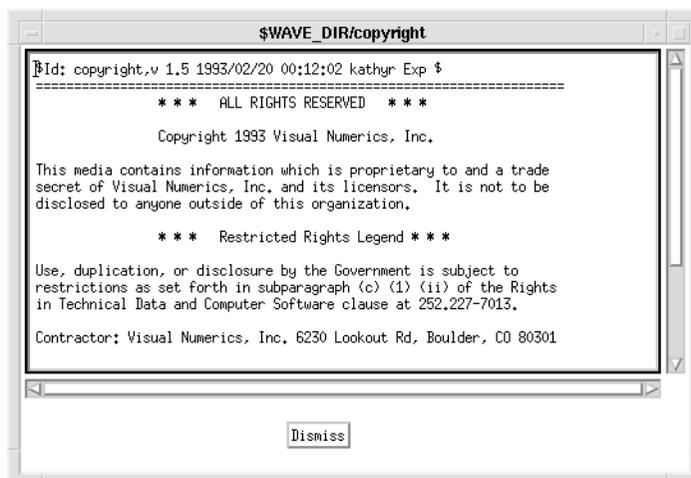
**Basecolor** — Specifies the base color.

**Font** — Specifies the name of the font used for text.

**Foreground** — Specifies the foreground color name.

## Discussion

WgTextTool is a window () that lets you view text from a string or from a file.



**Figure 2-90** WgTextTool displays either: 1) text from a file, or 2) text from an input string. In this illustration, WgTextTool is displaying text from the copyright file that is distributed with every copy of PV-WAVE.

---

**NOTE** The text displayed in the window is read only; it cannot be edited.

---

## Interacting with the Window

The standard text editing functions are available. In Motif, this means the left mouse button and the middle mouse button can be used to copy text. However, in the text edit popup menu, only the **Copy** function will be sensitized, because the text is read-only.

Use the **Dismiss** button (or the TextTool window's window manager menu) to destroy the window when you are finished viewing the text.

## Event Handling

You can use the TextTool widget in one of the following two ways:

- **From the WAVE> prompt** — Enter the procedure name at the WAVE> prompt to display the TextTool widget. The TextTool widget handles its own event loop by calling WwLoop.
- **Stand-alone widget in its own window created by another application** — The TextTool widget has its own Main window, but the application (not the TextTool widget) handles the event loop by calling WwLoop.

The output parameter *shell* can be returned only if you also supply the input parameter *parent*.

## Example

Enter the commands shown below into a file, and compile the procedure with the .RUN command. If the variable `parent` is defined, `WgTextTool` is created as a child of `parent`; otherwise, `WgTextTool` runs on its own (i.e., in its own event loop).

When you are finished interacting with the `WgTextTool` window, close it by clicking the **Dismiss** button.

```
PRO Sample_wgtexttool, parent, tool_shell
IF !Version.platform EQ 'VMS' THEN BEGIN
filename = GETENV('WAVE_DIR')+'copyright'
    ; Specify the pathname and filename using OpenVMS notation.
ENDIF ELSE BEGIN
filename = '$WAVE_DIR/copyright'
    ; Specify the pathname and filename using UNIX notation.
ENDELSE

IF N_ELEMENTS(parent) NE 0 THEN BEGIN

WgTextTool, File=filename, parent, tool_shell
    ; Create WgTextTool as a child of the widget known as "parent".
    ; The window of the newly created widget is returned via the optional
    ; output parameter "tool_shell".

ENDIF ELSE BEGIN

WgTextTool, File=filename
    ; Create WgTextTool and display it as its own Main window. In other
```

```
        ; words, the WgTextTool window runs on its own (i.e., in its own event
        ; loop).
ENDELSE
END
```

## See Also

WwText in the *PV-WAVE Application Developer's Guide*

For more information about how to write an application program based on WAVE Widgets, refer to . For more information about how to write an application program based on the PV-WAVE Widget Toolbox, refer to .

---

## WHERE Function

Returns a longword vector containing the one-dimensional subscripts of the non-zero elements of the input array.

### Usage

*result* = WHERE(*array\_expr* [, *count* ])

### Input Parameters

*array\_expr* — The array to be searched. May be of any data type. Both the real and imaginary parts of complex numbers must be zero for the number to be considered zero.

### Output Parameters

*count* — (optional) If present, *count* is converted into a longword integer containing the number of nonzero elements found in *array\_expr*. Must be a named variable.

### Returned Value

*result* — A longword vector containing the one-dimensional subscripts of the non-zero elements of *array\_expr*. The length is equal to the number of nonzero elements in *array\_expr*.

## Keywords

None.

## Discussion

Frequently the result of WHERE is used as a vector subscript to select elements of an array using given criteria.

As a side effect, the system variable !Err is set to the number of nonzero elements. This is for compatibility with previous versions of PV-WAVE. Therefore, it is recommended that the *Count* keyword be used in all new programs, rather than !Err.

## Example 1

The WHERE function can be used to select a range of values in an array. For example:

```
index = WHERE ((array GT 50) AND (array LT 100))
      ; Get the subscripts of those elements greater than 50 and less
      ; than 100.
result = array(index)
      ; Put the selected values into result.
```

## Example 2

If all the elements of *array\_expr* are zero, WHERE returns a scalar integer with a value of -1. If you attempt to use this result as an index into another array, you will get an error message about the subscripts being out of bounds. In some situations, code similar to the following can be used to avoid errors:

```
index = WHERE(array, count)
      ; Use count to get the number of nonzero elements.
IF (count GT 0) THEN result = array(index)
      ; Only subscript the array if it's safe to do so.
```

## Example 3

You can use WHERE to determine where an array of strings matches a specific pattern, or where the array of strings is non-NULL.

```
s = ['this', 'is', 'an', 'array', 'of', 'strings']
PRINT, WHERE(s eq 'an')
      2
s = ['this', '', '', 'array', '', 'strings']
```

```
PRINT, WHERE (s)
      0   3   5
```

## See Also

[QUERY\\_TABLE](#), [SORT](#), [WHEREIN](#)

System Variables: [!Err](#)

For more information, see .

---

## **WHEREIN Function**

Find the indices into an array where the values occur in a second array; keywords yield intersection, union, and complement

### **Usage**

$i = \text{WHEREIN}(a, b [,c])$

### **Input Parameters**

$a$  — An array.

$b$  — An array.

### **Output Parameters**

$c$  — (Optional) The number of indices.

### **Returned Value**

$i$  — The array of indices into  $a$  where the values occur in  $b$ ; if there are no such indices, then a scalar  $-1$  is returned.

### **Output Keywords**

$j$  — The array of indices into  $b$  where the values occur in  $a$ ; if there are no such indices then  $-1$  is returned.

**Intersection** — The vector of elements common to both  $a$  and  $b$ ; if there are no such elements then nothing is returned.

**Union** — the vector of elements in either  $a$  or  $b$ .

***A\_Complement*** — The array of indices into *a* where the values do not occur in *b*; if there are no such indices then -1 is returned.

***B\_Complement*** — The array of indices into *b* where the values do not occur in *a*; if there are no such indices then -1 is returned.

## Example

```
PM, WHEREIN( [0,1,2,3,2], [0,0,4,2,5,5,6,0] )
0
2
4
```

## See Also

[INDEX\\_AND](#), [WHERE](#)

---

## WIN32\_PICK\_FONT Function

Displays a Windows common font dialog.

### Usage

```
font_name = WIN32_PICK_FONT()
```

### Input Parameters

None.

### Returned Value

***fontname*** — The name of the font selected in the form “Face, Point Size [, bold, italic, underline ]” If the Cancel button is pressed a NULL string is returned.

### Keywords

***Default*** — A font specification that is used to initialize the font dialog.

***Device*** — If present and non-zero indicates the selected font should be set as the current hardware font for subsequent WIN32 driver text. Equivalent to using the command “DEVICE, Font=f”.

***Fixed*** — If present and non-zero indicates that only fixed width fonts should be displayed in the font dialog.

**Scalable** — If present and non-zero indicates that only scalable fonts should be displayed in the font dialog.

**TrueType** — If present and non-zero indicates that only TrueType fonts should be displayed in the font dialog.

### Example

```
font_name = WIN32_PICK_FONT(/TrueType, /Device)
```

Displays all TrueType fonts and sets the default font to the selected font.

```
font_name = WIN32_PICK_FONT(Default='Arial, 12, bold')
```

Opens a font dialog displaying Arial, 12 pt. Bold as the default selected font.

### See Also

[WIN32\\_PICK\\_PRINTER](#)

---

## **WIN32\_PICK\_PRINTER Function**

Displays a Windows printer dialog.

### Usage

```
printer_name = WIN32_PICK_PRINTER()
```

### Input Parameters

None.

### Returned Value

*printername* — The name of the printer.

### Keywords

None.

### See Also

[WIN32\\_PICK\\_FONT](#)

---

## **WINDOW Procedure**

Creates a window for the display of graphics or text.

### **Usage**

WINDOW [, *window\_index*]

### **Input Parameters**

*window\_index* — (optional) An integer specifying the index of the newly created window.

If *window\_index* is omitted, 0 is used as the index of the new window.

If the value of *window\_index* specifies an existing window, the existing window is deleted and a new window is created.

---

**NOTE** If a window with index zero (0) is already open and you call WINDOW with no parameters, the original window zero (0) will be deleted. To avoid this, specify a *window\_index* for the new window or use the *Free* keyword.

---

### **Keywords**

**Bitmap** — (Windows Only) Creates a bitmap stored in the display memory rather than a visible window. This invisible window is created with backing store (see the *Retain* keyword). See also [Appendix B, Output Devices and Window Systems](#). (Interchangeable with the *Pixmap* keyword.)

**Colors** — The maximum number of color table indices to be used. This keyword has effect only if it is supplied when the first window is created. Otherwise, PV-WAVE uses all of the available color indices.

To use monochrome windows on a color display, use `Colors=2` when creating the first window.

---

**UNIX and OpenVMS USERS** If the X Window System is being used, a negative value for *Colors* specifies that all but the given number of colors from the shared color table should be allocated.

---

**Free** — If nonzero, creates a window using the largest unused window index. This keyword can be used instead of specifying the *window\_index* parameter.

---

**UNIX and OpenVMS USERS** The default position of the new window is opposite the current window.

---

**Get\_Win\_ID** — Returns the window ID for the window just created. For example:

```
WINDOW, Get_Win_ID=New_Win_ID
```

**Get\_Xpixmap\_ID** — (X Window System only) Returns the X pixmap ID for the pixmap just created. You must use the *Pixmap* keyword to set this ID. For example:

```
WINDOW, Get_Xpixmap_ID=New_Xpixmap_ID, /Pixmap
```

**Get\_Xwin\_ID** — (X Window System only) Returns the X Window ID for the window just created. For example:

```
WINDOW, Get_Xwin_ID=New_Xwin_ID
```

**NoMeta** — (Windows only) Turns metafiles off for the window. Use this keyword when running animations or displaying images.

A metafile is an internal, vector-based record of all the graphics commands sent to a window. By default, a metafile is kept for each window to speed the redrawing of the window when it is resized. The metafile is also used when printing to avoid resolution problems that occur when printing a bitmap image.

**Noretain** — (UNIX/OpenVMS Only) An obsolete keyword (see the *Retain* keyword).

**Noshow** — (Windows only) If present and nonzero, the graphics window is not displayed on the screen.

**Pixmap** — This keyword specifies that the window being created is actually an invisible portion of the display memory called a pixmap. (Interchangeable with the *Bitmap* keyword.)

**Retain** — Specifies how backing store for the window should be handled. Possible values for this keyword are listed below:

- 0 No backing store (same as *Noretain* keyword).
- 1 The server or window system is requested to make the window retained.
- 2 PV-WAVE should provide a backing pixmap and handle the backing store directly (X Window System only).

**Set\_Win\_ID** — (Windows Only) The window associated with the window ID assigned to this keyword for the PV-WAVE window. For example:

WINDOW, Set\_Win\_ID=1234567L

**Set\_Xpixmap\_ID** — (X Window System only) Uses the X pixmap associated with the X Pixmap ID assigned to this keyword for the PV-WAVE pixmap. This keyword forces the *Pixmap* keyword to be set. For example:

WINDOW, Set\_Xpixmap\_ID=1234567L

**Set\_Xwin\_ID** — (X Window System only) Uses the X Window associated with the X Window ID assigned to this keyword for the PV-WAVE window. For example:

WINDOW, Set\_Xwin\_ID=1234567L

**Title** — A scalar string specifying the window's label. If not specified, the window is given a label of the form "WAVE *n*", where *n* is the index number of the window. For example, to create a window with a label of PV-WAVE Graphics:

WINDOW, Title='PV-WAVE Graphics'

**XPos, YPos** — The *x* and *y* positions of the lower-left corner of the new window, specified in device coordinates.

If no position is specified, the position of the window is determined from the value of *window\_index*, using the following rules:

- Window 0 is placed in the upper right-hand corner.
- Even-numbered windows are placed on the top half of the screen and odd-numbered windows are placed on the bottom half.
- Windows with  $window\_index = 4 * i$  (e.g., 0, 4, 8, etc.), and windows with  $window\_index = 4 * i + 1$  (where *i* is any whole number) are placed on the right side of the screen. Windows with  $window\_index = 4 * i + 2$  (e.g., 2, 6, 10, etc.), and windows with  $window\_index = 4 * i + 3$  (where *i* is any whole number) are placed on the left side of the screen.

**XSize** — The width of the window, in pixels. (Default: 640)

**YSize** — The height of the window, in pixels. (Default: 512)

## Discussion

You can create any number of windows; however, valid window indices are governed by the following rules. First, windows are allocated in sets of 32. You always have available to you:

- the windows in the currently available window set that are not already "used up" (beginning with the first set (0..31)) and
- the next available set of 32 windows.

For example, the following two commands are both valid because the first window is created from the first available set of windows (0..31) and the second window is created from the next available set (32..63).

```
WINDOW, 1
    ; Creates window 1 from window set (0..31).
```

```
WINDOW, 63
    ; Creates window 63 from window set (32..63), the next available set.
```

On the other hand, the following commands break the rule and produce an error because the second window created is not in the next available set after (0..31); it is in the set (64..95).

```
WINDOW, 1
WINDOW, 64
    % WINDOW: Window number out of range or no more
    % windows Execution halted at $MAIN$ (WINDOW)
```

You can use the command:

```
DEVICE, Window_State = winarr
```

to determine the number of windows (and sets of windows) that are currently allocated.

You only need to use `WINDOW` if you want to display more than one `PV-WAVE` window simultaneously or to set specific characteristics of windows.

The behavior of `WINDOW` varies slightly depending on the window system in effect. You can use the `DEVICE` procedure to change many of the `WINDOW` procedure's defaults.

The newly created window becomes the current `PV-WAVE` window, and the system variable `!D.Window` is set to the window index associated with it. (See the `WSET` procedure for a discussion of the current `PV-WAVE` window.)

## Example 1

This example creates a single window. The window's ID is zero (0). The output of any graphics commands appear in this window.

```
WINDOW, 0
```

## Example 2 (Windows only)

In this example, graphics are output at a size that exceeds the width and height, respectively, of the actual display device. The `Xsize` and `Ysize` keywords set the width and height and the `Noshow` keyword causes the graphics to be created with-

out displaying them on the screen. The WPRINT command is used to print the graphics. (This example only works under Windows.)

```
WINDOW, 1, XSize=3000, YSize=3000, /Noshow
PLOT, x, y
WPRINT, 1
```

---

**Windows USERS** If you want to see the current contents of window 1 in this example at any time, create a visible window with a smaller size and pass the contents of window 1 to the smaller window using the WCOPY and WPASTE functions. These commands transfer graphics using the Clipboard. For example:

```
status = WCOPY(1)
      ; Copy current window (window 1) to the Clipboard.
WINDOW, 2
      ; Create a visible window that will fit on the screen.
status = WPASTE(2)
      ; Paste the Clipboard contents into the visible window.
WSET, 1
      ; Set the current graphics window back to window 1 and continue.
```

---

## See Also

[ERASE](#), [WDELETE](#), [WSET](#), [WSHOW](#)

System Variables: [!D.Window](#)

For additional information on PV-WAVE graphics devices, see [Appendix B, Output Devices and Window Systems](#).

---

**Windows USERS** For information on the graphics window Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## ***WMENU Function (UNIX/OpenVMS)***

Displays a menu inside the current window whose choices are given by the elements of a string array and which returns the index of the user's response.

### **Usage**

*result* = WMENU(*strings*)

### **Input Parameters**

*strings* — A string array, with each element containing the text of one menu choice. Both the maximum number of elements and the maximum element length are constrained by how large a display area will be used.

### **Returned Value**

*result* — A value ranging from 0 to the number of elements in *strings* minus one. The value -1 is returned, if no menu item was selected.

### **Keywords**

*Initial\_Selection* — The index of the initial selection.

- If this keyword is specified and within the range of *strings* indices, the initial menu display is made with the designated item selected.
- If this keyword is not specified, the menu is initially displayed with the mouse cursor at the immediate left of the first (top) selection.

*Title* — The index of the *strings* element that is the title, normally 0. The title element is not selectable and is displayed reversed and centered. If this keyword is omitted, all items are selectable.

*XPos* — (X Window System only) The position on the *x*-axis of the display device where the menu is to be placed.

*YPos* — (X Window System only) The position on the *y*-axis of the display device where the menu is to be placed.

### **Discussion**

WMENU can be used only with X Window System displays.

To use, select a menu item with the mouse and click the left mouse button.

## Example

The following statement displays a menu containing the selections Yes and No and entitled Do you wish to continue?:

```
i = WMENU(['Do you wish to continue?', 'Yes', $  
          'No'], Title=0, Init=1)
```

The menu is displayed with Yes initially selected. The result is as follows:

- 1 if the user clicks on Yes .
- 2 if the user clicks on No.
- if the user clicks the left mouse button outside the menu.
- 1

---

## ***WPASTE Function (Windows)***

Pastes the contents of the Clipboard into a graphics window.

### **Usage**

*status* = WPASTE( [*window\_index*] )

### **Input Parameters**

*window\_index* — (optional) The index of the window to which the contents of the Clipboard are to be pasted. If not specified, the current window is assumed.

### **Returned Value**

*status* — A value indicating success or failure of the paste operation; expected values are:

- < 0    Indicates an error. For example, an error value is returned if no graphics are on the clipboard.
- 0      Indicates a successful paste.

### **Keywords**

None.

### **Discussion**

You can paste graphics from the Clipboard in two ways:

- The WPASTE function
- The **Paste from Clipboard** option on the graphics window Control menu.

### **Example**

See the [WCOPY](#) function for an example that demonstrates an application of WPASTE.

### **See Also**

[WCOPY](#)

---

**Windows USERS** The graphics window Control menu includes a command that pastes graphics from the Clipboard. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## **WPRINT Procedure (Windows)**

Prints the contents of a specified window.

### **Usage**

WPRINT [, *window\_index*]

### **Input Parameters**

*window\_index* — (optional) The index number of the window to be printed. If not specified, then the value of !D.Window is used.

### **Keywords**

***Collate*** — If present and nonzero, enables collating for printers that support this feature.

***Color*** — If present and nonzero, enables color output for printers that support this feature.

***Copies*** — Specifies the number of copies to print; the default value is set to the default value of the current printer.

***Duplex*** — If present and nonzero, enables duplex printing. A positive value specifies horizontal (long side) duplexing; a negative value specifies vertical (short side) duplexing.

***Inches*** — By default, the *Xsize* and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, *Xsize* and *Ysize* are taken to be inches instead.

***Landscape*** — If present and nonzero, then landscape orientation is used. The default is portrait orientation.

***Paper\_Size*** — A string value that determines the size of the paper. Valid string values are:

---

Letter (default)	Legal	Tabloid	Ledger
Statement	Executive	A3	A4
A5	B4	B5	Folio
Quarto	10x14	11x17	Note
CSheet	DSheet	ESheet	

---

---

**NOTE** The page dimensions set with this keyword can be overridden with the *XSize* and *YSize* keywords.

---

**Portrait** — If present and nonzero then portrait orientation is used. This is the default.

**Printer\_Name** — Specifies the name of the print queue to which the graphics should be printed. If no printer name is specified, then the default printer is used.

---

**NOTE** If this keyword is not specified, a Print dialog box appears from which you can interactively select a print queue and other options.

---

**Quality** — A string value that specifies the printer resolution; the default value is set to the default value of the current printer. Possible string values are: High, Low, Medium, and Draft.

**Scale\_Factor** — Specifies a scale factor that affects the entire graphics area. The default value is 1.0, which allows output to appear at its normal size.

**Source** — A string value that specifies the paper bin from which the paper is fed by default; the default value is set to the default value of the current printer. Possible string values include:

---

Upper	Middle	Lower	Manual
Auto	Tractor	Cassette	

---

**XSize, YSize** — Specifies the width and height of the output page. By default, these values are specified in centimeters unless the *Inches* keyword is used.

## Discussion

This command allows you to print the contents of a specified window.

Not all output devices support all of the features that can be configured using the WPRINT keywords. If you use a keyword to set a feature that is not supported by the output device, WPRINT simply ignores that keyword, and no error message is displayed. See your printer's documentation for a complete list of its features.

## Example

```
WPRINT, Printer_Name = 'lzl', /Landscape,$
      Copies=3
```

## See Also

[PRINT](#), [PRINTF](#), [WINDOW](#)

---

**Windows USERS** The graphics window Control menu includes a command that prints the contents of a graphics window. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## **WREAD\_DIB Function (Windows)**

Loads a Device Independent Bitmap (DIB) from a file into a graphics window.

### **Usage**

*status* = WREAD\_DIB( [*window\_index*] )

### **Input Parameters**

*window\_index* — (optional) The index of the window to receive the image. If not specified, the current window is assumed. If no window is currently open, an error results.

### **Returned Value**

*status* — A value indicating success or failure; expected values are:

- < 0 Indicates an error.
- 0 Indicates a successful read.

### **Keywords**

***Filename*** — A string containing the name of the DIB (Device Independent Bitmap) file. If not specified, a file named `wave . bmp` is assumed to be in the current directory.

***Interactive*** — If present and nonzero, open the Import Graphics dialog box. This dialog box lets you interactively select a file to import.

## Discussion

Device Independent Bitmap (DIB) is a bitmap format that is useful for transporting graphics and color table information between different devices and applications. DIB files can be produced by graphics applications such as Microsoft Image Editor, Microsoft Paintbrush, and PV-WAVE.

---

**NOTE** This function loads a DIB from a file into a graphics window. To load a DIB directly into a variable, use the function `DC_READ_DIB`.

---

## Example

Assume that the file `map.bmp` is a DIB file that was exported from a graphics application. The following command reads the contents of that file directly into the graphics window with index number 2.

```
status = WREAD_DIB(2, Filename='map.bmp')
```

## See Also

[DC\\_READ\\_DIB](#), [DC\\_WRITE\\_DIB](#), [WWRITE\\_DIB](#)

---

**Windows USERS** The graphics window Control menu includes a command that imports DIB data from a file. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## ***WREAD\_META Function (Windows)***

Loads an enhanced-format metafile (EMF) into a graphics window.

### Usage

```
status = WREAD_META( [window_index] )
```

### Input Parameters

***window\_index*** — (optional) The index of the window to receive the graphics. If not specified, the current window is assumed. If no window is currently open, an error results.

## Returned Value

*status* — A value indicating success or failure of the read; expected values are:

- < 0 Indicates an error.
- 0 Indicates a successful read.

## Keywords

*Filename* — A string containing the name of the enhanced-format metafile. If not specified, a file named `wave.emf` is assumed to be in the current directory.

*Interactive* — If present and nonzero, open the Import Graphics dialog box. This dialog box lets you interactively select a file to import.

## Discussion

This function loads an enhanced-format metafile (EMF) into a graphics window. For more information on metafiles, see the description of the `WINDOW` procedure.

## Example

Assume that the file `map.emf` is an enhanced-format metafile that was exported from a graphics application. The following command reads the contents of that file directly into a graphics window.

```
status = WREAD_META(2, Filename='map.emf')
```

## See Also

[WWRITE\\_META](#)

---

**Windows USERS** The graphics window Control menu includes a command that imports EMF data from a file. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## **WRITEU Procedure**

Writes binary (unformatted) data from an expression into a file.

### **Usage**

WRITEU, *unit*, *expr*<sub>1</sub>, ... , *expr*<sub>n</sub>

### **Input Parameters**

*unit* — The file unit to which the output will be sent.

*expr*<sub>*i*</sub> — Expressions to be output. For nonstring variables, the number of bytes contained in *expr* is output. For string variables, the number of bytes contained in the existing string is output.

### **Keywords**

None.

### **Discussion**

WRITEU performs a direct transfer, with no processing of any kind being done to the data.

### **Example**

In this example, WRITEU is used to write some data to a file. The READU procedure could then be used to read the data from the file.

```
d = BYTSCL(REFORM(FIX(100 * RANDOM(40000))), 200, 200)
      ; Create some data. Argument d contains a 200-by-200 byte array.
```

```
OPENW, unit, 'wux.dat', /Get_Lun
      ; Open the file wux.dat for writing.
```

```
WRITEU, unit, d
      ; Write the data in d to the wux.dat file.
```

```
FREE_LUN, unit
      ; Close the file and free the file unit number.
```

### **See Also**

[OPEN \(UNIX/OpenVMS\)](#), [OPEN \(Windows\)](#), [READ](#)

For more information and examples, see .

---

## ***WRITE\_XBM Procedure***

Writes an image to an X-bitmap (XBM) file.

### **Usage**

`WRITE_XBM, file, image`

### **Input Parameters**

*file* — A scalar string giving the filename of the XBM image.

*image* — The variable containing the input image.

### **Keywords**

None.

### **Discussion**

Since XBM is a monochrome (2-color) format, the input image is forced into 2 colors, if it is not already. Any extra colors are re-mapped using the median point between the minimum and maximum color values; anything less than the median value is converted to 0 (background), while the median and higher values are converted to 1 (black).

### **Example**

```
image = BYTE(DIST(100))  
WRITE_XBM, 'your.xbm', image  
; Write a 2D byte array to an XBM file.
```

### **See Also**

[IMAGE\\_CREATE](#), [IMAGE\\_READ](#), [IMAGE\\_WRITE](#), [READ\\_XBM](#)

---

## **WSET Procedure**

Used to select the current, or “active” window to be used by the graphics and imaging routines.

### **Usage**

WSET, *window\_index*

### **Input Parameters**

*window\_index* — The window index of the new current window.

### **Keywords**

*Resize* — If present and nonzero, notifies PV-WAVE that a window being used to display PV-WAVE graphics (e.g., with the WINDOW command’s *Set\_Xwin\_Id* or *Set\_Win\_Id* keyword) has been resized. PV-WAVE updates the window size in the !D system variable so that subsequent PV-WAVE graphics commands use the new window size.

### **Discussion**

WSET can be used only on displays with window systems.

The window-index number of the current window is given by the read-only system variable !D.Window.

### **Window Resizing in Noninteractive Applications**

On UNIX, if you are writing a PV-WAVE noninteractive application, you must use the *Resize* keyword in order for PV-WAVE to recognize when a user resizes a graphic window.

### **Example 1 — UNIX/OpenVMS**

This example shows how the keyword *Resize* is used in a PV-WAVE Widgets application written in the C programming language. The *cwavec* function is used to call the PV-WAVE functions from the C program.

```
static void      resizeCB_drawingArea( widget, data, cbs )
                Widget          widget;
                XtPointer        data;
                XmDrawingAreaCallbackStruct *cbs;
```

```

{
    int action, numcmds, istat, cwavec();
    char *cmds[2];
    if (!XtIsRealized(widget)) return;
    if (cbs->reason == XmCR_RESIZE) {
        action = 2;
        cmds[0] = 'erase';
        istat = cwavec (action, 1, cmds);
        /*
         * Update display, to solve Motif geometry changes
         * and give PV-WAVE the proper window size.
         */
        XmUpdateDisplay(widget);
        cmds[0] = 'WSET,0,/Resize';
        cmds[1] = 'SHADE_SURF, HANNING(20,20)';
    istat = cwavec (action, 2, cmds);
    }
}

```

## Example 2 — UNIX/OpenVMS

This example shows how the keyword *Resize* is used in a PV-WAVE Widgets application written in the C programming language. The *wavecmds* function is used to call the PV-WAVE functions from the C program.

```

static void    resizeCB_drawingArea( widget, data, cbs )
    Widget      widget;
    XtPointer    data;
    XmDrawingAreaCallbackStruct    *cbs;
{
    if (!XtIsRealized(widget)) return;
    if (cbs->reason == XmCR_RESIZE) {
        wavecmd('ERASE');
        /*
         * Update display, to solve Motif geometry changes
         * and give PV-WAVE the proper window size.

```

```
        */
        XmUpdateDisplay(widget);
        wavecmd('WSET,0,/Resize');
        wavecmd('SHADE_SURF, HANNING(20,20)');
    }
}
```

## See Also

[WDELETE](#), [WINDOW](#), [WSHOW](#)

System Variables: [!D.Window](#)

---

## ***WSHOW Procedure***

Exposes or hides the designated window.

### **Usage**

`WSHOW` [, *window\_index* [, *show*]]

### **Input Parameters**

*window\_index* — (optional) The window index of the window to be hidden or exposed. If not specified, the current window is used.

*show* — (optional) A flag indicating whether a window is hidden or exposed:

- 0 Hides the window.
- 1 Exposes the window.

### **Keywords**

*Iconic* — (UNIX/OpenVMS Only) If present and nonzero, turns the window into an icon.

### **Discussion**

WSHOW does not automatically make the specified window the active window — the window into which new graphics are drawn. You can use WSET to specify the active window.

---

**UNIX and OpenVMS USERS** The definition of “hidden” is machine-dependent. On Sun Workstations, for example, the window disappears, although it can still be written to if it is the active window. On machines supporting the X Window System server, the window is simply pushed to the back of the window display list so that it appears to be located behind other windows on the display.

---

## See Also

[WDELETE](#), [WINDOW](#), [WSET](#)

System Variables: [!D.Window](#)

---

## ***WWRITE\_DIB Function (Windows)***

Saves the contents of a graphics window to a file as a Device Independent Bitmap (DIB).

### Usage

*status* = WWRITE\_DIB( [*window\_index*] )

### Input Parameters

*window\_index* — (optional) The index of the window from which the image is to be saved. If not specified, the current window is assumed.

### Returned Value

*status* — A value indicating success or failure of the write; expected values are:

- < 0 Indicates an error.
- 0 Indicates a successful write.

### Keywords

***Filename*** — A string containing the name of the output DIB file. If not specified, a file named `wave.bmp` is created in the current directory.

***Interactive*** — If present and nonzero, open the Import Graphics dialog box. This dialog box lets you interactively create or select an export file.

## Discussion

This function saves the contents of a graphics window to a DIB format file. To export the contents of a variable to a DIB file, use the `DC_WRITE_DIB` function.

## Example

The following commands demonstrate how to export the contents of graphics window 2 to a DIB file:

```
WINDOW, 2
SHADE_SURF, DIST(40)
    ; Create a window and display graphics in it.
status = WWRITE_DIB(2, Filename='image.bmp')
    ; Save the contents of the window directly in a DIB format file.
```

## See Also

[DC\\_READ\\_DIB](#), [DC\\_WRITE\\_DIB](#), [WREAD\\_DIB](#)

---

**Windows USERS** The graphics window Control menu includes a command that exports DIB graphics to a file. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## ***WWRITE\_META Function (Windows)***

Saves the contents of a graphics window to a file as an enhanced-format metafile (EMF).

### Usage

```
status = WWRITE_META( [window_index] )
```

### Input Parameters

**window\_index** — (optional) The index of the window from which the graphics are to be saved. If not specified, the current window is assumed.

### Returned Value

**status** — A value indicating success or failure of the write; expected values are:

- < 0 Indicates an error.
- 0 Indicates a successful write.

## Keywords

**Filename** — A string containing the name of the output EMF file. If not specified, a file named `wave.emf` is created in the current directory.

**Interactive** — If present and nonzero, open the Import Graphics dialog box. This dialog box lets you interactively create or select an export file.

## Discussion

This function saves the contents of a graphics window in an EMF file.

## Example

In this example, the contents of window 2 are written to an EMF format file called `image.emf`.

```
WINDOW, 2,  
SHADE_SURF, DIST(40)  
status = WWRITE_META(2, Filename='image.emf')
```

## See Also

[WREAD\\_META](#)

---

**Windows USERS** The graphics window Control menu includes a command that exports EMF graphics to a file. For information on the Control menu, see , in the *PV-WAVE User's Guide*.

---

---

## **WzAnimate Procedure**

Starts a VDA Tool used for animating a sequence of images.

### **Usage**

WzAnimate, *var*

### **Input Parameters**

*var* — The name of a 3D byte variable to plot or an equivalent expression.

### **Keywords**

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies the *x* and *y* (horizontal and vertical) coordinates in pixels for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure that has been saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies the width of the drawing area, in pixels.

**YSize** — Specifies the height of the drawing area, in pixels.

### **Discussion**

The *Parent* keyword is used to connect WzAnimate to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzAnimate in:

**(UNIX)** <wavedir>/lib/vdatools/wzanimate.pro

**(OpenVMS)** <wavedir>:[LIB.VDATOOLS]WZANIMATE.PRO

**(Windows)** <wavedir>\lib\vdatools\wzanimate.pro

Where `<wavedir>` is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
WzAnimate, head
    ; The variable head exists on the $MAIN$ level of PV-WAVE, and the
    ; animation is created in the following manner.
OPENR, 3, 'wave/data/headspin.dat'
head = BYTARR(256, 256, 32)
READU, 3, head
CLOSE, 3
```

## See Also

[TV](#)

---

## WzBar Procedure

Starts a VDA Tool used for plotting a bar chart. This VDA Tool creates simple, stacked, and grouped bar charts.

## Usage

WzBar, *var*

## Input Parameters

*var* — The name of a 1D, 2D, or 3D variable, with a maximum of 400 elements. See the *Discussion* section for information on the effects of these different dimensioned variables.

## Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

**YSize** — Specifies, in pixels, the height of the drawing area.

## Discussion

If *var* is a 1D array, a simple bar chart is plotted.

If *var* is a 2D array, the bar chart can either consist of grouped bars or stacked bars. For grouped bars (the default) the first dimension is construed as a group and the second dimension as a bar value. If the **Stack** option is selected in the VDA Tool, then the first dimension is construed as a stack and the second dimension as a bar value.

If *var* is a 3D array, the bar chart consists of grouped, stacked bars. The first dimension is construed as a group, the second as the stack, and the third as a bar in a stack in a group.

The *Parent* keyword is used to connect WzBar to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzHistogram in:

```
(UNIX)      <wavedir>/lib/vdatools/wzbar.pro
(OpenVMS)   <wavedir>:[LIB.VDATOOLS]WZBAR.PRO
(Windows)   <wavedir>\lib\vdatools\wzbar.pro
```

Where `<wavedir>` is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = INDGEN(5)
WzBar, var, XSize=400, YSize=400
```

## See Also

[BAR, ,](#) [BAR2D,](#) [WzBar3D,](#) [WzPie](#)

---

## WzBar3D Procedure

Starts a VDA Tool used for plotting a 3D bar chart.

### Usage

WzBar3D, *var*

### Input Parameters

*var* — The name of a 2D variable, with a maximum size of 200 by 200.

### Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the  $x$  and  $y$  (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

**YSize** — Specifies, in pixels, the height of the drawing area.

## Discussion

The *var* parameter is a 2D array of elevation values. The 3D effect is established by modifying the colortable to create darker color values for use on the top and left sides of the bars. By default, the bars are displayed vertically (upward).

The *Parent* keyword is used to connect WzBar3D to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzHistogram in:

```
(UNIX)      <wavedir>/lib/vdatools/wzbar3d.pro
(OpenVMS)   <wavedir>:[LIB.VDATOOLS]WZBAR3D.PRO
(Windows)   <wavedir>\lib\vdatools\wzbar3d.pro
```

Where <wavedir> is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = DIST(5)
WzBar3d, var, XSize=400, YSize=400
```

## See Also

[BAR3D](#), , [BAR](#), [WzBar](#), [WzPie](#)

---

## WzColorEdit Procedure

Starts a VDA Tool used for editing the image and plot color tables used in other VDA Tools.

### Usage

```
WzColorEdit [, var1[, var2, var3]]
```

### Input Parameters

**var1** — (optional) If only one variable is specified, it must be a 2D array (or equivalent expression) containing 3-by-*n* or *n*-by-3 elements, where *n* is the number of color values to use when initializing the color table.

---

**NOTE** If three variables are specified, all three must be 1D arrays of equal length or equivalent expressions. The input variables are used as a triplet to specify the three components of the color table: (red, green, blue) for the RGB model; (hue, saturation, value) for the HSV model; or (hue, lightness, saturation) for the HLS model. See the *Discussion* for the ranges of these variables.

---

**var2** — (optional) A 1D array or equivalent expression used with both *var1* and *var3* that is of the same length as *var1* and *var3*. The array contains the number of color values to use for the second component of the color model when initializing the color table.

**var3** — (optional) A 1D array or equivalent expression used with both *var1* and *var2* that is of the same length as *var1* and *var2*. The array contains the number of color values to use for the third component of the color model when initializing the color table.

### Keywords

**Cmap** — The index of a predefined color map to load when the color table is initialized. This color map is loaded before the input variables, if any, are written into the color map.

**Hls** — If set, the HLS color system (hue, lightness, saturation) is used, instead of the default (RGB) color system.

**Hsv** — If set, the HSV color system (hue, saturation, value) is used, instead of the default (RGB) color system.

**Image** — If set, the image color table is displayed.

---

**NOTE** If neither *Image* nor *Plot* is specified, both color tables are displayed.

---

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — If used, specifies the relationship of the WzColorEdit Tool to the widget ID of the parent shell. Otherwise, a new top level shell is created for the tool.

**Plot** — If set, the plot color table is displayed.

**Position** — Specifies the *x* and *y* (horizontal and vertical) coordinates in pixels for the starting location of the upper-left corner of the VDA Tool window.

**Range** — A 1D array specifying the range of colors to display. This array contains four elements:

```
[image_start_color, image_end_color, plot_start_color, plot_end_color].
```

If the range is not specified, the `IMAGE_RANGE` and `PLOT_RANGE` global variables are used.

**Restore** — A data structure that has been saved in the Tools Manager with the `TmSaveTools` function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — If specified, the contents of the named template are restored.

## Discussion

The *Restore* keyword is used specifically by the `TM_RESTORE` method. For information on the `TM_RESTORE` method, refer to the source code for `WzColorEdit` in:

```
(UNIX)      <wavedir>/lib/vdatools/wzcoloredit.pro
(OpenVMS)   <wavedir>:[LIB.VDATOOLS]WZCOLOREDIT.PRO
(Windows)   <wavedir>\lib\vdatools\wzcoloredit.pro
```

Where `<wavedir>` is the main **PV-WAVE** directory.

The ranges of the input variables depend on the type of color system that is specified. The components of each color system can have specific ranges of values, shown in the following table.

Color System Component	Input Variable Range
r, g, b	0 – 255
hue	0 – 360
saturation, lightness, value	0 – 1.0

The WzColorEdit Tool can be called from either a Navigator Tool, or directly from the command line. This VDA Tool allows you to select and modify the color table to be used by all other VDA Tools for both the image and the plot colors.

Using a menu bar selection, you can pick from a number of predefined system color tables, or create and save your own custom color tables. As you make selections and/or changes to those selections, all other VDA Tools that are running are updated, so you can see the results of the color edits right after making them.

The editing capabilities included in this VDA Tool let you change a single cell or a range of cells at the same time. Changes to a single cell can be made using control area sliders, or a color wheel dialog box. Changes to a range of color table cells are made by applying a ramp function between two specified color table indices. The ramp method can be linear, logarithmic, or exponential.

---

**NOTE** If the beginning index you specify for the ramp function is larger than the ending index, the VDA Tool ignores the index order when applying the ramp.

---

## Example

In this example, the WzColorEdit Tool with the HSV color model is called from the WAVE> prompt.

```
hue = [0, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330]
saturation = FLTARR(12)
saturation(*) = 1.0
value = saturation
    ; The three 1D variables are defined with equal lengths of n = 12.

WzColorEdit, hue, saturation, value, /Hsv
    ; The VDA Tool appears with the first 6 color indices in the image colors set to
    ; the values specified and the color model controls set for hue, saturation, and
    ; value.
```

## See Also

[COLOR\\_EDIT](#), [LOADCT](#)

---

## WzContour Procedure

Starts a VDA Tool used for plotting contours.

### Usage

WzContour, *var*

### Input Parameters

*var* — The name of a 2D variable to plot or an equivalent expression.

### Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

**YSize** — Specifies, in pixels, the height of the drawing area.

### Discussion

The *Parent* keyword is used to connect WzContour to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzContour in:

(UNIX) <wavedir>/lib/vdatools/wzcontour.pro  
(OpenVMS) <wavedir>:[LIB.VDATOOLS]WZCONTOUR.PRO  
(Windows) <wavedir>\lib\vdatools\wzcontour.pro

Where <wavedir> is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = DIST(40)
WzContour, var, XSize=400, YSize=400
```

## See Also

[CONTOUR](#)

---

## WzExport Procedure

Starts a VDA Tool used for exporting a PV-WAVE variable to an external file in a specified format.

### Usage

WzExport, *var*

### Input Parameters

*var* — The name of the variable to export.

### Keywords

*Directory* — A string containing the name of the destination directory.

*Filename* — A string containing the name of the destination file.

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Pattern** — A string containing the filter pattern for locating existing filenames within a specified directory.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use. It is not to be used at the command line.

**Template** — A string containing the name of a VDA Tool template file.

**Type** — A string containing the type of file to be written.

## Discussion

The *Parent* keyword is used to connect WzExport to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzExport in:

```
(UNIX)      <wavedir>/lib/vdatools/wzexport.pro
(OpenVMS)   <wavedir>: [LIB.VDATOOLS] WZEXPORT.PRO
(Windows)   <wavedir>\lib\vdatools\wzexport.pro
```

Where `<wavedir>` is the main PV-WAVE directory.

A template is a VDA Tool with no data associated with it. The template contains all of the modifications to the VDA Tool that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

The *Type* keyword specifies the format for the output file. Valid file types include the following:

- ASCII-CSV (comma-separated values)
- Binary
- FORTTRAN Binary

8 Bit Image  
XOR Binary  
24 Bit Image  
Image  
Windows DIB (Windows Only)

The names are not case sensitive. The smallest number of unique characters will be recognized. For example: `Type=' 8 BIT'`

The *Filename* keyword specifies the name of the output file.

The *Directory* keyword specifies the path to the directory in which the output file will be saved.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = DIST(40)
WzExport, var
```

## See Also

[WzVariable](#)

---

## ***WzHistogram Procedure***

Starts a VDA Tool used for plotting a histogram.

### **Usage**

WzHistogram, *var*

### **Input Parameters**

*var* — The name of a 1D, 2D, or 3D variable to plot or an equivalent expression.

## Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

**YSize** — Specifies, in pixels, the height of the drawing area.

## Discussion

The *Parent* keyword is used to connect WzHistogram to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzHistogram in:

```
(UNIX)      <wavedir>/lib/vdatools/wzhistogram.pro
(OpenVMS)   <wavedir>:[LIB.VDATOOLS]WZHISTOGRAM.PRO
(Windows)   <wavedir>\lib\vdatools\wzhistogram.pro
```

Where `<wavedir>` is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = DIST(40)
WzHistogram, var, XSize=400, YSize=400
```

## See Also

[HISTOGRAM](#), [PLOT\\_HISTOGRAM](#)

---

## WzImage Procedure

Starts a VDA Tool used for displaying image data.

### Usage

WzImage, *var*

### Input Parameters

*var* — The name of the 2D image variable to plot or an equivalent expression. A 3D variable or equivalent expression is required if the *True* keyword is specified.

### Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**True** — If present and nonzero, indicates that a true-color (24-bit) image is to be displayed and specifies the index of the dimension over which color is interleaved:

- 1 Displays pixel-interleaved images of dimensions (3, *m*, *n*).

- 2 Displays row-interleaved images of dimensions  $(m, 3, n)$ .
- 3 Displays image-interleaved images of dimensions  $(m, n, 3)$ . (Image interleaving is also known as band interleaving.)

---

**NOTE** To use *True*, the *var* parameter must have three dimensions, one of which is equal to 3.

---

*XSize* — Specifies, in pixels, the width of the drawing area.

*YSize* — Specifies, in pixels, the height of the drawing area.

## Discussion

The *Parent* keyword is used to connect WzImage to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzImage in:

**(UNIX)**        <wavedir>/lib/vdatools/wzimage.pro

**(OpenVMS)**   <wavedir>: [LIB.VDATOOLS] WZIMAGE.PRO

**(Windows)**   <wavedir>\lib\vdatools\wzimage.pro

Where <wavedir> is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = DIST(30)
WzImage, var, XSize=400, YSize=400
```

## See Also

[TV](#)

---

## WzImport Procedure

Starts a VDA Tool used for importing data into PV-WAVE.

### Usage

WzImport [, *var*<sub>1</sub>, *var*<sub>2</sub>, ... , *var*<sub>*n*</sub>]

### Input Parameters

*var*<sub>*i*</sub> — (optional) The names of variables into which data is read, up to 100.

### Keywords

**Directory** — A string containing the name of the directory to read data from.

**Filename** — A string containing the name of the file to read.

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock = 2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Pattern** — A string containing the filter pattern for files. For example, ' \* .dat ' will cause files with a .dat extension to be listed in the import tool.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**Type** — A string containing the type of file to be read. Valid file types include the following. The names are not case sensitive. The smallest number of unique characters will be recognized. For example: `Type=' 8 BIT '`.

ASCII-CSV (comma-separated values)

Binary

FORTRAN Binary

8 Bit Image  
XDR Binary  
24 Bit Image  
Image  
Windows DIB (Windows only)

## Discussion

The *Parent* keyword is used to connect WzImport to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzImport in:

**(UNIX)**        <wavedir>/lib/vdatools/wzimport.pro  
**(OpenVMS)**   <wavedir>: [LIB.VDATOOLS] WZIMPORT.PRO  
**(Windows)**   <wavedir>\lib\vdatools\wzimport.pro

Where <wavedir> is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to VDA Tool default settings that were set when the template file was saved. Template files are saved with the **File=>Save Template As** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var1=INTARR(100)
var2=BYTARR(512, 512)
var3=FLTARR(100, 100)
WzImport, var1, var2, var3
```

## See Also

[WzPreview](#), [WzVariable](#)

---

## WzMultiView Procedure

Starts a VDA Tool used to display multiple plots.

### Usage

WzMultiView

### Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See `WwLoop` in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the `TmSaveTools` function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

**YSize** — Specifies, in pixels, the height of the drawing area.

### Discussion

The *Parent* keyword is used to connect WzMultiView to another application, such as the Navigator.

The *Restore* keyword is used specifically by the `TM_RESTORE` method. For information on the `TM_RESTORE` method, refer to the source code for WzMultiView in:

```
(UNIX)      <wavedir>/lib/vdatools/wzmultiview.pro
(OpenVMS)   <wavedir>:[LIB.VDATOOLS]WZMULTIVIEW.PRO
(Windows)   <wavedir>\lib\vdatools\wzmultiview.pro
```

Where `<wavedir>` is the main PV-WAVE directory.

---

**NOTE** For information on how to use this VDA Tool, use Online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
WzMultiView, XSize = 400, YSize = 400
```

## See Also

[WzContour](#), [WzHistogram](#), [WzImage](#), [WzPlot](#), [WzSurface](#)

---

## WzPie Procedure

Starts a VDA Tool used for plotting pie charts.

## Usage

WzPie, *var*

## Input Parameters

*var* — The name of a 1D variable, with a maximum size of 400 elements.

## Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock = 2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

*YSize* — Specifies, in pixels, the height of the drawing area.

## Discussion

The *var* parameter is a 1D array specifying the size of each “slice” of the pie.

The *Parent* keyword is used to connect WzPie to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzPie in:

(UNIX) `<wavedir>/lib/vdatools/wzpie.pro`

(OpenVMS) `<wavedir>:[LIB.VDATOOLS]WZPIE.PRO`

(Windows) `<wavedir>\lib\vdatools\wzpie.pro`

Where `<wavedir>` is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = INDGEN(5)
WzPie, var, XSize=400, YSize=400
```

## See Also

[PIE](#), [PIE\\_CHART](#), [WzBar](#), [WzBar3D](#)

---

## WzPlot Procedure

Starts a VDA Tool used for 2D plotting.

### Usage

WzPlot,  $var_1$  [,  $var_2$ , ...,  $var_n$ ]

### Input Parameters

$var_i$  — The name of the 2D variable to plot or an equivalent expression.

### Keywords

**Independent** — Specifies the name of the variable to be taken as the independent variable.

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the  $x$  and  $y$  (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

**YSize** — Specifies, in pixels, the height of the drawing area.

### Discussion

---

**NOTE** The maximum number of variables you can plot simultaneously in a WzPlot tool is 10.

---

The *Parent* keyword is used to connect WzPlot to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzPlot in:

(UNIX) `<wavedir>/lib/vdatools/wzplot.pro`

(OpenVMS) `<wavedir>:[LIB.VDATOOLS]WZPLOT.PRO`

(Windows) `<wavedir>\lib\vdatools\wzplot.pro`

Where `<wavedir>` is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements — that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = HANNING(40)
WzPlot, var, XSize=400, YSize=400
```

## See Also

[PLOT](#)

---

## WzPreview Procedure

Starts a VDA Tool used to view an ASCII file's contents and select which parts of the file are to be read in as PV-WAVE variables.

## Usage

```
WzPreview [, filename]
```

## Parameters

*filename* — (optional) A string containing the name of the ASCII file to preview. If not specified, WzPreview will start without displaying a file.

## Keywords

**AutoDefine** — If set along with *filename*, then import definitions will be automatically defined when the file is previewed. If no file is specified, this keyword has no effect.

**Columns** — An integer specifying the number of visible columns in the WzPreview window.

**Fixed** — Set this keyword if the file contains fixed-width values that are column-oriented.

**Free** — Set this keyword if the file contains non-fixed width values (either column-oriented) that are separated by delimiters.

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Rows** — An integer specifying the number of visible rows in the WzPreview window.

**Template** — A string containing the name of a VDA Tool template file.

## Discussion

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzPreview in:

```
(UNIX)      <wavedir>/lib/vdatools/wzpreview.pro
(OpenVMS)   <wavedir>:[LIB.VDATOOLS]WZPREVIEW.PRO
(Windows)   <wavedir>\lib\vdatools\wzpreview.pro
```

Where <wavedir> is the main PV-WAVE directory.

## Example

```
WzPreview, !Data_Dir + 'epa.dat', /Fixed
```

## See Also

[WtPreview](#), [WwPreview](#) (in the *PV-WAVE Application Developer's Guide*),  
[WzImport](#)

---

## WzSurface Procedure

Starts a VDA Tool used for surface plots.

### Usage

```
WzSurface, z [, x, y]
```

### Input Parameters

**z** — A 2D array containing the values that make up the surface or an equivalent expression. If *x* and *y* are supplied, the surface is plotted as a function of the X,Y locations specified by their contents. Otherwise, the surface is generated as a function of the array index of each element of *z*.

**x** — (optional) A vector or 2D array (or equivalent expression) specifying the *x*-coordinates for the surface.

If *x* is a vector, each element of *x* specifies the *x*-coordinate for a column of *z*. For example, *x*(0) specifies the *x*-coordinate for *z*(0, \*).

If *x* is a 2D array, each element of *x* specifies the *x*-coordinate of the corresponding point in *z* (*x*<sub>*ij*</sub> specifies the *x*-coordinate for *z*<sub>*ij*</sub>).

**y** — (optional) A vector or 2D array (or equivalent expression) specifying the *y*-coordinates for the surface.

If *y* is a vector, each element of *y* specifies the *y*-coordinate for a row of *z*. For example, *y*(0) specifies the *y*-coordinate for *z* (\*, 0).

If *y* is a 2D array, each element of *y* specifies the *y*-coordinate of the corresponding point in *z* (*y*<sub>*ij*</sub> specifies the *y*-coordinate for *z*<sub>*ij*</sub>).

## Keywords

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE GUI Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the  $x$  and  $y$  (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved by the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Shade** — Used to specify how to shade the surface. This keyword can be set to a variable (2D byte array) used to shade the surface, or to one of the following values:

- 1 Gouraud shading
- 2 Elevation shading

(Default: not shaded)

**Template** — A string containing the name of a VDA Tool template file.

**XSize** — Specifies, in pixels, the width of the drawing area.

**YSize** — Specifies, in pixels, the height of the drawing area.

## Discussion

The *Parent* keyword is used to connect WzSurface to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzPlot in:

```
(UNIX)      <wavedir>/lib/vdatools/wzsurface.pro
(OpenVMS)   <wavedir>:[LIB.VDATOOLS]WZSURFACE.PRO
(Windows)   <wavedir>\lib\vdatools\wzsurface.pro
```

Where `<wavedir>` is the main PV-WAVE directory.

A template is a VDA Tool without any data associated with it. The template contains all of the modifications to the VDA Tool — colors, axes, graphical elements

— that were set when the template file was saved. Template files are saved with the **File=>Save Template As ...** function on the VDA Tool.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
var = DIST(40)
WzSurface, var, XSize=400, YSize=400
```

## See Also

[SURFACE](#)

---

## ***WzTable Procedure***

Starts a VDA Tool used for creating an editable 2D array of cells containing string data.

### Usage

WzTable, *var*

### Input Parameters

*var* — A vector, 2D array, or scalar variable (or equivalent expression).

### Keywords

***Alignments*** — A 1D array [0, ..., *cols*-1] of column alignments. Valid values are:

- 0     Align cell contents to cell's left edge (left justify).
- 1     Center the cell contents (center justify).
- 2     Align cell contents to the cell's right edge (right justify).

***ColLabels*** — A 1D string array [0, ..., *cols*-1] of column labels.

***CWidth*** — A 1D array [0, ..., *cols*-1] of column widths. If *CWidth* is not specified, the default column width is 10 characters.

**Format** — A FORTRAN-style format specification for the output. For more information, see the [STRING](#) function.

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock =2`. (See WwLoop in the *PV-WAVE GUI Application Developer's Guide* for more information.)

**NumCols** — The number of columns in the table. If *NumCols* is not specified, the number of columns is calculated from the dimensions of *var*.

**NumRows** — The number of rows in the table. If *NumRows* is not specified, and the *var* parameter is specified, the number of rows is calculated from the dimensions of *var*. If neither *NumRows* nor *var* is specified, the size of the table is set to one row.

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**RowLabels** — A 1D string array [0, ..., *cols*-1] of row labels.

**Template** — A string containing the name of a VDA Tool template file.

**Vertical** — If this keyword is present and nonzero, the contents of *var* are displayed as transposed (*var* rows are vertical and columns are horizontal).

**VisibleCols** — The number of columns displayed in the view window. If *VisibleCols* is not specified, four columns are displayed.

---

**NOTE** If the table size is bigger than the number of visible columns and rows, scrollbars are placed at the right and bottom edges of the view window.

---

**VisibleRows** — The number of rows displayed in the view window. If *VisibleRows* is not specified, four rows are displayed.

## Discussion

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzTable in:

(UNIX) `<wavedir>/lib/vdatools/wztable.pro`  
(OpenVMS) `<wavedir>:[LIB.VDATOOLS]WZTABLE.PRO`  
(Windows) `<wavedir>\lib\vdatools\wztable.pro`

Where `<wavedir>` is the main PV-WAVE directory.

The type of variable used determines the numbers of columns and rows in the table:

scalar — 1 cell

vector — 1 row with  $n$  columns

2D array — a matrix of  $n$  rows by  $n$  columns

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
a = FINDGEN(10,20)
WzTable, a, Format = 'f10.2'
```

## See Also

[WtTable](#), [WwTable](#) (in the *PV-WAVE GUI Application Developer's Guide*)

---

## WzVariable Procedure

Starts a VDA Tool used for viewing and exporting variables.

### Usage

WzVariable

### Input Parameters

None.

## Keywords

**Associated\_With** — A string containing the name of a VDA Tool. The WzVariable window will only show variables from this specified VDA Tool. Note that the string you specify is case-sensitive.

**NoBlock** — If specified, the event loop (WwLoop) that is started by the VDA Tool will use the given value. By default a value of 1 (non-blocking loop) will be used. However, it may be necessary in certain circumstances to force a non-blocking loop by specifying `NoBlock = 2`. (See WwLoop in the *PV-WAVE GUI Application Developer's Guide* for more information.)

**Parent** — The widget ID of the parent widget.

**Position** — Specifies, in pixels, the *x* and *y* (horizontal and vertical) coordinates for the starting location of the upper-left corner of the VDA Tool window.

**Restore** — A data structure previously saved in the Tools Manager with the TmSaveTools function. This keyword is reserved for internal use; it is not to be used at the command line (see *Discussion*).

**Type** — An integer, or array of integers, that specify the data types to display. These integers and their corresponding data types are:

<b>Data Type Code</b>	<b>Data Type</b>
0	Undefined
1	Byte
2	Integer
3	Longword Integer
4	Floating Point
5	Double-Precision Floating
6	Complex single-precision floating
7	String
12	Complex double-precision floating

## Discussion

The *Parent* keyword is used to connect WzVariable to another application, such as the Navigator.

The *Restore* keyword is used specifically by the TM\_RESTORE method. For information on the TM\_RESTORE method, refer to the source code for WzVariable in:

**(UNIX)**        <wavedir>/lib/vdatools/wzvariable.pro

**(OpenVMS)** <wavedir>:[LIB.VDATOOLS]WZVARIABLE.PRO

**(Windows)** <wavedir>\lib\vdatools\wzvariable.pro

Where <wavedir> is the main PV-WAVE directory.

---

**NOTE** For information on how to use this VDA Tool, use online Help. Select the **On Window** command from the VDA Tool Help menu to bring up Help on this VDA Tool.

---

## Example

```
WzVariable, Associated_With='WzPlot_1'
```

## See Also

[WzImport](#)

---

## ***XYOUTS Procedure***

Draws text on the currently selected graphics device starting at the designated data coordinate.

### **Usage**

XYOUTS, *x*, *y*, *string*

### **Input Parameters**

*x, y* — Specifies the column, *x* and the row, *y* at which the output string should start. Both *x* and *y* are normally taken to be in data coordinates; however, the *Device* and *Normal* keywords can be used to change this unit.

### **Output Parameters**

*string* — The scalar string containing the text that is to be output to the display surface. If not of string type, it is converted prior to use.

### **Keywords**

***Background*** — Fills the area behind the text with a specified color. The default, *-1*, indicates no fill.

Additional XYOUTS keywords are listed below. For a description of each keyword, see [Chapter 3, \*Graphics and Plotting Keywords\*](#).

<a href="#">Alignment</a>	<a href="#">Color</a>	<a href="#">Normal</a>	<a href="#">Text_Axes</a>
<a href="#">Channel</a>	<a href="#">Data</a>	<a href="#">Orientation</a>	<a href="#">Width</a>
<a href="#">Charsize</a>	<a href="#">Device</a>	<a href="#">PClip</a>	<a href="#">Z</a>
<a href="#">Charthick</a>	<a href="#">Font</a>	<a href="#">Size</a>	
<a href="#">Clip</a>	<a href="#">Noclip</a>	<a href="#">T3d</a>	

---

## Discussion

XYOUTS is machine-dependent when you are using hardware fonts. This means that on two different machines, the same commands may produce text that does not appear the same. To guarantee similar appearance, use software fonts.

---

**UNIX and OpenVMS USERS** You may notice that under X Windows the size of the software fonts varies from device to device. When you start PV-WAVE, the PV-WAVE hardware font is set to the current hardware font of the X server. Not all X servers will have the same default font size because users can reconfigure the default font and the default font can differ between X servers. Therefore, you may discover that the hardware font size, and therefore the software font size, may vary across different workstations. You can avoid this by explicitly setting the X font using the DEVICE procedure. For example:

```
DEVICE, font='-adobe-courier-medium-r-normal--14-*'
```

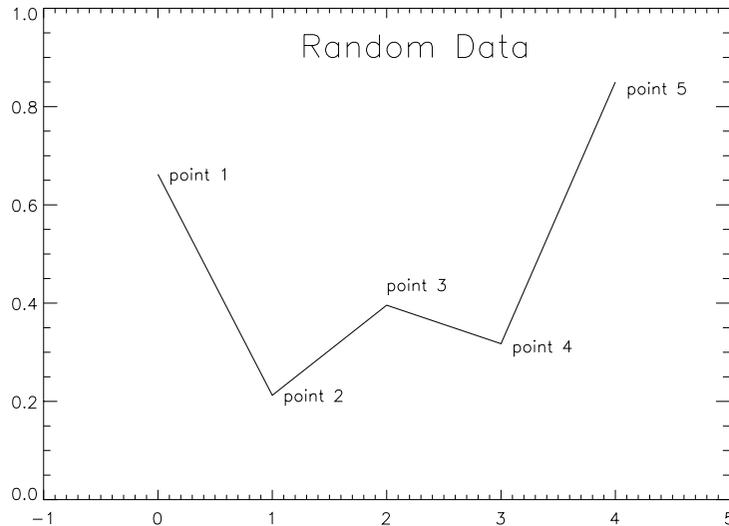
---

## Example

In this example, XYOUTS is used to label a plot of random data. Procedure XYOUTS also is used to place a title on the plot. Text placement is relative to the  $x$  and  $y$  coordinates of the plot, which is the default for XYOUTS. Note that the CURSOR procedure was used to determine the proper coordinates at which to place labels. This example uses PV-WAVE:IMSL Statistics Toolkit procedure RANDOMOPT.

```
RANDOMOPT, Set = 1234567
x = RANDOM(5)
      ; Create a 5-element vector of random data.
PLOT, x, XRange = [-0.5, 4.5]
      ; Plot the data.
XYOUTS, 0.1, 0.65, "point 1"
      ; Label the first data point.
XYOUTS, 1.1, 0.20, "point 2"
      ; Label the second data point.
XYOUTS, 2.0, 0.425, "point 3"
      ; Label the third data point.
XYOUTS, 3.1, 0.30, "point 4"
      ; Label the fourth data point.
```

```
XYOUTS, 4.1, 0.825, "point 5"  
    ; Label the fifth data point.  
XYOUTS, 1.25, 0.9, "Random Data", Charsize = 2  
    ; Place a title on the plot. Make the title twice the default character size using the  
    ; Charsize keyword.
```



**Figure 2-91** Example of plot labeling using XYOUTS.

## See Also

[LEGEND](#), [PLOT](#)

For more information on using XYOUTS to annotate plots, see .

---

## **ZOOM Procedure**

Expands and displays part of an image or graphic plot.

### **Usage**

ZOOM

### **Input Parameters**

None.

### **Keywords**

**Fact** — The zoom expansion factor, an integer. (Default: 4)

**Interp** — Specifies the interpolation method to be used. If nonzero, uses the bilinear interpolation method. Otherwise, uses the nearest neighbor method (the default).

**XSize** — The width of the new window, in pixels. (Default: 512 unless NoNew is set, where the window size doesn't change)

**YSize** — The height of the new window, in pixels. (Default: 512 unless NoNew is set, where the window size doesn't change)

**Continuous** — If set, causes the zoom window to track the mouse cursor. This obviates the need to press the left mouse button to mark the center of the zoom. No effect if NoNew is set.

**NoClose** — If set, the newly created window will not be closed. No effect if NoNew is set.

**NoNew** — If set, the original window is used to show the zoomed image. If this keyword is set, Continuous, NoClose, XSize, and YSize are ignored.

**Restore** — If set and NoNew is set, the original image is restored when ZOOM exits. It has no effect unless NoNew is set.

**Rubberband** — If set, allows the user to select the zoom region by dragging a rubberband box around that region.

## Discussion

ZOOM works only on windowing systems. It provides a quick way to get a close look at your image.

ZOOM lets you click with your mouse button on the center point of a region in a previously displayed window to bring up an enlarged view of this region in a new window. Then use your mouse buttons as follows:

- Use the left mouse button to choose the center point of the region to be zoomed in on from the original image.
- Use the middle mouse button to display a window for choosing the zoom factor.

---

**Windows USERS** If you have a two-button mouse, <Shift> in combination with the left mouse button works the same as a middle button.

---

- Use the right button to exit the ZOOM procedure.

## Example

```
OPENR, 1, !Data_dir + 'mandril.img'
mandril = BYTARR(512, 512)
READU, 1, mandril
      ; Read in the image file.

TVSCL, mandril
      ; Display the image file.

ZOOM
      ; Use ZOOM with the default values; the mouse button functions will
      ; be described in the original window.

ZOOM, Fact=2
      ; Use ZOOM with the zoom factor set to 2.

ZOOM, Interp=0
      ; Use ZOOM with the zoomed region displayed using nearest neighbor sampling.

ZOOM, Interp=1
      ; Use ZOOM with the zoomed region displayed using bilinear interpolation.

ZOOM, XSize=200, YSize=200
      ; Display the new image in a window that is 200-by-200 pixels.

ZOOM, /Continuous
      ; Run ZOOM where it continually samples the cursor; there is no need to click.

ZOOM, Fact=2, /NoNew, /Restore
```

- ; Run ZOOM without creating a new window, which allows subsequent zooming.
- ; The original image is restored when ZOOM ends.

## See Also

[ROT](#), [ROT\\_INT](#), [TVRD](#)

For details on interpolation methods. see Chapter 6, *Displaying Images*, in the *PV-WAVE User's Guide*.

---

## ZROOTS Procedure

Finds the roots of the  $m$ -degree complex polynomial, using Laguerre's method.

### Usage

ZROOTS,  $a$ ,  $roots$  [,  $polish$ ]

### Input Parameters

$a$  — A vector containing the  $m + 1$  coefficients of the polynomial. This 1D array may be either real or complex. If the input is of single-precision data type, the result is single-precision; if the input is double-precision, the result is double-precision.

$polish$  — (optional) Specifies whether polishing is to be done. Set to 0 if you want to prevent polishing of the roots. If set to 1 or omitted, roots are polished.

### Output Parameters

$roots$  — The result of ZROOTS, which is set to an  $m$ -element complex vector on exit.

### Keywords

None.

### Discussion

ZROOTS returns the roots of the  $m$ -degree complex polynomial:

$$\sum_{i=0}^m a_i x^i$$

## See Also

### [POLY](#)

ZROOTS is based on a routine of the same name in *Numerical Recipes in C: The Art of Scientific Computing*, by Flannery, Press, Teukolsky, and Vetterling, Cambridge University Press, Cambridge, MA, 1988. It is used by permission.



## Graphics and Plotting Keywords

This chapter describes the keywords that can be used with the graphics and plotting system routines. For information on the corresponding system variables that are listed for some keywords, see [Chapter 4, System Variables](#).

### Alignment Keyword

**Used With Routines:** [XYOUTS](#)

**Corresponding System Variable:** None.

Specifies the horizontal alignment of the text in relation to the point  $x, y$ , which is specified as input to the [XYOUTS](#) procedure.

An alignment of 0.0 (the default) places the left edge of the text on the given  $(x, y)$  coordinate (left-justifies). An alignment of 1.0 right-justifies the text, while 0.5 centers the text over point  $(x, y)$ .

### Ax Keyword

**Used With Routines:** [BAR3D](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [vtkSCATTER](#), [vtkSLICEVOL](#), [vtkSURFACE](#), [vtkPOLYSHADE](#)

**Corresponding System Variable:** None.

Specifies the angle of rotation about the  $x$ -axis, in degrees, towards the viewer.

The  $Ax$  keyword parameter defaults to +30 degrees if omitted and `!P.T3d` is 0.

---

**NOTE** This keyword is effective only if !P.T3d is *not* set. If !P.T3d is set, the three-dimensional to two-dimensional transformation used by SURFACE is contained in the 4-by-4 array !P.T.

---

The surface represented by the two-dimensional array is first rotated,  $A_z$  (see the next section) degrees about the  $z$ -axis, then by  $A_x$  degrees about the  $x$ -axis, tilting the surface towards the viewer ( $A_x > 0$ ), or away from the viewer.

The 3D to 2D transformation represented by  $A_x$  and  $A_z$  can be saved in !P.T by including the *Save* plotting keyword.

## Az Keyword

**Used With Routines:** [BAR3D](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [vtkSCATTER](#), [vtkSLICEVOL](#), [vtkSURFACE](#), [vtkPOLYSHADE](#)

**Corresponding System Variable:** None.

Specifies the counterclockwise angle in degrees of rotation about the  $z$ -axis (when looking down the  $z$ -axis toward the origin).

This keyword is effective only if !P.T3d is *not* set. The order of rotation is  $A_z$  first, then  $A_x$ .

## Background Keyword

**Used With Routines:** [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** !P.Background.

The background color index to which the screen is set when the ERASE procedure is called.

---

**NOTE** Not all devices support erasing the background to a color index.

---

## Example

To produce a black plot with a white background on a color display:

```
PLOT, y, Background = 255, Color = 0
```

## Bottom Keyword

Used With Routines: [SURFACE](#)

Corresponding System Variable: None.

The color index used to draw the lower part of the surface. If not specified, the bottom is drawn with the same color as the top.

---

**NOTE** If the X rotation is between 90 and 270 degrees, the top of the surface will be colored with the color set by the *Bottom* keyword.

---

## Box Keyword

Used With Routines: [PLOT](#)

Corresponding System Variable: [!PDT.Box](#)

Places a box around the labels in a Date/Time axis. If you set the keyword to a value of 1, boxes are drawn around all the labels of the Date/Time axis. (Default: no boxes are drawn)

## C\_Annotation Keyword

Used With Routines: [CONTOUR](#), [CONTOUR2](#), [MAP\\_CONTOUR](#)

Corresponding System Variable: None.

Sets the label that will be drawn on each contour.

Usually, contours are labeled with their value. This parameter, a vector of strings, allows any text to be specified. The first label is used for the first contour drawn, and so forth. If the *Levels* keyword is specified, the elements of *C\_Annotation* correspond directly to the levels specified, otherwise, they correspond to the default levels chosen by the contour procedure. If there are more contour levels than elements in *C\_Annotation*, the remaining levels are labeled with their values.

---

**NOTE** If the *CONTOUR2 Fill* keyword is used, labeling is disabled. Refer to the description of [CONTOUR2](#) in the *PV-WAVE Reference* for an example of how to create a filled contour plot with labels.

---

### **Example**

To produce a contour plot with three levels labeled “low”, “medium”, and “high”:

```
CONTOUR, Z, Levels = [0.0, 0.5, 1.0], $  
    C_Annotation = ["low", "medium", "high"]
```

Use of this keyword implies use of the *Follow* keyword.

## C\_Charsize Keyword

**Used With Routines:** [CONTOUR](#), [CONTOUR2](#), [MAP\\_CONTOUR](#)

**Corresponding System Variable:** None.

Sets the size of the characters used to annotate contour labels.

Normally, contour labels are drawn at three-fourths the size used for the axis labels (specified by the *Charsize* keyword or `!P.Charsize` system variable). This keyword allows the contour label size to be specified independently. Use of this keyword implies use of the *Follow* keyword.

---

**NOTE** If the `CONTOUR2 Fill` keyword is used, labeling is disabled. Refer to the description of [CONTOUR2](#) in the *PV-WAVE Reference* for an example of how to create a filled contour plot with labels.

---

## C\_Charthick Keyword

**Used With Routines:** [CONTOUR](#), [CONTOUR2](#)

**Corresponding System Variable:** `!P.Charthick`

Sets the thickness of contour label characters drawn with the software fonts. Normal thickness is 1.0, double thickness is 2.0, etc. If this keyword is omitted, the value of the system variable `!P.Charthick` is used.

---

**NOTE** If the `CONTOUR2 Fill` keyword is used, labeling is disabled. Refer to the description of [CONTOUR2](#) in the *PV-WAVE Reference* for an example of how to create a filled contour plot with labels.

---

## C\_Colors Keyword

**Used With Routines:** [CONTOUR](#), [CONTOUR2](#), [MAP\\_CONTOUR](#)

**Corresponding System Variable:** None.

A vector of color indices used to set the color index used to draw each contour.

This parameter is a vector, converted to integer type if necessary. If there are more contour levels than elements in *C\_Colors*, the elements of the color vector are cyclically repeated.

### **Example**

If *C\_Colors* contains three elements, and there are seven contour levels to be drawn, the colors  $c_0, c_1, c_2, c_0, c_1, c_2, c_0$  will be used for the seven levels. To call CONTOUR and set the colors to [100, 150, 200]:

```
CONTOUR, Z, C_Colors = [100, 150, 200]
```

## **C\_Labels Keyword**

Used With Routines: [CONTOUR](#), [CONTOUR2](#), [MAP\\_CONTOUR](#)

Corresponding System Variable: None.

Specifies which contour levels should be labeled. By default, every other contour level is labeled.

*C\_Labels* allows you to override this default and explicitly specify the levels to label. This parameter is a vector, converted to integer type if necessary. If the *Levels* keyword is specified, the elements of *C\_Labels* correspond directly to the levels specified, otherwise, they correspond to the default levels chosen by CONTOUR. Setting an element of the vector to zero causes that contour label to not be labeled. A nonzero value forces labeling.

---

**NOTE** If the CONTOUR2 *Fill* keyword is used, labeling is disabled. Refer to the description of [CONTOUR2](#) in the *PV-WAVE Reference* for an example of how to create a filled contour plot with labels.

---

### **Example**

To produce a contour plot with four levels where all but the third level is labeled:

```
CONTOUR, Z, Levels = [0.0, 0.25, 0.75, 1.0], $  
      C_Labels = [1, 1, 0, 1]
```

Use of this keyword implies use of the *Follow* keyword.

## **C\_Linestyle Keyword**

Used With Routines: [CONTOUR](#), [CONTOUR2](#)

Corresponding System Variable: None.

Specifies the linestyle used to draw each contour.

As with *C\_Colors*, *C\_Linestyle* is a vector of linestyle indices. If there are more contour levels than linestyles, the linestyles are cyclically repeated. The following table lists the available linestyles and their keyword indices:

Index	X Windows Style	Windows Style
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

---

**NOTE** The current contouring algorithm draws all the contours in each cell, rather than following contours. Hence, some of the more complicated linestyles will not be suitable for some applications.

---

### **Example**

To produce a contour plot, with the contour levels directly specified in a vector *V*, with all negative contours drawn with dotted (UNIX/OpenVMS) or dashed (Windows) lines, and with positive levels in solid lines:

```
CONTOUR, Z, Levels = V, C_Linestyle = V LT 0.0
```

## **C\_Thick Keyword**

**Used With Routines:** [CONTOUR](#), [CONTOUR2](#)

**Corresponding System Variable:** None.

Specifies the line thickness of lines used to draw each contour level. As with *C\_Colors*, *C\_Thick* is a vector of line thickness values, although the values are floating-point. If there are more contours than thickness elements, elements are repeated. If omitted, the overall line thickness specified by the *Thick* keyword parameter or *!P.Thick* is used for all contours.

## Channel Keyword

**Used With Routines:** [AXIS](#), [CONTOUR](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE\\_SURF](#), [SURFACE](#), [XYOUTS](#)

**Corresponding System Variable:** None.

Specifies the destination channel index or mask for the operation. This parameter is used only with devices that have multiple display channels. (Default: zero)

## Charsize Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PIE](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [XYOUTS](#), [vtkSCATTER](#), [vtkSURFACE](#), [vtkTEXT](#)

**Corresponding System Variable:** `!P.Charsize`

Sets the overall character size for the annotation. A *Charsize* of 1.0 is normal. The size of the annotation on the axes may be set, relative to *Charsize*, with *XCharsize*, *YCharsize*, and *ZCharsize*. The main title is written with a character size of 1.25 times this parameter.

---

**NOTE** If you use `!P.Multi` to create a multiple plot of more than two rows or columns, `PV-WAVE` decreases the character size by a factor of two.

---

## Charthick Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#), [XYOUTS](#)

**Corresponding System Variable:** `!P.Charthick`

Sets the thickness of characters drawn with the software fonts. Normal thickness is 1.0, double thickness is 2.0, etc. If this keyword is omitted, the value of the system variable `!P.Charthick` is used.

## Clip Keyword

**Used With Routines:** [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SURFACE](#), [XYOUTS](#)

**Corresponding System Variable:** None.

Specifies the coordinates of a rectangle used to clip the graphics output. Graphics that fall inside the rectangle are displayed; graphics that fall outside the clipping rectangle are not displayed.

The rectangle is specified as a vector of the form  $[X_0, Y_0, X_1, Y_1]$ , giving coordinates of the lower-left and upper-right corners, respectively. Coordinates are specified in data coordinate units unless an overriding coordinate keyword is present, such as *Normal* or *Device*.

## Color Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PIE\\_CHART](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [XYOUTS](#)

**Corresponding System Variable:** `!P.Color`

Sets the color index of text, lines, solid polygon fill, data, axes, and annotation. If this keyword is omitted, `!P.Color` specifies the color index.

When used with `PLOT`, `OPLOT`, and `PLOTS`, the *Color* keyword can specify an array of color values. If an array is used, each color value in the array is applied, in order, to color the line segments and/or plot symbols that make up the graph. The colors are repeated, as needed, to complete the entire graph of the data set. In addition, when an array is specified, the `!P.Color` system variable is used to color the axes (rather than using one of the data colors). Negative values in a *Color* array create transparent data segments (in other words, no line segment is drawn for that data interval).

When used with `PIE_CHART`, you can only specify an array value for this keyword. The specified colors are applied, in order, to the slices of the pie chart.

---

**NOTE** You cannot specify an array of values for the `!P.Color` system variable. The array of color values can only be used with the *Color* keyword.

---

### Example 1

```
TEK_COLOR
```

```
!P.Color=10
```

```
PLOT, FINDGEN(10), Color=[2,3,5]
```

```
    ; Axes drawn in color 10.
```

```
    ; The 9 line segments are drawn in colors 2, 3, 5, 2, 3, 5, 2, 3, 5.
```

## Example 2

Plot data containing “missing” values by using the fact that negative colors are “transparent”.

```
data = HANNING(50)
      ; Create some data.

data([5, 6, 10, 20, 25, 30, 31]) = -1
      ; Flag some data as “missing” (= -1).

PLOT, data, PSym=-5, YRange=[0,1]
      ; Normal plot with lots of spikes.

col = REPLICATE(!P.Color, 50)
      ; Array of plot colors.

missing = WHERE(data eq -1)
      ; Indices of missing points.

col(missing) = -1
      ; Symbols on the missing points and line segments to the right of
      ; missing points are transparent.

PLOT, data, PSym=5, YRange=[0,1], Color=col
      ; Draw plot symbols.

col(missing-1) = -1
      ; Line segments to the left of missing points are also transparent.

OPLOT, data, Color=col
      ; Overlay lines with segments missing.
```

## Compress Keyword

Used With Routines: [PLOT](#), [OPLOT](#)

Corresponding System Variable: [!PDT.Compress](#)

Compresses out weekends and holidays from a Date/Time axis. Before you can use this keyword, you must define holidays or weekends with the procedures [CREATE\\_HOLIDAYS](#) and [CREATE\\_WEEKENDS](#).

## Data Keyword

Used With Routines: [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [XYOUTS](#)

Corresponding System Variable: None.

A keyword flag, which if present, indicates that the coordinates are specified in data coordinates (the default). When used with `AXIS`, `CONTOUR`, `CONTOUR2`, `O PLOT`, `PLOT`, `SHADE_SURF`, and `SURFACE`, this keyword specifies that the *Position* and *Clip* coordinates are in data units.

## Device Keyword

**Used With Routines:** `AXIS`, `BAR3D`, `CONTOUR`, `CONTOUR2`, `O PLOT`, `PLOT`, `PLOTS`, `POLYFILL`, `SHADE_SURF`, `SHADE_SURF_IRR`, `SURFACE`, `XYOUTS`

**Corresponding System Variable:** None.

Express coordinates in device coordinates. When used with `AXIS`, `CONTOUR`, `CONTOUR2`, `O PLOT`, `PLOT`, `SHADE_SURF`, and `SURFACE`, this keyword specifies that the *Position* and *Clip* coordinates are in device units.

### Example

The following code displays an image contained in the variable `A` and then draws a contour plot of pixels (100:499, 100:399) registered over the pixels:

```
TV, A
    ; Display the image.
CONTOUR, A(100:499, 100:399), Position = $
    [100,100, 499,399], /Device, /Noerase, $
    XStyle = 1, YStyle = 1
    ; Draw the contour plot, specify the coordinates of the plot, in
    ; device coordinates, do not erase, set the X and Y axis styles
    ; to EXACT.
```

Note that in the above example, the keyword specification `/Device` is equivalent to `Device = 1`.

## DT\_Range Keyword

**Used With Routines:** `PLOT`

**Corresponding System Variable:** `!PDT.DT_Range`

Sets an exact range of values in a Date/Time axis. You must specify the desired start and end values from a Date/Time Julian value. The range may be adjusted slightly by the `PLOT` procedure, depending on the data. To obtain an exact range set the *XStyle* plotting keyword to 1 (one). For more information, see *XStyle*.

## Fill\_Pattern Keyword

**Used With Routines:** [PLOTS](#), [POLYFILL](#)

**Corresponding System Variable:** None.

The hardware-dependent fill pattern index for the [POLYFILL](#) and [PLOTS](#) procedures. If omitted or set to 0, a solid fill results.

## Follow Keyword

**Used With Routines:** [CONTOUR](#)

**Corresponding System Variable:** None.

If present and nonzero, forces the [CONTOUR](#) procedure to use the line-following method instead of the cell-drawing method.

[CONTOUR](#) can draw contours using one of two different methods:

- The cell-drawing method, used by default, examines each array cell and draws all contours emanating from that cell before proceeding to the next cell. This method is efficient in terms of computer resources but does not allow contour labeling.
- The line-following method searches for each contour line and then follows the line until it reaches a boundary or closes. This method gives better looking results with dashed linestyles, and allows contour labeling, but requires more computer time. It is used if any of the following keywords is specified: *C\_Annotation*, *C\_Charsize*, *C\_Charthick*, *C\_Labels*, *Follow*, or *Path\_Filename*.

Although these two methods both draw correct contour maps, differences in their algorithms can cause small differences in the resulting plot.

## Font Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [MAP\\_CONTOUR](#), [MAP\\_XYOUTS](#), [OPLOT](#), [PIE](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [XYOUTS](#)

**Corresponding System Variable:** `!P.Font`

An integer that specifies the graphics text font index.

Font index `-1` selects the software fonts, which are drawn using vectors. Font number `0` selects the hardware font of the output device. See for a complete description of the software fonts. See [Appendix B, Output Devices and Window](#)

*Systems* for more information on the hardware fonts available with each supported output device.

---

**NOTE** Hardware font drivers that support 3D transformations include X Windows, WIN32 (on Windows NT platforms only), PostScript, and WMF (on Windows NT platforms only).

---

## Gridstyle Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

**Corresponding System Variable:** `!P.Gridstyle`

Lets you change the linestyle of tick intervals.

The default is a solid line. Other linestyle choices and their index values are listed in the following table:

Index	X Windows Style	Windows Style
0	Solid (default)	Solid (default)
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

One possible use for this keyword is to create an evenly spaced grid consisting of dashed lines across your plot region. To do this, first set the *Ticklen* keyword to 0.5. This ensures that the dashed tick style will appear correctly on your plot. Then set the *Gridstyle* keyword to the style you want to use. For example:

```
PLOT, mydata, Ticklen = 0.5, Gridstyle = 2
```

produces a plot with a dashed grid across the entire plot region.

See also .

## Horizontal Keyword

**Used With Routines:** [BAR3D](#), [SURFACE](#)

**Corresponding System Variable:** None.

If set, causes SURFACE to only draw lines across the plot perpendicular to the line of sight. The default is for SURFACE to draw both across the plot and from front to back.

## Levels Keyword

**Used With Routines:** [CONTOUR](#), [CONTOUR2](#)

**Corresponding System Variable:** None.

Specifies a vector containing the contour levels (maximum of 150) drawn by the CONTOUR and CONTOUR2 procedures.

A contour is drawn for each level specified in *Levels*. If omitted, the data range is divided into approximately six equally-spaced levels.

### ***Example***

To draw a contour plot with levels at 1, 100, 1000, and 10000:

```
CONTOUR, Z, Levels = [1, 100, 1000, 10000]
```

To draw a contour plot with levels at 50, 60, ..., 90, 100:

```
CONTOUR, Z, Levels = FINDGEN(6) * 10 + 50
```

## Line\_Fill Keyword

**Used With Routines:** [PLOTS](#), [POLYFILL](#)

**Corresponding System Variable:** None.

Indicates that polygons are to be filled with parallel lines, rather than using solid or patterned filling methods.

When using the line-drawing method of filling, the thickness, linestyle, orientation, and spacing of the lines may be specified with keywords.

## Linestyle Keyword

**Used With Routines:** [BAR3D](#), [OPLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SURFACE](#)

**Corresponding System Variable:** [!P.Linestyle](#)

Specifies the linestyle used to draw the lines or connect data points.

---

**UNIX and OpenVMS USERS** The line join style is “miter,” i.e., the outer edges of two lines extend to meet at an angle.

---

---

**Windows USERS** The line join style is “round.”

---

The linestyle index is an integer, as shown in the following table:

<b>Index</b>	<b>X Windows Style</b>	<b>Windows Style</b>
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

### **Lower\_Only Keyword**

**Used With Routines:** [SURFACE](#)

**Corresponding System Variable:** None.

Indicates that only the lower surface of the object is to be drawn.

### **Max\_Levels Keyword**

**Used With Routines:** [PLOT](#)

**Corresponding System Variable:** [!PDT.Max\\_Levels](#)

Sets the maximum number of levels on a Date/Time axis. For example, assume that the Date/Time data contains years, months, days, hours, minutes, and seconds. If this keyword is set to three, then the Date/Time axis will show three levels: seconds, minutes, and hours.

## Max\_Value Keyword

Used With Routines: [CONTOUR](#), [CONTOUR2](#)

Corresponding System Variable: None.

Data points with values equal to or above this value are ignored when contouring. Cells containing one or more corners with values above *Max\_Value* will have no contours drawn through them.

## Month\_Abbr Keyword

Used With Routines: [PLOT](#)

Corresponding System Variable: [!PDT.Month\\_Abbr](#)

Abbreviates month or quarter names to either three characters or one character, depending on the space available on the Date/Time axis. If a one-character abbreviation does not fit, no label is used. If the complete label can fit, it is not abbreviated, even if the keyword is specified. Month names are specified in the system variable [!Month\\_Names](#). Quarter names are specified in the [!Quarter\\_Names](#) system variable.

## NLevels Keyword

Used With Routines: [CONTOUR](#), [CONTOUR2](#)

Corresponding System Variable: None.

The number of equally-spaced contour levels that are produced by [CONTOUR](#) and [CONTOUR2](#). The maximum is 150. (Default: 6)

If the *Levels* parameter, which explicitly specifies the value of the contour levels, is present this keyword has no effect. If neither parameter is present approximately six levels are drawn.

If the minimum and maximum Z values are  $Z_{\min}$  and  $Z_{\max}$ , then the value of the *i*th level is:

$$Z_{\min} + (i + 1)(Z_{\max} - Z_{\min}) / (NLevels + 1)$$

where *i* ranges from 0 to  $NLevels - 1$ .

## Noclip Keyword

Used With Routines: [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SURFACE](#), [XYOUTS](#)

**Corresponding System Variable:** `!P.Noclip`

For PLOT, CONTOUR, CONTOUR2, and SURFACE, enforces the default clipping behavior, which is to clip graphics at the boundary of the Plot Data Region (area bounded by the coordinate axes). For OPLOT, PLOTS, POLYFILL, and XYOUTS, disables clipping altogether, allowing text and graphics to be drawn anywhere in the Device Area.

## Nodata Keyword

**Used With Routines:** `AXIS`, `CONTOUR`, `CONTOUR2`, `OPLOT`, `PLOT`, `SHADE_SURF`, `SHADE_SURF_IRR`, `SURFACE`

**Corresponding System Variable:** None.

If this keyword is set, only the axes, titles, and annotation are drawn. No data points are plotted.

### *Example*

To draw an empty set of axes between some given values:

```
PLOT, [XMIN, XMAX], [YMIN, YMAX], /Nodata
```

## Noerase Keyword

**Used With Routines:** `AXIS`, `BAR`, `CONTOUR`, `CONTOUR2`, `OPLOT`, `PIE`, `PLOT`, `SHADE_SURF`, `SHADE_SURF_IRR`, `SURFACE`

**Corresponding System Variable:** None.

Specifies that the screen or page is not to be erased. By default the screen is erased, or a new page is begun, before a plot is produced.

## Normal Keyword

**Used With Routines:** `AXIS`, `BAR3D`, `CONTOUR`, `CONTOUR2`, `OPLOT`, `PLOT`, `PLOTS`, `POLYFILL`, `SHADE_SURF`, `SHADE_SURF_IRR`, `SURFACE`, `XYOUTS`

**Corresponding System Variable:** None.

Indicates that the coordinates are in the normalized coordinate system and range from 0.0 to 1.0.

When used with `AXIS`, `CONTOUR`, `CONTOUR2`, `OPLOT`, `PLOT`, `SHADE_SURF`, and `SURFACE`, indicates that the *Clip* and/or *Position* coordinates are in the normalized coordinate system and range from 0.0 to 1.0.

## **Nsum Keyword**

**Used With Routines:** `OPLOT`, `PLOT`

**Corresponding System Variable:** `!P.Nsum`

Indicates the number of data points to average when plotting.

If *Nsum* is larger than 1, every group of *Nsum* points is averaged to produce one plotted point. If there are *m* data points, then  $m / Nsum$  points are displayed. On logarithmic axes a geometric average is performed.

It is convenient to use *Nsum* when there is an extremely large number of data points to plot because it plots fewer points, the graph is less cluttered, and it is quicker.

## **Orientation Keyword**

**Used With Routines:** `MAP_XYOUTS`, `PLOTS`, `POLYFILL`, `XYOUTS`

**Corresponding System Variable:** None.

Specifies the angle in degrees, counterclockwise from horizontal, of the text baseline and the lines used to fill polygons.

When used with the `POLYFILL` procedure, this keyword forces the *Linestyle* type of fill, rather than solid or patterned fill.

## **Overplot Keyword**

**Used With Routines:** `CONTOUR`, `CONTOUR2`

**Corresponding System Variable:** None.

Indicates the `CONTOUR` or `CONTOUR2` procedure is to overplot.

No axes are drawn and the previously established scaling remains in effect. You must explicitly specify the values of the contour levels with the *Levels* keyword when using this option.

## **Path\_Filename Keyword**

**Used With Routines:** `CONTOUR`

**Corresponding System Variable:** None.

Specifies the name of a file to contain the contour positions.

If *Path\_Filename* is present, CONTOUR does not draw the contours, but rather, opens the specified file and writes the positions, in normalized coordinates, into it. The file consists of a series of logical records containing binary data. Each record is preceded with a header structure defining the contour as follows:

```
{CONTOUR_HEADER, TYPE : 0B, HIGH : 0B, $  
    LEVEL : 0, NUM : 0L, VALUE : 0.0}
```

The fields are:

- TYPE — A byte which is zero if the contour is open, and one if it is closed.
- HIGH — A byte which is 1 if the contour is closed and above its surroundings, and is 0 if the contour is below. This field is meaningless if the contour is not closed.
- LEVEL — A short integer with value greater than or equal to zero. (It is an index into the *Levels* array).
- NUM — The longword number of data points in the contour.
- VALUE — The contour value. This a single-precision floating-point value.

Following the header in each record are NUM pairs of single-precision floating (*x,y*) values, expressed in normalized coordinates.

The CONTOURFILL procedure can be used along with this file to fill the contours with specified colors or patterns. You can use CONTOUR with the *Path\_Filename* keyword to get the path information and then use CONTOURFILL to fill the closed contours.

Use of this keyword implies use of the *Follow* keyword.

## Pattern Keyword

**Used With Routines:** [PLOTS](#), [POLYFILL](#), [CONTOURFILL](#)

**Corresponding System Variable:** None.

A rectangular array of pixels (3D for CONTOURFILL) giving the fill pattern.

If this keyword parameter is omitted, POLYFILL fills the area with a solid color. The pattern array may be of any size; if it is smaller than the filled area the pattern array is cyclically repeated.

---

**Windows USERS** The *Pattern* keyword is not available for the PLOTS or POLYFILL procedure.

---

## Example

To fill the current plot window with a grid of dots:

```
Pattern = BYTARR(10, 10)
        ; Define pattern array as 10-by-10.

Pattern(5,5) = 255
        ; Set center pixel to bright.

POLYFILL, !X.Window([0, 1, 1, 0]), $
        !Y.Window([0, 0, 1, 1]), /Normal, Pattern = Pattern
        ; Fill the rectangle defined by the four corners of the window with the pattern.
```

## PClip Keyword

**Used With Routines:** [PLOTS](#), [POLYFILL](#), [XYOUTS](#)

**Corresponding System Variable:** None.

Forces [PLOTS](#), [POLYFILL](#), and [XYOUTS](#) to accept the value of !P.Clip as the clipping rectangle. Usually, this is the area bounded by the coordinate axes. By default, these routines ignore !P.Clip, allowing you to place text and graphics outside the Data Plot Region.

## Polar Keyword

**Used With Routines:** [O PLOT](#), [PLOT](#)

**Corresponding System Variable:** None.

Polar plots are produced when this keyword is present and nonzero.

The X and Y vector parameters, both of which must be present, are first converted from polar to cartesian coordinates. The first parameter is the radius, and the second is  $\theta$ , expressed in radians.

To make a polar plot:

```
PLOT, /Polar, R, Theta
```

## Position Keyword

**Used With Routines:** [AXIS](#), [BAR](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** !P.Position

Allows direct specification of the plot window.

*Position* is a four-element vector giving, in order, the coordinates  $[(x_0, y_0), (x_1, y_1)]$  of the lower-left and upper-right corners of the data window. Coordinates are expressed in normalized units ranging from 0.0 to 1.0, unless the keyword *Device* is present, in which case they are in actual device units.

When setting the position of the window, be sure to allow space for the annotation, which resides outside the window. PV-WAVE outputs the message

```
%, Warning: Plot truncated.
```

if the plot region is larger than the screen or page size. The plot region is the rectangle enclosing the plot window and the annotation.

When plotting in three dimensions, the *Position* keyword is a six-element vector with the first four elements describing, as above, the XY position, and with the last two elements giving the minimum and maximum  $z$ -coordinates. The  $Z$  specification is always in normalized coordinate units.

When making more than one plot per page it is more convenient to set !P.Multi than to manipulate the position of the plot directly with the *Position* keyword.

### **Example**

The following statement produces a contour plot with data plotted in only the upper-left quarter of the screen:

```
CONTOUR, Z, Position = [0.0, 0.5, 0.5, 1.0]
```

Because no space on the left or top edges was allowed for the axes or their annotation, the warning message described above results.

## **Psym Keyword**

**Used With Routines:** [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#)

**Corresponding System Variable:** [!P.Psym](#)

Specifies by reference number a symbol used to mark each data point. The available symbols and their corresponding reference numbers are shown in the following figure.

+	1	□	14	⊗	26	◡	39
*	2	◇	15	▷	27	◡	40
-	3	◇	16	◁	28	◡	41
◇	4	◇	17	◁	29	◡	42
△	5	◇	18	◁	30	△	43
□	6	◁	19	▷	31	▷	44
×	7	×	20	▷	32	▽	45
(User)	8	×	21	◇	33	▷	46
◡	9	×	22	⊞	34	△	47
(Histogram)	10	×	23	▷	35	▷	48
□	11	×	24	▷	36	▽	49
□	12	⊗	25	◁	37	▷	50
□	13			◁	38		

**Figure 3-1** The plot symbols and their corresponding reference numbers.

Normally,  $Psym$  is 0, data points are connected by lines, and no symbols are drawn to mark the points. Set !P.Psym to the symbol index as given in [Figure 3-1](#) to mark points with symbols. The keyword *Symsize* is used to set the size of the symbols.

Negative values of  $Psym$  cause the symbol designated by  $|Psym|$  to be plotted at each point with solid lines connecting the symbols. For example, a  $Psym$  value of  $-5$  plots triangles at each data point and connects the points with lines.

The USERSYM procedure is used to create a user-defined symbol (number 8).

For symbol number 10, Histogram, data points are plotted in the histogram mode. Horizontal and vertical lines are connect the plotted points, as opposed to the normal method of connecting points with straight lines.

When used with PLOT, OPLOT, and PLOTS, the *Psym* keyword can specify an array of plot symbols. If an array is used, each plot symbol value in the array is applied, in order, to create the plot symbols that make up the graph. The symbols are repeated, as needed, to complete the entire graph of the data set.

---

**NOTE** You cannot specify an array of values for the !P.Psym system variable. The array of color values can only be used with the *Psym* keyword.

---

See also [Solid\\_Psym](#).

### **Example**

The following code plots an array using points, and then overplots the smoothed array, connecting the points with lines:

```
PLOT, A, Psym = 3
    ; Plot using points.
OPLOT, SMOOTH(A, 7)
    ; Overplot smoothed data.
```

## **Save Keyword**

**Used With Routines:** [AXIS](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** None.

Saves the 3D to 2D transformation matrix established by SURFACE and SHADE\_SURF, and specified by the *Ax* and *Az* keywords, in the system variable field !P.T.

Use this keyword when combining the output of SURFACE and SHADE\_SURF with the output of other routines in the same plot.

When used with AXIS, the *Save* keyword parameter saves the scaling parameters established by the call back in the appropriate axis system variable, !X, !Y, or !Z. This causes subsequent overplots to be scaled to the new axis.

## **Example**

To display a two-dimensional array using SURFACE, and to then superimpose contours over the surface: (This example assumes that !P.T3d is zero, its default value.)

```
SURFACE, Z, /Save
    ; Make a surface plot and save the transformation.
CONTOUR, Z, /Noerase, /T3d
    ; Make contours, don't erase, use the 3D to 2D transform placed in
    ; !P.T by SURFACE.
```

To display a surface and to then display a flat contour plot, registered above the surface:

```
SURFACE, Z, /Save
    ; Make the surface, save transform.
CONTOUR, Z, /Noerase, /T3d, ZValue = 1.0
    ; Now display a flat contour plot, at the maximum Z value (normalized
    ; coordinates). You can display the contour plot below the surface
    ; with a ZValue of 0.0.
```

## **Size Keyword**

**Used With Routines:** [XYOUTS](#)

**Corresponding System Variable:** None.

Specifies the character size as a factor of the normal character size. Normal size is 1.0.

## **Skirt Keyword**

**Used With Routines:** [SURFACE](#)

**Corresponding System Variable:** None.

A skirt around the array at a given  $z$  value is drawn if this keyword parameter is present. The  $z$  value is expressed in data units.

For example:

```
SURFACE, A, Skirt = 100
```

draws the surface of A with a skirt at the Z value 100.

If the skirt is drawn, each point on the four edges of the surface is connected to a point on the skirt which has the given  $z$  value, and the same  $x$  and  $y$  values as the edge point. In addition, each point on the skirt is connected to its neighbor.

## Solid\_Psym Keyword

Used With Routines: [PLOT](#), [OPLOT](#), [PLOTS](#)

Corresponding System Variable: None.

When present and nonzero, symbols are drawn with solid lines no matter which linestyle is used to connect the symbols. By default, symbols are drawn with the currently specified linestyle.

## Spacing Keyword

Used With Routines: [PLOTS](#), [POLYFILL](#)

Corresponding System Variable: None.

Specifies the spacing, in centimeters, between the parallel lines used to fill polygons.

## Spline Keyword

Used With Routines: [CONTOUR](#)

Corresponding System Variable: None.

Specifies that contour paths are to be interpolated using cubic splines.

Use of this keyword implies the use of the *Follow* keyword. The appearance of contour plots of arrays with low resolution may be improved by using spline interpolation. In rare cases, contour lines that are close together may cross because of interpolation.

Splines are especially useful with small data sets (less than 15 array dimensions). With larger data sets the smoothing is not as noticeable and the expense of splines increases rapidly with the number of data points.

You may specify the length of each interpolated line segment in normalized coordinates by including a value with this keyword. The default value is 0.005 which is obtained when the parameter *Spline* is present. Smaller values for this parameter yield smoother lines, up to the resolution of the output device, at the expense of more computations.

## Start\_Level Keyword

Used With Routines: [PLOT](#)

Corresponding System Variable: [!PDT.Start\\_Level](#)

Specifies the initial level of tick labels to be displayed on a Date/Time axis. The subsequent levels depend on the data range and the first level selected.

Index Values for the Start\_Level Keyword

Index	Start Level
7	Year
6	Quarter
5	Month
4	Week
3	Day
2	Hour
1	Minute
0	Second
-1	Auto-level

## Subtitle Keyword

Used With Routines: [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PIE](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

Corresponding System Variable: [!P.Subtitle](#)

Produces a subtitle under the  $x$ -axis containing the text in this string parameter.

## Symsize Keyword

Used With Routines: [OPLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#)

Corresponding System Variable: None.

Specifies the size of the symbols drawn when *Psym* is set. The default size of 1.0 produces symbols approximately the same size as a character.

When used with [PLOT](#), [OPLOT](#), and [PLOTS](#), the *Symsize* keyword can specify an array of symbol sizes. If an array is used, each plot symbol size in the array is

applied, in order, to size the plot symbols that make up the graph. The symbol sizes are repeated, as needed, to complete the entire graph of the data set.

---

**NOTE** You cannot specify an array of values for the !P.Psym system variable. The array of color values can only be used with the *Psym* keyword.

---

## T3d Keyword

**Used With Routines:** [AXIS](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [PLOTS](#), [POLYFILL](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [XYOUTS](#)

**Corresponding System Variable:** !P.T3d

---

**NOTE** When the *T3d* keyword is in effect, the *CONTOUR2 Fill* keyword is disabled.

---

A keyword flag which, if present, indicates that the generalized transformation matrix in !P.T is to be used.

!P.T *must* contain a valid transformation matrix before the *T3d* keyword can be used. The matrix can be set by using the *Save* plotting keyword with the appropriate plotting routine.

If *T3d* is not present the user-supplied coordinates are simply scaled to screen coordinates. See the examples in the description of the *Save* plotting keyword.

A valid transformation matrix can be placed in !P.T in several ways:

- Use the *Save* keyword to save the transformation matrix from an earlier graphics operation.
- Establish a transformation matrix using the T3D user library procedure.
- Set the value of !P.T directly.

## Text\_Axes Keyword

**Used With Routines:** [XYOUTS](#)

**Corresponding System Variable:** None.

Specifies the plane of vector-drawn text when three-dimensional plotting is enabled. By default, text is drawn in the plane of the XY axes. The horizontal text direction is in the X plane, and the vertical text direction is in the Y plane.

Values of this keyword may range from 0 to 5, with the following effect: 0 for XY, 1 for XZ, 2 for YZ, 3 for YX, 4 for ZX, and 5 for ZY. The notation ZY means that the horizontal direction of the text lies in the Z plane, and the vertical direction of the text is drawn in the Y plane.

## Thick Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** `!P.Thick`

Controls the thickness of the lines connecting points. A thickness of 1.0 is normal, 2.0 is double-wide, etc.

## Tickformat Keyword

**Used With Routines:** [AXIS](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

**Corresponding System Variable:** `!P.Tickformat`

Lets you use FORTRAN-style format specifiers to change the format of tick labels on the x-, y-, and z-axes. For example:

```
PLOT, mydata, Tickformat = '(F5.2)'
```

The resulting plot's tick labels are formatted with a total width of five characters carried to two decimal places. As expected, the width field expands automatically to accommodate larger values. For more information on format specifiers, see . See also .

Note that only the I (integer), F (floating-point), and E (scientific notation) format specifiers can be used with *Tickformat*. Also, you cannot place a quoted string inside a tick format. For example, ("`<`", `F5.2`, "`>`") is an invalid *Tickformat* specification.

See also [\[XYZ\]Tickformat](#).

## Ticklen Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** `!P.Ticklen`

Controls the length of the axis tick marks, expressed as a fraction of the window size. The default value is 0.02. Ticklen of 0.5 produces a grid, while a negative *Ticklen* makes tick marks that extend outside the plot region, rather than inwards.

### **Example**

To produce outward-going tick marks of the normal length:

```
PLOT, X, Y, Ticklen = -0.02
```

To provide a new default tick length, set the system variable !P.Ticklen.

## **Title Keyword**

**Used With Routines:** [AXIS](#), [BAR](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [O PLOT](#), [PIE](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

**Corresponding System Variable:** !P.Title

Produces a main title centered above the plot window.

The text size of this main title is larger than the other text by a factor of 1.25.

For example:

```
PLOT, X, Y, Title = 'Final Results'
```

## **Upper\_Only Keyword**

**Used With Routines:** [SURFACE](#)

**Corresponding System Variable:** None.

Indicates that only the upper surface of the object is to be drawn. By default, both surfaces are drawn.

## **Week\_Boundary Keyword**

**Used With Routines:** [PLOT](#)

**Corresponding System Variable:** !PDT.Week\_Boundary

Sets the day of the week on which week tick marks are drawn for the week level on a Date/Time axis.

For example, if you set the week boundary to 'Sunday', weekly tick marks are drawn for each Sunday and each one is labeled with the date.

## Week Boundary Indices

<b>Index</b>	<b>Day of Week</b>
0	Sunday
1 (the default)	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

## Width Keyword

**Used With Routines:** [MAP\\_XYOUTS](#), [XYOUTS](#)

**Corresponding System Variable:** None.

Returns the width of the text string, in normalized coordinate units, to the designated variable.

For example, to put the width of a text string in the variable `w_title`, use:

```
XYOUTS, x, y, 'Title of Graph', Width = w_title
```

## [XY]Axis Keyword

**Used With Routines:** [AXIS](#)

**Corresponding System Variable:** None.

The *XAxis* and *YAxis* keywords indicate which type of axis is to be drawn by the `AXIS` procedure and its placement.

See also [ZAxis](#).

## [XYZ]Charsize Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** `![XYZ].Charsize`

The size of the characters used to annotate the axis and its title.

This field is a scale factor applied to the global scale factor set by !P.Charsize or the keyword *Charsize*.

See also *Charsize*.

## [XYZ]Gridstyle Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

**Corresponding System Variable:** [!\[XYZ\].Gridstyle](#)

Lets you change the linestyle of tick intervals on the *x*-, *y*-, and *z*-axes.

The default is a solid line.

Index	X Windows Style	Windows Style
0	Solid (default)	Solid (default)
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

See also *Gridstyle*.

## [XYZ]Margin Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PIE](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** [!\[XYZ\].Margin](#)

A two-element array specifying the margin around the sides of the plot window, in units of character size. Default margins are 10 (left margin) and 3 (right margin) for the *x*-axis, 4 (bottom margin) and 2 (top margin) for the *y*-axis. For the *z*-axis the default margins are both 0.

## [XYZ]Minor Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** `![XYZ].Minor`

The number of minor tick intervals on the particular axis. If set to 0, the default, PV-WAVE automatically determines the number of minor ticks in each major tick mark interval. Setting this parameter to  $-1$  suppresses the minor ticks, and setting it to a positive, nonzero number  $n$  produces  $n$  minor tick intervals, and  $n - 1$  minor tick marks.

## [XYZ]Range Keyword

**Used With Routines:** [AXIS](#), [BAR](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [vtkSCATTER](#), [vtkSURFACE](#)

**Corresponding System Variable:** `![XYZ].Range`

The desired data range of the particular axis, a two-element vector. The first element is the axis minimum, and the second is the maximum. PV-WAVE will frequently round this range.

## [XYZ]Style Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#)

**Corresponding System Variable:** `![XYZ].Style`

Allows specification of axis options such as rounding of tick values and selection of a box axis. Each option is encoded in a bit. See the following table for details:

Axis Options

Bit	Value	Function
0	1	Exact. By default the end points of the axis are rounded in order to obtain even tick increments. Setting this bit inhibits rounding, making the axis fit the data range exactly.
1	2	Extend. If this bit is set, the axes are extended by 5% in each direction, leaving a border around the data.
2	4	None. If this bit is set, the axis and its text is not drawn.

## Axis Options (Continued)

Bit	Value	Function
3	8	No box. Normally, PLOT, CONTOUR, and CONTOUR2 draw a box style axis with the data window surrounded by axes. Setting this bit inhibits drawing the top or right axis.
4	16	Inhibits setting the y-axis minimum value to zero, when the data are all positive and nonzero. The keyword <i>YNozero</i> sets this bit temporarily.

---

**NOTE** The *ZStyle* keyword has no effect in Date/Time plots.

---

## [XYZ]Tickformat Keyword

**Used With Routines:** [AXIS](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

**Corresponding System Variable:** [!\[XYZ\].Tickformat](#)

Lets you use FORTRAN-style format specifiers to change the format of tick labels for the particular axis. For example:

```
PLOT, mydata, XTickformat = '(F5.2)'
```

This keyword works basically the same way as the *Tickformat* keyword.

See also [Tickformat](#).

## [XYZ]Ticklen Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

**Corresponding System Variable:** [!\[XYZ\].Ticklen](#)

Functions the same as the keyword *Ticklen*. *[XYZ]Ticklen*, however, can be applied to the particular axis. *[XYZ]Ticklen* supersedes the value of the *Ticklen* setting.

## [XYZ]Tickname Keyword

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [vtkAXES](#)

**Corresponding System Variable:** [!\[XYZ\].Tickname](#)

A string array, of up to 30 elements, containing the annotation of each major tick mark.

If omitted, or if a given string element that contains the null string, PV-WAVE labels the tick mark with its value. To suppress the tick label, supply a string array of one-character-long blank strings. You can do this with the command:

```
REPLICATE ( ' ', N)
```

(Null strings cause PV-WAVE to number the tick mark with its value.) Note that if there are  $n$  tick mark intervals, there are  $n + 1$  tick marks and labels.

### **[XYZ]Ticks Keyword**

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [vtkAXES](#)

**Corresponding System Variable:** `![XYZ].Ticks`

The number of major tick intervals to draw for the axis. If omitted PV-WAVE will select from three to six tick intervals. Setting this field to  $n$ , where  $n > 0$ , produces exactly  $n$  tick intervals, and  $n + 1$  tick marks.

### **[XYZ]Tickv Keyword**

**Used With Routines:** [AXIS](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [vtkAXES](#)

**Corresponding System Variable:** `![XYZ].Tickv`

The data values for each tick mark, an array of up to 30 elements.

This keyword allows you to directly specify tick data values, producing graphs with non-linear tick marks. PV-WAVE scales the axis from the first tick value to the last, unless you directly specify a range. If you specify  $n$  tick intervals, you must specify  $n + 1$  tick values.

### **[XYZ]Title Keyword**

**Used With Routines:** [AXIS](#), [BAR](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SHADE\\_SURF\\_IRR](#), [SURFACE](#), [vtkAXES](#), [vtkSCATTER](#), [vtkSURFACE](#)

**Corresponding System Variable:** `![XYZ].Title`

Places a title below the particular axis.

See also *Title*.

## [XYZ]Type Keyword

**Used With Routines:** [AXIS](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#)

**Corresponding System Variable:** ![XYZ].Type.

Specifies a linear axis if zero; specifies a logarithmic axis if one; and if set to 2, enables compressed Julian numbers to be used directly with the PLOT or OPLOT procedures. Holds the value of the last plot's [XYZ]Type used, but is not used for plotting of subsequent plots (except by OPLOT).

---

**NOTE** *YType* has no effect in Date/Time plots.

---

## YNozero Keyword

**Used With Routines:** [AXIS](#), [OPLOT](#), [PLOT](#)

**Corresponding System Variable:** None.

Inhibits setting the minimum y-axis value to zero when the y data are all positive and nonzero, and no explicit minimum y value is specified (using *Yrange*, or !Y.Range).

By default, the y-axis spans the range of 0 to the maximum value of y, in the case of positive y data. Set bit 4 in !Y.Style to make this option the default.

## YLabelCenter Keyword

**Used With Routines:** [AXIS](#), [BAR](#), [BAR2D](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [PLOT\\_FIELD](#), [SURFACE](#)

**Corresponding System Variable:** None.

Controls whether the top and bottom major tick labels on a Y axis will be positioned within the boundaries of the axis box or centered across from the corresponding major tick.

If this keyword is set, the top and bottom Y axis major tick labels will be centered vertically with corresponding major ticks. If this keyword is not set, the default behavior is to position the top and bottom Y axis major tick labels within the boundaries of the axis box.

### **Example**

To produce a plot with with top and bottom major tick labels on the Y axis centered across from the corresponding major tick:

```
PLOT, DIST(20), YTICKS=10, /YLabelCenter
```

### **Z Keyword**

**Used With Routines:** [PLOTS](#), [POLYFILL](#), [XYOUTS](#)

**Corresponding System Variable:** None.

Provides the  $z$ -coordinate if a  $z$  parameter is not present in the call. This is of use only if the three-dimensional transformation is in effect.

### **ZAxis Keyword**

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [SURFACE](#)

**Corresponding System Variable:** None.

Specifies the existence of a  $z$ -axis for [CONTOUR](#) and [CONTOUR2](#), and the placement of the  $z$ -axis for [SURFACE](#). For [AXIS](#), the *ZAxis* keyword indicates that a  $z$ -axis is to be drawn and where it should be placed.

[CONTOUR](#) and [CONTOUR2](#) draw no  $z$ -axis by default. Include the *ZAxis* keyword in the call to [CONTOUR](#) and [CONTOUR2](#) to draw a  $z$ -axis. This is of use only if a three-dimensional transformation is established.

By default, [SURFACE](#) draws the  $z$ -axis at the upper-left corner of the axis box. To suppress the  $z$ -axis, use `ZAxis = -1` in the call. The position of the  $z$ -axis is determined from *ZAxis* as follows:

1 = lower-right, 2 = lower-left, 3 = upper-left, and 4 = upper-right.

### **ZValue Keyword**

**Used With Routines:** [AXIS](#), [BAR3D](#), [CONTOUR](#), [CONTOUR2](#), [OPLOT](#), [PLOT](#), [SHADE\\_SURF](#), [SURFACE](#)

**Corresponding System Variable:** None.

Sets the  $z$ -coordinate, in normalized coordinates in the range of 0 to 1, of the axis and data output from [PLOT](#), [OPLOT](#), [CONTOUR](#), and [CONTOUR2](#).

This has an effect only if `!P.T3d` is set and the three-dimensional to two-dimensional transformation is stored in `!P.T`. If *ZValue* is not specified, [CONTOUR](#) and

CONTOUR2 will output each contour at its z-coordinate, and the axes and title at a z-coordinate of 0.0.

## System Variables

This chapter discusses the PV-WAVE system variables. For more information on system variables, see Chapter 2, *Constants and Variables*, in the *PV-WAVE Programmer's Guide*.

### !C

The cursor system variable. Currently, its only function is to contain the subscript of the largest or smallest element found by the MAX and MIN functions.

### !Century\_Divider

The !Century\_Divider system variable is intended to help ensure that two-digit-year format dates do not adversely affect existing PV-WAVE applications. This is intended as a temporary solution to the year 2000 problem. We recommend that all PV-WAVE code be updated with four-digit year dates.

If set to -1 (the default), PV-WAVE behaves as in versions before 6.21. That is, any date in a two-digit year format is interpreted as being between January 1, 1901 and December 31, 1999. Furthermore, when !Century\_Divider is set to -1, a two-digit year of 00 is interpreted as Julian date zero (September 9, 1752). For more information on how dates are handled in PV-WAVE, refer to the chapter *Chapter 8, Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

If set in the range {0..99}, the !Century\_Divider value represents the maximum of the hundred year span encompassed by two-digit year dates. In other words, if !Century\_Divider is set to 35, then all two-digit year dates encountered in PV-WAVE are interpreted to be in the range {1935..2034}. Therefore, the date

01-31-35 refers to January 31, 1935; 01-31-99 refers to January 31, 1999; and 01-31-34 refers to January 31, 2034.

If !Century\_Divider is set to 0 or a value greater than 99, the two-digit year is added to 2000.

## **!D**

A structure containing information about the current graphics output device. Fields are described in the following sections.

The fields of the !D structure are read-only.

### **!D.Display\_Depth**

Stores the current display depth.

### **!D.Fill\_Dist**

The line interval, in device coordinates, required to obtain a solid fill.

### **!D.Flags**

A longword of flags. Each bit is a flag encoded as shown in the following table:

!D.Flags Bit Definitions

<b>Bit</b>	<b>Value</b>	<b>Function</b>
0	1	Device has scalable pixel size (e.g., PostScript).
1	2	Device can control line thickness with hardware.
2	4	Device can output text at an arbitrary angle using hardware.
3	8	Device can display images.
4	16	Device supports color.
5	32	Device can polygon fill with hardware.
6	64	Device hardware characters are monospace.
7	128	Device can read pixels (TVRD).
8	256	Device supports windows.

## **!D.N\_Colors**

The number of simultaneously available colors. In the case of devices with windows, this field is set after the window is initialized. For monochrome systems, !D.N\_Colors is 2, and for color systems it is normally 256.

## **!D.Name**

A string containing the name of the device.

## **!D.Table\_Size**

This field contains the number of color table indices available on the device. Devices without color tables have this field set to 0.

## **!D.Unit**

The logical number of the file open for output by the current graphics device. This field only has meaning if the file is accessible to the user from PV-WAVE, and is 0 if no file is open. For example, the PostScript driver maintains this field with the unit number of the file open for PostScript output. In the case of Tektronix output to a file, !D.Unit may be set to either + or – the logical unit number.

## **!D.Window**

The index of the currently open window. Set to –1 if no window is open. Used only with devices that support windows.

## **!D.X\_Ch\_Size / !D.Y\_Ch\_Size**

The normal width and height of a character in device units. These fields are set after the window is initialized.

## **!D.X\_Px\_Cm / !D.Y\_Px\_Cm**

The number of pixels per centimeter in the  $x$  and  $y$  directions.

## **!D.X\_Size / !D.Y\_Size**

Reports the width and height of the current graphics window in pixels. The values are updated when the window is resized.

## **!D.X\_Vsize / !D.Y\_Vsize**

Reports the height and width of the current graphics window in pixels. The values are updated when the window is resized.

## **!Date\_Separator**

A string containing, by default, a slash (/) character. This character is used to separate the parts of a date on output (for example, 5/15/1992). To use a different character, change the value of this variable. This variable is used by the DT\_PRINT, DT\_TO\_STR, and DC\_WRITE\_\* routines.

## **!Day\_Names**

An array of strings containing the names of the days of the week. This system variable is used by the DAYNAME function to return the names of the days for a specified date/time variable.

## **!Dir**

Contains the name of the main PV-WAVE directory (that is, the directory containing the files PV-WAVE needs to run).

## **!Display\_Size**

Contains the pixel dimensions of the display screen.

## **!Dpi**

Contains the double-precision value of pi. This is a read-only system variable.

## **!DT\_Base**

Contains the value of Julian Day 1 (September 14, 1752) as a !DT structure. Used in various Date/Time calculations. This value can be overridden using the *Base* keyword as a parameter to the SEC\_TO\_DT and DT\_TO\_SEC routines.

This variable can also be modified directly; however, if you do this, you must set the last field (the recalc flag) of the !DT structure to 1. For more information, see the section *Recalc Flag* in *Chapter 8, Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

## **!Dtor**

Contains the conversion factor to convert degrees to radians. The value is  $\pi / 180$ , which is approximately 0.01745. This is a read-only system variable.

## **!Edit\_Input**

Enables or disables the keyboard line-editing feature.

## **!Err**

Contains the code of the last I/O error message. The `CURSOR` function also uses `!Err` to store the return value of the function. This enables the user to determine which mouse button was pushed.

## **!Err\_String**

Contains the text of the last I/O error message. This is a read-only system variable.

## **!Holiday\_List**

This system variable is created by the `CREATE_HOLIDAYS` procedure. It is a Date/Time variable containing holidays as defined by `CREATE_HOLIDAYS`. This system variable does not have a default value. This variable can hold up to 50 holiday definitions.

## **!Journal**

Contains the logical unit number of journal output. If there is no journal output, `!Journal = 0`. This is a read-only system variable.

## **!Lang**

Identifies the language currently being used. The default is `american`.

## **!Month\_Names**

An array of strings containing the names of the months. This system variable is used by the `MONTH_NAME` function to return the names of the months for a specified date/time variable.

## **!Mouse**

Used by the CURSOR function to store the *x* and *y* position of the mouse, the mouse button status, and a date/time stamp. The fields for this variable are:

!Mouse.X  
!Mouse.Y  
!Mouse.Button  
!Mouse.Time

For additional information on this variable and its fields, see the CURSOR procedure.

## **!Msg\_Prefix**

Contains a prefix string for error messages issued by PV-WAVE. The default is a percent sign. This system variable provides a way to distinguish error messages from normal output.

## **!Order**

Controls the direction of image transfers. If !Order is 0, images are transferred from bottom to top (i.e., the row with a 0 subscript is written on the bottom). Setting !Order to 1 transfers images from top to bottom.

## **!P**

The main system variable structure for plotting.

All fields, except !P.Multi, have a directly corresponding keyword parameter in the main plot procedures: AXIS, PLOT, OPLOTT, CONTOUR, and SURFACE.

The !P structure fields are explained in the following sections.

### **!P.Background**

**Corresponding Plot Keyword:** [Background](#)

The background color index. When erasing a window, all pixels are set to this color. The default value is 0.

### **!P.Charsize**

**Corresponding Plot Keyword:** [Charsize](#)

The overall character size of all annotation. The normal size is 1. The main plot title size is 1.25 times this parameter.

---

**NOTE** If you use !P.Multi to create a multiple plot of more than two rows or columns, PV-WAVE decreases the character size by a factor of two.

---

## **!P.Charthick**

**Corresponding Plot Keyword:** [Charthick](#)

The thickness of characters drawn with the vector fonts. Normal thickness is 1.0, double thickness is 2.0, and so on.

## **!P.Clip**

**Corresponding Plot Keyword:** None.

Contains a six-element vector. The first four elements specify in device coordinates a rectangle used to clip the graphics window. The rectangle is specified in the form  $[(x_0, y_0), (x_1, y_1)]$ . The coordinates specify the lower-left and upper-right corners of the clipping rectangle, respectively. Normally the clipping rectangle is set to the Data Plot Area (the area bounded by the coordinate axes). The last two elements of this system variable are reserved for internal use by PV-WAVE.

## **!P.Color**

**Corresponding Plot Keyword:** [Color](#)

Color index used to draw data, axes, and annotation.

## **!P.Font**

**Corresponding Plot Keyword:** [Font](#)

The index of the graphics text font. !P.Font = -1 uses the software-drawn fonts (also called Hershey or vector-drawn fonts). !P.Font = 0 uses the hardware-drawn fonts. For information on hardware-drawn fonts available for a particular output device, see [Appendix B, Output Devices and Window Systems](#). See for a complete description of the vector-drawn fonts.

---

**NOTE** Hardware font drivers that support 3D transformations include X Windows, WIN32 (on Windows NT platforms only), PostScript, and WMF (on Windows NT platforms only).

---

## **!P.Gridstyle**

**Corresponding Plot Keyword:** [Gridstyle](#)

Lets you change the default linestyle of  $x$ -,  $y$ -, and  $z$ -axis tick marks. The default is a solid line. Other linestyle choices and their index values are listed in the following table:

Index	X Windows Style	Windows Style
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

## !P.Linestyle

**Corresponding Plot Keyword:** [Linestyle](#)

The style of the lines used to connect points.

---

**UNIX and OpenVMS USERS** The line join style is “miter,” i.e., the outer edges of two lines extend to meet at an angle.

---



---

**Windows USERS** The line join style is “round.”

---

The linestyle index is an integer, as described in the following table:

Index	X Windows Style	Windows Style
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

## !P.Multi

Allows making multiple plots on a page or a screen. It is a five-element integer array defined as follows:

- !P.Multi(0) contains the number of plots remaining on the page. If !P.Multi(0) is less than or equal to 0, the page is cleared, the next plot is placed the upper-left-hand corner, and !P.Multi(0) is reset to the number of plots per page.
- !P.Multi(1) is the number of plot columns per page. If  $\leq 0$ , one is assumed. If more than two plots are positioned in either the  $x$  or  $y$  direction, the character size is halved.
- !P.Multi(2) is the number of rows of plots per page. If  $\leq 0$ , one is assumed.
- !P.Multi(3) is the number of plots stacked in the  $z$  dimension.
- !P.Multi(4) is 0 to make plots from left to right (column major), and top to bottom, and is 1 to make plots from top to bottom, left to right (row major).

---

**NOTE** If more than two rows or columns of plots are produced, PV-WAVE decreases the character size by a factor of 2.

---

### Example

To position two plots across the page:

```
!P.Multi = [0, 2, 0, 0, 0]
PLOT, X0, Y0 ;Make left plot
PLOT, X1, Y1 ;Right plot
```

To position two plots vertically:

```
!P.Multi = [0, 0, 2, 0, 0]
PLOT, X0, Y0 ;Make top plot
PLOT, X1, Y1 ;Bottom plot
```

To make four plots per page, two across and two up and down:

```
!P.Multi = [0, 2, 2, 0, 0]
```

and then call plot four times.

To reset !P.Multi back to the normal one plot per page:

```
!P.Multi = 0
```

For more information on !P.Multi, see the *PV-WAVE User's Guide*.

### !P.NoClip

Corresponding Plot Keyword: [NoClip](#)

If set, this keyword disables the default clipping for the OPLOT procedure. By default, OPLOT uses the value of !P.Clip for its clipping rectangle. If you set !P.NoClip=1, then this default clipping is disabled for OPLOT. The *Clip* keyword takes precedence over the setting of !P.NoClip.

## !P.Nsum

**Corresponding Plot Keyword:** [Nsum](#)

The number of adjacent points to sum to obtain a plotted point. If !P.Nsum is larger than 1, every group of !P.Nsum points is averaged to produce one plotted point. If there are  $m$  data points, then  $m / !P.Nsum$  points are displayed. On logarithmic axes a geometric average is performed.

It is convenient to use *Nsum* when there is an extremely large number of data points to plot because it plots fewer points, the graph is less cluttered, and it is quicker.

## !P.Position

**Corresponding Plot Keyword:** [Position](#)

Specifies the position of the plot within the graphics window. It is called by specifying a four-element vector as follows:

```
!P.Position = [Xmin, Ymin, Xmax, Ymax]
```

The minimum and maximum position values are specified in normalized coordinates. For example, to position a plot in the center of the screen, type:

```
!P.Position = [0.2, 0.2, 0.8, 0.8]
```

---

**NOTE** The !P.Position variable will only position the plot itself, and the annotation on the axes may be cut off. If the plot and its associated annotation must be positioned, use the Region field of the !P system variable.

---

## !P.Psym

**Corresponding Plot Keyword:** [Psym](#)

Specifies by reference number a symbol used to mark each data point. Each point drawn by PLOT and OPLOT is marked with a symbol if this field is non-zero. The available symbols and their corresponding reference numbers are shown in the following figure.

+	1	□	14	⊗	26	◡	39
*	2	◇	15	▷	27	◡	40
-	3	◇	16	◁	28	◡	41
◇	4	◇	17	◁	29	◡	42
△	5	◁	18	△	30	△	43
□	6	▷	19	▷	31	▷	44
×	7	×	20	▷	32	▽	45
(User)	8	×	21	◇	33	▷	46
◡	9	×	22	⊞	34	△	47
(Histogram)	10	×	23	▷	35	▷	48
□	11	×	24	▷	36	▽	49
□	12	⊗	25	◁	37	▷	50
□	13			◁	38		

**Figure 4-1** The plot symbols and their corresponding reference numbers.

The USERSYM procedure is used to create a user-defined symbol (number 8).

For symbol number 10, Histogram, data points are plotted in the histogram mode. Horizontal and vertical lines are connect the plotted points, as opposed to the normal method of connecting points with straight lines.

Negative values of !P.Psym cause the symbol designated by !|P.Psym| to be plotted at each point with solid lines connecting the symbols. For example, a !P.Psym value of -5 plots triangles at each data point and connects the points with lines.

## **!P.Region**

Specifies the positioning of the plot and its associated annotation (i.e., anything that can be specified with the PLOT command). The difference between !P.Region and !P.Position is that !P.Region provides a margin around the plot to accommodate plot annotation. It is called by specifying !P.Region's maximum and minimum values in normalized coordinates:

```
!P.Region = [Xmin, Ymin, Xmax, Ymax]
```

For example, to position a plot with its associated annotation in the center of the screen, type:

```
!P.Region = [0.2, 0.2, 0.8, 0.8]
```

## **!P.Subtitle**

**Corresponding Plot Keyword:** [Subtitle](#)

The plot subtitle, placed under the  $x$ -axis label.

## **!P.T**

Contains the homogeneous 4-by-4 transformation matrix.

## **!P.T3D**

**Corresponding Plot Keyword:** [T3d](#)

Enables the three-dimensional to two-dimensional transformation contained in the homogeneous 4-by-4 matrix !P.T.

## **!P.Thick**

**Corresponding Plot Keyword:** [Thick](#)

The thickness of the lines connecting points. A thickness of 1.0 is normal, 2 is double-wide, and so on.

## **!P.Tickformat**

**Corresponding Plot Keyword:** [Tickformat](#)

Changes the default format for  $x$ -,  $y$ -, and  $z$ -axis tick labels using FORTRAN-style format specifiers.

See [!\[XYZ\].Tickformat](#) for more information.

## **!P.Ticklen**

**Corresponding Plot Keyword:** [Ticklen](#)

The length of the tick marks, expressed as a fraction of the plot size. The default value is 0.02. A !P.Ticklen of 0.5 produces a grid, while a negative value for !P.Ticklen makes tick marks that extend outside the window, rather than inwards.

See also [!\[XYZ\].Ticklen](#).

## **!P.Title**

**Corresponding Plot Keyword:** [Title](#)

The main plot title. The text size of this main title is larger than the other text by a factor of 1.25.

## **!Path**

Contains the colon-separated directory path to search for procedures, functions, and arguments of executive commands.

## **!PDT**

The main system variable structure for Date/Time plotting attributes. The fields of this structure are described below.

### **!PDT.Box**

**Corresponding Plot Keyword:** [Box](#)

If zero, the background box for the Date/Time axis labels is off; if one, the background box is turned on. (Default: 0—no box)

### **!PDT.Compress**

**Corresponding Plot Keyword:** [Compress](#)

If zero, compression is off; if one, compression is on. (Default: 0—no compression)

### **!PDT.Exclude\_Holiday**

If compression is set with the *Compress* keyword or !PDT.Compress, and this system variable is set to one, holidays are excluded from the results of Date/Time routines such as DT\_ADD, DT\_SUBTRACT, and DT\_DURATION. (Default: 1)

## **!PDT.Exclude\_Weekend**

If compression is set with the *Compress* keyword or !PDT.Compress, and this system variable is set to one, weekends are excluded from the results of Date/Time routines such as DT\_ADD, DT\_SUBTRACT, and DT\_DURATION. (Default: 1)

## **!PDT.Max\_Levels**

**Corresponding Plot Keyword:** [Max\\_Levels](#)

Sets the maximum number of levels on a Date/Time axis. For more information, see the *Max\_Levels* keyword in [Chapter 3, Graphics and Plotting Keywords](#).

## **!PDT.Month\_Abbr**

**Corresponding Plot Keyword:** [Month\\_Abbr](#)

If one, month names are abbreviated if there is not enough room to draw them on a plot axis; if zero, month names will not be abbreviated, and only the months which fit the space available on the axis will be displayed. (Default: 0—months are not abbreviated)

## **!PDT.Start\_Level**

**Corresponding Plot Keyword:** [Start\\_Level](#)

Allows you to override the starting level of the Date/Time axis labels that PV-WAVE derives. The default is -1, which causes starting levels to be selected automatically.

## **!PDT.DT\_Crange**

Contains axis ranges generated by the PLOT procedure.

## **!PDT.DT\_Range**

**Corresponding Plot Keyword:** [DT\\_Range](#)

Can be used to specify an exact Date/Time axis range. You must pass in the desired start and end values from a Date/Time Julian value. The specified range may be adjusted slightly depending on the data. To force an exact axis range (exactly as specified), set the *XStyle* keyword to two. See the description of *XStyle* in [Chapter 3, Graphics and Plotting Keywords](#).

## **!PDT.DT\_Offset**

This is a read-only system variable, which is mainly for internal use.

## **!PDT.Week\_Boundary**

**Corresponding Plot Keyword:** [Week\\_Boundary](#)

Allows you to set a different day of the week as the boundary for numbering the start of the week on the axis levels. (0 = Sunday, 1= Monday, etc.)

## **!Pi**

The floating-point value of pi. This is a read-only system variable.

## **!Prompt**

A string variable containing the prompt used by PV-WAVE.

## **!Quarter\_Names**

Array of strings containing the names for fiscal quarters. The default names of these labels are Q1, Q2, Q3, and Q4.

## **!Quiet**

This system variable is used to suppress error messages. It can have the following values:

<b>Value</b>	<b>Purpose</b>
0	Print all messages.
1	Suppress compiler and informational messages only.
2	Suppress error messages only.
3	Suppress compiler, informational, and error messages.

## **!Radeq**

A floating-point value for converting radians to degrees. The value is  $180 / \pi$  or approximately 57.2958. This is a read-only system variable.

## **!Start**

Contains the value of the time at which you started PV-WAVE as a date/time structure.

## **!Time\_Separator**

A string containing, by default, a colon (:) character. This character is used to separate the parts of a time (for example, 07:54:58.000). To use a different character, change the value of this variable. This variable is used by the DT\_PRINT, DT\_TO\_STR, and DC\_WRITE\_\* routines.

## **!Version**

A structure whose five string fields contain the:

- architecture
- current operating system
- current PV-WAVE release number
- current PV-WAVE revision level
- name (platform) of the machine running PV-WAVE.

This is a read-only system variable.

```
PRINT, !Version.release + !Version.revision
```

## **!Weekend\_List**

This system variable is created by the CREATE\_WEEKENDS procedure. It contains an array of long integers, where ones represent weekends and zeros represent weekdays. The values are defined by the CREATE\_WEEKENDS routine.

## **![XYZ]**

The system variables !X, !Y, and !Z, are structures that affect the appearance and scaling of the three axes. The fields for !X are described here. !Y and !Z have identical fields with identical meanings and usage.

In addition, almost all fields have corresponding keyword parameters, with identical function, but with temporary effect. For example, to suppress the minor tick marks on the *x*-axis using the !X system variable:

```
!X.Minor = -1
```

while to suppress them in the call to PLOT:

```
PLOT, X, Y, XMinor = -1
```

The name of the keyword parameter is simply the name of the system variable field, prefixed with the letter X, Y, or Z.

The fields for the !X system variable are explained in the following sections.

## ![XYZ].Charsize

**Corresponding Plot Keyword:** [\[XYZ\]Charsize](#)

The size of the characters used to annotate the axis and its title. This field is a scale factor applied to the global scale factor. For example, setting !P.Charsize to 2.0 and !X.Charsize to 0.5 results in a character size of 1.0 for the  $x$ -axis.

## ![XYZ].Crange

The *output* axis range. Setting this variable has no effect; set !X.Range to change the range. !X.Crange(0) always contains the minimum axis value, and !X.Crange(1) contains the maximum axis value of the last plot.

## ![XYZ].Gridstyle

**Corresponding Plot Keyword:** [\[XYZ\]Gridstyle](#)

Lets you change the default linestyle of tick marks along the  $x$ -,  $y$ -, and  $z$ -axes. The default is a solid line. Other linestyle choices and their index values are listed in the following table:

Index	X Windows Style	Windows Style
0	Solid	Solid
1	Dotted	Short dashes
2	Dashed	Long dashes
3	Dash dot	Long-short dashes
4	Dash-dot-dot-dot	Long-short-short dashes
5	Long dashes	Long dashes

## ![XYZ].Margin

**Corresponding Plot Keyword:** [\[XYZ\]Margin](#)

A two-element array specifying the margin around the sides of the plot window, in units of character size. Default margins are 10 (left margin) and 3 (right margin) for the  $x$ -axis, 4 (bottom margin) and 2 (top margin) for the  $y$ -axis. For the  $z$ -axis the default margins are both 0.

When calculating the size and position of the plot window, PV-WAVE first determines the plot region, the area enclosing the window plus the axis annotation and titles. It then subtracts the appropriate margin from each side, obtaining the window.

## ![XYZ].Minor

**Corresponding Plot Keyword:** [XYZ]Minor

The number of minor tick intervals. If ![XYZ].Minor is 0, the default, the number of minor ticks is automatically determined. You can force a given number of minor tick intervals by setting this field to the desired number. To suppress minor tick marks, set ![XYZ].Minor to -1.

## ![XYZ].Range

**Corresponding Plot Keyword:** [XYZ]Range

The input axis range, a two-element vector. The first element is the axis minimum, and the second is the maximum. Set this field, or use the corresponding keyword parameter, to specify the data range to plot. Because the endpoints of axes are rounded, the final axis range may not be equal to this input range. The field ![XYZ].Range contains the axis range used for the plot.

Set both elements equal to 0 for automatic axis ranges:

```
!X.Range = 0
```

### Example

To force the  $x$ -axis to run from 5.5 to 8.3:

```
!X.Range = [5.5, 8.3]
```

```
PLOT, X, Y
```

Alternatively, by using keywords:

```
PLOT, X, Y, XRange = [5.5, 8.3]
```

Note that even though the range was set to (5.5, 8.3), the resulting plot has a range of (5.5, 8.5), because of the axis rounding. To inhibit rounding, set !X.Style to 1.

## ![XYZ].Region

**Corresponding Plot Keyword:** None.

Contains the normalized coordinates of the region. This field is similar to !X.Window in that it is set by the graphics procedures and is a two-element floating-point array.

## ![XYZ].S

The scaling factors for converting between data coordinates and normalized coordinates (a two-element array). The formula for conversion from data ( $X_d$ ) to normalized ( $X_n$ ) coordinates is:

$$X_n = S_i X_d + S_0$$

If logarithmic scaling is in effect, substitute  $\log_{10} X_d$  for  $X_d$ .

## ![XYZ].Style

**Corresponding Plot Keyword:** `[XYZ]Style`

The style of the axis encoded as bits in a longword. The axis style may be set to exact, extended, none, or no box using this field. See the following table for details.

Settings for Axis Style

Bit	Value	Function
0	1	Exact. By default the end points of the axis are rounded in order to obtain even tick increments. Setting this bit inhibits rounding, making the axis fit the data range exactly.
1	2	Extend. If this bit is set, the axes are extended by 5% in each direction, leaving a border around the data.
2	4	None. If this bit is set, the axis and its annotation is not drawn.
3	8	No box. Normally, PLOT and CONTOUR draw a box style axis with the data window surrounded by axes. Setting this bit inhibits drawing the top or right axis.
4	16	Inhibits setting the y-axis minimum value to zero, when the data are all positive and non-zero. The keyword <i>YNozero</i> sets this bit temporarily.

### Example

To set the *x*-axis style to exact, use:

```
!X.Style = 1
```

or by using a keyword parameter:

```
PLOT, X, Y, XStyle = 1
```

## ![XYZ].Thick

The thickness of the axis line and tick marks. 1.0 is normal.

## ![XYZ].Tickformat

**Corresponding Plot Keyword:** [\[XYZ\]Tickformat](#)

Changes the default format for axis tick labels using FORTRAN-style format specifiers. For example, the following statement changes the default tick label format to floating-point numbers carried to two decimal places.

```
!X.Tickformat = '(F5.2)'
```

The specified width of the format is at least five characters; however, this width expands automatically to accommodate larger values.

For more information on format specifiers, see [.](#) See also [.](#)

Note that only the I (integer), F (floating-point), and E (scientific notation) format specifiers can be used with !X.Tickformat. Also, you cannot place a quoted string inside a tick format. For example, (" $<$ ", F5.2, " $>$ ") is an invalid !X.Tickformat specification.

## ![XYZ].Ticklen

**Corresponding Plot Keyword:** [\[XYZ\]Ticklen](#)

The length of tick marks, expressed as a fraction of the plot size. The default value is 0.02. A negative value makes tick marks that extend outside the window, rather than inward.

## ![XYZ].Tickname

**Corresponding Plot Keyword:** [\[XYZ\]Tickname](#)

The annotation for each tick. A string array of up to 30 elements. Setting elements of this array allows direct specification of the tick label. If this element contains a null string, the default value, PV-WAVE annotates the tick with its numeric value. Setting the element to a one-blank string suppresses the tick annotation.

### Example

To produce a plot with an abscissa labeled with the days of the week:

```
!X.Tickname = ['SUN', 'MON', 'TUE', 'WED', 'THU', 'FRI', 'SAT']  
            ; Set up x-axis tick labels.  
  
!X.Ticks = 6  
            ; Use six tick intervals, requiring seven tick labels.  
  
PLOT, Y  
            ; Plot the data, this assumes that Y contains 7 elements.
```

The same plot can be produced, using keyword parameters, with:

```
PLOT, Y, XTickname = ['SUN', 'MON', 'TUE', $  
    'WED', 'THU', 'FRI', 'SAT'], XTicks = 6  
    ; Set fields, as above, only temporarily.
```

See [for information on creating axes with date and time data.](#)

## **![XYZ].Ticks**

**Corresponding Plot Keyword:** [\[XYZ\]Ticks](#)

The number of major tick intervals to draw for the axis. If ![XYZ].Ticks is set to 0, the default, PV-WAVE will select from three to six tick intervals. Setting this field to  $n$ , where  $n > 0$ , produces exactly  $n$  tick intervals, and  $n + 1$  tick marks.

## **![XYZ].Tickv**

**Corresponding Plot Keyword:** [\[XYZ\]Tickv](#)

The data values for each tick mark, an array of up to 30 elements. You can directly specify the location of each tick by setting ![XYZ].Ticks to the number of tick marks (the number of intervals +1) and storing the data values of the tick marks in ![XYZ].Tickv. If, by default, ![XYZ].Tickv(0) is equal to ![XYZ].Tickv(1), PV-WAVE will automatically determine the value of the tick mark.

## **![XYZ].Title**

**Corresponding Plot Keyword:** [\[XYZ\]Title](#)

A string containing the axis title.

## **![XYZ].Type**

**Corresponding Plot Keyword:** [\[XYZ\]Type](#)

Specifies the type of axis, 0 for linear, 1 for logarithmic, 2 for Date/Time.

## **![XYZ].Window**

Contains the normalized coordinates of the axis end points, the plot window. This field is set by PLOT, CONTOUR, and SURFACE. Changing its value has no effect. It is a two-element floating-point array.



# Software Character Sets

---

## Software Character Sets

Font 3, Simplex Roman

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
_	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	!	"
#	\$	%	&	'	(	)	*	+	,
-	.	/	0	1	2	3	4	5	6
7	8	9	:	;	<	=	>	?	@

### Font 4, Simplex Greek

A	B	Γ	Δ	E	Z	H	Θ	I	J	K									
K	Λ	L	M	N	Ξ	O	Π	Q	Ρ	Σ	T	Υ							
U	Φ	V	X	Ψ	X	Ω	Y	∞	Z	[	[	\	]	]	^				
_	—	.	'	a	α	b	β	c	γ	d	δ	e	ε	f	ξ	g	η	h	θ
i	ι	j	κ	k	λ	l	μ	m	ν	n	ξ	o	ο	p	π	q	ρ	r	σ
s	τ	t	υ	u	φ	v	χ	w	ψ	x	ω	y	∞	z	ι	!	!	"	"
#	#	\$	\$	%	%	&	&	'	'	(	(	)	)	*	*	+	+	,	,
-	—	.	·	/	/	0	0	1	1	2	2	3	3	4	4	5	5	6	6
7	7	8	8	9	9	:	:	;	;	<	<	=	=	>	>	?	?	@	@

### Font 5, Duplex Roman

A	B	C	D	E	F	G	H	I	J	J									
K	L	M	N	O	P	Q	R	S	T	T									
U	V	W	X	Y	Z	[	[	\	]	]	^	^							
_	—	.	'	a	a	b	b	c	c	d	d	e	e	f	f	g	g	h	h
i	i	j	i	k	k	l	l	m	m	n	n	o	o	p	p	q	q	r	r
s	s	t	t	u	u	v	v	w	w	x	x	y	y	z	z	!	!	"	"
#	#	\$	\$	%	%	&	&	'	'	(	(	)	)	*	*	+	+	,	,
-	—	.	·	/	/	0	0	1	1	2	2	3	3	4	4	5	5	6	6
7	7	8	8	9	9	:	:	;	;	<	<	=	=	>	>	?	?	@	@

## Font 6, Complex Roman

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
_	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	!	"
#	\$	%	&	'	(	)	*	+	,
-	.	/	0	1	2	3	4	5	6
7	8	9	:	;	<	=	>	?	@

## Font 7, Complex Greek

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
_	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	!	"
#	\$	%	&	'	(	)	*	+	,
-	.	/	0	1	2	3	4	5	6
7	8	9	:	;	<	=	>	?	@

## Font 8, Complex Italic

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
_	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	!	"
#	\$	%	&	'	(	)	*	+	,
-	.	/	0	1	2	3	4	5	6
7	8	9	:	;	<	=	>	?	@

## Font 9, Math and Special

A	~	B	□	C	✓	D	∂	E	∃	F	ℱ	G	∇	H	∮	I	∫	J	ℳ
K		L		M	≡	N	≡	O	†	P	∅	Q	⌈	R	√	S	⌋	T	∴
U	♠	V	◇	W	♣	X	×	Y		Z		[	\	]					^
_	---	'		a	∠	b	≅	c	α	d	∂	e	€	f	♀	g		h	∩
i	∫	j	j	k	≅	l	≅	m	♂	n	⊙	o	‡	p	ℙ	q	♀	r	✓
s	∫	t	∅	u	♥	v	♣	w	♣	x	⊥	y		z		!		"	
#		\$	∞	%	°	&	§	'		(	)	*	*	+	±	,	,		
-	∓	.	°	/	÷	0	∩	1	∪	2	∩	3	∩	4	←	5	↓	6	→
7	↑	8		9	:	≡	;	<	{	=	≠	>	}	?	∞	@	ℵ		

Font 10, Special

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
_	.	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	!	"
#	\$	%	&	'	(	)	*	+	,
-	.	/	0	1	2	3	4	5	6
7	8	9	:	;	<	=	>	?	@

Font 11, Gothic English

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
_	~	'	a	b	c	d	e	f	g
h	i	j	k	l	m	n	o	p	q
r	s	t	u	v	w	x	y	z	!
"	#	\$	%	&	'	(	)	*	+
,	-	.	/	0	1	2	3	4	5
6	7	8	9	:	;	<	=	>	?
@									

## Font 12, Simplex Script

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
°	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	!	"
#	\$	%	&	'	(	)	*	+	,
-	.	/	0	1	2	3	4	5	6
7	8	9	:	;	<	=	>	? ?	@

## Font 13, Complex Script

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	[	\	]	^
°	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r
s	t	u	v	w	x	y	z	!	"
#	\$	%	&	'	(	)	*	+	,
-	.	/	0	1	2	3	4	5	6
7	8	9	:	;	<	=	>	? ?	@

## Font 14, Gothic Italian

A	Ħ	B	Ĭ	C	Ī	D	Ī	E	Ī	F	Ī	G	Ī	H	Ī	I	Ī	J	Ī
K	Ī	L	Ī	M	Ī	N	Ī	O	Ī	P	Ī	Q	Ī	R	Ī	S	Ī	T	Ī
U	Ī	V	Ī	W	Ī	X	Ī	Y	Ī	Z	Ī	[	{	\	§	]	}	^	Ī
-	~	'	a	b	c	d	e	f	g	h									
i	j	k	l	m	n	o	p	q	r										
s	t	u	v	w	x	y	z	!	'										
#	#	\$	%	&	'	(	)	*	+	,									
-	.	/	/	0	1	2	3	4	5	6									
7	8	9	:	;	<	=	>	?	@										

## Font 15, Gothic German

A	Ŧ	B	Ĭ	C	Ī	D	Ī	E	Ī	F	Ī	G	Ī	H	Ī	I	Ī	J	Ī
K	Ī	L	Ī	M	Ī	N	Ī	O	Ī	P	Ī	Q	Ī	R	Ī	S	Ī	T	Ī
U	Ī	V	Ī	W	Ī	X	Ī	Y	Ī	Z	Ī	[	{	\	Ī	]	}	^	Ī
-	~	'	a	b	c	d	e	f	g	h									
i	j	k	l	m	n	o	p	q	r										
s	t	u	v	w	x	y	z	!	'										
#	#	\$	%	&	'	(	)	*	+	,									
-	.	/	/	0	1	2	3	4	5	6									
7	8	9	:	;	<	=	>	?	@	Ŧ									

### Font 16, Cyrillic

A	А	B	Б	C	В	D	Г	E	Д	F	Е	G	Ж	H	З	I	И	J	Й
K	К	L	Л	M	М	N	Н	O	О	P	П	Q	Р	R	С	S	Т	T	У
U	Ф	V	Х	W	Ц	X	Ч	Y	Ш	Z	Щ	[	Ы	\	Э	]	Ь	^	Ю
-	Я	.	'	a	а	b	б	c	в	d	д	e	е	f	ж	g	з	h	з
i	и	j	й	k	к	l	л	m	м	n	н	o	о	p	п	q	р	r	с
s	т	t	у	u	ф	v	х	w	ц	x	ч	y	ш	z	щ	!	"	”	Ю
#	Ъ	\$	\$	%	Э	&	&	'	'	(	(	)	)	*	*	+	+	,	,
-	-	.	.	/	/	0	0	1	1	2	2	3	3	4	4	5	5	6	6
7	7	8	8	9	9	:	:	;	;	<	<	=	=	>	>	?	?	@	Ъ

### Font 17, Triplex Roman

A	A	B	B	C	C	D	D	E	E	F	F	G	G	H	H	I	I	J	J
K	K	L	L	M	M	N	N	O	O	P	P	Q	Q	R	R	S	S	T	T
U	U	V	V	W	W	X	X	Y	Y	Z	Z	[	{	\	\	]	}	^	^
-	°	'	'	a	a	b	b	c	c	d	d	e	e	f	f	g	g	h	h
i	i	j	j	k	k	l	l	m	m	n	n	o	o	p	p	q	q	r	r
s	s	t	t	u	u	v	v	w	w	x	x	y	y	z	z	!	!	"	"
#	#	\$	\$	%	%	&	&	'	'	(	(	)	)	*	*	+	+	,	,
-	-	.	.	/	/	0	0	1	1	2	2	3	3	4	4	5	5	6	6
7	7	8	8	9	9	:	:	;	;	<	<	=	=	>	>	?	?	@	@

## Font 18, Triplex Italic

A	B	C	D	E	F	G	H	I	J	
K	L	M	N	O	P	Q	R	S	T	
U	V	W	X	Y	Z	[	\	}	^	
_	°	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r	
s	t	u	v	w	x	y	z	!	"	
#	\$	%	&	'	(	)	*	+	,	
-	.	/	0	1	2	3	4	5	6	
7	8	9	:	;	<	=	>	?	@	

## Font 20, Miscellaneous

A	B	C	D	E	F	G	H	I	J	
K	L	M	N	O	P	Q	R	S	T	
U	V	W	X	Y	Z	[	\	}	^	
_	°	'	a	b	c	d	e	f	g	h
i	j	k	l	m	n	o	p	q	r	
s	t	u	v	w	x	y	z	!	"	
#	\$	%	&	'	(	)	*	+	,	
-	.	/	0	1	2	3	4	5	6	
7	8	9	:	;	<	=	>	?	@	



## Special Characters

This chapter describes characters with special interpretation and their function in PV-WAVE.

**ampersand (&)** — The ampersand separates multiple statements on one line. Statements may be combined until the maximum line length of 511 characters is reached. For example, the following line contains two statements:

```
I = 1 & PRINT, 'VALUE: ', I
```

**apostrophe (')** — Delimits string literals and indicates part of an octal or hexadecimal constant.

**asterisk (\*)** — In addition to denoting multiplication, designates an ending subscript range equal to the size of the dimension. For example,  $A(3 : *)$  represents all elements of the vector  $A$  except the first three elements.

**“at” sign (@)** — When the “at” sign is the very first character in a PV-WAVE command line, it causes the compiler to substitute the contents of the command file whose name appears after @. In addition to searching the current directory for the file, PV-WAVE searches a list of locations where procedures are kept.

**colon (:)** — Ends label identifiers. Labels may only be referenced by GOTO and ON\_ERROR statements. The following line contains a statement with the label LOOP1:

```
LOOP1: x = 2.5
```

In addition, the colon is used in CASE statements.

The colon also separates the starting and ending subscripts in subscript range specifiers. For example  $A(3 : 6)$  designates the fourth, fifth, sixth, and seventh elements of the variable  $A$ .

**dollar sign (\$)** — At the end of a line indicates that the current statement is continued on the following line. The dollar sign character may appear anywhere a space is legal except within a string constant (where it is interpreted literally). Any number of continuation lines are allowed.

**exclamation point (!)** — Begins the names of system-defined variables. System variables are predefined variables of a fixed type. Their purpose is to override defaults for system procedures, to return status information, and to control the action of PV-WAVE.

**period or decimal point (.)** — Indicates in a numeric constant that the number is of floating-point or double-precision type. Example: 1.0 is a floating-point number.

Also, in response to the WAVE> prompt, the period, if it is the first character on the line, begins an executive command. For example:

```
WAVE> .RUN myfile
```

causes PV-WAVE to compile the file `myfile.pro`. If `myfile.pro` contains a main program, the program will also be executed.

However, if the period is not the first character on the line as in the following example,

```
WAVE>         .RUN myfile
```

you receive a syntax error.

Also, the period precedes the name of a tag when referring to a field within a structure. For example, a reference to a tag called NAME in a structure stored in the variable A is: `A.NAME`.

**quotation mark (")** — The quotation mark precedes octal numbers which are always integers and delimits string constants. Examples: `"100B` is a byte constant equal to `6410`, `"Don't drink the water."` is a string constant.

**semicolon (;)** — Begins a comment field of a statement. All text on a line following a semicolon is ignored by PV-WAVE. A line may consist of just a comment or may contain both a valid statement followed by a comment.

When the \$ character is entered as the first character after the PV-WAVE prompt, the rest of the line is sent to the operating system as a command. To send an operating system command from within a procedure, use the SPAWN command.

## *Executive Commands*

PV-WAVE executive commands compile programs, continue stopped programs, and start previously compiled programs. All executive commands begin with a period. Under UNIX, file names are case sensitive, while under OpenVMS and Windows, either case may be used. Executive commands can be executed from files or from the WAVE> prompt.

---

### *Using Executive Commands*

Executive commands are summarized in the following table:

#### Executive Commands

<b>Command</b>	<b>Action</b>
.CON	Continues execution of a stopped program.
.GO	Executes previously compiled main program from the beginning of the program.
.LOCALS	Resizes the data area in terms of local variables and common block symbols.
..LOCALS	Allocates space for procedures executed from within procedures.
.RNEW	Compiles and possibly executes text from files or from the WAVE> prompt.
.RUN	Compiles and possibly executes text from files or from the WAVE> prompt.
.SIZE	Resizes the code area and the data area used to compile programs in terms of bytes.
.SKIP	Skips over the next statement and then single steps.
.STEP	Executes a single statement. This command may be abbreviated as .S.

---

## **.CON**

The `.CON` command continues execution of a program that has stopped because of an error, a `STOP` statement, or a keyboard interrupt. `PV-WAVE` saves the location of the beginning of the last statement executed before an error. If it is possible to correct the error condition in the interactive mode, the offending statement may be re-executed by typing `.CON`. After `STOP` statements, `.CON` continues execution at the next statement.

---

**NOTE** Execution of a program interrupted by typing `<Control>-C` may also be resumed at the point of interruption with the `.CON` command.

---

## **.GO**

The `.GO` command starts execution at the beginning of a previously compiled main program.

## **.LOCALS**

The syntax of the `.LOCALS` command is:

```
.LOCALS local_vars common_symbols
```

The `.LOCALS` command is similar to the `.SIZE` command, in that it resizes the data area (the data area is described in the previous section, *Using .SIZE*). The `.LOCALS` command, however, lets you specify the data area size in terms of local variables and common block symbols rather than in bytes. This command affects the size of the data area for the `$MAIN$`-level (commands entered from the `WAVE>` prompt), and the initial size of the data area for compiled procedures and functions.

The two parameters are positional, but not required. If you execute `.LOCALS` with no parameters, the data area is set back to its default value, which is 500 local variables. If you want to use 700 variables at the `$MAIN$` level, enter:

```
.LOCALS 700
```

`.LOCALS` clears and frees the current `$MAIN$` data area and code area. It then allocates a new code area of the same size as the previous one and a new data area of the specified size.

For compiled procedures and functions, the compiler initially allocates code and data areas of the same size as those that `$MAIN$` is currently using. If you get compiler error messages stating that the code and/or data area of a procedure or function is full, you must first make the `$MAIN$` code and/or data areas larger with the `.SIZE` or `.LOCALS` executive command. Then when you recompile the procedure or function, the compiler starts with the larger code and/or data areas.

## **..LOCALS**

The syntax of the `..LOCALS` compiler directive is:

```
..LOCALS local_vars common_symbols
```

This command is useful when you want to place the EXECUTE function inside a procedure or function. EXECUTE takes a string parameter containing a PV-WAVE command. This command argument is compiled and executed at runtime, allowing the possibility for command options to be specified by the user. Because the data area is compressed after compilation, there may not be enough room for additional local variables and common block symbols created by EXECUTE. The `..LOCALS` command provides a method of allocating extra space for these additional items.

The `..LOCALS` compiler directive is similar to the `.LOCALS` executive command, except:

- `..LOCALS` is only used inside procedures and functions.
- Its arguments specify the number of *additional* local variables and common block symbols that will be needed at “interpreter” time (when the already-compiled instructions are interpreted).
- It is used in conjunction with the EXECUTE function, which can create new local variables and common block symbols at runtime.

## **.RNEW**

The `.RNEW` command compiles and saves procedures and programs in the same manner as `.RUN`. However, all variables in the main program unit, including those in common blocks, are erased. The `-t`, `-l`, and `-c` switches have the same effect as with `.RUN`. See the examples below. Its syntax is:

```
.RNEW file1, ..., filen
```

See [Sample Usage of .RUN and .RNEW on page 1270](#).

## **.RUN**

The `.RUN` command compiles procedures, functions and main programs. The `.RUN` command also executes main programs. The command may be followed by a list of files to be compiled. Separate the filenames with blanks or commas:

```
.RUN file1, ..., filen
```

If no files are specified with the `.RUN` command, input is accepted from the keyboard at the `WAVE>` prompt until a complete program unit is entered. The values of all the variables are retained.

Files containing `PV-WAVE` procedures, programs, and functions are assumed to have the filename extension (suffix) `.pro`. If the filename is the same as the actual function or procedure name, the function or procedure is compiled and executed.

The command arguments `-t` for terminal listing, or `-l` for listing to a named file, may be used after the command name, and before the program file names, to produce a numbered program listing directed to the terminal or to a file. For instance, to see a listing on the screen as a result of compiling a procedure contained in a file named `analyze.pro`:

```
.RUN -t analyze
```

To compile the same procedure and save the listing in a file named: `analyze.lis`:

```
.RUN -l analyze.lis analyze
```

In listings produced by `PV-WAVE`, the line number of each statement is printed at the left margin. This number is the same as that printed in error statements, simplifying location of the statement causing the error.

Each level of block nesting is indented four spaces to the right of the preceding block level to improve the legibility of the program's structure.

---

**UNIX and OpenVMS USERS** Use the command argument `-c` after `.RUN` to compile a main program without executing it.

---

### ***Sample Usage of .RUN and .RNEW***

Some examples of the `.RUN` and `.RNEW` commands are:

```
.RUN
    ; Accept a program from the keyboard (WAVE> prompt). Retain the
    ; present variables.

.RUN myfile
    ; Compile the file myfile.pro. If myfile.pro is not found in the current
    ; directory, PV-WAVE looks for the file in the directory search path.

.RUN -t a, b, c
    ; Compiles the files a.pro, b.pro, and c.pro. Lists the programs on the
    ; terminal.

.RNEW -l myfile.lis myfile, yourfile
    ; Erases all variables. Compiles the files myfile.pro and yourfile.pro.
    ; Produces a listing of myfile in the file myfile.lis.
```

## **.SKIP**

The `.SKIP` command skips one or more statements and then single steps. This command is useful for continuing over a program statement which caused an error. If the optional argument *n* is present, it gives the number of statements to skip, otherwise, a single statement is skipped. The syntax is:

```
.SKIP [n]
```

For example, consider the following program segment:

```
... ..  
    OPENR, 1, 'missing'  
    READF, 1, xxx, ..., ...  
... ..
```

If the `OPENR` procedure fails because the specified file does not exist, program execution will halt with the `OPENR` procedure as the current procedure. Execution may not be resumed with the executive command `.CON` because it attempts to re-execute the offending `OPENR` procedure, causing the same error.

The remainder of the program can be executed by:

- Opening the correct file manually by typing in a valid `OPENR` procedure.
- Entering `.SKIP`, which skips over the incorrect `OPENR` procedure.
- Entering `.CON`, which resumes execution of the program at the `READF` procedure.

## **.SIZE**

The syntax of the `.SIZE` command is:

```
.SIZE code_size data_size
```

The `.SIZE` command resizes the *code area* and *data area*. These memory areas are used when `PV-WAVE` programs are compiled. The code area holds internal instruction codes that the compiler generates. The data area, also used by the compiler, contains variable name, common block, and keyword information for each compiled function, procedure, and main program.

After successful compilation, a new memory area of the required size is allocated to hold the newly compiled program unit.

By default, the size of the code area is about 800 bytes, and it grows dynamically as needed to accommodate the activity of your session. The initial size of the data area is 8,000 bytes (enough space to hold 500 local variables).

---

**CAUTION** Resizing the code and data areas erases the currently compiled main program and all main program variables.

---

For example, to extend the code and data areas to 40,000 and 10,000 bytes respectively:

```
.SIZE 40000 10000
```

The upper limit for both *code\_size* and *data\_size* is over 2 billion bytes.

## **.STEP**

The `.STEP` command executes one or more statements in the current program starting at the current position, stops, and returns control to the interactive mode. This command is useful in debugging programs. If the optional argument *n* is present, it gives the number of statements to execute, otherwise, a single statement is executed. The syntax of the `.STEP` command is:

```
.STEP [n]
```

or

```
.S [n]
```

## ***The PV-WAVE HDF Interface***

This appendix discusses:

- What is the PV-WAVE HDF interface?
- Where to find example programs and NCSA documentation in the PV-WAVE distribution.
- Using PV-WAVE HDF functions.
- The PV-WAVE HDF Base functions.

---

**NOTE** The PV-WAVE HDF interface is designed for the experienced HDF, NetCDF, and PV-WAVE user.

---

---

### ***What is the PV-WAVE HDF Interface?***

PV-WAVE provides an extensive interface to the NCSA (National Center for Supercomputer Applications) HDF (Hierarchical Data Format) C library, which is in the public domain. PV-WAVE's interface to the HDF library is fully supported by Visual Numerics; however, Visual Numerics is not responsible for questions or problems specifically related to the HDF library. Please direct such problems or questions directly to NSCA.

The PV-WAVE HDF interface is divided into two sets of functions: *convenience routines* and *base functions*.

The convenience routines are a set of PV-WAVE routines developed for general situations where you need to access HDF functionality from PV-WAVE. The convenience routines are described in this reference. For a list of these routines, see [Chapter 1, \*Functional Summary of Routines\*](#).

The extensive set of base functions provides direct access within PV-WAVE to the HDF library, which is dynamically linked to PV-WAVE and invoked at runtime. The calling sequence for each base function is listed in this appendix.

---

## **Example Programs Are Available**

To help you get the most out of the PV-WAVE HDF base functions, a collection of example programs is available in:

**(UNIX)**        `$VNI_DIR/hdf-4_00/test`

**(OpenVMS)** `VNI_DIR: [HDF-4_00.TEST]`

**(Windows)** `$VNI_DIR\hdf-4_00\test`

To run these test programs, either add the test directory path to the PV-WAVE system variable !Path, or simply start your PV-WAVE session directly from the test directory.

In addition, the convenience routines themselves provide good examples of how the base functions can be used in other applications. The convenience routines are located in:

**(UNIX)**        `$VNI_DIR/hdf-4_00/lib`

**(OpenVMS)** `VNI_DIR: [HDF-4_00.LIB]`

**(Windows)** `$VNI_DIR\hdf-4_00\lib`

## **Printing NCSA Documentation**

An extensive collection of NCSA documentation on HDF and NetCDF is located in the subdirectories of:

**(UNIX)**        `$VNI_DIR/hdf-4_00/doc`

**(OpenVMS)** `VNI_DIR: [HDF-4_00.DOC]`

**(Windows)** `$VNI_DIR\hdf-4_00\doc`

These documents are stored as compressed PostScript files that you can print on any PostScript laser printer. Simply uncompress the files before printing them.

The NCSA documents that are included with PV-WAVE are:

- *HDF User's Guide*, Version 4.0r2, July 1996, University of Illinois at Urbana-Champaign.
- *HDF Reference Manual*, Version 4.0r2, July 1996, University of Illinois at Urbana-Champaign.
- *NetCDF User's Guide*, Version 2.3, April 1993, University Corporation for Atmospheric Research.

For more information on these documents, refer to the README files in:

(UNIX)        \$VNI\_DIR/hdf-4\_00/doc

(OpenVMS)    VNI\_DIR: [HDF-4\_00.DOC]

(Windows)    \$VNI\_DIR\hdf-4\_00\doc

## ***Other Sources of Information on HDF and NetCDF***

NCSA documentation is directly available from:

NCSA Documentation Orders

152 Computing Applications Building

605 East Springfield Avenue

Champaign, IL 61820

(217) 244-0072

or from FTP site `ftp.ncsa.uiuc.edu`

NetCDF documentation is available from the FTP site `unidata.ucar.edu` in the directory `pub/netcdf`.

---

## ***Using the PV-WAVE HDF Functions***

### ***Initializing the HDF Module***

Before you can use any of the PV-WAVE HDF base or convenience routines, you must initialize the HDF module. To do this, type:

```
WAVE> @hdf_startup
      % Compiled module: HDF_INIT.
      PV-WAVE:HDF 4.00 Module Initialized
```

After the module is initialized, you can use any of the PV-WAVE HDF routines.

---

**TIP** Include a call to HDF\_STARTUP in your PV-WAVE system startup file, or your personal PV-WAVE startup file.

---

## ***HDF\_STARTUP Initializes Common Block Variables***

HDF\_STARTUP calls a function HDF\_INIT which initializes the variables in the hdf\_common common block. These variables match the flags that are defined by the header files to the HDF and NetCDF C libraries. For example, the NetCDF flag NC\_WRITE is set to the value 1 in the file when the PV-WAVE HDF module is initialized.

These variables are used to define:

- Default palette and string sizes
- Flags for Raster 8 encoding schemes
- Flags for Raster 24 bit interlace schemes
- Flags for SDS number types
- Flags for HOPEN file opening status
- Tag values, as found in the header file

**(UNIX)**      \$VNI\_DIR/hdf-4\_00/src/hdf

**(OpenVMS)**  VNI\_DIR: [HDF-4\_00.SRC.HDF]

**(Windows)**  \$VNI\_DIR\hdf-4\_00\hdf

The *HDF Calling Interfaces* manual refers to these common variables in HDF calls. These variables are used frequently in many of the calls to PV-WAVE HDF convenience routines and base routines.

---

**NOTE** hdf\_common must be included in all PV-WAVE procedures and functions that use PV-WAVE HDF routines.

---

## ***Input Data Is Converted to Required Data Type***

Input data to the PV-WAVE HDF base functions is usually converted to the required data type. This alters the input parameters if they are not the correct data type.

An error message

XXX must be declared in the calling procedure.

indicates that the data type and space required for the supplied variable is used within, but not allocated by, the service HDF routine. By defining the PV-WAVE variable with a dummy value of the proper data type, the PV-WAVE HDF wrapper will automatically convert the fetched value to the desired data type.

## ***Using the Usage and Help Keywords***

If you know the name of the routine you want to call, but do not know the calling sequence, use the *Usage* or *Help* keywords. For example, the following command displays the parameters for the NCCREATE function:

```
tmp = NCCREATE (/Usage)
      % NCCREATE: usage: status = NCCREATE (path, cmode, Help=help,
      Usage=usage)
```

These keywords are not part of HDF or NetCDF, but are provided for convenience to PV-WAVE HDF users.

## ***Ensure Correct Data Types with SDS GET Routines***

When using an SDS “GET” routine, be sure that the data types with which you fetch the annotated data are the same as your main dataset. Otherwise, you will get unexpected results.

## ***Use FORTRAN Array Alignment***

Arrays in PV-WAVE are aligned similar to FORTRAN (i.e., column-major order or first subscript varying fastest), therefore, datasets and images should be aligned as discussed in the HDF documentation for FORTRAN. For data read or written by C, this will result in the data being transposed from the PV-WAVE representation. For example, a 24-bit raster image array in PV-WAVE is defined as “3 X width X height” for pixel interlacing, however, in C, the same image array would be defined as “height X width X 3”.

## ***Annotation Routines May Require Further Processing***

Annotation routines which read or write descriptions (i.e., DFANADDFDS, DFANGETDESC, HDFPUTANN, etc.) operate on byte arrays rather than strings and may require further processing.

## ***Slab Routines Replace Slice Routines***

NCSA has replaced the slice routines for reading and writing hyperslabs of scientific data with equivalent slab routines. The slab routines are supported in PV-WAVE HDF; however, the slice routines are not.

## ***Base Functions Assume Valid Input***

The PV-WAVE HDF convenience routines are written to check input data for errors before continuing execution. The PV-WAVE HDF base routines assume that the input data is valid, and performs minimal tests on the input data. Because of this, the possibility for failure is present. If you are using PV-WAVE HDF base routines and are experiencing core dumps, double check your input data for correct size and data type.

---

## ***PV-WAVE HDF Base Function Interface***

The PV-WAVE HDF base functions let you access all of the functions described in the NCSA *HDF Reference Manual* from within PV-WAVE.

These functions are located in the library:

**(UNIX)**        \$VNI\_DIR/hdf-4\_00/lib

**(OpenVMS)**    VNI\_DIR: [HDF-4\_00.LIB]

**(Windows)**    \$VNI\_DIR\hdf-4\_00\lib

For detailed information on these routines, refer to the *HDF Reference Manual*.

## ***24-bit Raster Image Set: The DF24 Interface***

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = DF24ADDIMAGE (*filename*, *image*, *xdim*, *ydim*, *Help=help*,  
*Usage=usage*)

*status* = DF24GETDIMS (*filename*, *pxdim*, *pydim*, *pil*, *Help=help*, *Usage=usage*)

*status* = DF24GETIMAGE (*filename*, *image*, *xdim*, *ydim*, *Help=help*,  
*Usage=usage*)

*ref* = DF24LASTREF (*Help=help*, *Usage=usage*)

*nimages* = DF24NIMAGES (*filename*, *Help=help*, *Usage=usage*)

*status* = DF24PUTIMAGE (*filename*, *image*, *xdim*, *ydim*, *Help=help*,  
*Usage=usage*)

*status* = DF24READREF (*filename*, *ref*, *Help=help*, *Usage=usage*)

*status* = DF24REQIL (*il*, *Help=help*, *Usage=usage*)

*status* = DF24RESTART (*Help=help*, *Usage=usage*)

*status* = DF24SETCOMPRESS (*type*, *cinfo*, *Help=help*, *Usage=usage*)

*status* = DF24SETDIMS (*xdim*, *ydim*, *Help=help*, *Usage=usage*)

*status* = DF24SETIL (*il*, *Help=help*, *Usage=usage*)

## **Annotations: The DFAN Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = DFANADDFDS (*file\_id*, *desc*, *desclen*, *Help=help*, *Usage=usage*)

*status* = DFANADDFID (*file\_id*, *id*, *Help=help*, *Usage=usage*)

*status* = DFANGETDESC (*filename*, *tag*, *ref*, *desc*, *maxlen*, *Help=help*,  
*Usage=usage*)

*desclen* = DFANGETDESCLEN (*filename*, *tag*, *ref*, *Help=help*, *Usage=usage*)

*desclen* = DFANGETFDS (*file\_id*, *desc*, *maxlen*, *isfirst*, *Help=help*,  
*Usage=usage*)

*fidlen* = DFANGETFDSLEN (*file\_id*, *isfirst*, *Help=help*, *Usage=usage*)

*fidlen* = DFANGETFID (*file\_id*, *id*, *maxlen*, *isfirst*, *Help=help*, *Usage=usage*)

*fidlen* = DFANGETFIDLEN (*file\_id*, *isfirst*, *Help=help*, *Usage=usage*)

*status* = DFANGETLABEL (*filename*, *tag*, *ref*, *label*, *maxlen*, *Help=help*,  
*Usage=usage*)

*lablen* = DFANGETLABLEN (*filename*, *tag*, *ref*, *Help=help*, *Usage=usage*)

*numrefs* = DFANLABLIST (*filename*, *tag*, *reflist*, *labellist*, *listsizes*, *maxlen*,  
*startpos*, *Help=help*, *Usage=usage*)

*ref* = DFANLASTREF (*Help=help*, *Usage=usage*)

*status* = DFANPUTDESC (*filename, tag, ref, desc, desclen, Help=help, Usage=usage*)

*status* = DFANPUTLABEL (*filename, tag, ref, label, Help=help, Usage=usage*)

## **Palettes: The DFP Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = DFPADDPAL (*filename, palette, Help=help, Usage=usage*)

*status* = DFPGETPAL (*filename, palette, Help=help, Usage=usage*)

*ref* = DFPLASTREF (*Help=help, Usage=usage*)

*npals* = DFPNPALS (*filename, Help=help, Usage=usage*)

*status* = DFPPUTPAL (*filename, palette, overwrite, filemode, Help=help, Usage=usage*)

*status* = DFPREADREF (*filename, ref, Help=help, Usage=usage*)

*status* = DFPRESTART (*Help=help, Usage=usage*)

*status* = DFPWRITEREF (*filename, ref, Help=help, Usage=usage*)

## **8-bit Raster Image Sets: The DFR8 Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = DFR8ADDIMAGE (*filename, image, xdim, ydim, compress, Help=help, Usage=usage*)

*status* = DFR8GETDIMS (*filename, pxdim, pydim, pispal, Help=help, Usage=usage*)

*status* = DFR8GETIMAGE (*filename, image, xdim, ydim, pal, Help=help, Usage=usage*)

*ref* = DFR8LASTREF (*Help=help, Usage=usage*)

*nimages* = DFR8NIMAGES (*filename, Help=help, Usage=usage*)

*status* = DFR8PUTIMAGE (*filename, image, xdim, ydim, compress, Help=help, Usage=usage*)

*status* = DFR8READREF (*filename, ref, Help=help, Usage=usage*)

*status* = DFR8RESTART (*Help=help, Usage=usage*)

*status* = DFR8SETCOMPRESS (*type, cinfo, Help=help, Usage=usage*)

*status* = DFR8SETPALETTE (*pal, Help=help, Usage=usage*)

*status* = DFR8WRITEREF (*filename, ref, Help=help, Usage=usage*)

## **Scientific Data Sets: Single File DFSD Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = DFSDADDDATA (*filename, rank, dimsizes, data, Help=help, Usage=usage*)

*status* = DFSDCLEAR (*Help=help, Usage=usage*)

*status* = DFSDENDSLAB (*Help=help, Usage=usage*)

*status* = DFSDGETCAL (*pcal, pcal\_err, pioff, pioff\_err, cal\_nt, Help=help, Usage=usage*)

*status* = DFSDGETDATA (*filename, rank, dimsizes, data, Help=help, Usage=usage*)

*status* = DFSDGETDATALEN (*llabel, lunit, lformat, lcoordsys, Help=help, Usage=usage*)

*status* = DFSDGETDATASTRS (*label, unit, format, coordsys, Help=help, Usage=usage*)

*status* = DFSDGETDIMLEN (*dim, llabel, lunit, lformat, Help=help, Usage=usage*)

*status* = DFSDGETDIMS (*filename, prank, sizes, maxrank, Help=help, Usage=usage*)

*status* = DFSDGETDIMSCALE (*dim, maxsize, scale, Help=help, Usage=usage*)

*status* = DFSDGETDIMSTRS (*dim, label, unit, format, Help=help, Usage=usage*)

*status* = DFSDGETFILLVALUE (*fill\_value, Help=help, Usage=usage*)

*status* = DFSDGETNT (*pnumbertype, Help=help, Usage=usage*)

*status* = DFSDGETRANGE (*pmax, pmin, Help=help, Usage=usage*)

*ref* = DFSDLASTREF (*Help=help, Usage=usage*)  
*ndatasets* = DFSDNDATASETS (*filename, Help=help, Usage=usage*)  
*status* = DFSDPRE32SDG (*filename, ref, ispre32, Help=help, Usage=usage*)  
*status* = DFSDPUTDATA (*filename, rank, dimsizes, data, Help=help, Usage=usage*)  
*status* = DFSDREADREF (*filename, ref, Help=help, Usage=usage*)  
*status* = DFSDREADSLAB (*filename, start, slab\_size, stride, buffer, buffer\_size, Help=help, Usage=usage*)  
*status* = DFSDRESTART (*Help=help, Usage=usage*)  
*status* = DFSDSETCAL (*cal, cal\_err, ioff, ioff\_err, cal\_nt, Help=help, Usage=usage*)  
*status* = DFSDSETDATASTRS (*label, unit, format, coordsys, Help=help, Usage=usage*)  
*status* = DFSDSETDIMS (*rank, dimsizes, Help=help, Usage=usage*)  
*status* = DFSDSETDIMSCALE (*dim, dimsize, scale, Help=help, Usage=usage*)  
*status* = DFSDSETDIMSTRS (*dim, label, unit, format, Help=help, Usage=usage*)  
*status* = DFSDSETFILLVALUE (*fill\_value, Help=help, Usage=usage*)  
*status* = DFSDSETLENGTHS (*maxlen\_label, maxlen\_unit, maxlen\_format, maxlen\_coordsys, Help=help, Usage=usage*)  
*status* = DFSDSETNT (*numbertype, Help=help, Usage=usage*)  
*status* = DFSDSETRANGE (*maxi, mini, Help=help, Usage=usage*)  
*status* = DFSDSTARTSLAB (*filename, Help=help, Usage=usage*)  
*status* = DFSDWRITEREF (*filename, ref, Help=help, Usage=usage*)  
*status* = DFSDWRITESLAB (*start, stride, count, data, Help=help, Usage=usage*)

## **The H Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = HCLOSE (*file\_id*, *Help=help*, *Usage=usage*)

*status* = HGETFILEVERSION (*file\_id*, *majorv*, *minorv*, *release*, *string*,  
*Help=help*, *Usage=usage*)

*status* = HGETLIBVERSION (*majorv*, *minorv*, *release*, *string*, *Help=help*,  
*Usage=usage*)

*status* = HISHDF (*filename*, *Help=help*, *Usage=usage*)

*fileid* = HOPEN (*path*, *access*, *nnds*, *Help=help*, *Usage=usage*)

*status* = HXSETCREATEDIR (*dirname*, *Help=help*, *Usage=usage*)

*status* = HXSETDIR (*dirname*, *Help=help*, *Usage=usage*)

## **Scientific Data Sets: The NetCDF Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = NCABORT (*cdfid*, *Help=help*, *Usage=usage*)

*status* = NCATTCOPY (*incdf*, *invar*, *name*, *outcdf*, *outvar*, *Help=help*,  
*Usage=usage*)

*status* = NCATTDEL (*cdfid*, *varid*, *name*, *Help=help*, *Usage=usage*)

*status* = NCATTGET (*cdfid*, *varid*, *name*, *value*, *Help=help*, *Usage=usage*)

*status* = NCATTINQ (*cdfid*, *varid*, *name*, *datatype*, *len*, *Help=help*,  
*Usage=usage*)

*status* = NCATTNAME (*cdfid*, *varid*, *attnum*, *name*, *Help=help*, *Usage=usage*)

*status* = NCATTPUT (*cdfid*, *varid*, *name*, *datatype*, *len*, *value*, *Help=help*,  
*Usage=usage*)

*status* = NCATTRENAME (*cdfid*, *varid*, *name*, *newname*, *Help=help*,  
*Usage=usage*)

*status* = NCCLOSE (*cdfid*, *Help=help*, *Usage=usage*)

*status* = NCCREATE (*path*, *cmode*, *Help=help*, *Usage=usage*)

*status* = NCDIMDEF (*cdfid*, *name*, *length*, *Help=help*, *Usage=usage*)

*status* = NCDIMID (*cdfid*, *name*, *Help=help*, *Usage=usage*)

*status = NCDIMINQ (cdfid, dimid, name, length, Help=help, Usage=usage)*

*status = NCDIMRENAME (cdfid, dimid, name, Help=help, Usage=usage)*

*status = NCENDEF (cdfid, Help=help, Usage=usage)*

*status = NCINQUIRE (cdfid, ndims, nvars, natts, recdim, Help=help, Usage=usage)*

*status = NCOPEN (path, mode, Help=help, Usage=usage)*

*status = NCREDEF (cdfid, Help=help, Usage=usage)*

*status = NCSETFILL (cdfid, fillmode, Help=help, Usage=usage)*

*status = NCSYNC (cdfid, Help=help, Usage=usage)*

*status = NCTYPELEN (datatype, Help=help, Usage=usage)*

*status = NCVARDEF (cdfid, name, datatype, ndims, dim, Help=help, Usage=usage)*

*status = NCVARGET (cdfid, varid, start, count, value, Help=help, Usage=usage)*

*status = NCVARGET1 (cdfid, varid, coords, value, Help=help, Usage=usage)*

*status = NCVARGETG (cdfid, varid, start, count, stride, imap, values, Help=help, Usage=usage)*

*status = NCVARGETS (cdfid, varid, start, count, stride, values, Help=help, Usage=usage)*

*status = NCVARID (cdfid, name, Help=help, Usage=usage)*

*status = NCVARINQ (cdfid, varid, name, datatype, ndims, dim, natts, Help=help, Usage=usage)*

*status = NCVARPUT (cdfid, varid, start, count, value, Help=help, Usage=usage)*

*status = NCVARPUT1 (cdfid, varid, coords, value, Help=help, Usage=usage)*

*status = NCVARPUTG (cdfid, varid, start, count, stride, imap, values, Help=help, Usage=usage)*

*status = NCVARPUTS (cdfid, varid, start, count, stride, values, Help=help, Usage=usage)*

*status = NCVARRENAME (cdfid, varid, name, Help=help, Usage=usage)*

## Scientific Data Sets: The SD Interface

For complete descriptions of these functions, see the *HDF Reference Manual*.

*size* = DFKNTSIZE (*data\_type*, *Help=help*, *Usage=usage*)

*status* = SDATTRINFO (*id*, *index*, *name*, *nt*, *count*, *Help=help*, *Usage=usage*)

*status* = SDCREATE (*fid*, *name*, *nt*, *rank*, *dimsizes*, *Help=help*, *Usage=usage*)

*status* = SDDIMINFO (*id*, *name*, *l\_size*, *nt*, *nattr*, *Help=help*, *Usage=usage*)

*status* = SDEND (*id*, *Help=help*, *Usage=usage*)

*status* = SDENDACCESS (*id*, *Help=help*, *Usage=usage*)

*status* = SDFILEINFO (*fid*, *datasets*, *attrs*, *Help=help*, *Usage=usage*)

*status* = SDFINDATTR (*id*, *attrname*, *Help=help*, *Usage=usage*)

*status* = SDGETCAL (*sdsid*, *cal*, *cale*, *ioff*, *ioffe*, *nt*, *Help=help*, *Usage=usage*)

*status* = SDGETDATASTRS (*sdsid*, *l*, *u*, *f*, *c*, *len*, *Help=help*, *Usage=usage*)

*status* = SDGETDIMID (*sdsid*, *number*, *Help=help*, *Usage=usage*)

*status* = SDGETDIMSCALE (*id*, *data*, *Help=help*, *Usage=usage*)

*status* = SDGETDIMSTRS (*id*, *l*, *u*, *f*, *len*, *Help=help*, *Usage=usage*)

*status* = SDGETFILLVALUE (*sdsid*, *val*, *Help=help*, *Usage=usage*)

*status* = SDGETINFO (*sdsid*, *name*, *rank*, *dimsizes*, *nt*, *nattr*, *Help=help*,  
*Usage=usage*)

*status* = SDGETRANGE (*sdsid*, *pmax*, *pmin*, *Help=help*, *Usage=usage*)

*status* = SDIDTOREF (*id*, *Help=help*, *Usage=usage*)

*status* = SDISCOORDVAR (*id*, *Help=help*, *Usage=usage*)

*status* = SDNAMETOINDEX (*fid*, *name*, *Help=help*, *Usage=usage*)

*status* = SDREADATTR (*id*, *index*, *buf*, *Help=help*, *Usage=usage*)

*status* = SDREADDATA (*sdsid*, *start*, *stride*, *l\_end*, *data*, *Help=help*,  
*Usage=usage*)

*status* = SDSELECT (*fid*, *index*, *Help=help*, *Usage=usage*)

*status* = SDSETATTR (*id, name, nt, count, data, Help=help, Usage=usage*)  
*status* = SDSETCAL (*sdsid, cal, cale, ioff, ioffe, nt, Help=help, Usage=usage*)  
*status* = SDSETDATASTRS (*sdsid, l, u, f, c, Help=help, Usage=usage*)  
*status* = SDSETDIMNAME (*id, name, Help=help, Usage=usage*)  
*status* = SDSETDIMSCALE (*id, count, nt, data, Help=help, Usage=usage*)  
*status* = SDSETDIMSTRS (*id, l, u, f, Help=help, Usage=usage*)  
*status* = SDSETEXTERNALFILE (*id, filename, offset, Help=help, Usage=usage*)  
*status* = SDSETFILLVALUE (*sdsid, val, Help=help, Usage=usage*)  
*status* = SDSETRANGE (*sdsid, pmax, pmin, Help=help, Usage=usage*)  
*status* = SDSTART (*name, HDFmode, Help=help, Usage=usage*)  
*status* = SDWRITEDATA (*sdsid, start, stride, l\_end, data, Help=help, Usage=usage*)

## **Vgroups: The V interface**

For complete descriptions of these routines, see the *HDF Reference Manual*.

*status* = VADDTAGREF (*vg, tag, ref, Help=help, Usage=usage*)  
*status* = VATTACH (*f, vgid, accesstype, Help=help, Usage=usage*)  
VDETACH, *vg, Help=help, Usage=usage*  
VEND, *f, Help=help, Usage=usage*  
VGETCLASS, *vkey, vgclass, Help=help, Usage=usage*  
*status* = VGETID (*f, vgid, Help=help, Usage=usage*)  
VGETNAME, *vkey, vgname, Help=help, Usage=usage*  
*status* = VGETNEXT (*vg, id, Help=help, Usage=usage*)  
*status* = VGETTAGREF (*vg, which, tag, ref, Help=help, Usage=usage*)  
*status* = VGETTAGREFS (*vg, tagarray, refarray, n, Help=help, Usage=usage*)  
*status* = VINQTAGREF (*vg, tag, ref, Help=help, Usage=usage*)  
*status* = VINQUIRE (*vg, nentries, vgname, Help=help, Usage=usage*)

*status* = VINSERT (*vgroup\_id*, *v\_id*, *Help=help*, *Usage=usage*)  
*status* = VISVG (*vg*, *id*, *Help=help*, *Usage=usage*)  
*status* = VISVS (*vg*, *id*, *Help=help*, *Usage=usage*)  
*status* = VLONE (*f*, *idarray*, *asize*, *Help=help*, *Usage=usage*)  
*status* = VNTAGREFS (*vg*, *Help=help*, *Usage=usage*)  
*status* = VSETCLASS(*vkey*, *vgclass*, *Help=help*, *Usage=usage*)  
*status* = VSETNAME(*vkey*, *vgname*, *Help=help*, *Usage=usage*)  
VSTART, *f*, *Help=help*, *Usage=usage*

## **Vdata: The VS Interface**

For complete descriptions of these routines, see the *HDF Reference Manual*.

*status* = VSATTACH (*f*, *vsid*, *accessstype*, *Help=help*, *Usage=usage*)  
VSDETACH, *vs*, *Help=help*, *Usage=usage*  
*status* = VSELTS (*vs*, *Help=help*, *Usage=usage*)  
*status* = VSFDEFINE (*vs*, *field*, *localtype*, *order*, *Help=help*, *Usage=usage*)  
*status* = VSFEXIST (*vs*, *fields*, *Help=help*, *Usage=usage*)  
*status* = VSFIND (*f*, *vsname*, *Help=help*, *Usage=usage*)  
VSGETCLASS, *vs*, *vsclass*, *Help=help*, *Usage=usage*  
*status* = VSGETFIELDS (*vs*, *fields*, *Help=help*, *Usage=usage*)  
*status* = VSGETID (*f*, *vsid*, *Help=help*, *Usage=usage*)  
*status* = VSGETINTERLACE (*vs*, *Help=help*, *Usage=usage*)  
VSGETNAME, *vs*, *vsname*, *Help=help*, *Usage=usage*  
*status* = VSINQUIRE (*vs*, *nelt*, *interlace*, *fields*, *eltsize*, *vsname*, *Help=help*,  
*Usage=usage*)  
*status* = VSLONE (*f*, *idarray*, *asize*, *Help=help*, *Usage=usage*)  
*status* = VSREAD (*vs*, *buf*, *nelt*, *interlace*, *Help=help*, *Usage=usage*)  
*status* = VSSEEK (*vs*, *eltpos*, *Help=help*, *Usage=usage*)

VSSETCLASS, *vs, vsclass, Help=help, Usage=usage*  
*status = VSSETFIELDS (vs, fields, Help=help, Usage=usage)*  
*status = VSSETINTERLACE (vs, interlace, Help=help, Usage=usage)*  
VSSETNAME, *vs, vsname, Help=help, Usage=usage*  
*status = VSSIZEOF (vs, fields, Help=help, Usage=usage)*  
*status = VSWRITE (vs, buf, nelt, interlace, Help=help, Usage=usage)*

### **Vdata Fields: The VF Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status = VFFIELDESIZE (vkey, index, Help=help, Usage=usage)*  
*status = VFFIELDISIZE (vkey, index, Help=help, Usage=usage)*  
*status = VFFIELDNAME (vkey, index, Help=help, Usage=usage)*  
*status = VFFIELDORDER (vkey, index, Help=help, Usage=usage)*  
*status = VFFIELDTYPE (vkey, index, Help=help, Usage=usage)*  
*status = VFNFIELDS (vkey, Help=help, Usage=usage)*

### **Vdata Query: The VSQ Interface**

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status = VSQUERYCOUNT (vs, count, Help=help, Usage=usage)*  
*status = VSQUERYFIELDS (vs, flds, Help=help, Usage=usage)*  
*status = VSQUERYINTERLACE (vs, intr, Help=help, Usage=usage)*  
*status = VSQUERYNAME (vs, name, Help=help, Usage=usage)*  
*status = VSQUERYREF (vkey, Help=help, Usage=usage)*  
*status = VSQUERYTAG (vkey, Help=help, Usage=usage)*  
*status = VSQUERYVSIZE (vs, size, Help=help, Usage=usage)*

## ***High Level Vdata/Vgroups: The VH interface***

For complete descriptions of these functions, see the *HDF Reference Manual*.

*status* = VHMAKEGROUP (*f*, *tagarray*, *refarray*, *n*, *vgname*, *vgclass*, *Help=help*,  
*Usage=usage*)

*status* = VHSTOREDATA (*f*, *field*, *buf*, *n*, *datatype*, *vsname*, *vsclass*, *Help=help*,  
*Usage=usage*)

*status* = VHSTOREDATAM (*f*, *field*, *buf*, *n*, *datatype*, *vsname*, *vsclass*, *order*,  
*Help=help*, *Usage=usage*)



# *Output Devices and Window Systems*

This appendix discusses the following output devices and window systems that are supported by PV-WAVE.

Supported Output Devices and Window Systems

<b>Device Name</b>	<b>See Page</b>	<b>Description</b>
NULL	N/A	No graphic output
CGM	<a href="#">B-5</a>	Computer Graphics Metafile generator
HP	<a href="#">B-8</a>	Hewlett-Packard Graphics Language (HPGL) plotters
PCL	<a href="#">B-14</a>	Hewlett-Packard Printer Control Language (PCL)
PM	<a href="#">B-17</a>	Pixel Map output
PS	<a href="#">B-19</a>	PostScript devices
REGIS	<a href="#">B-34</a>	Regis graphics output devices
TEK	<a href="#">B-36</a>	Tektronix or compatible terminals
WIN32	<a href="#">B-39</a>	Microsoft Windows WIN32 driver
WMF	<a href="#">B-53</a>	Windows Metafile
X	<a href="#">B-58</a>	X Window System
Z	<a href="#">B-86</a>	Z-buffer device

---

## Window System Features

PV-WAVE utilizes the native window system by creating and using one or more largely independent windows, each of which can be used for the display of graphics and/or images. One color table is shared among these windows. Up to 32 separate windows can be active at any time. Windows are referenced using their index, which is an integer value between 0 and 31.

“Dithering” or halftoning techniques are used to display images with multiple shades of gray on monochrome displays — displays that can only display white or black. This topic is discussed in the *PV-WAVE User’s Guide*.

Graphic and image output is always directed to the current window. When a window system is selected as the current graphics device, the index number of the current window is found in the !D.Window system variable. This variable equals -1 if no window is open or selected. The WSET procedure is used to change the current window. WSHOW hides or displays a window. WDELETE deletes a window.

The WINDOW procedure creates a new window with a given index. If a window already exists with the same index, it is first deleted. The size, position, title, and number of colors may also be specified. If you access the display before creating the first window, PV-WAVE automatically creates a window with an index number of 0 and with the default attributes.

### How Is Backing Store Handled?

One of the features that distinguishes various window systems is how they handle the issue of backing store. When part of a window that was previously not visible is exposed, there are two basic approaches that a window system can take. Some keep track of the current contents of all windows and automatically repair any damage to their visible regions (retained windows). This saved information is known as the backing store. Others simply report the damage to the program that created the window and leave repairing the visible region to the program (non-retained windows). There are convincing arguments for and against both approaches. It is generally more convenient for PV-WAVE if the window system handles this problem automatically, but this often comes at a performance penalty. The actual cost of retained windows varies between systems and depends partially on the application.

The X Window and Microsoft Windows systems do not by default keep track of window contents. Therefore, when a window on the display is obscured by another window, the contents of its obscured portion is lost.

---

**UNIX and OpenVMS USERS** Re-exposing the window causes the X server to fill the missing data with the default background color for that window, and request the application to redraw the missing data. Applications can request a backing store for their windows, but servers are not required to provide it. Most current X servers do not provide backing store, and even those that do cannot necessarily provide it for all requesting windows. Therefore, requesting backing store from the server might help, but there is no certainty.

---

**Windows USERS** Re-exposing the window causes the Microsoft Windows to fill the missing data with the default background color for that window, and request the application to redraw the missing data.

---

The window system drivers allow you to control backing store using the *Retain* keyword to the `DEVICE` and `WINDOW` procedures. Using *Retain* with `DEVICE` allows you to set the default action for all windows, while using it with `WINDOW` lets you override the default for the new window. The possible values for this keyword are summarized in the following table, and are described in greater detail following the table.

<b>Value</b>	<b>Description</b>
0	No backing store.
1	(The Default) The server or window system is requested to retain the window.
2	PV-WAVE should provide a backing pixmap and handle the backing store directly (X Window System only).

---

**0** — A value of 0 specifies that no backing store is kept. In this case, exposing a previously obscured window leaves the missing portion of the window blank. Although this behavior can be inconvenient, it usually has the highest performance because there is no need to keep a copy of the window contents.

**1** — (The Default) Setting the *Retain* keyword to 1 causes PV-WAVE to request that a backing store be maintained. If the window system decides to accept the request, it automatically repairs the missing portions when the window is exposed. X Windows may or may not, depending on the capabilities of the server and the resources available to it.

**2** — Specifies that PV-WAVE should keep a backing store for the window itself, and repair any window damage when the window system requests it. This option

exists mainly for the X Window System. Under X, a pixmap (off-screen display memory) the same size as the window is created at the same time the window is created, and all graphics operations sent to the window are also sent to the pixmap. When the server requests PV-WAVE to repair freshly exposed windows, this pixmap is used to fill in the missing contents. Pixmap is a precious resource in the X server, so backing pixmaps should only be requested for windows with contents that must absolutely be preserved.

If the type of backing store to use is not explicitly specified using the *Retain* keyword, PV-WAVE assumes Option 1 and requests the window system to keep a backing store.

---

**UNIX USERS** Some IBM AIX systems do not have backing store enabled in the X server. For this reason, `DEVICE, Retain=2` is set by the Standard Library procedure `SETDEMO_RS6000.pro`. With backing store enabled, you can achieve increased performance by setting `DEVICE, Retain=1`. Your system administrator can help you determine whether or not backing store is enabled on your system. To enable backing store, kill your workstation's X server and restart with:

```
xinit -- -bs -fn fixed
```

---

---

## CGM Output

Computer Graphics Metafile, or CGM, is a standard for storing graphics output. To direct graphics output from PV-WAVE to a CGM file, enter the command:

```
SET_PLOT, 'CGM'
```

This causes PV-WAVE to use the CGM driver for producing graphical output, including line plots, contour plots, surface plots, and raster images. Once the CGM driver is enabled via `SET_PLOT`, the `DEVICE` procedure is used to control its actions, as described in [Controlling CGM Output with DEVICE Keywords](#) on page B-5. The default settings for the CGM driver are listed in the following table:

Setting	Default Value
Output file name	wave.cgm
File status	Closed
Metafile type	Clear Text
Number of colors in color table	254

Setting	Default Value
Color table offset	1
Clip to VDC range	ON
Horizontal offset	0 Coordinates
Vertical offset	0 Coordinates
Scale factor	None
Standard Cell Array	Standard

The CGM driver supports clear text and binary CGM output. To see the driver's current settings, enter:

```
INFO, /Device
```

---

**TIP** If you want to send a CGM file to a hardcopy printer, and you want to modify the color table, it is a good idea to modify the color table before sending CGM graphics output to the file. For more information about modifying color tables, see the *PV-WAVE User's Guide*.

---

**TIP** Run the [MSWORD\\_CGM\\_SETUP](#) procedure before importing a CGM file into Microsoft Word. For example:

```
SET_PLOT, 'CGM'
DEVICE, File='myplot.cgm'
MSWORD_CGM_SETUP
PLOT, dist(20)
DEVICE, /Close
```

---

## Controlling CGM Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the CGM driver:

**Clip** — Specifies that all coordinates will be reduced to fit within the VDC coordinate range. This keyword is primarily used with the *Xoffset* and *Yoffset* keywords to protect the metafile from containing coordinates which are outside the range of the VDC system.

**Close** — Closes the current CGM output file.

**Colors** — Specifies the number of colors in the color table. Be sure to specify the number of colors in the color table before any graphics output is written to the file.

**Ct\_Offset** — Specifies the location (or offset) of the first element in the color table. As with the *Colors* keyword, this value should be specified before any graphics output is written to the file.

**Filename** — Specifies the name of the CGM output file to be opened. If the keyword is not specified, the default name `wave.cgm` is used.

**Index\_From\_Zero** — If nonzero, maps the PV-WAVE colortable indices directly to the CGM color table indices. That is, the indices are mapped 0→0, 1→1, and so on. By default, the PV-WAVE to CGM color indices are mapped 0→1, 1→2, and so on. This keyword disables the *Color* and *Ct\_Offset* keywords.

**Metafile\_Type** — Specifies the metafile type. Options are `clear_text` and `binary`. The default value is `clear_text`. The `binary` type is machine-specific and is more difficult to transfer.

**Scale\_Factor** — Includes a floating-point value in the metafile for the Scale Mode Metric command. This allows for CGM interpreters to scale the graphics output appropriately for “exact sizing”. The metric scale-factor represents the distance (in millimeters) in the displayed picture — this corresponds to one VDC unit.

**Std\_Cell\_Array** — Specifies that images created with the binary output (CELL\_ARRAY commands) are compliant with the CGM standard.

*Std\_Cell\_Array* is enabled by default. Disabling this keyword may result in some non-standard cell array commands for odd-sized images. The non-standard cell array option is provided for downward compatibility.

**Xoffset** — Specifies the number of horizontal coordinate units to offset the graphics output. *Xoffset* is specified by a positive normalized coordinate in the range {0.0...1.0}.

**Yoffset** — Specifies the number of vertical coordinate units to offset the graphics output. *Yoffset* is specified by a positive normalized coordinate in the range {0.0...1.0}.

## Using the CGM Driver

The CGM output file is automatically opened when the CGM driver is selected and PV-WAVE commands are issued that result in graphics output; there is no explicit OPEN command. If more than one CGM file is to be created, use the following command sequence:

```

SET_PLOT, 'CGM'
    ; Select the CGM driver for graphics output.
DEVICE, Filename='filename1'
    ; Open the first file.
...
    ; Graphics output commands here.
DEVICE, /Close
    ; Close the first file.
DEVICE, Filename='filename2'
    ; Open another file.
...
    ; Graphics output commands here.
DEVICE, /Close
    ; Close the second file, etc.

```

Note also that color table changes should be made every time a new CGM file is opened. In the absence of changes, the new CGM file contains the default color table, rather than the current color table.

## Using Color with CGM Output

To create and load a color table with four elements, black, red, green, and blue:

```
TVLCT, [0,255,0,0], [0,0,255,0], [0,0,0,255]
```

Drawing text or graphics with a color index of 1 results in black, 2 in red, 3 in green, and 4 in blue.

### ***Changing the Image Background Color***

Images that are displayed with a black background on a monitor frequently look better in hardcopy form if the background is changed to white. This is easily accomplished with CGM output by issuing the statement, where A is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

### ***Changing the CGM Background Color***

To set the background color for a CGM metafile, use the ERASE command with a color index. The index is used to define the background color. For example:

```
ERASE, 150
```

## Binary CGM Output for VAX/OpenVMS Machines

Binary CGM output is written to a fixed-length file with a record length of 512 bytes. To display the CGM metafile with Visual Numerics' VAX/OpenVMS Extended Metafile System, provide the CGM metafile filename in quotes followed by format 2, record 512.

For example,

```
CGM> set metafile "cgmfile" format 2 record 512
```

```
CGM> interpret
```

---

## HPGL Output

HPGL (Hewlett-Packard Graphics Language) is a plotter control language used to produce graphics on a wide family of pen plotters.

To use HPGL as the current graphics device, issue the command:

```
SET_PLOT, 'HP'
```

This causes PV-WAVE to use HPGL for producing graphical output. Once the HPGL driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, as described in [Controlling HPGL Output with DEVICE Keywords](#) on page B-9. The default settings for the HPGL driver are given in the following table:

Setting	Default Value
Output file name	wave.hp
Orientation	portrait
Erase	no action
Polygon fill	software
Turn plotter logically on/off	no
Specify xon/xoff flow control	yes
Horizontal offset	.3175 cm (.125 in.)
Vertical offset	11.43 cm (4.5 in.)
Width	17.78 cm (7 in.)
Height	12.7 cm (5 in.)

Use the statement:

```
INFO, /Device
```

to view the current HPGL driver settings.

## Controlling HPGL Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the HPGL driver:

**Close\_File** — PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close\_File* keyword outputs any buffered commands and closes the file.

---

**CAUTION** Under UNIX, if you close the output file with the *Close\_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET\_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

---

**NOTE** See the discussion of printing output files in the *PV-WAVE User's Guide* for more information on this topic.

---

**Eject** — In order to perform an erase operation on a plotter, it is necessary to remove the current sheet of paper and load a fresh sheet. The ability of various plotters to do this varies, so the *Eject* keyword allows you to specify what should be done. The following table gives the possible values:

Value	Meaning
0	(Default) Do nothing. Note that this is likely to cause one page to plot over the previous one, so you should limit yourself to one page of output per file.
1	Use the sheet feeder to load the next page.
2	Put the plotter off-line at the beginning of each page, except the first.

---

Many HPGL plotters lack a sheet feeder, and require you to load the next page manually. Therefore, the default action is for PV-WAVE to not issue any page eject

instructions. In this case, you must restrict yourself to generating only a single plot at a time.

If your plotter has a sheet feeder, you need to issue the command:

```
DEVICE, /Eject
```

to tell PV-WAVE that it should use the sheet feeder instead of placing the plotter off-line. If your plotter does not have a sheet feeder, but it does understand the HPGL NR command, use the command:

```
DEVICE, Eject=2
```

to place the plotter off-line at the start of every plot except the first one. This causes the plotter to wait between plots for you to replace the paper. When you put the plotter back on-line, the graphics commands for the new page are executed by the plotter. Consult the programming manual for your plotter to determine if this instruction is provided.

**Filename** — By default all generated output is sent to a file named `wave.hp`. The *Filename* keyword can be used to change this default. If you specify a filename, the following occurs:

- If the file is already open (as happens if plotting commands have been directed to the file since the call to SET\_PLOT), then the file is completed and closed as if *Close\_File* had been specified.
- The specified file is opened for subsequent graphics output.

**Inches** — By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, these keywords are taken to be in inches instead.

**Landscape** — PV-WAVE normally generates plots with portrait orientation (the *x*-axis is along the short dimension of the page). If *Landscape* is present, landscape orientation (the *x*-axis is along the long dimension of the page) is used instead.

**Output** — Specifies a scalar string that is sent directly to the graphics output file without any processing, allowing you to send arbitrary commands to the file. Since PV-WAVE does not examine the string, it is your responsibility to ensure that the string is correct for the target device.

**Polyfill** — Some plotters (e.g., HP7550A) can perform polygon filling in hardware, while others (e.g., HP7475) cannot. PV-WAVE therefore assumes that the plotter cannot, and generates all polygon operations in software using line drawing. Specifying a nonzero value for the *Polyfill* keyword causes PV-WAVE to use the hardware polygon filling. Setting it to zero reverts to software filling.

Different implementations of HPGL plotters may have different limits for the number of vertices that can be specified for a polygon region before the plotter runs out of internal memory. Since this limit can vary, the HPGL driver cannot check for calls to *Polyfill* that specify too many points. Therefore, it is possible for you to produce HPGL output that causes an error when sent to the plotter. To avoid this situation, minimize the number of points used. On the HP7550A, the limit is about 127 points. If you do generate output that exceeds the limit imposed by your plotter, you have to break that polygon filling operation into multiple smaller operations.

***Plotter\_On\_Off*** — There are some configurations in which an HPGL plotter is connected between the computer and a terminal. In this mode (known as eavesdrop mode), the plotter ignores everything it is sent and passes it through to the terminal — the plotter is logically off. This state continues until an escape sequence is sent that turns the plotter logically on. At this point the plotter interprets and executes all input as HPGL commands. Another escape sequence is sent at the end of the HPGL commands to return the plotter to the logically off state.

Most configurations do not use eavesdrop mode, and the plotter is always logically on. However, if you are using this style of connection, you must use *Plotter\_On\_Off* to instruct PV-WAVE to generate the necessary on/off commands. If present and nonzero, *Plotter\_On\_Off* causes each output page to be bracketed by device control commands that turn the plotter logically on and off. Specifying a value of zero stops the issuing of such commands. You should only use this keyword before any output has been generated.

***Portrait*** — If *Portrait* is present, PV-WAVE generates plots using portrait orientation, the default.

***Xoffset*** — Specifies the *x* position on the page of the lower-left corner of output. *Xoffset* is specified in centimeters unless *Inches* is specified. (In some cases, offset is taken from the origin. See the Note at the end of this section for details.)

***Xon\_Xoff*** — If present and nonzero, *Xon\_Xoff* causes each output page to start with device control commands that instruct the plotter to obey xon/xoff (^S/^Q) style flow control. Specifying a value of zero stops the issuing of such commands. You should only use this keyword before any output has been generated. Such handshaking is the default. To turn it off, use the command:

```
DEVICE, Xon_Xoff=0
```

Often, it is not necessary to tell the plotter to obey flow control because the printing facilities on the system handle such details for you, but it is usually harmless.

***Xsize*** — Specifies the width of output PV-WAVE generates. *Xsize* is specified in centimeters unless *Inches* is specified.

***Yoffset*** — Specifies the y position on the page of the lower-left corner of output generated by PV-WAVE. *Yoffset* is specified in centimeters unless *Inches* is specified. (In some cases, offset is taken from the origin. See the Note following the keyword descriptions for details.)

***Ysize*** — Specifies the height of output generated by PV-WAVE. *Ysize* is specified in centimeters unless *Inches* is specified.

If you are plotting to an HP plotter in the series 757X, 758X, and 759X, you will need to specify the *Xoffset* and *Yoffset* from the origin instead of the lower-left corner of output. Because the default values for *Xoffset* and *Yoffset* are assume a lower-left corner origin, you will have to set these keywords to appropriate values whenever plotting to one of the above listed plotters. For example, to center a plot, use the following equations to find the offsets:

$$Xoffset = -(Xsize/2)$$

$$Yoffset = -(Ysize/2)$$

where *Xsize* and *Ysize* are the width and height, respectively, of the plot to be generated.

---

**NOTE** If you have trouble with plots not completing and you have a plotter that allows you to set switches to enable pen buffering, make sure that this switch is set. If it is not, turn the plotter off, set the switch, and send your plot to the plotter again.

---

## Supported Features of HPGL

PV-WAVE is able to produce a wide variety of graphical output using HPGL.

Here is a list of what is supported:

- All types of vector graphics can be generated, including line plots, contours, surfaces, etc.
- HPGL plotters can draw lines in different colors selected from the pen carousel. It should be noted that color tables are not used with HPGL. Instead, each color index refers directly to one of the pens in the carousel.
- Some HPGL plotters can do polygon filling in hardware. Others can rely on the software polygon filling provided by PV-WAVE.
- It is possible to generate graphics using the hardware-generated text characters, although such characters do not give much improvement over the standard software fonts. To use hardware characters, set the !P.Font system variable to zero, or set the *Font* keyword to the plotting routines to zero. For more information on fonts, see the *PV-WAVE User's Guide*.

- Here is a list of what is not supported:
- Since HPGL is designed to drive pen plotters, it does not support the output of raster images. Therefore the TV and TVSCL procedures do not work with HPGL.
- Since pen plotters are not interactive devices, they cannot support such operations as cursors and windows.

## Specifying Linestyles in HPGL Output

The *Linestyle* graphics keyword allows you to specify any of 6 linestyles. This keyword is documented in [Chapter 3, Graphics and Plotting Keywords](#).

HPGL is not able to support all of the standard PV-WAVE linestyles. The following table summarizes the differences between the PV-WAVE linestyles and those supported by HPGL.

### HPGL Supported Linestyles

Index	Normal Line Style	HPGL Style
0	Solid	same
1	Dotted	same
2	Dashed	same
3	Dash Dot	The relative size of the dash and dot are different.
4	Dash Dot Dot Dot	Dash Dot Dot
5	Long Dashes	same

**TIP** If your HPGL plotter is connected to an HP-IB interface, you must run PV-WAVE's HPGL output through a filter before you can plot it. The following UNIX command accomplishes this task:

```
tr -d '/012' <wave.hp >newwave.hp
```

---

## PCL Output

PCL (Printer Control Language) is used by Hewlett-Packard laser and ink jet printers to produce graphics output. This driver does not support the use of color and does not support the use of hardware fonts.

---

**NOTE** When printing from VDA Tools to a PCL device, all output that appears in black in the VDA Tool will appear in white on the printed page (i.e., it will not be visible).

---

To direct graphics output to a PCL file, issue the command:

```
SET_PLOT, 'PCL'
```

This causes PV-WAVE to use the PCL driver for producing graphical output. Once the PCL driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, as described in [Controlling PCL Output with DEVICE Keywords](#) on page B-15. The default settings for the PCL driver are given in the following table:

Setting	Default Value
Output file name	wave.pcl
Mode	Portrait
Optimization Level	0 (None)
Dither Method	Floyd-Steinberg
Resolution	300 dpi
Horizontal Offset	1.27 cm (1/2 in.)
Vertical Offset	2.54 cm (1 in.)
Width	17.78 cm (7 in.)
Height	12.7 cm (5 in.)

By default, PCL writes black on white paper. The default color index when PCL is selected is 0. If the color index is set to 255 when PCL is selected, the result is white writing on white, which is invisible on white paper. Color tables are not used with PCL.

Use INFO, /Device to view the driver's current settings.

---

**UNIX USERS** When you print files to a PCL device from a UNIX system, you may need to specify the `-v` option in the print command. This option indicates that a raster image is being transmitted. For example:

```
lpr -Plj250_q -v wave.pcl
```

---

**OpenVMS USERS** Use the `PASSALL` parameter. For example:

```
PRINT /QUEUE=LJ250_Q /PASSALL WAVE.PCL
```

---

## Controlling PCL Output with DEVICE Keywords

The following keywords to the `DEVICE` procedure provide control over the PCL driver:

***Close\_File*** — `PV-WAVE` creates, opens and writes a file containing the generated graphics output. The *Close\_File* keyword outputs any buffered commands and closes the file.

---

**CAUTION** Under UNIX, if you close the output file with the *Close\_File* `DEVICE` keyword, and then execute a command (such as `PLOT`) that creates more output, `PV-WAVE` reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use `SET_PLOT` to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

---

**NOTE** For more information on this topic, see the discussion of printing output files in the *PV-WAVE User's Guide*.

---

***Filename*** — By default all generated output is sent to a file named `wave.pcl`. The *Filename* keyword can be used to change this default. If you specify a filename, the following occurs:

- If the file is already open (as happens if graphics have been directed to the file since the call to `SET_PLOT`), then the file is completed and closed as if *Close\_File* had been specified.
- The specified file is opened for subsequent graphics output.

***Floyd*** — If present and nonzero, selects the Floyd-Steinberg method of dithering. For information on this dithering method, see the *PV-WAVE User's Guide*.

**Inches** — By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, they are taken to be in inches instead.

**Landscape** — PV-WAVE normally generates plots with portrait orientation (the *x*-axis is along the short dimension of the page). If *Landscape* is present, landscape orientation (the *x*-axis is along the long dimension of the page) is used instead.

**Optimize** — It is desirable, though not always possible, to compress the size of the output file. Such optimization reduces the size of the output file, and improves I/O speed to the printer. There are three levels of optimization:

- **0** — No optimization is performed. This is the default because it will work with any PCL device. However, users of devices which can support optimization should use one of the other optimization levels.
- **1** — Optimization is performed using PCL optimization primitives. This gives the best output compression and printing speed. Unfortunately, not all PCL devices support it. On those that can't, the result will be garbage printed on the page.

The required sequences are: <ESC>\*b0M (Select full graphics mode), <ESC>\*b1M (Select compacted graphics mode 1), and <ESC>\*b2M (Select compacted graphics mode 2). To determine if your printer supports the required escape sequences, consult the programmers manual for the device.

The HP LaserJet II does not support this optimization level. The DeskJet PLUS does.

- **2** — PV-WAVE attempts to optimize the output by explicitly moving the left margin and then outputting non-blank sections of the page. This is primarily intended for use with the LaserJet II, which does not support optimization level 1.

---

**NOTE** Level 2 optimization can be very slow on some devices (such as the DeskJet PLUS). On such devices, it is best to avoid this optimization level.

---

**Ordered** — Selects the Ordered Dither method of dithering when displaying images on a monochrome display. For information on this dithering method, see the *PV-WAVE User's Guide*.

**Pixels** — By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Pixels* is present and nonzero, they are taken to be in

pixels instead. Note that the selected resolution will determine how large a region is actually written on the page.

**Portrait** — If *Portrait* is present, PV-WAVE will generate plots using portrait orientation, the default.

**Resolution** — The resolution at which the PCL printer will print. PCL supports resolutions of 75, 100, 150, and 300 dots per inch. The default is 300 dpi. Lower resolution gives smaller output files, while higher resolution gives superior quality.

**Threshold** — Specifies use of the threshold dithering algorithm. For information on this dithering method, see the *PV-WAVE User's Guide*.

**Xoffset** — Specifies the *x* position on the page of the lower-left corner of output generated by PV-WAVE. *Xoffset* is specified in centimeters unless *Inches* or *Pixels* is specified.

**Xsize** — Specifies the width of the PV-WAVE output. *Xsize* is specified in centimeters unless *Inches* or *Pixels* is specified.

**Yoffset** — Specifies the *y* position on the page of the lower-left corner of output generated by PV-WAVE. *Yoffset* is specified in centimeters unless *Inches* or *Pixels* is specified.

**Ysize** — Specifies the height of the PV-WAVE output. *Ysize* is specified in centimeters unless *Inches* or *Pixels* is specified.

## PCL Image Background Color

Images that are displayed with a black background on a monitor frequently look better if the background is changed to white when displayed with PCL. This is easily done with the following statement, where A is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

---

## Pixel Map Output

The Pixel Map buffer allows you to create PV-WAVE graphical output in memory. This output buffer is useful for creating images using batch mode methods or background processes.

To direct graphics output to the PM buffer, enter the command:

```
SET_PLOT, 'PM'
```

This causes PV-WAVE to use the PM buffer driver for producing graphical output. Once the PM buffer driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, as described in [Controlling PM buffer Output with DEVICE Keywords](#) on page B-18.

Use INFO, /Device to view the driver's current settings.

## Controlling PM buffer Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control of the PM buffer driver:

**Close** — Deallocates the memory used by the buffers. The PM buffer device is reinitialized if subsequent graphics operations are directed to the device.

**Get\_Graphics\_Function** — See the description of the *Get\_Graphics\_Function* keyword in [X Window System](#) on page B-58.

**Get\_Write\_Mask** — See the description of the *Get\_Write\_Mask* keyword in [X Window System](#) on page B-58.

**Set\_Character\_Size** — A two-element vector that changes the standard width and height of the vector-drawn fonts. The first element in the vector contains the new character width, and the second element contains the height. By default, characters are approximately 8-pixels wide, with 12 pixels between lines.

**Set\_Colors** — Sets the number of pixel values, !D.N\_Colors. This value is used by a number of routines to determine the scaling of pixel data and the default drawing index. Allowable values range from 2 to 256, and the default value is 256. Use this parameter to make the PM buffer device compatible with devices with fewer than 256 color indices. The number of colors can be changed at any time without affecting any accumulated graphics present in the buffer (it does not delete the current buffer).

**Set\_Graphics\_Function** — See the description of the *Set\_Graphics\_Function* keyword in [X Window System](#) on page B-58.

The PM buffer allows you to use all graphics functions supported by the X driver.

**Set\_Resolution** — Two-element vector that sets the width and height of the buffers. The default size is 640-by-512. If this size is not the same as the existing buffers, the current buffers are destroyed and the device is reinitialized.

**Set\_Write\_Mask** — See the description of the *Set\_Write\_Mask* keyword in [X Window System](#) on page B-58.

### **Example 1**

```
SET_PLOT, 'PM'  
    ; Switch output to the PM buffer.  
DEVICE, Set_Resolution = [256,256], $  
    Set_Colors=128  
    ; Make a 256x256 buffer with 128 colors.  
LOADCT, 15  
    ; Load a color table.  
SHADE_SURF, data  
    ; Make a plot.  
TVLCT, r, g, b, /Get  
    ; Get the color table RGB values.  
cmap =TRANSPOSE([[r],[g],[b]])  
    ; Put RGB values into a colormap array.  
imgarr = TVRD()  
    ; Read the image from the PM buffer.  
stat = IMAGE_WRITE('output.gif', $  
    IMAGE_CREATE(imgarr, Colormap=cmap))  
    ; Write the image and colormap to a GIF file.  
DEVICE, /Close  
    ; De-allocate memory used by the device.
```

---

## **PostScript Output**

---

**NOTE** The default PostScript fonts changed with PV-WAVE 6.21. The previous default PostScript font was 12 point Helvetica. The new default PostScript font is 14 point Times Roman. You can change the default font by editing the file `fontmap_ps`, which is discussed in the chapter *Using Fonts* in the *PV-WAVE User's Guide*. To see which

---

PostScript is a programming language designed to convey a description of virtually any desired page containing text and graphics. It is widely available on laser printers and typesetters.

To direct graphics output to a PostScript file, enter the command:

```
SET_PLOT, 'PS'
```

This causes PV-WAVE to use the PostScript driver for producing graphical output. Once the PostScript driver is enabled via SET\_PLOT, the DEVICE procedure controls its actions, as described in *Controlling PostScript Output with DEVICE Keywords* on page B-21. The default settings for the PostScript driver are given in the following table.

#### Postscript Driver Default Settings

Setting	Default Value
Output file name	wave.ps
Mode	portrait, non-encapsulated, no color
Paper	letter
Horizontal offset	1.905 cm (3/4 in.)
Vertical offset	12.7 cm (5 in.)
Width	17.78 cm (7 in.)
Height	12.7 cm (5 in.)
Scale factor	1.0
Font size	14 pt.
Font	Times Roman
bits / image pixel	4

---

**NOTE** Unlike monitors where white is the most visible color, PostScript writes black on white paper. Setting the output color index to 0, the default when PostScript output is selected, writes black. A color index of 255 writes white which is invisible on white paper. Color tables are not used with PostScript unless the color mode has been enabled using the DEVICE procedure. See below for information on using PV-WAVE with color PostScript.

---

---

**NOTE** All PostScript printers impose a limit on the number of vertices a polygon may contain. This limit (750 vertices for most printers) is checked, and an error message is printed if it is exceeded.

---

---

**TIP** Use INFO, /Device to view the driver's current settings.

---

---

**NOTE** PostScript hardware fonts can be rotated and transformed by general 3D transforms in PV-WAVE.

---

### ***Additional Text Formatting Commands***

The following text formatting commands are new and can be used with the PostScript, WIN32, WMF, and X drivers:

<b>Formatting Command</b>	<b>Description</b>
!FB	Switch to the bold face of the current font.
!FI	Switch to the italic face of the current font.
!FU	Underline the current font.
!FN	Switch to the normal form of the current font.
!Pxx	Switch to point size <i>xx</i> of the current font, where <i>xx</i> is a two digit integer (01-99).

Refer to the *PV-WAVE User's Guide* for a comprehensive list of text formatting commands.

### **Controlling PostScript Output with DEVICE Keywords**

The following keywords to the DEVICE procedure provide control over the PostScript driver:

PostScript fonts are selected using DEVICE keywords. For example:

```
DEVICE, /Helvetica, /Bold
```

See *Using PostScript Fonts* on page B-26 for a complete list of these keywords.

***Bits\_Per\_Pixel*** — PV-WAVE is capable of producing PostScript images with 1, 2, 4, or 8 bits per pixel. Using more bits per pixel gives higher resolution at the cost of generating larger files. *Bits\_Per\_Pixel* is used to specify the number of bits to use. The default number of bits is four.

The Apple Laserwriter is capable of only 32 different shades of gray (which can be represented by 5 bits). Thus, specifying 8 bits per pixel does not give 256 levels of gray as might be expected, only 32, at a cost of sending twice the number of bits to

the printer. Often, 4 bits (16 levels of gray) will give acceptable results with a large savings in file size.

**Close\_File** — PV-WAVE creates, opens and writes a file containing the generated graphics output. The *Close\_File* keyword outputs any buffered commands and closes the file.

---

**CAUTION** Under UNIX, if you close the output file with the *Close\_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET\_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

---

**Color** — Enables color PostScript output if present and nonzero. For details on using color PostScript devices see *Using Color PostScript Devices* on page B-29.

**Encapsulated** — Specifies that the PostScript produced by PV-WAVE is to be included in another PostScript document, such as one produced by T<sub>E</sub>X, L<sub>A</sub>T<sub>E</sub>X, FrameMaker, Microsoft Word, or Ventura Publisher.

By default PV-WAVE assumes that its PostScript-generated output will be sent directly to a printer. It therefore includes PostScript commands to center the plot on the page and to eject the page from the printer. These commands are undesirable if the output is going to be inserted into the middle of another PostScript document. If *Encapsulated* is present and nonzero, PV-WAVE does not generate these commands.

PV-WAVE follows the standard PostScript convention for encapsulated files. It assumes the standard PostScript scaling is in effect (72 points per inch). In addition, it declares the size, or bounding box, of the plotting region at the top of the output file. This size is determined when the output file is opened (when the first graphics command is given) by multiplying the size of the plotting region (as specified with the *Xsize* and *Ysize* keywords) by the current scale factor (as specified by the *Scale\_Factor* keyword). Changing the size of the plotting region or scale factor once any graphics have been output will not be reflected in the declared bounding box, and will confuse programs that attempt to import the resulting graphics. Therefore, when generating encapsulated PostScript do not change the plot region size of scaling factor after issuing graphics commands. If you want to change these parameters, use the *Filename* keyword to start a new file.

To explicitly disable the encapsulated PostScript option, use DEVICE, *encaps=0*.

---

**NOTE** You cannot reposition encapsulated output (including EPSI output) via *Xoffset* or *Yoffset* keywords. This allows full positioning control for the including product, such as MS Word.

---

***Epsi*** — Produces an output file in encapsulated PostScript interchange (EPSI) format. When the *Epsi* keyword is present and nonzero, the *Encapsulated* keyword is automatically assumed. For detailed information on the *Encapsulated* keyword, see the description given previously in this section.

Like normal encapsulated PostScript files, EPSI files can be imported into many desktop publishing and word processing systems; however, EPSI format provides a previewing capability that allows an approximation of the printed PostScript output to be displayed on the screen. Previewed EPSI graphics are displayed in monochrome only.

---

**TIP** EPSI images sometimes appear differently when printed than they do in windows on the screen. This occurs because the driver's EPSI bitmap size is fixed no matter what the size of the original PV-WAVE image, and the PV-WAVE image is scaled into this bitmap. In addition, printed PostScript files can take advantage of scalable pixels, while pixels on the screen are static. Try experimenting with the other DEVICE keywords when you specify */Epsi* to see which combination gives the best results with your particular size image. The driver's EPSI bitmap size (888-by-635 pixels) works well in many cases.

---

---

**NOTE** You can print EPSI files directly, but only when the file is generated in portrait mode (see the description of the *Portrait* keyword below).

---

***Filename*** — By default all generated output is sent to a file named *wave.ps*. The *Filename* keyword can be used to change this default. If you specify a filename, one or both of the following actions occur:

- If the file is already open (as happens if plotting commands have been directed to the file since the call to SET\_PLOT), then the file is completed and closed as if *Close\_File* had been specified.
- The specified file is opened for subsequent graphics output.

***Font\_Size*** — Specifies the default height used for displayed text. *Font\_Size* is given in points (a common unit of measure used in typesetting). The default size is 12 point text.

**Get\_Fontmap** — Returns a list of the current font map settings. If `/Get_Fontmap` is specified, the information is printed to the screen. If a variable name is specified (e.g., `Get_Fontmap=varname`), a string array is returned.

The information returned by `Get_Fontmap` is in the same form as the strings specified with `Set_Fontmap`. The only exception to this pattern is that the first string returned in a string array is labeled as font 0 and is the current default font.

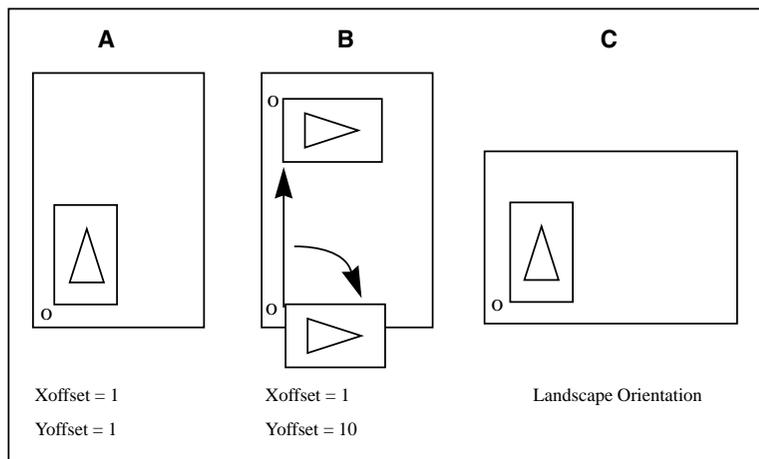
**Inches** — By default, the `Xoffset`, `Xsize`, `Yoffset`, and `Ysize` keywords are specified in centimeters. However, if `Inches` is present and nonzero, they are taken to be in inches instead.

**Landscape** — PV-WAVE normally generates plots with portrait orientation (with the  $x$ -axis along the short dimension of the page). If `Landscape` is present, landscape orientation (with the  $x$ -axis along the long dimension of the page) is used instead.

---

**TIP** In both portrait and landscape mode, the  $x$  offset is measured as a displacement along the page's short dimension, and  $y$  offset is measured as a displacement along the page's long dimension. This may cause you some confusion when you are trying to orient graphics on a landscape page. The following figure demonstrates the correct way to specify offset for PostScript's landscape mode.

---



**Output** — Specifies a scalar string that is sent directly to the graphics output file without any processing, allowing you to send arbitrary commands to the file. Since PV-WAVE does not examine the string, it is your responsibility to ensure that the string will be recognized by the target device.

**Papername** — Specifies the size of paper to be used for printing. If you set the keyword value to 'Letter', then standard 8.5x11-inch paper is used; 'tabloid' is 11x17-inch paper (Default: 'Letter')

---

**NOTE** For 11x17 output, you must be using a PostScript Level 2-compliant device.

---

**TIP** The PostScript driver does not automatically scale up for 11x17 output. You may want to adjust the *Xsize*, *Ysize*, *Xoffset*, *Yoffset* keywords accordingly.

---

**Path\_Points** — Lets you change the maximum number of points in any PostScript path. The *PostScript Language Reference Manual* defines this limit as “Maximum number of points specified in all active path descriptions, including the current path, clip path, and paths saved by save and gsave”. The default value is 750 because that is the maximum that some printers can handle. The *PostScript Language Reference Manual* states that 1500 is a typical limit for Level 1 implementations. The maximum for any particular printer can depend on the PostScript implementation being used, the amount of memory in the printer itself and other software being used to access the printer. If you set *Path\_Points* to a number larger than the default and your printer reports a PostScript memory error, then reduce the *Path\_Points* number.

**Portrait** — If *Portrait* is present, PV=WAVE generates plots using portrait orientation (the default).

**Scale\_Factor** — Specifies a scale factor applied to the entire plot. Its default value is 1.0, allowing output to appear at its normal size. *Scale\_Factor* magnifies or shrinks the resulting output.

**Set\_Fontmap** — Associates specific font characteristics with a particular hardware font command. This keyword can be used to specify a single string containing the information for one font, or an array of strings containing the information for multiple fonts.

For example, to associate 16 point Helvetica italic with font command !5, use the following DEVICE call:

```
DEVICE, Set_Fontmap=' 5 Helvetica-Oblique, 16'
```

Now, whenever the !5 font command appears in a text string, the text output appears in 16 point Helvetica italic. This keyword only affects hardware fonts (that is, when !P.Font=0).

---

**NOTE** A default font mapping is defined in a configuration file that is read when the PS driver is initialized. For information on this configuration file, see the *Using Fonts* chapter in the *PV-WAVE User's Guide*.

---

**User\_Font** — A scalar string that gives the name of a PostScript font to use. This font name must be specified exactly as the PostScript interpreter expects it, with the correct case and spelling. *User\_Font* lets you use fonts not known to PV-WAVE.

**Xoffset** — Specifies the *x* position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Xoffset* is specified in centimeters unless the *Inches* keyword is specified. *Scale\_Factor* does not affect the value of *Xoffset*.

**Xsize** — Specifies the width of output. By default, *Xsize* is specified in centimeters unless the *Inches* keyword is specified. *Scale\_Factor* modifies the value of *Xsize*. Hence, the statement:

```
DEVICE, /Inches, Xsize=7.0, Scale_Factor=0.5
```

results in a real width of 3.5 inches.

**Yoffset** — Specifies the *y* position on the page of the lower-left corner of output generated by PV-WAVE. By default, *Yoffset* is specified in centimeters unless the *Inches* keyword is specified. *Scale\_Factor* does not affect the value of *Yoffset*.

**Ysize** — Specifies the height of output. By default, *Ysize* is specified in centimeters unless the *Inches* keyword is specified. *Scale\_Factor* modifies the value of *Ysize*. Hence, the statement:

```
DEVICE, /Inches, Ysize=5.0, Scale_Factor=0.5
```

results in a real width of 2.5 inches.

## Using PostScript Fonts

By default, PV-WAVE uses software characters for annotating plots (i.e., !P.Font is -1). These characters are of good quality and are extremely flexible. However, if this flexibility is not required, higher quality characters are available via PostScript fonts. In order to get PV-WAVE to use the PostScript fonts, do *one* of the following:

- Set !P.Font to 0 by entering:

```
!P.Font = 0
```

- Use the *Font* keyword with the plotting and graphics procedures to specify font 0. For example:

```
PLOT, temps, Title='Average Temp', Font=0
```

The default PostScript font is 12-point Helvetica. To change this font, use the DEVICE procedure keywords, as shown in the following table. Note that not all fonts may be available on a particular device.

---

**NOTE** When generating three-dimensional plots, it is best to use the software fonts, because PV-WAVE can draw them in perspective with the rest of the plot. For details on software fonts, see the *PV-WAVE User's Guide*.

---

#### PostScript Fonts

<b>PostScript Font</b>	<b>DEVICE Keywords</b>
Courier	/Courier
Courier Bold	/Courier, /Bold
Courier Oblique	/Courier, /Oblique
Courier Bold Oblique	/Courier, /Bold, /Oblique
Helvetica	/Helvetica
Helvetica Bold	/Helvetica, /Bold
Helvetica Oblique	/Helvetica, /Oblique
Helvetica Bold Oblique	/Helvetica, /Bold, /Oblique
Helvetica Narrow	/Helvetica, /Narrow
Helvetica Narrow Bold	/Helvetica, /Narrow, /Bold
Helvetica Narrow Oblique	/Helvetica, /Narrow, /Oblique
Helvetica Narrow Bold Oblique	/Helvetica, /Narrow, /Bold, /Oblique
ITC Avant Garde Gothic Book	/Avantgarde, /Book
ITC Avant Garde Gothic Book Oblique	/Avantgarde, /Book, /Oblique
ITC Avant Garde Gothic Demi	/Avantgarde, /Demi
ITC Avant Garde Gothic Demi Oblique	/Avantgarde, /Demi, /Oblique
ITC Bookman Demi	/Bkman, /Demi
ITC Bookman Demi Italic	/Bkman, /Demi, /Italic
ITC Bookman Light	/Bkman, /Light
ITC Bookman Light Italic	/Bkman, /Light, /Italic

## PostScript Fonts (Continued)

<b>PostScript Font</b>	<b>DEVICE Keywords</b>
ITC Zapf Chancery Medium Italic	/Zapfchancery, /Medium, /Italic
ITC Zapf Dingbats	/Zapfdingbats
New Century Schoolbook	/Schoolbook
New Century Schoolbook Bold	/Schoolbook, /Bold
New Century Schoolbook Italic	/Schoolbook, /Italic
New Century Schoolbook Bold Italic	/Schoolbook, /Bold, /Italic
Palatino	/Palatino
Palatino Bold	/Palatino, /Bold
Palatino Italic	/Palatino, /Italic
Palatino Bold Italic	/Palatino, /Bold, /Italic
Symbol	/Symbol
Times	/Times
Times Bold	/Times, /Bold
Times Italic	/Times, /Italic
Times Bold Italic	/Times, /Bold, /Italic

When using PostScript fonts, commands may be inserted into the text in order to specify positioning and special characters. These commands are described in the following table. They are similar to the commands provided for the standard software characters.

### PostScript Text Positioning Commands

<b>Command</b>	<b>Description</b>
!A	Shift above the division line.
!E	Shift up to the exponent level and decrease the character size by a factor of 0.44.
!MX	Insert a bullet character.
!N	Shift back to the normal level and original character size.

## PostScript Text Positioning Commands (Continued)

Command	Description
!B	Shift below the division line.
!I	Shift down to the index level and decrease the character size by a factor of 0.44.
!!	Display the ! symbol.

## Using Color PostScript Devices

If you have a color PostScript device you can enable the use of color with the statement:

```
DEVICE, /Color
```

Enabling color also enables the color tables. Text and graphic color indices are translated to RGB by dividing the red, green, and blue color table values by 255. As with most display devices, color indices range from 0 to 255. Zero is normally black and white is normally represented by an index of 255.

## PostScript Supports Color Images

As with black and white PostScript, images may be output with 1, 2, 3, 4 or 8 bits, yielding 1, 2, 16, or 256 possible colors. In addition, images can be either pseudo-color or true color. For a thorough comparison of pseudo-color and true color, see the *PV-WAVE User's Guide*.

## Changing the Image Background Color

Images that are displayed with a black background on a monitor frequently look better if the background is changed to white when displayed with PostScript. This is easily done with the following statement, where A is the image variable:

```
A(WHERE(A EQ 0B)) = 255B
```

## Creating Publication-quality Documents

The combination of PV-WAVE and the PostScript page description language gives you a powerful tool for creating publication-quality documents with text, graphics, and images arranged together on the page. Three examples of how to insert PostScript output into text documents follow, but these are not exclusive. Consult your word processing or desktop publishing manual to see how your software handles encapsulated PostScript files.

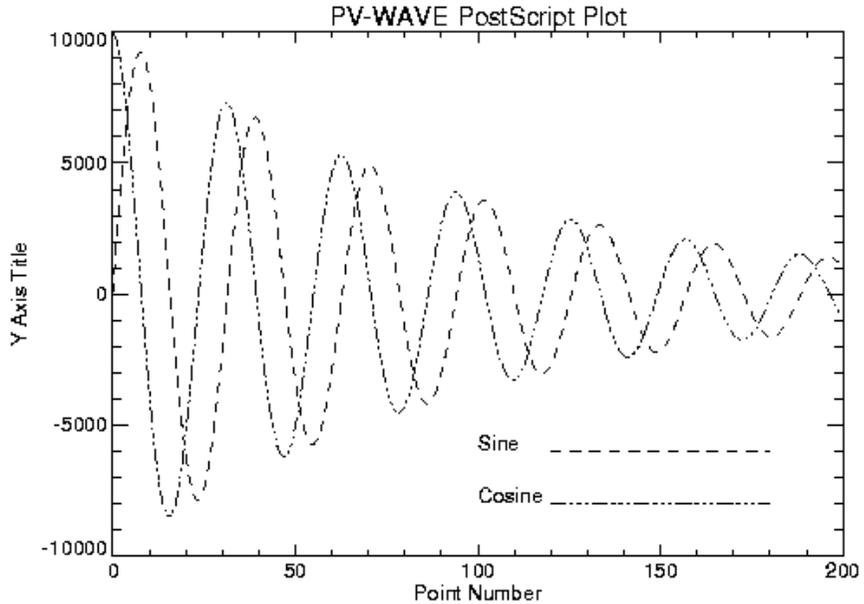
## ***Inserting PV-WAVE Plots into L<sub>A</sub>T<sub>E</sub>X Documents***

L<sub>A</sub>T<sub>E</sub>X is a special version of the T<sub>E</sub>X page formatting language developed by Donald E. Knuth for producing publication-quality documents. L<sub>A</sub>T<sub>E</sub>X was developed by Leslie Lamport to make the T<sub>E</sub>X language easier to use. Although T<sub>E</sub>X is released into the public domain, commercial versions of T<sub>E</sub>X are supported by private vendors. One such vendor is ArborText, Inc., of Ann Arbor, Michigan. The examples below describe how Visual Numerics has used ArborText's T<sub>E</sub>X, L<sub>A</sub>T<sub>E</sub>X, and DVILASER/PS software on Sun workstations to incorporate PV-WAVE graphs and images into publication-quality documents.

is an example of a PV-WAVE-generated PostScript plot that has been inserted into a L<sub>A</sub>T<sub>E</sub>X document. It was produced with the following statements:

```
SET_PLOT, 'PS'  
    ; Select the PostScript driver.  
  
DEVICE, /Encapsulated, Filename='pic1.ps'  
    ; Use ENCAPSULATED because output is for use with LATEX.  
  
x = FINDGEN(200)  
PLOT, 10000 * SIN(x/5) / EXP(x/100), $  
    Linestyle=2, Title='PV-WAVE ' + $  
    'PostScript plot', Xtitle='Point Number', $  
    Ytitle='Y Axis Title', Font=0  
    ; Plot the sine wave. Set the font to hardware font.  
  
OPLOT, 10000 * COS(x/5) / EXP(x/100), Linestyle=4  
    ; Add the cosine.  
  
XYOUTS, 100, -6000, 'Sine', Font=0  
    ; Annotate the plot.  
  
OPLOT, [120, 180], [-6000, -6000], Linestyle=2  
    ; Annotate the plot  
  
XYOUTS, 100, -8000, 'Cosine', Font=0  
    ; Annotate the plot  
  
OPLOT, [120, 180], [-8000, -8000], Linestyle=4
```

Note the use of the *Encapsulated* keyword in the call to DEVICE. This is what allows you to import the file into a L<sub>A</sub>T<sub>E</sub>X document. Simply omit the *Encapsulated* keyword from the call to DEVICE if you want to produce a plot that can be printed directly.



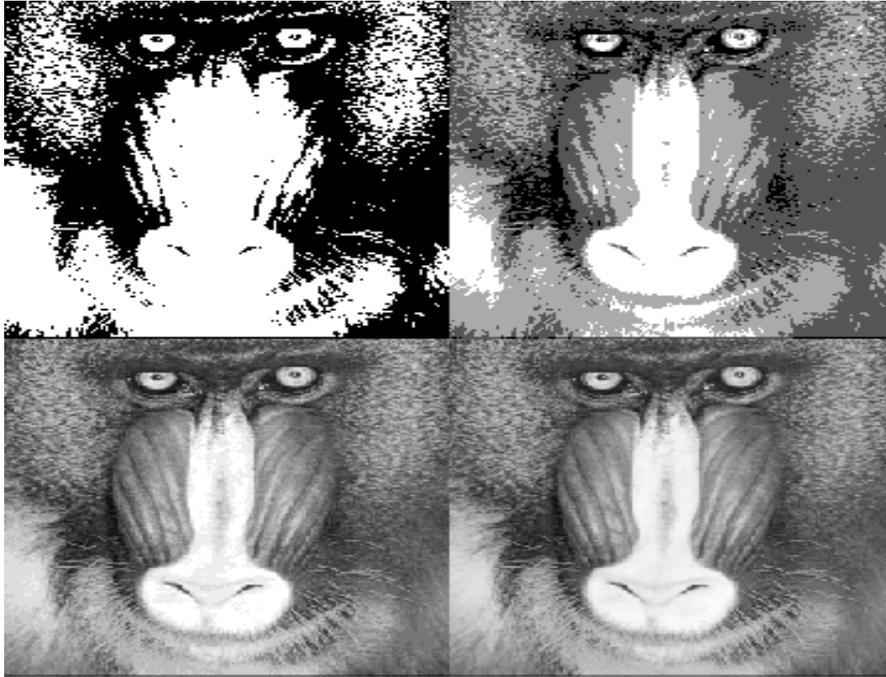
**Figure B-1** Sample PostScript plot using Helvetica font.

Any kind of PostScript plot can be included in L<sup>A</sup>T<sub>E</sub>X documents. In , a PV-WAVE PostScript image has been included. In this case, the same image is reproduced four times. Each time, a different number of bits are used per image pixel. The illustration was produced with the following statements:

```

SET_PLOT, 'PS'
DEVICE, /Encapsulated, Filename='pic4.ps'
OPENR, 1, !Dir+'/data/mandril.img'
    ; Open image file.
a = BYTARR(256, 256, /Nozero)
    ; Variable to hold image.
READU, 1, a
    ; Input the image.
CLOSE, 1
    ; Done with the file.
FOR i = 0,3 DO BEGIN
    ; Output the image four times.
    DEVICE, Bits_Per_Pixel=2^i
        ; Use 1, 2,4, and 8 bits per pixel.
    TV,a,i ,Xsize=2.5, Ysize=2.5, /Inches
        ; Output using TV with position numbers 0, 1, 2, and 3.
    ENDFOR

```



**Figure B-2** 1, 2, 4, and 8-bit PostScript images.

## The L<sub>A</sub>T<sub>E</sub>X Insertplot Macro

The following L<sub>A</sub>T<sub>E</sub>X macro, named `insertplot`, is used to insert PV-WAVE-generated PostScript files into L<sub>A</sub>T<sub>E</sub>X documents. The definition of this macro depends upon the T<sub>E</sub>X DVI to PostScript translation program used and is therefore not portable between various DVI programs.

However, given familiarity with T<sub>E</sub>X it is relatively easy to modify the macro to suit the various DVI programs encountered in practice. The `insertplot` macro is as follows:

```
% Insert a PostScript plot made by
% PV-WAVE into a LATEX document:
% For the ArborText program dvips.
%
% This macro creates a captioned figure
% of the specified size and uses the
% \special command to insert the Post
% Script.
%
% Usage: \insertplot{file}{caption}
```

```

% {label}{width}{height}
% file = name of file containing the PostScript.
% caption = caption of figure
% label = latex \label{} for figure to
% be used by \ref{} macro.
% width = width of figure, in inches.
% height = height of figure, in inches.
%
% Insert a plot.
\newcommand{\insertplot}[5]{%
% Usage: \insertplot{file}{caption}
% {label}{width in inches}{height}
\begin{figure}%
\hfill\hbox to 0.05in{\vbox to #5in{\vfil%
\inputplot{#1}{#4}{#5}%Include the plot
% file
}\hfill}%
\hfill\vspace{-.1in}% Fudge factor to
% tighten things up a bit.
\caption{#2}\label{#3}
\end{figure}}
%
%
%           Include a PostScript File, this varies
%           according to the DVI program: Usage:
%           \inputplot{filename}{width}{height}
%           When called from insertplot, the current
%           position is at the bottom CENTER of the
%           figure box.
\newcommand{\inputplot}[3]{%
%           Output PostScript commands to scale
%           default sized (7 wide by 5 high) PV-WAVE
%           plot into the specified size. Also
%           set origin to the current point less
%           half the width of the box, centering the
%           box above the current point.
\special{ps: gsave #2 -36 mul 0 rmoveto
currentpoint translate #2 7.0 div #3 5.0
div scale}%
\special{ps: plotfile #1}\special{ps:
grestore}}
%
%

```

---

## Regis Output

PV-WAVE provides Regis graphics output for the Digital VT240, VT330, and VT340 series of terminals. To output graphics to such terminals, enter the command:

```
SET_PLOT, 'REGIS'
```

This causes PV-WAVE to use the Regis driver for producing graphical output. Once the Regis driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, as described in the next section, *Controlling Regis Output with DEVICE Keywords*.

### Controlling Regis Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the Regis driver:

**Average\_Lines** — Controls the method of writing images to the VT240. If this keyword is set, as it is by default, even and odd pairs of image lines are averaged and written to a single line. If clear, each image line is written to the screen (see the discussion below).

This keyword has no effect when using a VT300 series terminal.

**Close\_File** — PV-WAVE creates, opens, and writes a file containing the generated graphics output. The *Close\_File* keyword outputs any buffered commands and closes the file.

---

**CAUTION** Under UNIX, if you close the output file with the *Close\_File* DEVICE keyword, and then execute a command (such as PLOT) that creates more output, PV-WAVE reopens the same file, erasing the previous contents. To avoid losing the contents of an output file, use the *Filename* keyword to specify a different filename, or use SET\_PLOT to switch to a different graphics driver, or be sure to print the closed output file before creating more output.

---

---

**NOTE** See the discussion of printing output files in the the *PV-WAVE User's Guide*.

---

**Filename** — By default all generated output is sent to a file named `wave.regis`. The *Filename* keyword can be used to change this default. When you specify a filename, the following occurs:

- If the file is already open (as happens if graphics have been directed to the file since the call to SET\_PLOT), then the file is completed and closed as if *Close\_File* had been specified.
- The specified file is opened for subsequent graphics output.

**Plot\_To** — Directs the Regis output that would normally go to the user's terminal to the specified I/O unit. The logical unit specified should be open with write access to a device or file.

*Do not* use the interactive graphics cursor when graphic output is not directed to your terminal. To direct the graphic data to both the terminal and the file, set the unit to the negative of the actual unit number. If the specified unit number is zero, then Regis output to the file is stopped.

**Tty** — Directs output to both a file and the terminal.

**VT240** — Sets driver for VT240 series terminals.

**VT241** — Same as *VT240*.

**VT340** — Sets driver for VT340 series terminals (the default).

**VT341** — Same as *VT340*.

The default setting for Regis output is: VT340, 16 colors, 4 bits per pixel.

## Limitations of REGIS Output

- Four colors are available with VT240 and VT241 terminals, sixteen colors are available with the VT330 and VT340.
- Thick lines are emulated by filling polygons. There may be a difference in line-style appearance between thick and normal lines.
- Image output is slow and poor quality, especially on the VT240 series.
- The VT240 is only able to write pixels on even numbered screen lines. PV-WAVE offers two methods of writing images to the VT240: Even and odd pairs of rows are averaged and written to the screen. An  $n, m$  image will occupy  $n$  columns and  $m$  screen rows. If this method is selected, graphics and image coordinates coincide. This method is the default: *Average\_Lines=1*. Routines that rely on a uniform graphics and image coordinate system, such as SHADE\_SURF, will only work in this mode. Each line of the image is written to the screen, displaying every image pixel. An  $n, m$  image occupies  $2m$  lines on the screen (*Average\_Lines=0*). Graphics and image coordinates coincide only at the lower left corner of the image.
- Pixel values may not be read back from the terminal, rendering the TVRD function inoperable.

---

## Tektronix Terminals

The Tektronix 4000 (4010, 4014, etc.), 4100 and 4200 series of graphics terminals (and the multitude of terminals and microcomputers that emulate them) are among the most common graphics devices available. To use PV-WAVE graphics with such terminals, enter the command:

```
SET_PLOT, 'TEK'
```

This causes PV-WAVE to use the Tektronix driver for producing graphical output.

### Controlling Tektronix Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the Tektronix driver:

**Colors** — The number of colors supported by the terminal. Only used with 4100 series terminals. For example, if your terminal has 4-bit planes, the number of colors is  $2^4 - 16$ :

```
DEVICE, Colors=16
```

Valid values of this parameter are: 2, 4, 8, 16, or 64; other values cause problems. Some Tektronix terminals do not operate properly if this parameter does not exactly match the number of colors available in the terminal hardware.

This parameter sets the field !D.N\_Colors, which affects the loading of color tables via the Standard Library procedures, the scaling used by the TVSCL procedure, and the number of bits output by the TV procedure to the terminal. It also changes the default color, !P. Color, to the number of colors minus one.

**Gin\_Chars** — The number of characters PV-WAVE reads when accepting a GIN (Graphics Input) report. The default is 5. If your terminal is configured to send a carriage return at the end of each GIN report, set this parameter to 6. If the number of GIN characters is too large, the CURSOR procedure will not respond until two or more keys are struck. If it is too small, the extra characters sent by the terminal will appear as input to the next PV-WAVE prompt.

**Plot\_To** — Directs the Tektronix graphic output that would normally go to the user's terminal to the specified I/O unit. The logical unit specified should be open with write access to a device or file. Save graphic output in files for later playback, redirection to other terminals, or to devices that accept Tektronix graphic commands.

*Do not* use the interactive graphics cursor when graphic output is not directed to your terminal. To direct the graphic data to both the terminal and the file, set the

unit to the negative of the actual unit number. If the specified unit number is zero, then Tektronix output to the file is stopped.

**Reset\_String** — The string used to place the terminal back into the normal interactive mode after drawing graphics. Use this parameter, in conjunction with the *Set\_String* keyword, to control the mode switching of your terminal. For example, the GraphON 200 series terminals require the string <ESC>2 to activate the alpha-numeric window after drawing graphics. The call to set this is:

```
DEVICE, Reset=string(27b) + '2'
```

If the 4100 series mode switch is set, using the keyword *Tek4100*, the default mode re-setting string is <ESC>%!1, which selects the ANSI code mode.

**Set\_String** — The string used to place the terminal into the graphics mode from the normal interactive terminal mode. If the 4100 series mode switch is set, using the keyword *Tek4100*, the default graphic mode setting string is <ESC>%!0, which selects the Tektronix code mode.

**Tek4014** — If nonzero, specifies that coordinates are to be output with full 12-bit resolution. If this keyword is not present or is zero, 10-bit coordinates are output. By default, PV-WAVE sends 10-bit coordinates. 12-bit coordinates are compatible with most terminals, even those without the full resolution, but require more characters to send.

---

**NOTE** The 4014 and the 4100 modes may be used together. The coordinate system PV-WAVE uses for the Tektronix is 0 to 4095 in the *x* direction and 0 to 3120 in the *y* direction, even when not in the 4014 mode — in the 10-bit case the internal coordinates are divided by 4 prior to output.

---

**Tek4100** — Indicates that the terminal is a 4100 series terminal. The use of color, ANSI and Tektronix mode switching, hardware line styles, and pixel output with the TV procedure is supported with 4100 series terminals. Also, text is output differently.

The default setting for Tektronix output is: 10-bit coordinates, 4000 series terminals, and no use of color.

## Notes on the Tektronix Driver

Once the Tektronix driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, and to configure PV-WAVE for the specific features of your terminal. See [Controlling Tektronix Output with DEVICE Keywords](#) on page B-36 for more information. If you never call the DEVICE procedure, PV-WAVE assumes a standard Tektronix 4000 series compatible terminal.

The 4200 series is upwardly compatible with the 4100 series; all references to the 4100 series also include the 4200 series. To set up PV-WAVE for use with a 4100 series compatible terminal with  $n$  colors:

```
SET_PLOT, 'TEK'  
DEVICE, /TEK4100, Colors = n
```

The number of colors should be set to  $2^B$  where  $B$  is the number of bit planes in your terminal. If you use a Tektronix compatible terminal that requires calling the DEVICE procedure for configuration, you should probably create and use a start-up procedure that calls the DEVICE procedure, see the section on modifying your environment in the *PV-WAVE Programmer's Guide*.

The line drawing procedures work with all models. Color and the display of images (albeit very slowly and frequently of a poor quality because of the low number of colors) is usable only with 4100 series terminals. Hardware polygon fill works only on the 4100 series.

Because of the tremendous variation among the requirements and abilities of these terminals, it is crucial that you configure PV-WAVE properly for your terminal.

## Limitations of Tektronix and Tektronix-compatible Terminals

Because of hardware restrictions, you will encounter the following limitations when using Tektronix and Tektronix-compatible terminals with PV-WAVE:

- Pixel coordinates do not match the coordinates used by the rest of the graphic procedures. This is because no two models of Tektronix terminals are alike. The graphics procedures use the default coordinate system of 1024-by-780, or 4096-by-3120 in the 12-bit mode. The size of the pixel memory and coordinate system vary widely between models. The *Position* parameter for the TV and TVSCL procedures does not work.
- The cursor cannot be positioned from the computer, meaning the TVCRS procedure may not be used in the Tektronix mode.
- Pixel values may not be read back from the terminal, rendering the TVRD function inoperable.

---

**CAUTION** If you try to display images produced with the SHADE\_SURF and SHOW3 procedures, PV-WAVE may abort. Because of a limitation in the range of image coordinates available on Tektronix devices, they are not well suited to the display of images.

---

---

## WIN32 Driver

WIN32 is the default driver for PV-WAVE running under Microsoft Windows. This driver controls graphics output from PV-WAVE to the Windows workspace.

---

**NOTE** If you are running a Windows NT system, WIN32 hardware fonts can be rotated and transformed by general 3D transforms in PV-WAVE.

---

### Selecting the WIN32 Driver

To enable WIN32 as the current graphics device, enter the following command:

```
SET_PLOT, 'WIN32'
```

This causes PV-WAVE to use the WIN32 driver for producing subsequent graphics output. Once the WIN32 driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, as described in [Controlling the WIN32 Driver with DEVICE Keywords on page 40](#) below.

---

**NOTE** WIN32 is the default device for Windows, so selecting WIN32 as the current graphics device is not necessary unless you have previously selected some other device and want to return to WIN32.

---

### Listing the Current Settings for the WIN32 Driver

Use the command:

```
INFO, /Device
```

to view the current WIN32 driver settings.

### Additional Text Formatting Commands

The following text formatting commands are new and can be used with the PostScript, WIN32, WMF, and X drivers. These commands can only be used when hardware fonts are enabled (!P.Font=0).

Formatting Command	Description
!FB	Switch to the bold face of the current font.
!FI	Switch to the italic face of the current font.
!FU	Underline the current font.
!FN	Switch to the normal form of the current font.
!Pxx	Switch to point size xx of the current font, where xx is a two digit integer (01-99).

---

Refer to the *PV-WAVE User's Guide* for a comprehensive list of text formatting commands.

## Controlling the WIN32 Driver with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the WIN32 driver:

**Close\_Display** — Deletes all generated windows from the Windows workspace and returns PV-WAVE to the initial graphics state.

**Copy** — Copies a rectangular area of pixels from one region of a window to another. The argument of *Copy* is a six- or seven-element array:  $[X_s, Y_s, N_x, N_y, X_d, Y_d, W]$ , where:  $(X_s, Y_s)$  is the lower-left corner of the source rectangle,  $(N_x, N_y)$  are the number of columns and rows in the rectangle, and  $(X_d, Y_d)$  is the coordinate of the destination rectangle. Optionally,  $W$  is the index of the window from which the pixels should be copied to the current window. If it is not supplied, the current window is used for both source and destination. *Copy* can be used to increase the pace of animation. This is described in [Using Pixmaps to Improve Application Performance](#) on page B-75.

**Cursor\_Crosshair** — Selects the crosshair cursor type.

**Cursor\_Image** — Specifies the cursor pattern. The value of this keyword must be a 16-line by 16-column bitmap, contained in a 16-element short integer vector. Each line of the bitmap must be a 16-bit pattern of ones and zeros, where one (1) is white and zero (0) is black. The offset from the upper-left pixel to the point that is considered the “hot spot” can be provided via the *Cursor\_XY* keyword.

**Cursor\_Original** — Selects the Windows default cursor — the cursor in use by the window manager when PV-WAVE starts. This cursor pattern is used by default.

**Cursor\_Wait** — Specifies the wait cursor, usually an hourglass icon.

**Cursor\_XY** — A two-element integer vector giving the  $x, y$  pixel offset of the cursor “hot spot”, the point which is considered to be the mouse position, from the upper-left corner of the cursor image. This parameter is only applicable if *Cursor\_Image* is provided. The cursor image is displayed top-down — in other words, the first row is displayed at the top.

**Direct\_Color** — When specified with a *Depth* value, selects the DirectColor visual with *Depth* bits per pixel. For example:

```
Direct_Color=24
```

The *Direct\_Color* keyword has effect only if no windows have been created. The *Direct\_Color* and *True\_Color* keywords are essentially synonymous when used with the WIN32 driver: they perform identical functions.

**Font** — Specifies the name of the font used when the hardware font is selected (that is, when !P.Font=0). For example, to select the font 14 point New Times Roman bold, use the following DEVICE call:

```
DEVICE, Font='Times New Roman, 14, bold'
```

Windows provides the Character Map accessory tool that can be used by all applications to show the fonts available on your system.

---

**NOTE** The size of the font selected also affects the size of software-drawn text (e.g., the Hershey fonts). The “!” commands accepted for software fonts for subscripts and superscripts do not work for hardware fonts.

---

TrueType hardware fonts can be rotated but not projected. When generating 3D plots, it is best to use the software characters because PV-WAVE can draw them in perspective with the rest of the plot. For more information on software fonts, see the *PV-WAVE Programmer's Guide*.

**Get\_Fontmap** — Returns a list of the current font map settings. If */Get\_Fontmap* is specified, the information is printed to the screen. If a variable name is specified (e.g., *Get\_Fontmap=varname*), a string array is returned.

The information returned by *Get\_Fontmap* is in the same form as the strings specified with *Set\_Fontmap*. The only exception to this pattern is that the first string returned in a string array is labeled as font 0 and is the current default font.

**Get\_Graphics\_Function** — Returns the value of the current graphics function (set with the *Set\_Graphics\_Function* keyword). This keyword can be used to save the value of the current graphics function, change it temporarily, and then restore it. See the example in the section [Using Graphics Functions to Manipulate Color](#) on page B-83.

**Get\_Window\_Position** — Places the *x* and *y* device coordinates (relative to the lower-left corner of the display) of the window's lower-left corner into a named variable.

**Get\_Write\_Mask** — Specifies the name of a variable to receive the current value of the write mask. For example:

```
DEVICE, Get_Write_Mask=mask
```

For more information on the write mask, refer to [Using the Write Mask and Graphics Functions to Manipulate Color](#) on page B-82.

**Max\_Lines** — Sets the maximum number of identical line segments that will be cached before forcing a screen redraw. The WIN32 driver uses an internal cache for drawing line segments. Instead of drawing each line individually, lines with common characteristics are drawn as a group. This feature increases drawing efficiency and rendering speed. To disable *Max\_Lines*, set the keyword equal to one. (Default: 500 lines.)

**Max\_Pens** — Sets the maximum number of pens that will be cached before forcing a screen redraw. (Default: 10 pens, each of which can hold 500 line segments.) To disable *Max\_Pens*, set the keyword equal to one.

---

**NOTE** The line caching feature may affect some PV-WAVE behavior that is familiar to you. In general, this applies to any operation that depends on the order in which lines are drawn. For example, using OPLOT with the color set to the background color to erase drawn lines may not work as expected. This is because some or all of the second set of lines may be drawn before the first set they were intended to overwrite. To avoid this, use the EMPTY procedure to force the driver to empty the line cache first.

---

**Pseudo\_Color** — When specified with a *Depth* value, selects the PseudoColor visual with *Depth* bits per pixel. For example:

```
Pseudo_Color=8
```

The *Pseudo\_Color* keyword has an effect only if no windows have been created.

**Set\_Character\_Size** — A two-element vector that changes the standard width and height of the vector-drawn fonts. The first element in the vector contains the new character width, and the second element contains the height. By default, character size is set to the size of the default hardware font.

**Set\_Fontmap** — Associates specific font characteristics with a particular hardware font command. This keyword can be used to specify a single string containing the information for one font, or an array of strings containing the information for multiple fonts.

For example, to associate 16 point Helvetica italic with font command !5, use the following DEVICE call:

```
DEVICE, Set_Fontmap='5 Helvetica,16,italics'
```

Now, whenever the !5 font command appears in a text string, the text output appears in 16 point Helvetica italic. This keyword only affects hardware fonts (that is, when !P.Font=0).

---

**NOTE** A default font mapping is defined in a configuration file that is read when the WIN32 driver is initialized. For information on this configuration file, see the *Fonts* chapter in the *PV-WAVE User's Guide*.

---

***Set\_Graphics\_Function*** — Windows allows applications to specify the *Graphics Function*. This is a logical function which specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen. The complete list of possible values is given in the following table. The default graphics function is *GXcopy*, which causes new pixels to completely overwrite any previous pixels.

Graphics Functions

Logical Function	Code	Definition	Windows ROP Code
GXclear	0	0	R2_BLACK
GXand	1	src AND dst	R2_MASKPEN
GXandReverse	2	src AND (NOT dst)	R2_MASKPENNOT
GXcopy	3	src	R2_COPYPEN
GXandInverted	4	(NOT src) AND dst	R2_MASKNOTOPEN
GXnoop	5	dst	R2_NOP
GXxor	6	src XOR dst	R2_XORPEN
GXor	7	src OR dst	R2_MERGEPEN
GXnor	8	(NOT src) AND (NOT dst)	R2_NOTMASKPEN
GXequiv	9	(NOT src) XOR dst	R2_NOTXORPEN
GXinvert	10	(NOT dst)	R2_NOT
GXorReverse	11	src OR (NOT dst)	R2_MERGEPENNOT
GXcopyInverted	12	(NOT src)	R2_NOTCOPYPEN
GXorInverted	13	(NOT src) OR dst	R2_MERGENOTPEN
GXnand	14	(NOT src) OR (NOT dst)	R2_NOTMERGEPEN
GXset	15	1	R2_WHITE

---

**Set\_Write\_Mask** — Sets the write mask to the specified value. Under Windows, the write mask can range from 0 to 255. See also *Using the Write Mask and Graphics Functions to Manipulate Color* on page B-82.

**True\_Color** — When specified with a *Depth* value, selects the TrueColor visual with *Depth* bits per pixel. For example: `True_Color=24`

The *True\_Color* keyword has an effect only if no windows have been created. The *Direct\_Color* and *True\_Color* keywords are essentially synonymous when used with the WIN32 driver: they perform identical functions.

**Window\_State** — Returns an array containing values indicating the status for all available PV-WAVE windows. Each element of the array contains a value equal to the sum of the following values:

- 0 Window closed
- 1 Window active on display
- 2 Backing store active
- 4 Metafile active

In the following example, the fourth element of `winarray` contains the value 3, or 1 + 2. This indicates that the window is active and backing store is active.

```
WINDOW, 3
      ; Open window 3.

DEVICE, Window_State=winarray
PRINT, winarray
```

```
0      0      0      3      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0      0      0      0      0
0      0      0      0      0
```

## Resizing Graphics

Prior to Version 6.0 of PV-WAVE for Microsoft Windows, it was necessary to use special keywords so that displayed graphics would resize whenever the user resized a window using its border. This is no longer the case. Window resizing is now the default behavior and is automatically provided for all normal PV-WAVE graphics windows.

## **Windows Metafile Limitations**

A Windows metafile is used to retain graphics in a window. The metafile is “replayed” into the graphic window whenever the window is resized. Metafiles are also used when you print the window. If you expect that your graphics might be subject to resizing, you should be aware of two limitations inherent in metafiles.

### **Resolution Dependency for Plots**

Metafiles are limited by the resolution of the graphics window in which you initially constructed a plot. So if you draw a plot to a small (low-resolution) window and then resize that window to full screen, you will likely see “aliasing” effects due to the low resolution of the original plot. For example, you might see “stairsteps” in diagonal lines or squared-off corners in small arcs or circles. You will see the same kinds of effects if you print the plot to a printer that has a higher resolution than your graphics window.

You can solve this problem by plotting to a high-resolution (large) window when you intend to make high-resolution plots. If you do not wish to view this large window (or you need to make it larger than your display) then you can use the */Pixmap* keyword with the WINDOW command to make the window invisible.

### **Disappearing “Dots”**

Another problem with metafiles is that they cannot reproduce a “dot” (a single-pixel point). So, if you use, for example,  $P_{sym}=3$  (a dot) in a PLOT command, it will appear on the screen, but when you resize the window or print the graph, the dot will disappear. This is because these pixels are treated differently by the metafile. (Note that this problem does not affect images, such as those you might display with the TV procedure, for example.)

To solve the disappearing-dot problem, you can turn off metafiles by using the */NoMeta* keyword with the WINDOW command. This results in a bitmap (DIB file) being used for repainting and printing. Since bitmaps are not resizable, pixels can’t be lost. But keep in mind that when the graphic is printed, it will be scaled to fit the printed region, so you will see effects such as wider lines and possibly boxes in place of dots. This problem can also be solved by using a higher resolution window for the initial plot.

## **Graphics Window Commands**

The following commands operate upon the contents of graphics windows. Most of these commands allow for the interchange of graphics between PV-WAVE and other graphics applications.

For detailed information on these commands, see their individual descriptions in this *Reference*.

- **WCOPY** — Copies the contents of a graphics window onto the Clipboard.
- **WPASTE** — Pastes the contents of the Clipboard into a graphics window.
- **WREAD\_DIB** — Loads a Device Independent Bitmap (DIB) from a file directly into a specified graphics window.
- **WREAD\_META** — Loads a Windows enhanced metafile (EMF) into a specified graphics window. Note that EMF graphics are designed to be used only with 32-bit applications; they cannot be pasted into a 16-bit application.
- **WWRITE\_DIB** — Saves the contents of a graphics window into a file as a Device Independent Bitmap (DIB).
- **WWRITE\_META** — Saves the contents of a graphics window into a file as a Windows enhanced metafile (EMF). Note that EMF graphics are designed to be used only with 32-bit applications; they cannot be pasted into a 16-bit application.
- **WPRINT** — Prints the contents of the specified graphics window.

---

**TIP** You can also use the graphics window Control menu to import, export, and print graphics.

---

## Use of Color in the WIN32 Driver

This section discusses how color tables and color palettes are handled by the WIN32 driver. For general information on color tables and how PV-WAVE handles color, see the *PV-WAVE Programmer's Guide*.

---

**NOTE** The color model used by the WIN32 driver in PV-WAVE Version 6.0 has changed from previous releases of PV-WAVE. Earlier releases, specifically version 4.2 and PV-WAVE Personal Edition, used a color model that was very similar to that used by the X Windows X11 driver. For PV-WAVE 6.0, changes to the WIN32 driver were made to support high-color Windows video drivers and required a simplification of the way color tables and palettes are handled.

---

### **Windows Video Modes**

Windows graphics output depends on a piece of software called a video driver that provides an interface between Windows Graphics Device Interface (GDI) calls and

the video hardware installed in the computer. This software is usually provided by the manufacturer of the video card. Depending on the capabilities of the hardware, video drivers usually support several video modes. A video mode is a combination of screen resolution, color depth and refresh rate.

Screen resolution is usually expressed in terms of the number of horizontal and vertical pixels displayed on the screen. Common resolutions are 640x480 (VGA), 800x600, 1024x768, and 1280x1024. Higher resolutions require more memory on the video card and a more capable monitor.

Color depth refers to the number of colors that can be displayed at once on the screen. This is usually one of 16, 256, 65536 (16-bit), or 16777216 (24-bit) and is directly related to the number of bits of video memory required to store the color information for a single pixel. In combination with the resolution, this determines what video modes are available for a given amount of memory on a video card. For instance, 1024x768 resolution with 256 colors requires 768KB of memory and so should be available on most video cards with at least 1 MB of video RAM while the same resolution with 24 bits of color information would require 2304KB and would not be available unless the video card has more than 2MB of video memory.

Refresh rate refers to how quickly the video card transfers information to the monitor and is not relevant to this discussion.

### ***PV-WAVE Color Model***

PV-WAVE inherently uses an 8-bit pseudo-color model. Color values are represented as an index into the current color table. The color table consists of three byte arrays that have 256 elements; the arrays represent the relative amounts of red, green and blue in a particular color. When a particular value is placed in a PV-WAVE graphics window (by the TV procedure, for instance) the graphics driver determines what color to display by using the value as an index into the color table arrays. When the color table changes (via the LOADCT procedure, for instance) the color represented by a particular value is likely to be different and, if so, will result in a new color being displayed in the graphics window.

In PV-WAVE's WIN32 graphics driver, the contents of a graphics window are stored in a DIB (Device Independent Bitmap) created using the DIB\_RGB\_COLORS format. This means that the window image is stored in an exact parallel to PV-WAVE's color model: an array of 8-bit values that act as indices into a color table which contains 8 bits each of red, green, and blue color information.

## ***Interaction Between PV-WAVE and the Video Driver***

When a graphics window is painted, the DIB section is transferred from normal memory to video memory where it appears on the screen. This operation is called a *blit*. As part of the blit operation, the video driver and Windows determine how to map the pixel information in the DIB into the current video mode. This is usually a highly optimized operation in the video driver.

The way a pixel value in the DIB gets mapped to a color on the video screen depends on the current color depth that the video driver is using. In general, Windows will map a given pixel color to the closest matching RGB value that the video driver can display. This is likely to be an unsatisfactory match for a driver in 16 color VGA mode, while a driver in 24-bit mode will be able to match any requested color exactly.

## ***256 Color Drivers and Palettes***

When Windows is using a video mode with 256 colors, pixel values in video memory represent an index into a palette (as opposed to directly representing the color as is the case in 16- and 24-bit modes). A palette serves exactly the same function as PV-WAVE's color table — it allows the selection of a subset of the 16 million colors that can be represented by the RGB triple.

The palette is controlled by the application that the user is currently using — the foreground application. With certain exceptions (discussed below), the foreground application can set the palette to represent whatever set of colors it wants. Background applications try to render their data as best they can given that the foreground application controls the palette; this often results in “flashing” as applications redraw their data when they move from background to foreground. When PV-WAVE is the foreground application, it directly maps the current color table to the Windows palette.

The exception is that Windows reserves the top and bottom 10 entries in the palette (for a total of 20 colors) for the system colors used to draw controls and window decorations. This results in less distraction for the user since only the contents of application windows will flash while window frames and controls will not.

## ***The PV-WAVE WIN32 Driver in 256 Color Mode***

The penalty for non-flashing window frames is that only 236 colors are available to PV-WAVE while it needs 256 to represent the color table. Earlier versions of the WIN32 driver used the color limiting mechanisms implemented to support the X driver to reduce the color table size to 236. The drawback to this approach is that it makes PV-WAVE's behavior dependent on the current video mode — the same

program will give different results with a different video mode. It also caused performance problems since it was necessary to continuously transform data transferred to and from the screen.

With PV-WAVE Version 6.0, the WIN32 driver always indicates that PV-WAVE has 256 colors available; if it needs to display in a palletized video mode, 20 colors evenly distributed through the color table are not mapped to the palette. The other 236 colors will be exactly represented but the 20 unmapped colors will map to the closest match available in the palette, either among the 236 PV-WAVE colors or the 20 system colors. For most of the PV-WAVE color tables, the closest match is immediately adjacent in the color table and the unmapped colors are very difficult to notice. Any discrepancy is purely visual; all PV-WAVE operations, including TV and TVRD, set and return the proper color and these operations return consistent results regardless of the current video model.

## Using Bitmaps

Windows can direct graphics to either windows or bitmaps:

- **Windows** — Windows are the usual windows that appear on the screen and contain graphics. Drawing to a window produces a viewable result.
- **Bitmaps** — Bitmaps are areas of invisible graphics memory. Drawing to a bitmap simply updates the bitmap memory.

Bitmaps are useful because it is possible to write graphics to a bitmap and then copy the contents of the bitmap to a window where it can be viewed. The process works by placing the desired images into bitmap memory and then copying them in succession to a visible window.

### *Creating a Bitmap*

To create a bitmap, use the *Bitmap* keyword with the WINDOW procedure. For example, to create a 1280-by-1280 bitmap in window 1:

```
WINDOW, 1, /Bitmap, Xsize=1280, Ysize=1280
```

Once they are created, bitmaps are treated just like normal windows, although some operations (WSHOW, for instance) don't have any noticeable effect when applied to a bitmap. Note also that the bitmap's resolution does not have to be restricted to the size of the display's resolution.

### *Example — Animating a Series of Bitmap Images*

The following example shows how animation can be performed using bitmap memory. It uses a series of 15 heart images taken from the file `abnorm.dat`. (This file is located in the `dat` subdirectory of the main PV-WAVE directory.) It

creates a bitmap and writes all 15 images to it. It then uses the *Copy* keyword of the *DEVICE* procedure to copy the images to a visible window. Pressing any key causes the animation to halt:

```
PRO animate_heart
    ; This procedure animates a series of heart data.
OPENR, u, !Data_Dir+'abnorm.dat',/Get_Lun
    ; Open the file containing the images.

frame = ASSOC(u, BYTARR(64, 64))
    ; Associate a file variable with the file. Each heart image is 64-by-64 pixels.

WINDOW, 0, /Bitmap, Xsize=7680, Ysize=512
    ; Window 0 is a bitmap large enough to hold one double-sized
    ; image tall and 15 double-sized images wide. The resized
    ; images will be placed in this bitmap.

FOR i = 0, 15-1 DO TVSCL, $
    REBIN(SMOOTH(frame(i), 3), 512, 512), i
        ; Write each image to bitmap memory. SMOOTH is used to
        ; improve the appearance of each image and REBIN is used to enlarge each image
        ; to the final display size.

FREE_LUN, u
    ; Close the image file and free the file unit.

WINDOW, 1, Xsize=512, Ysize=512, Title='Heart'
    ; Window 1 is a visible window that is used to display the
    ; animated data, creating the appearance of a beating heart.

i = 0L
    ; Load the current frame number.

WHILE GET_KBRD(0) EQ '' DO BEGIN
    DEVICE, Copy=[i * 512, 0, 512, 512, 0, 0, 0]
        ; Display frames until any key is pressed. Copy the next image
        ; from the bitmap to the visible window.
    i = (i + 1) MOD 15
        ; Keep track of total frame count.

ENDWHILE

END
```

## Using the Write Mask and Graphics Functions to Manipulate Color

The write mask can be used to superimpose (overlay) one graphics pattern over another when plotting to a graphics window, allowing you to create special effects.

Another possible application for the write mask is to simultaneously manage two 4-bit-deep images in a single graphics window instead of a single 8-bit-deep image. You could use the write mask to control whether the current graphics operation operates on the “top” image or the “bottom” image.

### ***Using Graphics Functions to Manipulate Color***

The WIN32 driver provides two keywords for inquiring and manipulating the graphics function — *Get\_Graphics\_Function* and *Set\_Graphics\_Function*.

The value of the *Set\_Graphics\_Function* keyword controls the logical graphics function; this function specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen. To see a complete list of graphics function codes, refer to the *Set\_Graphics\_Function* keyword description in the section *X Window System* on page B-58.

In the following example, POLYFILL is used to select the area to be inverted. Colors represented by color index 1 are XORed with the image currently displayed in that area.

```
DEVICE, Get_Graphics=oldg, Set_Graphics=6
    ; Set graphics function to exclusive or (GXxor), saving the old
    ; function.
POLYFILL, [[x0,y0], [x0,y1], [x1,y1], [x1,y0]], /Device, Color=1
DEVICE, Set_Graphics_Function=oldg
    ; Restore the previous graphics function.
```

The default value for the *Set\_Graphics\_Function* keyword is GXcopy, which means that the source graphics from the current operation get copied into the window, destroying any graphics that were previously displayed there.

### ***Interaction Between the Set\_Write\_Mask and the Set\_Graphics\_Function Keywords***

You use the *Set\_Write\_Mask* keyword to specify the planes whose bits you are manipulating or the plane you want to use for the special effects. The way the two graphics patterns are combined depends on the value you provide for the *Set\_Graphics\_Function* keyword. For example, you could use the following commands:

```
DEVICE, Set_Graphics_Function=6
DEVICE, Set_Write_Mask=8
```

to extract the fourth bit (the binary equivalent of the decimal value 8) of the image in the current graphics window. The extracted plane is XORed (XOR is the graphics function specified by setting the *Set\_Graphics\_Function* keyword equal to 6) with the source pattern (the result of the current graphics operation). After the graphics function is implemented, the result is drawn in the current graphics window using whichever color(s) in the color table match the resultant value(s).

### ***Interaction Between the Set\_Graphics\_Function Keyword and Colors***

The graphics functions specified by the *Set\_Graphics\_Function* keyword operate individual colors, not on hardware pixel values as they do with the X driver.

## **Window IDs**

The PV-WAVE WINDOW procedure has two keywords, *Get\_Win\_Id* and *Set\_Win\_Id* that provide access to the WIN32 window handle associated with a PV-WAVE graphics window.

*Get\_Win\_Id* returns the HWND that was returned from the WIN32 `CreateWindow` API call. If you are making PV-WAVE calls from another application via a connectivity mechanism, you can use this keyword to get the window handle and then use the handle as an argument to an API call like `HideWindow`. You can also use the handle if you need to subclass a PV-WAVE graphics window for any reason.

*Set\_Win\_Id* causes PV-WAVE to use a window that already exists as a graphics window. For instance, you might have created a window in a Visual Basic form in which you would like PV-WAVE to display graphics. You can do this by passing PV-WAVE the window handle of the existing window at runtime using the *Set\_Win\_Id* keyword. PV-WAVE will subclass the window referred to by the handle when the keyword is used with a WINDOW command. When the driver is closed or when PV-WAVE exits, subclassing is removed.

See the *WIN32 Programmer's Reference* for more information about window handles and subclassing.

---

## WMF Driver

The WMF driver creates Windows Enhanced Metafile output. The output can be saved either to a file or sent directly to a printer.

The WMF driver is a 24-bit device. For more information on 24-bit color and the WMF driver, see [Handling 24-bit Color](#) on page B-54.

---

**NOTE** If you are running a Windows NT system, WMF hardware fonts can be rotated and transformed by general 3D transforms in PV-WAVE.

---

### Selecting the WMF Driver

To enable WMF as the current graphics device, enter the following command:

```
SET_PLOT, 'WMF'
```

This causes PV-WAVE to use the WMF driver for producing subsequent graphics output. Once the WMF driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, as described in [Controlling the WMF Driver with DEVICE Keywords](#) below.

### Listing the Current Settings for the WMF Driver

Use the command:

```
INFO, /Device
```

to view the current WMF driver settings.

### Sending PV-WAVE Output Directly to a Printer

The following steps can be used to send Windows Metafile output directly to a Windows printer:

**Step 1** Select the WMF driver:

```
SET_PLOT, 'WMF'
```

**Step 2** Indicate output should be sent directly to the printer:

```
DEVICE, /Print
```

**Step 3** Select the printer to send output to (if other than the default printer):

```
printer = WIN32_PICK_PRINTER()
```

- or -

```
DEVICE, Set_Printer='printer_name'
```

**Step 4** Execute graphics commands (PLOT, SURFACE, XYOUTS, etc.)

**Step 5** Close the Windows Metafile and the output is sent to the printer:

```
DEVICE, /Close_File
```

## Handling 24-bit Color

The WMF driver is a 24-bit device. Colors specified with this device must conform to the PV-WAVE 24-bit color convention, which is a PV-WAVE long integer with the red, green, and blue color values occupying the lower three bytes, in order, from the least significant byte. For instance, color values less than 256 are considered to be shades of red. If the *Pseudo\_Color* keyword is set to 8, color values less than 256 are taken to be indices into the PV-WAVE color map. Colors greater than 256 are still interpreted as 24-bit colors.

Because the WMF driver is a 24-bit device, you need to specify application colors as RGB values in hexadecimal notation, rather than as an index into a color table.

To ensure that colors display correctly on all types of devices, we recommend that you use the `WoColorConvert` function whenever you specify a color index in your code.

`WoColorConvert` returns the RGB color value on 24-bit devices and the color table index on 8-bit devices. By using `WoColorConvert` with all color index values in your code, you ensure that your application colors will appear properly on all types of devices.

For example:

```
TEK_COLOR
```

```
PLOT, DIST(10), Color=WoColorConvert(5)
```

If your code might be used with 8 and 24-bit displays, and/or the WMF driver, we recommend you use the `WoColorConvert` function to ensure that colors display correctly.

## Additional Text Formatting Commands

The following text formatting commands are new and can be used with the PostScript, WIN32, WMF, and X drivers. These commands can only be used when hardware fonts are enabled (!P.Font=0).

Formatting Command	Description
!FB	Switch to the bold face of the current font.
!FI	Switch to the italic face of the current font.
!FU	Underline the current font.
!FN	Switch to the normal form of the current font.
!Pxx	Switch to point size <i>xx</i> of the current font, where <i>xx</i> is a two digit integer (01-99).

Refer to the *PV-WAVE User's Guide* for a comprehensive list of text formatting commands.

## Controlling the WMF Driver with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the WMF driver:

**Close\_File** — Closes the currently open Windows Metafile. If the *Print* keyword was specified, the file will be sent directly to the selected printer.

**Direct\_Color** — If set to 24, the WMF driver interprets all colors as 24-bit. For the WMF driver, this keyword is synonymous with the *True\_Color* keyword. (Example: `Direct_Color=24`)

**Filename** — Specifies the name of a file to save the Windows Metafile output in. This keyword must be used prior to any graphics commands. If a graphics command is used and neither the *Print* nor *Filename* keyword has been specified, a Windows File Selector will be displayed.

**Font** — Specifies the name of the font used when the hardware font is selected (that is, when !P.Font=0). For example, to select the font 14 point New Times Roman bold, use the following DEVICE call:

```
DEVICE, Font='Times New Roman, 14, bold'
```

---

**TIP** To avoid typing the font name, use the [WIN32\\_PICK\\_FONT](#) function to select a font interactively from a dialog box. For example, if you enter this command:

```
WAVE> DEVICE, Font=WIN32_PICK_FONT()
```

the Font dialog box appears. Pick the font and font characteristics that you wish to use, and click OK. The font that you select is automatically set by the DEVICE command.

---

**Get\_Fontmap** — Returns a list of the current font map settings. If */Get\_Fontmap* is specified, the information is printed to the screen. If a variable name is specified (e.g., *Get\_Fontmap=varname*), a string array is returned.

The information returned by *Get\_Fontmap* is in the same form as the strings specified with *Set\_Fontmap*. The only exception to this pattern is that the first string returned in a string array is labeled as font 0 and is the current default font.

**Get\_Graphics\_Function** — See the description of the *Get\_Graphics\_Function* keyword in [X Window System](#) on page B-58.

**Get\_Write\_Mask** — See the description of the *Get\_Write\_Mask* keyword in [X Window System](#) on page B-58.

**Inches** — By default, the *Xoffset*, *Xsize*, *Yoffset*, and *Ysize* keywords are specified in centimeters. However, if *Inches* is present and nonzero, these keywords are taken to be in inches instead.

**Landscape** — PV-WAVE normally generates plots with portrait orientation (the *x*-axis is along the short dimension of the page). If *Landscape* is present, landscape orientation (the *x*-axis is along the long dimension of the page) is used instead. This keyword only works if */Print* is used.

**Max\_Lines** — Sets the maximum number of identical line segments that will be cached before forcing a screen redraw. The WIN32 driver uses an internal cache for drawing line segments. Instead of drawing each line individually, lines with common characteristics are drawn as a group. This feature increases drawing efficiency and rendering speed. To disable *Max\_Lines*, set the keyword equal to one. (Default: 500 lines.)

**Max\_Pens** — Sets the maximum number of pens that will be cached before forcing a screen redraw. (Default: 10 pens, each of which can hold 500 line segments.) To disable *Max\_Pens*, set the keyword equal to one.

---

**NOTE** The line caching feature may affect some PV-WAVE behavior that is familiar to you. In general, this applies to any operation that depends on the order in which lines are drawn. For example, using OPLOTT with the color set to the background color to erase drawn lines may not work as expected. This is because some or all of the second set of lines may be drawn before the first set they were intended to overwrite. To avoid this, use the EMPTY procedure to force the driver to empty the line cache first.

---

**Portrait** — If *Portrait* is present, PV-WAVE generates plots using portrait orientation, the default. This keyword only works if */Print* is used.

**Print** — If present and non-zero, specifies that output should be sent directly to the selected Windows printer when the Windows Metafile is closed. As described in [Sending PV-WAVE Output Directly to a Printer](#) on page B-53, either the *Set\_Printer* keyword or the WIN32\_PICK\_PRINTER function can be used to select a destination printer. If neither is used, the default Windows printer is used. The *Print* keyword must be specified before any output is made to the Windows Metafile.

**Pseudo\_Color** — If set to 8, the WMF driver interprets 8-bit color values (that is, color values less than 256) as color map indices rather than as 24-bit colors. (Example: *Pseudo\_Color=8*)

**Set\_Character\_Size** — A two-element vector that changes the standard width and height of the vector-drawn fonts. The first element in the vector contains the new character width, and the second element contains the height. By default, character size is set to the size of the default hardware font.

**Set\_Fontmap** — Associates specific font characteristics with a particular hardware font command. This keyword can be used to specify a single string containing the information for one font, or an array of strings containing the information for multiple fonts.

For example, to associate 16 point Helvetica italic with font command !5, use the following DEVICE call:

```
DEVICE, Set_Fontmap=' 5 Helvetica,16,italics'
```

Now, whenever the !5 font command appears in a text string, the text output appears in 16 point Helvetica italic. This keyword only affects hardware fonts (that is, when !P.Font=0).

---

**NOTE** A default font mapping is defined in a configuration file that is read when the WMF driver is initialized. For information on this configuration file, see the *Fonts* chapter in the *PV-WAVE User's Guide*.

---

***Set\_Graphics\_Function*** — See the description of the *Set\_Graphics\_Function* keyword in *X Window System* on page B-58.

***Set\_Write\_Mask*** — See the description of the *Set\_Write\_Mask* keyword in *X Window System* on page B-58.

Under Windows, the write mask can range from 0 to 255.

***Thickness*** — Specifies the thickness (in millimeters) of lines drawn into the Windows Metafile. The default thickness is 2.

***True\_Color*** — If set to 24, the WMF driver interprets all colors as 24-bit. For the WMF driver, this keyword is synonymous with the *Direct\_Color* keyword. (Example: *True\_Color=24*)

***Xoffset*** — Specifies the *x* position on the page of the lower-left corner of output. *Xoffset* is specified in centimeters unless *Inches* is specified. This keyword only works if */Print* is used.

***Xsize*** — Specifies the width of output PV-WAVE generates. *Xsize* is specified in centimeters unless *Inches* is specified.

***Yoffset*** — Specifies the *y* position on the page of the lower-left corner of output generated by PV-WAVE. *Yoffset* is specified in centimeters unless *Inches* is specified. This keyword only works if */Print* is used.

***Ysize*** — Specifies the height of output generated by PV-WAVE. *Ysize* is specified in centimeters unless *Inches* is specified.

---

## ***X Window System***

PV-WAVE uses the X Window System to provide an environment in which you can create one or more independent windows, each of which can be used for the display of graphics and/or images.

The X Window System is the default windowing system for UNIX and OpenVMS PV-WAVE platforms.

In X there are two basic cooperating processes, clients and servers. A server usually consists of a display, keyboard, and pointer (such as a mouse) as well as the software that controls them. Client processes (such as PV-WAVE) display graphics

and text on the screen of a server by sending X protocol requests across the network to the server. Although in the simplest case, the server and client reside on the same machine, this network-based design allows more elaborate configurations.

---

**NOTE** If you are running an X11R6 system, X Windows fonts can be rotated and transformed by general 3D transforms in PV-WAVE.

---

## Controlling Where Graphics are Displayed

When you use X, the environment variable `$DISPLAY` (UNIX) or the logical `DECW$DISPLAY` (DECwindows Motif) must be set properly. Otherwise, PV-WAVE may appear to “hang”. The following command allows PV-WAVE windows to appear on the local display (the current workstation). Enter one of these commands at the prompt of the current workstation before you start PV-WAVE, depending on whether you are using UNIX or OpenVMS (DECwindows Motif):

**(UNIX)** `setenv DISPLAY nodename:0.0`

**(OpenVMS)** `SET DISPLAY /CREATE /NODE=nodename -  
/SCREEN=0.0 /TRANSPORT=transport_type`

You must also be sure that an X server is running on the host machine specified with the `$DISPLAY` environment variable or `DECW$DISPLAY` logical, and that the machine PV-WAVE is running on has permission to communicate with that X server. Refer to the documentation for your X-compatible window manager to learn how to modify your X server’s permissions list.

### Selecting the X Driver

To use X as the current graphics device, enter the following command:

```
SET_PLOT, 'X'
```

This causes PV-WAVE to use X for producing subsequent graphical output. Once the X driver is enabled via `SET_PLOT`, the `DEVICE` procedure is used to control its actions, as described in [Controlling the X Driver with DEVICE Keywords](#) on page B-61.

### Listing the Current Settings for the X Driver

Use the command:

```
INFO, /Device
```

to view the current X driver settings.

## Graphical User Interfaces (GUIs) for PV-WAVE Applications Running Under X

If you wish to include a graphical user interface (GUI) with a PV-WAVE application that you are writing for use with the X Window System, you have several choices:

- **WAVE Widgets** — A versatile, easy-to-use set of functions for creating Motif GUIs for PV-WAVE applications. WAVE Widgets are designed for PV-WAVE developers with little or no experience using the Motif GUI toolkits. See the *PV-WAVE Application Developer's Guide* for detailed information on the Widget Toolbox functions.
- **Widget Toolbox** — A set of highly flexible PV-WAVE functions used to create Motif Graphical User Interfaces (GUIs) for PV-WAVE applications. The Widget Toolbox functions call Motif routines directly, and are designed primarily for developers who are already experienced using either Motif. See the *PV-WAVE Application Developer's Guide* for detailed information on the Widget Toolbox functions.
- **C-based Applications** — PV-WAVE can be used to add visual data analysis capability to an existing C application. The application interface can be developed in C, and, via interapplication communication functions, the C application can call PV-WAVE to perform data processing and display functions.

---

**UNIX and OpenVMS USERS** For more details on interapplication communication, refer to the *PV-WAVE Application Developer's Guide*.

---

**NOTE** All the options listed above are fully compatible with the X Window System and can be used to facilitate access to your application. However, you are not required to use any of them — your application can still run under X, even though it does not have a Motif GUI.

---

### Additional Text Formatting Commands

The following text formatting commands are new and can be used with the PostScript, WIN32, WMF, and X drivers. These commands can only be used when hardware fonts are enabled (`!P.Font=0`).

Formatting Command	Description
!FB	Switch to the bold face of the current font.
!FI	Switch to the italic face of the current font.
!FU	Underline the current font.
!FN	Switch to the normal form of the current font.
!Pxx	Switch to point size <i>xx</i> of the current font, where <i>xx</i> is a two digit integer (01-99).

Refer to the *PV-WAVE User's Guide* for a comprehensive list of text formatting commands.

## Controlling the X Driver with DEVICE Keywords

The following keywords to the DEVICE procedure provide control over the X driver:

***Bypass\_Translation*** — When this keyword is set, the translation table is bypassed and color indices can be directly specified. Pixel values read via the TVRD function are not translated if this keyword is set, and thus the result contains the actual (hardware) pixel values present in the display. By default, the translation table is used with shared color tables. When displays with static (read-only) visual classes and with private color tables are used, the translation table is *always* bypassed. For more information about the translation table, refer to [Color Translation Table](#) on page B-74.

***Close\_Display*** — Causes PV-WAVE to sever the connection with the X server. This has the effect of deleting all generated windows from the screen of the server, and returns PV-WAVE to the initial graphics state. One use for this option is to change the number of colors used. See the section, [When Color Characteristics are Determined](#) on page B-73, for details.

***Copy*** — Copies a rectangular area of pixels from one region of a window to another. The argument of *Copy* is a six- or seven-element array: [ $X_s$ ,  $Y_s$ ,  $N_x$ ,  $N_y$ ,  $X_d$ ,  $Y_d$ ,  $W$ ], where: ( $X_s$ ,  $Y_s$ ) is the lower-left corner of the source rectangle, ( $N_x$ ,  $N_y$ ) are the number of columns and rows in the rectangle, and ( $X_d$ ,  $Y_d$ ) is the coordinate of the destination rectangle. Optionally,  $W$  is the index of the window from which the pixels should be copied to the current window. If it is not supplied, the current window is used for both source and destination. *Copy* can be used to increase the

pace of animation. This is described in *Using Pixmaps to Improve Application Performance* on page B-75.

***Cursor\_Crosshair*** — Selects the crosshair cursor type. The crosshair cursor style is the default.

***Cursor\_Image*** — Specifies the cursor pattern. The value of this keyword must be a 16-line by 16-column bitmap, contained in a 16-element short integer vector. Each line of the bitmap must be a 16-bit pattern of ones and zeros, where one (1) is white and zero (0) is black. The offset from the upper-left pixel to the point that is considered the “hot spot” can be provided via the *Cursor\_XY* keyword.

***Cursor\_Original*** — Selects the default cursor for the window system. Under X, it is the cursor in use by the window manager when PV-WAVE starts.

***Cursor\_Standard*** — Selects one of the predefined cursors provided by X. The available cursor shapes are defined in the file:

(UNIX) `/usr/include/X11/cursorfont.h`

(OpenVMS) `DECW$INCLUDE:CURSORFONT.H`

In order to use one of these cursors, select the number of the font and provide it as the value of the *Cursor\_Standard* keyword. For example, the `cursorfont.h` file gives the value of *Xc\_Cross* as being 30. In order to make that the current cursor, use the statement:

```
DEVICE, Cursor_Standard=30
```

***Cursor\_XY*** — A two-element integer vector giving the *x*, *y* pixel offset of the cursor “hot spot”, the point which is considered to be the mouse position, from the upper-left corner of the cursor image. This parameter is only applicable if *Cursor\_Image* is provided. The cursor image is displayed top-down — in other words, the first row is displayed at the top.

***Direct\_Color*** — When specified with a *Depth* value, selects the DirectColor visual with *Depth* bits per pixel. For example:

```
Direct_Color=12
```

The *Direct\_Color* keyword has effect only if no windows have been created.

***Floyd*** — If present and nonzero, selects the Floyd-Steinberg dithering method. For more information on this dithering method, the *PV-WAVE User's Guide*.

***Font*** — Specifies the name of the font used when the hardware font is selected. For example, to select the font 8X13:

```
DEVICE, Font='8X13'
```

Usually, the window system provides a directory of font files that can be used by all applications. List the contents of that directory to find the fonts available on your system. The size of the font selected also affects the size of software-drawn text (e.g., the Hershey fonts). On some machines, fonts are kept in subdirectories of:

**(UNIX)**        `/usr/lib/X11/fonts`

**(OpenVMS)** `SYS$SYSROOT: [SYSCOMMON.SYSFONT]`

***Get\_Fontmap*** — Returns a list of the current font map settings. If `/Get_Fontmap` is specified, the information is printed to the screen. If a variable name is specified (e.g., `Get_Fontmap=varname`), a string array is returned.

The information returned by `Get_Fontmap` is in the same form as the strings specified with `Set_Fontmap`. The only exception to this pattern is that the first string returned in a string array is labeled as font 0 and is the current default font.

***Get\_Graphics\_Function*** — Returns the value of the current graphics function (set with the `Set_Graphics_Function` keyword). This can be used to remember the current graphics function, change it temporarily, and then restore it. For an example, see the example in the section [Using Graphics Functions to Manipulate Color](#) on page B-83.

***Get\_Visuals*** — Displays the list of currently available visual classes. If a named variable is specified with this keyword, the information is stored in a 2D array of long values. The size of first dimension is 8, and the index is defined in the following table:

<b>Index</b>	<b>Description</b>
0	Visual ID
1	Visual class
2	Depth
3	Size of colormap
4	Red mask
5	Green mask
6	Blue mask
7	Significant bits in the specification

The value of visual class item above is shown in the following table:

Visual Class	Value
StaticGray	0
GrayScale	1
StaticColor	2
PseudoColor	3
TrueColor	4
DirectColor	5

The second dimension of the 2D array represents the number of available visual classes.

**Get\_Window\_Position** — Places the *x* and *y* device coordinates of the window's lower-left corner into a named variable.

**Get\_Write\_Mask** — Specifies the name of a variable to receive the current value of the write mask. For example:

```
Get_Write_Mask=mask
```

For more information on the write mask, refer to [Using the Write Mask and Graphics Functions to Manipulate Color](#) on page B-82.

**List\_Fonts** — Returns a list of the available X fonts. If */List\_Fonts* is specified, the information is printed to the screen. If a variable name is specified (e.g., *List\_Fonts=varname*), a string array is returned. This keyword works like the UNIX command `xlsfonts`. If the *Pattern\_Font* keyword is used to specify a pattern, that pattern is used to filter the list of fonts returned by *List\_Fonts*; otherwise, *List\_Fonts* returns all available fonts.

**Ordered** — If present and nonzero, selects the Ordered method of dithering. For more information on dithering, see the *PV-WAVE User's Guide*.

**Pattern\_Font** — Specifies a pattern that is used to filter the fonts that are returned by the *List\_Font* keyword. For example, to list all the fonts in the *adobe* foundry, enter this command:

```
DEVICE, /List_Fonts, Pattern='-adobe-*
```

---

**NOTE** X11R5 systems support scalable fonts. You can return the names of all scalable fonts by specifying a complete font name (one with all 14 fields separated by dashes) with a zero (0) in fields 7, 8, and 12 (the fields for pixel size, point size, and average width). To list all scalable fonts, enter the command:

```
DEVICE, /List_Fonts, Pattern_Font='-***-***-***-0-0-***-***-0-***'
```

---

**NOTE** X11R6 systems allow fonts to be rotated and transformed by a general 3D transform. (This is accomplished using the X Logical Font Description (XLFD) extensions and does not include perspective.) You can return the names of all fonts that support 3D transformation by specifying a matrix in either the pixel size or point size fields (fields 7 or 8). To list all fonts capable of 3D transformation, enter the command:

```
DEVICE, /List_Fonts, $
    Pattern_Font='-***-***-***-[1 0 0 1]-***-***-***'
```

In this case, the matrix is specified in the 7th field. All other fields contain the wildcard symbol (\*).

---

***Pseudo\_Color*** — When specified with a *Depth* value, selects the PseudoColor visual with *Depth* bits per pixel. For example:

```
Pseudo_Color=8
```

The *Pseudo\_Color* keyword has effect only if no windows have been created.

***Retain*** — Specifies the default method used for backing store when creating new windows. This is the method used when the *Retain* keyword is not specified with the WINDOW procedure. Backing store is discussed in more detail in the subsection, *How Is Backing Store Handled?* on page B-2. The possible values for this keyword are summarized in on page B-3.

If *Retain* is not used to specify the default method, method 1 (server-supplied backing store) is used.

***Set\_Character\_Size*** — A two-element vector that changes the standard width and height of the vector-drawn fonts. The first element in the vector contains the new character width, and the second element contains the height. By default, character size is set to the size of the default hardware font.

***Set\_Fontmap*** — Associates specific font characteristics with a particular hardware font command. This keyword can be used to specify a single string containing the information for one font, or an array of strings containing the information for multiple fonts.

For example, to associate 14 point Times bold with font command !5, use the following DEVICE call and the complete X driver font name:

```
Set_Fontmap=' 5 -adobe-times-bold-r-normal-*.14-*.**.*.*.*.*'
```

Now, whenever the !5 font command appears in a text string, the text output appears in 14 point Times bold. This keyword only affects hardware fonts (that is, when !P.Font=0).

---

**NOTE** A default font mapping is defined in a configuration file that is read when the X driver is initialized. For information on this configuration file, see the *Fonts* chapter in the *PV-WAVE User's Guide*.

---

**Set\_Graphics\_Function** — X allows applications to specify the *Graphics Function*. This is a logical function which specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen. The complete list of possible values is given in the following table. The default graphics function is GXcopy, which causes new pixels to completely overwrite any previous pixels.

#### Logical Graphics Functions and Values

Logical Function	Code	Definition	Windows ROP Code
GXclear	0	0	R2_BLACK
GXand	1	src AND dst	R2_MASKPEN
GXandReverse	2	src AND (NOT dst)	R2_MASKPENNOT
GXcopy	3	src	R2_COPYPEN
GXandInverted	4	(NOT src) AND dst	R2_MASKNOTOPEN
GXnoop	5	dst	R2_NOP
GXxor	6	src XOR dst	R2_XORPEN
GXor	7	src OR dst	R2_MERGEPEN
GXnor	8	(NOT src) AND (NOT dst)	R2_NOTMASKPEN
GXequiv	9	(NOT src) XOR dst	R2_NOTXORPEN
GXinvert	10	(NOT dst)	R2_NOT
GXorReverse	11	src OR (NOT dst)	R2_MERGEPENNOT
GXcopyInverted	12	(NOT src)	R2_NOTCOPYPEN
GXorInverted	13	(NOT src) OR dst	R2_MERGENOTPEN
GXnand	14	(NOT src) OR (NOT dst)	R2_NOTMERGEPEN
GXset	15	1	R2_WHITE

**Set\_Write\_Mask** — Sets the write mask to the specified value. For an  $n$ -bit system, the write mask can range from 0 to  $2^n - 1$ . For more information on the write mask, refer to [Using the Write Mask and Graphics Functions to Manipulate Color](#) on page B-82.

**Static\_Color** — When specified with a *Depth* value, selects the StaticColor visual with *Depth* bits per pixel. This keyword has effect only if no windows have been created.

**Static\_Gray** — When specified with a *Depth* value, selects the StaticGray visual with *Depth* bits per pixel. This keyword has effect only if no windows have been created.

**Threshold** — Specifies use of the threshold dithering algorithm — the simplest dithering method. For more information on this dithering method, see *PV-WAVE User's Guide*.

**Translation** — Using the shared colormap (which is normally recommended) causes PV-WAVE to translate between color indices (which always start with zero and are contiguous) and the pixel values actually present in the display. The *Translation* keyword specifies the name of a variable to receive the translation vector. To read the translation table:

```
DEVICE, Translation=Transarr
```

The result is a 256-element byte vector, `Transarr`. Element zero of `Transarr` contains the pixel value allocated for the first color in the colormap, and so forth.

For more information on the translation table, refer to [Color Translation Table](#) on page B-74.

**True\_Color** — When specified with a *Depth* value, selects the TrueColor visual with *Depth* bits per pixel. For example:

```
True_Color=12
```

The *True\_Color* keyword has effect only if no windows have been created.

**VisualID** — If specified with the visual class ID number, selects the visual class with visual class with given ID number.

---

**NOTE** To acquire visual class ID numbers use the X library client program `xdp-info`, or use the command `DEVICE, /Get_Visuals`.

---

**Window\_State** — Returns an array containing values indicating the status (open = 1, closed = 0) for all available PV-WAVE windows. For example:

```
WINDOW, 3
```

; Open window 3.

```
DEVICE, Window_State=winarray  
PRINT, winarray
```

## X Window Visuals

Visuals specify how the hardware deals with color. The X server of your display may provide colors or only gray scale (black and white), or both. The color tables may be changeable from within PV-WAVE (read-write), or they may be static (read-only). The value of each pixel may be mapped to any color (Undecomposed Colormap), or certain bits of each pixel are dedicated to the red, green, and blue primary colors (Decomposed Colormap).

The X server provides six visual classes — read-write and read-only visuals for three types of displays: Gray Scale, Undecomposed Color, and Decomposed Color. The names of the visual classes are listed in the following table:

### X Window System Visual Classes

Visual Class Name	Writable	Description
StaticGray	no	Gray scale
GrayScale	yes	Gray scale
StaticColor	no	Undecomposed color
PseudoColor	yes	Undecomposed color
TrueColor	no	Decomposed color
DirectColor	yes	Decomposed color

PV-WAVE supports all six types of visual classes, although not at all possible depths (e.g., 4-bit, 8-bit, 24-bit).

---

**NOTE** For more information on the differences between pseudo color and 24-bit (“true”) color, see *PV-WAVE User’s Guide*.

---

X servers from different manufacturers will each have a default visual class. Many servers may provide multiple visual classes. For example, a server with display hardware that supports an 8-bit deep, undecomposed, writable color map (Pseudo-

Color), may also easily support StaticColor, StaticGray, and GrayScale visual classes.

---

**TIP** For more detailed information about X visual classes, refer to *Volume 1 of the Xlib Programming Manual, Second Edition*, O'Reilly & Associates, Inc., Sebastopol, CA, 1990.

---

### **Selecting a Visual Class**

When opening the display, PV-WAVE asks the display for the following visual classes, in order, until a supported visual class is found:

1. DirectColor, 24-bit
2. TrueColor, 24-bit
3. PseudoColor, 8-bit, then 4-bit
4. StaticColor, 8-bit, then 4-bit
5. GrayScale, any depth
6. StaticGray, any depth

You can override this default choice by using the DEVICE procedure to specify the desired visual class and depth before you create a window.

For example, if you are using a display that supports both the 24-bit deep DirectColor visual class, and an 8-bit deep PseudoColor visual class, PV-WAVE will select the 24-bit deep DirectColor visual class. To use PseudoColor, enter the following command before creating a window:

```
DEVICE, Pseudo_Color=8
```

---

**NOTE** If a visual type and depth is specified, using the DEVICE procedure, and it does not match the default visual class of the screen, a new colormap is created.

---

## **Colormapped Graphics**

Colormaps define the mapping from color index to screen pixel value. In this discussion, the term *colormap* is used interchangeably with the terms *color table*, *color translation table*, or *color lookup table* — other terminology that you may be familiar with from working with other systems. Colormaps perform a slightly different role on 8-bit workstations than they do on 24-bit workstations.

## **8-bit Graphics Primer**

On an 8-bit workstation, the screen pixel value in video memory “looks up” the red-green-blue color combination in its corresponding color table index (hence, the term *color lookup table*). For example, a pixel value of 43 looks in color table location 43 and may find Red=255, Green=0, and Blue=255. The three values at that colortable location tell the red, green, and blue electron guns in the CRT what intensities to display for that pixel. In this example, any pixel with a value of 43 will be displayed as magenta (100% red, 0% green, 100% blue).

## **24-bit Graphics Primer**

On a 24-bit workstation, each pixel on the screen can be displayed in any one of a possible 16.7 million ( $2^{24}$ ) colors. The video memory on the machine is capable of addressing each pixel on the screen with a 24-bit value, “decomposed” to eight bits each for the red, green, and blue intensity values for that pixel.

Since each pixel in video memory directly references a set of three 8-bit red-green-blue intensities, there is no need for a color lookup table as in an 8-bit system. However, because PV-WAVE is an 8-bit colortable-based software package (instead of a “true” 24-bit software package), it performs similar conversions internally when drawing to a 24-bit display.

Some 24-bit displays allow the screen to be treated as two separate 12-bit Pseudo-Color visuals. This allows for “double-buffering”, a technique useful for animation, or for storing distance data to simplify hidden line and plane calculations in 3D applications.

Although a 24-bit display takes up three times as much video memory as an 8-bit system, the number of concurrent colors allowed on the screen is not limited by memory — the number is limited only by the number of pixels on the screen (assuming, of course, that the screen contains less than 16.7 million pixels). For example, a 24-bit X Window display with 1280-by-1024 pixel resolution contains 1,310,720 pixels, and thus it can potentially display that many colors on the screen at one time.

---

**NOTE** To see a formula for calculating how many colors a display is capable of displaying, given the number of bits it has for describing each pixel, refer to the *PV-WAVE User’s Guide*.

---

For more information about how colors are represented on 24-bit displays, refer to [Understanding 24-bit Graphics Displays](#) on page [B-79](#).

## How PV-WAVE Allocates the Colormap

Many factors affect how PV-WAVE chooses the type of colormap. For example, the keywords you supply with the DEVICE procedure control the way color is used in graphics windows throughout that session.

PV-WAVE colormaps can be either shared or private, and either read-write or read-only:

- Colormaps may be private or shared. This characteristic is determined by the number and type of application(s) running during your session.
- Colormaps may be static (read-only) or writable. This characteristic is controlled by the visual class that was invoked at the time the X server was started (or restarted).

For more information about how to select PV-WAVE's visual class, refer to [Selecting a Visual Class](#) on page B-69.

### **Shared Colormaps**

The window manager creates a colormap when it is started. This is known as the default colormap, and it can be shared by all applications using the display. When any application requires a colormap entry, it can allocate one from this *shared colormap*.

### **Advantages**

Using the shared colormap ensures that all applications share the available colors without conflict, and all color indices are available to every application. No application is allowed to change a color that is allocated to another application.

In other words, PV-WAVE can change the colors it has allocated without changing the colors that have been allocated by the window manager or other applications that are running.

### **Disadvantages**

On the other hand, using a shared colormap can involve the following disadvantages:

- The window system's interface routines must translate between internal pixel values and the values required by the server, significantly slowing the transfer of images.
- The number of available colors in the shared colormap depends on the window manager being used and the demands of other applications. Thus, the number

of available colors can vary, and the shared colormap might not always have enough colors available to perform the desired PV-WAVE operations.

- The allocated colors in a shared colormap do not generally start at 0 (zero) and they are not necessarily contiguous. This makes it difficult to use the write mask for certain operations.

For more information about the write mask, refer to [Using the Write Mask and Graphics Functions to Manipulate Color](#) on page B-82.

### **Private Colormaps**

An application can create its own *private colormap*. Most hardware can only display a single colormap at a time, so these private colormaps are called virtual colormaps, and only one at a time is visible. When the window manager gives the input focus to a window with a private colormap, the X server loads its virtual colormap into the hardware colormap.

### **Advantages**

The advantages of private colormaps include:

- Every color supported by the hardware is available to PV-WAVE, improving the quality of images.
- Allocated colormaps always start at 0 (zero) and use contiguous colors, which simplifies your use of the write mask.
- No translation between internal pixel values and the values required by the server is required, which optimizes the transfer of images.

### **Disadvantages**

On the other hand, using a private colormap can involve the following disadvantages:

- You may see “flashing” when you move the pointer in and out of PV-WAVE graphics windows. This happens because when the pointer moves inside a PV-WAVE window, the PV-WAVE colormap is loaded, and other applications are displayed using PV-WAVE’s colors. The situation is reversed when the pointer moves out of the PV-WAVE window into an area under the jurisdiction of a different application.
- Colors in a private colormap are usually allocated from the lower end of the map first. But these typically are the colors allocated by the window manager for such things as window borders, the color of text, and so forth.
- Since most PV-WAVE colormaps have very dark colors in the lower indices, the result of having other applications use the PV-WAVE colormap is that the

portions of the screen that are not PV-WAVE graphics windows look dark and unreadable.

### ***When Color Characteristics are Determined***

PV-WAVE decides how many colors and which combination of colormap and visual class to use when it creates its first graphics window of that session. You can create windows in two ways:

- Use the WINDOW procedure. WINDOW allows you to explicitly control many aspects of how the window is created, including its X visual class.
- If no windows exist and a graphics operation requiring a window is executed, PV-WAVE implicitly creates a window (window 0) using the default characteristics.

Once the first window is created, all subsequent PV-WAVE windows share the same colormap. The number of colors available is stored in the system variable !D.N\_Colors. For more information about !D.N\_Colors, see the *PV-WAVE User's Guide*.

For more information about when color characteristics are determined when drawing or plotting to a window running in a 24-bit visual class, refer to [24-bit Visual Classes](#) on page [B-77](#).

### ***Closing the Connection to the X Server to Reset Colors***

To change the type of colormap used or the number of colors, you must first completely close the existing connection to the X server using the following command:

```
DEVICE, /Close_Display
```

You can then use the WINDOW procedure to specify the new configuration. However, remember that if you enter the command shown above, it will cause every PV-WAVE graphics window that is currently open to be deleted.

---

**TIP** Another way to close the connection to the X server is to delete every PV-WAVE graphics window that is currently open; this automatically closes the connection.

---

### ***How Many Colors PV-WAVE Maps into the Color Table***

If the number of colors to use is explicitly specified using the *Colors* keyword with the WINDOW procedure, PV-WAVE attempts to allocate the number of colors specified from the shared colormap using the default visual class of the screen. If

enough colors aren't available, a private colormap with that number of colors is created instead.

For more information about the advantages and disadvantages of private colormaps, refer to *Private Colormaps* on page B-72.

By default, the shared colormap is used whenever possible, and PV-WAVE allocates all available colors from the shared colormap. The allocation occurs if no window currently exists and a graphics operation causes PV-WAVE to implicitly create one.

### ***Reserving Colors for Other Applications' Use***

Specifying a negative value for the *Colors* keyword to the WINDOW procedure causes PV-WAVE to use the shared colormap, allocating all but the specified number of colors. For example:

```
WINDOW, Colors = -16
```

allocates all but eight of the currently available colors. This allows other applications that might need their own colors (such as the window manager) to run in tandem with PV-WAVE.

### ***Color Translation Table***

Colors in the shared colormap do not have to start from index zero, nor are they necessarily contiguous. PV-WAVE preserves the illusion of a zero-based contiguous colormap by maintaining a translation table between applications' color indices and the actual pixel values allocated from the X server. The color indices range from 0 to !D.N\_Colors - 1. Normally, you need not be concerned with this translation table, but it is available using the statement:

```
DEVICE, Translation=Trans
```

This statement stores the current translation table, a 256-element byte vector, in the variable *Trans*. Element zero of this translation vector contains the value pixel allocated for the zero<sup>th</sup> color in the PV-WAVE colormap, and so forth.

---

**TIP** In the case of a private colormap, each element of the translation vector contains its own index value, because private colormaps start at zero and are contiguous.

---

The translation table may be bypassed, allowing direct access to the display's color hardware pixel values, by specifying the *Bypass\_Translation* keyword with the

DEVICE procedure. Translation is disabled by clearing the bypass flag, as shown in the following command:

```
DEVICE, Bypass_Translation=0
```

When a private or static (read-only) color table is initialized, the bypass flag is cleared. The bypass flag is always set when initializing a shared color table.

For more information about !D.N\_Colors, refer to the *PV-WAVE User's Guide*.

To see an example of how the translation table can affect displayed colors, see [Interaction Between the Set\\_Graphics\\_Function Keyword and Hardware Pixel Values](#) on page B-84.

## Using Pixmaps to Improve Application Performance

The X Window System can direct graphics to either windows or pixmaps.

- **Windows** — Windows are the usual windows that appear on the screen and contain graphics. Drawing to a window produces a viewable result.
- **Pixmaps** — Pixmaps are areas of invisible graphics memory contained in the server. Drawing to a pixmap simply updates the pixmap memory.

Pixmaps are useful because it is possible to write graphics to a pixmap and then copy the contents of the pixmap to a window where it can be viewed. This copy operation is very fast because it happens entirely within the server, instead of communicating across the network to the client. Provided enough pixmap memory is available for the server's use, this technique works very well for animating a series of images. The process works by placing the desired images into pixmap memory and then copying them to a visible window.

To read a brief description of the relationship between the X server and the X client, refer to [X Window System](#) on page B-58.

### ***Creating a Pixmap***

To create a pixmap, use the *Pixmap* keyword with the WINDOW procedure. For example, to create a 128-by-128 pixmap in PV-WAVE window 1:

```
WINDOW, 1, /Pixmap, XSize=128, YSize=128
```

Once they are created, pixmaps are treated just like normal windows, although some operations (WSHOW, for instance) don't have any noticeable effect when applied to a pixmap.

### **Example — Animating a Series of Pixmap Images**

The following example shows how animation can be performed using pixmap memory. It uses a series of 15 heart images taken from the file `heartbeat.dat`. (This file is located in the `data` subdirectory of the main `PV-WAVE` directory.) It creates a pixmap and writes all 15 images to it. It then uses the `Copy` keyword of the `DEVICE` procedure to copy the images to a visible window. Pressing any key causes the animation to halt:

```
PRO animate_heart
    ; This procedure animates a series of heart data.

IF !Version.platform EQ 'VMS' THEN $
    OPENR, u, GETENV('WAVE_DIR')+$
        '[data]heartbeat.dat', /Get_Lun $
ELSE $
    OPENR, u, '$WAVE_DIR/data/heartbeat.dat', /Get_Lun
        ; Open the file containing the images.

frame = ASSOC(u, BYTARR(256, 256))
    ; Associate a file variable with the file. Each heart image is
    ; 256-by-256 pixels.

WINDOW, 0, /Pixmap, XSize=7680, YSize=512
    ; Window 0 is a pixmap that is one double-sized image tall and
    ; 15 double-sized images wide. The images will be placed in this
    ; pixmap.

FOR i = 0, 15-1 DO TV, $
    REBIN(SMOOTH(frame(i), 3), 512, 512), i
        ; Write each image to pixmap memory. SMOOTH is used to
        ; improve the appearance of each image and REBIN is used to
        ; enlarge/shrink each image to the final display size.

FREE_LUN, u
    ; Close the image file and free the file unit.

WINDOW, 1, XSize=512, YSize=512, Title='Heart'
    ; Window 1 is a visible window that is used to display the animated
    ; data, creating the appearance of a beating heart.

i = 0L
    ; Load the current frame number.

WHILE GET_KBRD(0) EQ '' DO BEGIN
    DEVICE, Copy=[i * 512, 0, 512, 512, 0, 0, 0]
        ; Display frames until any key is pressed. Copy the next image
        ; from the pixmap to the visible window.
    i = (i + 1) MOD 15
        ; Keep track of total frame count.
```

ENDWHILE

END

In this example, the pixmap was made one image tall and 15 images wide for simplicity. However, some X servers will refuse to display a pixmap that is wider than the physical width of the screen. For this case, the above routine would have to be rewritten to either use a shorter, taller pixmap or to use several pixmaps.

## 24-bit Visual Classes

If you do not request a visual class (by entering a `DEVICE` command prior to opening the first graphics window), `PV-WAVE` uses the first class the device supports, regardless of the root visual class. `PV-WAVE` queries the device about the following visual classes, in order, until a supported visual class is found:

1. DirectColor, 24-bit
2. TrueColor, 24-bit
3. PseudoColor, 8-bit, then 4-bit
4. StaticColor, 8-bit, then 4-bit
5. GrayScale, any depth
6. StaticGray, any depth

`PV-WAVE` can operate in either of the two 24-bit visual classes. But because of the search order, if your workstation is configured to run in a 24-bit mode, `PV-WAVE` will choose DirectColor by default.

For more discussion concerning the search order for X visual classes, refer to [Selecting a Visual Class](#) on page B-69.

### **DirectColor Mode**

In DirectColor mode, color tables are always in effect and can be loaded. `PV-WAVE` loads the color tables using the same `LOADCT` and `TVLCT` commands you use when using `PV-WAVE` in 8-bit PseudoColor mode.

In this mode, color data is *decompressed*, meaning there are still only 256 slots in the `PV-WAVE` color table while operating in DirectColor mode. When color data is decompressed, 8 of the bits map to one of the 256 reds, 8 of the bits map to one of the 256 greens, and 8 of the bits map to one of the 256 blues that have been loaded into the 256 element color table (with `LOADCT` or `TVLCT`).

For more information about which bits are mapped to red, green, and blue, refer to [Specifying 24-bit Colors in Hexadecimal Notation](#) on page B-79.

## **TrueColor Mode**

In TrueColor mode, color tables cannot be loaded. However, PV-WAVE can load a translation table. You load the translation table using either the LOADCT or TVLCT commands. The translation table works just like a color table except that it takes effect at drawing time (TV, PLOT, CONTOUR, etc.) rather than when you enter the LOADCT or TVLCT command.

### **Example — Using PV-WAVE in TrueColor Mode**

In TrueColor mode, the color map takes effect when the graphics command (PLOT, SURFACE, TVSCL, etc.) is entered. An intriguing benefit of this behavior is that several PV-WAVE color tables loaded into translation tables (using LOADCT or TVLCT) may be used simultaneously during the same session, as shown in the following group of commands:

```
DEVICE, True_Color=24
    ; Define a TrueColor graphics window.

data = DIST(512)
    ; Create a sample data set that can be displayed as an image.

WINDOW, 0
LOADCT, 0
TVSCL, data
    ; Display the image in window 0 using grayish hues.

WINDOW, 1
LOADCT, 1
TVSCL, data
    ; Display the image in window 1 using bluish hues.

WINDOW, 2
LOADCT, 4
TVSCL, data
    ; Display the image in window 2 using shades of red, yellow, and
    ; green.
```

If the root window is also running in the 24-bit TrueColor visual class (just like PV-WAVE in this example), you will not see any “flashing”, even though you are entering LOADCT and TVSCL commands. For more information about the condition known as “window flashing”, refer to [Private Colormaps](#) on page [B-72](#).

### **PV-WAVE Does Not Inherit the X Visual Class**

PV-WAVE does not inherit the visual class of the X Window System root window. Thus, booting a machine with the root window set to any specific visual class has

no effect on the visual classes that are available for use by PV-WAVE (see the Note later in this section for an exception to this statement). For example, you could edit the `/etc/ttys` file of a Digital UNIX workstation (if it supported 24-bit visual classes) such that it included the line:

```
0: window = '/usr/bin/Xtm -class StaticGray'
```

and then proceed to simultaneously (and successfully) run two PV-WAVE sessions — one in DirectColor mode and the other in PseudoColor mode.

---

**NOTE** Windows that are created with calls to WAVE Widgets or Widget Toolbox functions and procedures are treated differently. These windows *do* inherit the X visual class from their top-level shell, which in turn, inherits the X visual class from the root window. In this sense, windows that are part of a graphical user interface (GUI) are different from ordinary PV-WAVE graphics windows.

---

## Understanding 24-bit Graphics Displays

A 24-bit raster image is actually made up of three component 8-bit images — a red, a green, and a blue image, which combine to create a “true” color picture. With 24-bit graphics displays, each pixel on the screen can be displayed in any one of a possible 16.7 million ( $2^{24}$ ) colors.

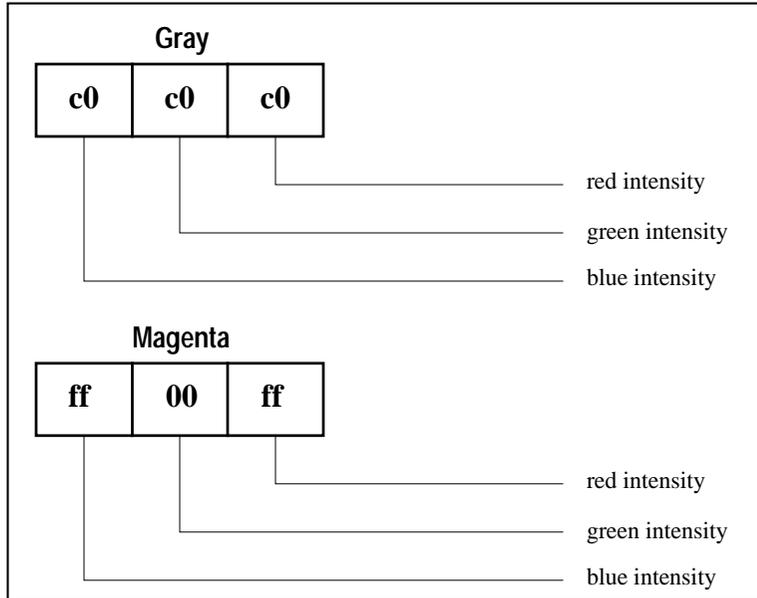
The video memory in a 24-bit machine is capable of addressing each pixel on the screen with 8 bits assigned to each of the red, green, and blue intensity values for that pixel. These sets of 8-bit values are known as *color planes*, e.g., the “red plane”, the “green plane”, and the “blue plane”. The three red-green-blue intensity values for each 24-bit pixel in video memory are translated directly into color intensities for the red, green, and blue electron guns in the CRT.

### ***Specifying 24-bit Colors in Hexadecimal Notation***

Colors in PV-WAVE graphics windows can be specified using the *Color* keyword. In the range of  $\{0..16,777,216\}$ , the color magenta has a decimal value of 16,711,935. But when using 24-bit color, the convention is to represent this value not as a decimal value, but as a 6-digit hexadecimal value. For example, the color magenta can be passed to one of the graphics routines using the construct:

```
Color = 'ff00ff'x
```

The first two digits in this hexadecimal value correspond to the blue intensity, the middle two digits correspond to the green intensity, and the right two digits correspond to the red intensity. The interpretation of the various digits is shown in .



**Figure B-3** Hexadecimal notation can be used to represent 24-bit numbers; each two digits describes either the red, green, or blue intensity. The top color shown in this figure is a light gray, since the red, green, and blue components are all set to an equal intensity. The lower color shown is the color magenta, where red and green are both set to full intensity (ff), but the color green is essentially turned “off” by setting it equal to zero (00).

The hexadecimal notations for some of the most commonly-used colors are shown in the following table.

Hexadecimal Notation for Comonly used Colors

Color	Hexadecimal Notation
Black	'000000'x
White	'ffffff'x
Red	'0000ff'x
Green	'00ff00'x
Blue	'ff0000'x
Cyan	'ffff00'x
Magenta	'ff00ff'x

## Hexadecimal Notation for Commonly used Colors (Continued)

Color	Hexadecimal Notation
Yellow	'00ffff'x
Medium Gray	'7f7f7f'x

---

**NOTE** When programming with WAVE Widgets or Widget Toolbox, you can only enter colors using color names. The hexadecimal form is not recognizable by the WAVE Widgets or Widget Toolbox routines. For more information about how to select colors for widgets in a graphical user interface (GUI), refer to the *PV-WAVE Application Developer's Guide*.

---

### **Specifying 24-bit Plot Colors**

In the TrueColor visual class, raster graphics colors go through the translation table, but vector graphics colors do not. This means that vector graphics colors in plots can be specified explicitly despite any translation table that has been loaded.

For more information about the translation table, refer to [Color Translation Table](#) on page [B-74](#).

### **Example — Plotting with 24-bit Colors**

The following example plots a line chart showing five data sets, each one plotted in a different color. (Assume that `mydata1`, `mydata2`, `mydata3`, `mydata4`, and `mydata5` have all been defined as integers or floating-point vectors prior to entering the commands shown below.)

```
DEVICE, True_Color=24
    ; Define a TrueColor graphics window.
PLOT, mydata1, Color='00ff00'x
    ; Draw mydata1 using a green line.
O PLOT, mydata2, Color='0000ff'x
    ; Draw mydata2 using a red line.
O PLOT, mydata3, Color='ff0000'x
    ; Draw mydata3 using a blue line.
O PLOT, mydata4, Color='00ffff'x
    ; Draw mydata4 using a yellow line.
O PLOT, mydata5, Color='007fff'x
    ; Draw mydata5 using an orange line.
```

---

**NOTE** The system variable !D.N\_Colors must still be initialized properly prior to opening the graphics window. For more information about how to initialize color characteristics, refer to *When Color Characteristics are Determined* on page B-73.

---

## Using the Write Mask and Graphics Functions to Manipulate Color

If you are using PV-WAVE in one of the 24-bit visual classes, you may want to consider using the write mask to isolate a certain group of bits, such as the red group of bits, or the green group. This is relatively easy to do, since each pixel in video memory directly references a set of three 8-bit red-green-blue intensities. For more information about how to address the various planes of a 24-bit (24-plane) workstation, refer to *Understanding 24-bit Graphics Displays* on page B-79.

In an 8-bit visual class, an analogous write mask operation is to use a write mask of 1 so that only the “lowest” plane is affected. But this is a suitable choice only if you want to force your application to display in monochrome on both color and monochrome displays.

---

**TIP** For best results when using the write mask, use a color table that uses all 256 available colors, and bypass the translation table to make sure the color table starts at zero (0). Unfortunately, a side effect of letting PV-WAVE allocate all 256 colors is that you may see “window flashing” when using your application, as explained in *Private Colormaps* on page B-72.

---

### Using the Write Mask to Create Special Effects

The write mask can be used to superimpose (overlay) one graphics pattern over another when plotting to a graphics window, allowing you to create special effects. For example, some 24-bit displays allow the screen to be treated as two separate 12-bit Pseudo\_Color visuals. This allows for “double-buffering”, a technique useful for animation, or for storing distance data to simplify hidden line and plane calculations in 3D applications.

Another possible application for the write mask is to simultaneously manage two 4-bit-deep images in a single graphics window instead of a single 8-bit-deep image. You could use the write mask to control whether the current graphics operation operates on the “top” image or the “bottom” image.

## ***Using Graphics Functions to Manipulate Color***

PV-WAVE's X (and Windows) driver provides two keywords for inquiring and manipulating the graphics function — *Get\_Graphics\_Function* and *Set\_Graphics\_Function*.

The value of the *Set\_Graphics\_Function* keyword controls the logical graphics function; this function specifies how the source pixel values generated by a graphics operation are combined with the pixel values already present on the screen. To see a complete list of graphics function codes, refer to the *Set\_Graphics\_Function* keyword description in the section *X Window System* on page B-58.

For example, the following code segment shows how to use the XOR graphics function to toggle the “low bit” of the pixel value that determines the color in a rectangle defined by its diagonal corners ( $x_0, y_0$ ) and ( $x_1, y_1$ ):

```
DEVICE, Get_Graphics=oldg, Set_Graphics=6
    ; Set graphics function to exclusive or (GXxor), saving the old
    ; function.

POLYFILL, [[x0,y0], [x0,y1], [x1,y1], $
    [x1,y0]], /Device, Color=1
    ; Use POLYFILL to select the area to be inverted, and
    ; immediately XOR a pixel value of 1 with the image currently
    ; displayed in that area. XORing every pixel with the binary
    ; equivalent of 1 ensures that only the lowest bit of the color is
    ; affected.

DEVICE, Set_Graphics_Function=oldg
    ; Restore the previous graphics function.
```

The default value for the *Set\_Graphics\_Function* keyword is *GXcopy*, which means that the source graphics from the current operation get copied into the window, destroying any graphics that were previously displayed there.

## ***Interaction Between the Set\_Write\_Mask and the Set\_Graphics\_Function Keywords***

Use the *Set\_Write\_Mask* keyword to specify the planes whose bits you are manipulating or the plane you want to use for the special effects. The way the two graphics patterns are combined depends on the value you provide for the *Set\_Graphics\_Function* keyword. For example, the following commands:

```
DEVICE, Set_Graphics_Function=6
DEVICE, Set_Write_Mask=8
```

extract the fourth bit (the binary equivalent of the decimal value 8) of the image in the current graphics window. The extracted plane is XORed (XOR is the graphics function specified by setting the *Set\_Graphics\_Function* keyword equal to 6) with the source pattern (the result of the current graphics operation). After the graphics function is implemented, the result is drawn in the current graphics window using whichever color(s) in the color table match the resultant value(s).

### **Interaction Between the *Set\_Graphics\_Function* Keyword and Hardware Pixel Values**

The graphics functions specified by the *Set\_Graphics\_Function* keyword operate on hardware pixel values. Unless the translation table is bypassed, like it is when displays with static (read-only) visual classes and private color tables are used, a color table index that represents a certain color will likely differ in value from the hardware pixel value that represents the same color. This can produce unexpected colors when you use the GX graphics functions.

An easy way to avoid getting such unexpected colors is to use Boolean OR, AND, XOR, and NOT operators, rather than the GX graphics functions. For details, see the following example.

### **Example — Understanding the Colors that are Produced by the GX Graphics Functions**

Suppose that you are using your X display in one of the 8-bit visual classes, and a simple color translation table has been defined in the following way:



**Figure B-4** An example color translation table.

Now enter the following commands:

```
data = [1, 2, 3]
; Define a simple dataset to experiment with.
GXand = 1
PLOT, data, Color=5
; The data is plotted in hardware color 1, because color 5 gets
; translated to hardware pixel value 1.
```

```
PLOT, data, Color=(5 AND 6)
; The data is plotted in hardware color 3, because 5 ANDed with 6
; equals 4, which then gets translated to hardware pixel value 3.

DEVICE, Set_Graphics_Function=GXand
PLOT, data, Color=6
; The data is plotted in hardware color 1, because color 5 gets
; translated to hardware pixel value 1 and color 6 gets translated to
; hardware pixel value 7. Then, in hardware pixel values, 7 is ANDed
; with 1 and the result equals 1.
```

---

**TIP** You can check the values in the current translation table using the *Translation* keyword; this keyword specifies the name of a variable to receive the translation vector. To read the translation table, enter this command:

```
DEVICE, Translation=Transarr
```

The result is a 256-element byte vector, *Transarr*. Element zero of *Transarr* contains the pixel value allocated for the first color in the **PV-WAVE** colormap, and so forth.

---

## X Window IDs

**PV-WAVE** provides methods for getting and setting X Window IDs for any **PV-WAVE** window. **PV-WAVE** also supports methods for setting and getting X Pixmap IDs for **PV-WAVE** windows.

To set the X Window ID for a **PV-WAVE** window, use the *Set\_Xwin\_Id* keyword with the **WINDOW** procedure. The X Window ID must be a valid, existing ID for the X server that **PV-WAVE** is using. When the *Set\_Xwin\_Id* keyword is used, **PV-WAVE** uses the X window associated with the ID; **PV-WAVE** does not create a new window. The programmer is responsible for synchronizing the use of this window by the two programs.

---

**NOTE** If the X Window ID is from another program, **PV-WAVE** color table changes may not affect the window.

---

To get the X Window ID for a **PV-WAVE** window, use the *Get\_Win\_ID* keyword with the **WINDOW** procedure. The X Window ID returned from the **WINDOW** procedure may be passed to another program. The other program may then write into the **PV-WAVE** window. The programmer is responsible for synchronizing the use of this window by the two programs.

Similarly, the *Get\_Xpix\_Id* keyword can be used to get the X Pixmap ID.

---

**CAUTION** The WDELETE procedure will delete all windows sharing a common X Window ID. For example:

---

```
WINDOW, 0, Get_Xwin_Id=New_Xwin_Id
WINDOW, 1, Set_Xwin_Id=New_Xwin_Id
WDELETE, 1
```

The command WDELETE, 1 deletes both windows 1 and 0.

---

## Z-buffer Output

The Z-buffer allows you to create complex 3D plots, image warping to polygons, and transparency effects without special hardware.

To direct graphics output to the Z-buffer, enter the command:

```
SET_PLOT, 'Z'
```

This causes PV-WAVE to use the Z-buffer driver for producing graphical output. Once the Z-buffer driver is enabled via SET\_PLOT, the DEVICE procedure is used to control its actions, as described in [Controlling Z-buffer Output with DEVICE Keywords](#) on page B-86.

Use INFO, /Device to view the driver's current settings.

## Controlling Z-buffer Output with DEVICE Keywords

The following keywords to the DEVICE procedure provide control of the Z-buffer driver:

**Close** — Deallocates the memory used by the buffers. The Z-buffer device is reinitialized if subsequent graphics operations are directed to the device.

**Get\_Graphics\_Function** — See the description of the *Get\_Graphics\_Function* keyword in [Controlling the X Driver with DEVICE Keywords](#) on page B-61.

**Get\_Write\_Mask** — See the description of the *Get\_Write\_Mask* keyword in [Controlling the X Driver with DEVICE Keywords](#) on page B-61.

**Set\_Character\_Size** — A two-element vector that changes the standard width and height of the vector-drawn fonts. The first element in the vector contains the new character width, and the second element contains the height. By default, characters are approximately 8-pixels wide, with 12 pixels between lines.

**Set\_Colors** — Sets the number of pixel values, !D.N\_Colors. This value is used by a number of routines to determine the scaling of pixel data and the default drawing index. Allowable values range from 2 to 256, and the default value is 256. Use this parameter to make the Z-buffer device compatible with devices with fewer than 256 color indices.

**Set\_Graphics\_Function** — See the description of the *Set\_Graphics\_Function* keyword in *Controlling the X Driver with DEVICE Keywords* on page B-61.

The Z-buffer allows you to use all graphics functions supported by the X driver.

**Set\_Resolution** — Two-element vector that sets the width and height of the buffers. The default size is 640-by-512. If this size is not the same as the existing buffers, the current buffers are destroyed and the device is reinitialized.

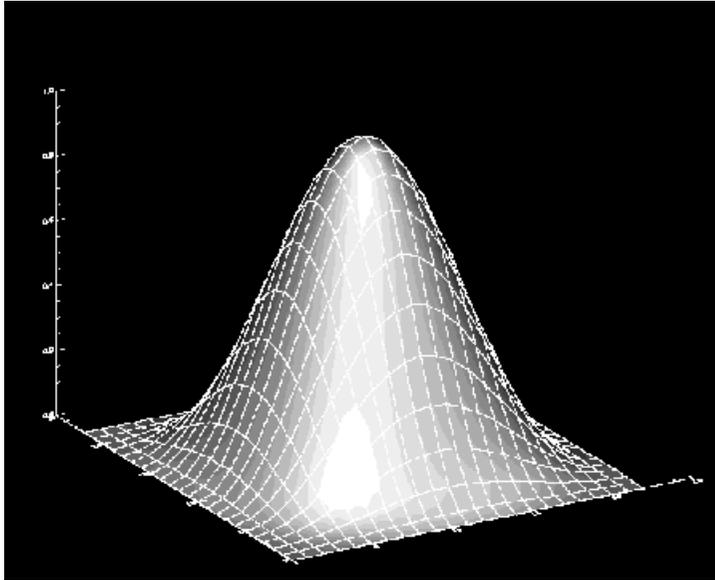
**Set\_Write\_Mask** — See the description of the *Set\_Write\_Mask* keyword in *Controlling the X Driver with DEVICE Keywords* on page B-61.

## Z-buffer Examples

### Example 1

This example demonstrates how graphics can be rendered in memory (into the Z-buffer) and then later displayed. The resulting image is shown in .

```
SET_PLOT, 'z'  
SHADE_SURF, HANNING(23,23)  
SURFACE, HANNING(23,23), /Noerase  
img = TVRD(0,0,640,512)  
SET_PLOT, 'x'  
TV, img
```



**Figure B-5** Resulting image from Example 1.

### ***Example 2***

This example creates a single image composed of two intersecting objects drawn with hidden surfaces removed. This effect can only be accomplished with the Z-buffer. The resulting image is shown in .

```

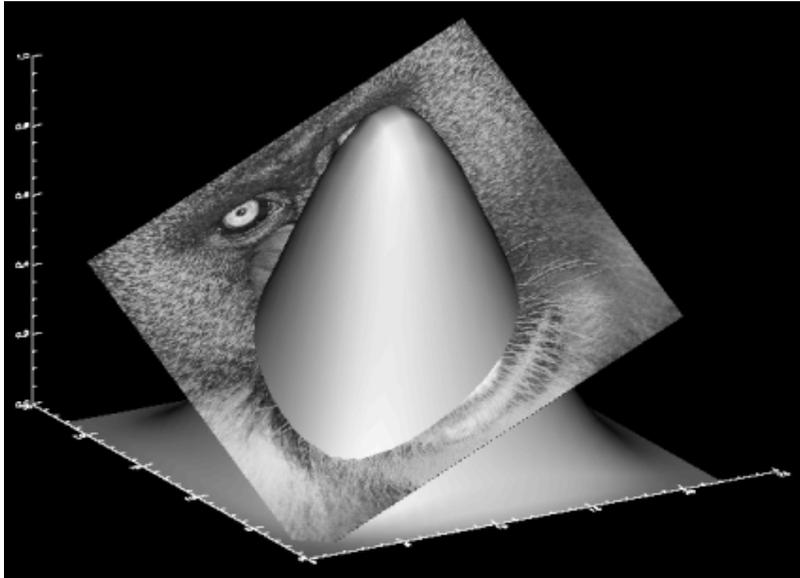
dev_name = !D.Name
          ; Remember current graphics device
im = BYTARR(512,512)
OPENR, 1, FILEPATH('mandril.img', SubDir='data' )
          ; Open the file containing the image.
READU, 1, im
CLOSE, 1
SET_PLOT, 'z'
ERASE
          ; Erase the Z-buffer in case there was something in there before.
SHADE_SURF, HANNING(23,23), /Save
          ; Draw a surface.
          ; Remember the 3D viewing transform (/Save).
verts=[ [ 0,0,0], [ 20,0,0.5], [ 20,20,1], $
        [ 0,20,0.5] ]
          ; Create a 3D quadrilateral in data coordinates.

```

```

POLYFILL, Verts, Pattern = im, $
    Image_coord=[[0,0],[511,0],[511,511], $
    [0,511]], /T3d
    ; Draw the transformed quadrilateral with an image mapped
    ; onto it.
res = TVRD(0,0,640,512)
SET_PLOT, dev_name
TVSCL, res
    ; Notice how the two objects intersect.

```



**Figure B-6** Resulting image from Example 2.

### ***Example 3***

In this example, the same image from the previous example is created; however, this time maximum intensity projection is used to produce a transparency effect. The resulting image is shown in .

```

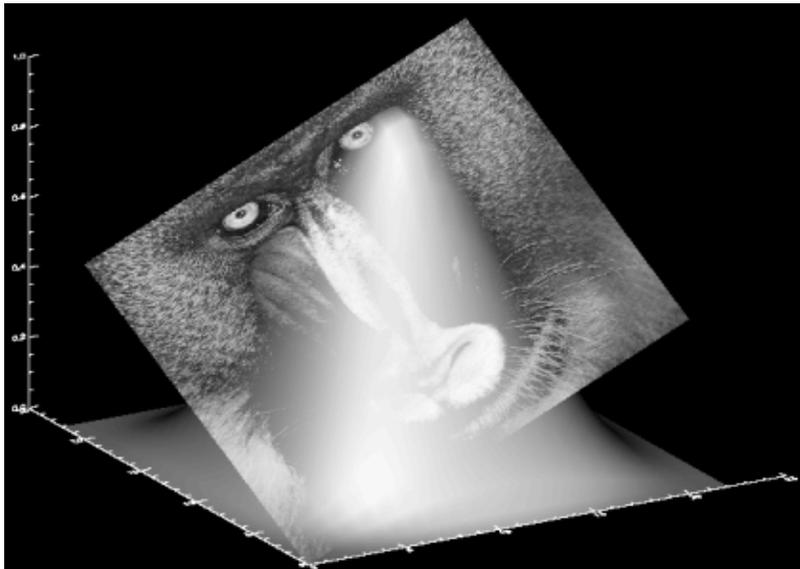
dev_name = !D.Name
    ; Remember current graphics device
im = BYTARR(512,512)
OPENR, 1, FILEPATH('mandril.img', SubDir='data' )
READU, 1, im
CLOSE, 1
SET_PLOT, 'z'

```

```

ERASE
    ; Erase the Z-buffer in case there was something in there before.
SHADE_SURF, HANNING(23,23), /Save
    ; Draw a surface.
    ; Remember the 3D viewing transform (/Save).
verts=[ [0,0,0], [20,0,0.5], [20,20,1], [0,20,0.5] ]
    ; Create a 3D quadrilateral in data coordinates.
POLYFILL, Verts, Pattern = im, $
    Image_coord=[ [0,0], [511,0], [511,511], [0,511] ], /T3d
    ; Draw the transformed quadrilateral with an image mapped
    ; onto it.
res = TVRD(0,0,640,512)
SET_PLOT, dev_name
TVSCL, res
    ; Notice how the two objects interact this time. You can partially see
    ; through the surface to the image passing through it.

```



**Figure B-7** Resulting image from Example 3.

# Reference Index

## A

- ABS function [46](#)
- absolute value [46](#)
- ACOS function [47](#)
- ADD\_EXEC\_ON\_SELECT procedure [48](#)
- ADDVAR procedure [49](#)
- AFFINE function [50](#)
- ALOG function [51](#)
- ALOG10 function [52](#)
- ampersand, separating multiple statements [1265](#)
- animation
  - See also* images
  - controlling the pace [1077](#), [1102](#)
  - cycling through images [1077](#)
  - data in pixmaps [1074](#), [1103](#)
  - double-buffering on 24-bit displays [B-70](#)
  - image [1159](#)
  - improving using pixmaps [B-75](#)
  - of images using bitmaps [B-49](#)
  - sequences of images [1073](#), [1102](#)
- annotation
  - alignment of the text [1197](#)
  - angle of [1213](#)
  - axis tick marks [1225](#)
  - axis title [1229](#), [1253](#)
  - centering of text [1197](#)
  - character size [1203](#), [1219](#), [1239](#), [1249](#)
  - color of [1204](#)
  - contour plots [1199](#)
  - display without data [1212](#)
  - formatting commands [B-39](#)
  - map [574](#)
  - margin for [1249](#)
  - plots [1189](#)
  - size of title [1203](#), [1239](#)
  - 3D orientation of text [1222](#)
  - tick marks [1252](#)
  - title of plot [1224](#), [1245](#)
  - width of text string [1225](#)
  - X axis [1221](#)
- application programming. *See* programming
- arc-cosine [47](#)
- arcsine [55](#)
- area of polygon [676](#)
- array
  - shift along one dimension [818](#)
  - subset [835](#)
- arrays
  - arbitrary type, creating [553](#)
  - average of [61](#)
  - building tables from [93](#)
  - calculating determinant of [282](#)
  - complement of [1135](#)
  - compute logical AND [502](#)
  - conversion of indices [503](#)
  - correlation coefficient of [166](#)
  - creating
    - associative [53](#)
    - dynamically [553](#)
    - from unique elements [960](#)
    - integer type [505](#), [509](#)
    - list arrays [533](#)
    - longword type [547](#)
    - strings [830](#), [868](#)
    - with arbitrary initialization [756](#)
  - days of the month [590](#)
  - days of the week [183](#)
  - diagonal, making or extracting [284](#)
  - double-precision complex type [190](#)
  - double-precision floating type [187](#)
  - double-precision type [186](#)

- elements, number of [597](#)
  - expanding into higher dimensions [342](#)
  - extrema [344](#)
  - finding the maximum value [575](#)
  - floating type [364](#), [366](#)
  - homogeneous region, counting [87](#)
  - homogeneous region, isolating [86](#)
  - integer type [505](#), [509](#)
  - intersection of [1135](#)
  - linear log scaling [651](#)
  - log-linear scaling [651](#)
  - log-log scaling [651](#)
  - longword type [547](#)
  - mean of [865](#)
  - median value of [577](#)
  - minimum value of [584](#)
  - non-zero elements, locating [1133](#)
  - OR together [504](#)
  - padding [634](#)
  - reading
    - from display [951](#)
  - reformatting [745](#)
  - replicating values of [756](#)
  - resample to new dimensions [758](#)
  - resizing of [160](#), [743](#)
  - reversing direction of [762](#)
  - rotating [772](#)
  - scaling to byte values [102](#)
  - shifting elements of [819](#)
  - size and type of [832](#)
  - smoothing of [838](#)
  - sorting contents of [854](#), [858](#)
  - standard deviation of [865](#)
  - string [830](#), [868](#)
  - subscripts
    - of points inside polygon region [683](#), [690](#)
  - sum elements of [931](#)
  - summing [901](#)
  - table grouping [399](#)
  - table sorting [631](#)
  - tensor product [927](#)
  - transformation [50](#)
  - transposition of [936](#)
  - tuples, finding unique [959](#)
  - union of [1135](#)
  - ASARR function [53](#)
  - ASCII
    - fixed format I/O [242](#)
    - free format data, writing to a file [250](#)
    - reading files [203](#), [218](#)
  - ASIN function [55](#)
  - ASKEYS function [56](#)
  - ASSOC function [58](#)
  - ATAN function [60](#)
  - average, boxcar [838](#)
  - AVG function [61](#)
  - axes
    - See also* tick marks
    - adding to plot [63](#)
    - annotation of [1225](#), [1229](#)
    - color of [1204](#)
    - date/time [1253](#)
    - endpoints of [1253](#)
    - global control of [1248–1253](#)
    - linear [1253](#)
    - logarithmic [1230](#), [1253](#)
    - saving scaling parameters [1218](#)
    - without data [1212](#)
  - AXIS procedure [63](#)
- ## B
- backing store, for windows [B-2](#)
  - back-substitution for linear equations [910](#)
  - bandpass filters [286](#)
  - BAR procedure [66](#)
  - BAR2D procedure [73](#)
  - BAR3D procedure [75](#)
  - base 10 logarithm [52](#)
  - basis function, example of [170](#)
  - BESEL1 function [77](#)
  - BESELJ function [78](#)
  - BESELY function [80](#)
  - BILINEAR function [82](#)
  - bilinear interpolation [82](#)
  - BINDGEN function [85](#)
  - bit shifting operation [516](#)
  - bitmap
    - See also* pixmap
    - animation of series [B-49](#)
    - creating with WINDOW procedure [B-49](#)
    - directed to graphics window [1138](#)
    - examples of [B-49](#)
    - used for cursor pattern [B-40](#)
    - with WIN32 driver [B-49](#)
  - BLOB function [86](#)

BLOBCOUNT function [87](#)  
 boundary [88](#)  
 BOUNDARY function [88](#)  
 BREAKPOINT procedure [89](#)  
 breakpoints  
     entered from command line [89](#)  
 buffer, flushing output [321](#), [367](#)  
 BUILD\_TABLE function [93](#)  
 BUILDRESOURCEFILENAME function [90](#)  
 BYTARR function [96](#)  
 byte  
     arrays, creating [96](#)  
     converting data to [97](#)  
     extracting data from [97](#)  
     scaling [102](#)  
 BYTE function [97](#)  
 BYTEORDER procedure [100](#)  
 BYTSCL function [102](#)

## C

C\_EDIT procedure [109](#)  
 CALL\_UNIX function [105](#)  
 callback  
     procedures [336](#)  
     registering [334](#)  
 Cartesian product [171](#)  
 CD procedure [107](#)  
 CeditTool procedure [1086](#)  
 center 3D data in display [113](#)  
 CENTER\_VIEW procedure [113](#)  
 CGM output [B-4–B-7](#)  
 characters  
     converting parameters to [872](#)  
     special [1265](#)  
 CHEBYSHEV function [115](#)  
 CHECK\_MATH function [118](#)  
 CHECKFILE procedure [116](#)  
 child processes, spawning [855](#), [859](#)  
 CINDGEN function [121](#)  
 city names [969](#)  
 clear output buffer [321](#), [367](#)  
 clear screen [324](#)  
 client  
     in X Window Systems [B-58](#)  
 clipboard  
     copy graphics to [1070](#)  
     pasting graphics from [1145](#)  
 clipping

    defining a rectangle for [1204](#)  
     graphics output [1204](#)  
     of graphics window [1215](#), [1239](#)  
     strings [889](#)  
     suppressing [1242](#)  
 CLOSE procedure [122](#)  
 closing  
     files [122](#)  
 code area, PV-WAVE [1271](#)  
 color  
     *See also* colormaps; color systems;  
     color tables  
     as 6-digit hexadecimal value [B-79](#)  
     background  
         plotting background color [1198](#)  
         PostScript output [B-29](#)  
         setting with system variable  
         [1238](#)  
     bar, purpose of [1079](#)  
     characteristics, determining [B-73](#)  
     decomposed into red, green, and  
         blue [B-70](#)  
     editing interactively [1083](#), [1092](#)  
     8-bit display [B-70](#)  
     histogram equalizing [434](#)  
     images [1102](#)  
     index, set for graphics elements  
         [1204](#)  
     model, PV-WAVE [B-47](#)  
     number available on graphics device  
         [1235](#)  
     PostScript devices [B-29](#)  
     pseudo [B-29](#)  
     raster graphics [B-81](#)  
     reserving for other applications [B-74](#)  
     rotate with color bar [1081](#), [1094](#)  
     running out of [B-72](#)  
     specifying by name [B-81](#)  
     surface bottom [1199](#)  
     translation  
         in X Windows [B-74](#)  
         table [B-61](#), [B-69](#), [B-74](#)  
     true-color [B-29](#)  
     24-bit display [B-70](#)  
     unexpected [B-84](#)  
     vector graphics [B-81](#)  
     WMF driver corrections [B-54](#)  
     X Windows [B-68](#)  
 color tables

- convert HSV to RGB [451](#)
- convert RGB to HLS or HSV [123](#)
- copying [792](#)
- creating [1086](#)
- editing [1083](#), [1092](#)
- expanding [870](#)
- histogram equalizing [434](#), [437](#)
- HLS based system [448](#), [723](#)
- indices [1087](#)
- loading
  - custom color table [537](#)
  - from variables [949](#)
  - predefined tables [535](#)
- lookup table [B-70](#)
- modifying [587](#)
- palette window [129](#), [636](#), [1086](#)
- replacing [587](#)
- stretching [870](#), [1094](#)
- supplied with PV-WAVE [535](#), [587](#)
- Tektronix 4115 color table [925](#)
- tools, interactive [109](#), [124](#), [635](#)
- 24-bit devices [B-70](#)
- COLOR\_CONVERT procedure [123](#)
- COLOR\_EDIT procedure [124](#)
- COLOR\_PALETTE procedure [129](#)
- colormap
  - compared to color table [B-69](#)
  - how PV-WAVE
    - chooses one [B-71](#)
    - obtains [B-73](#)
  - private, advantages/disadvantages [B-72](#)
  - shared, advantages/disadvantages [B-71](#)
- column-oriented data
  - reading [210](#), [225](#), [245](#), [253](#)
- commands
  - .RNEW [1270](#)
- comment, adding a [1266](#)
- compare [780](#)
- COMPILE procedure [131](#)
- compiling
  - memory requirements [1271](#)
  - one or more statements [337](#)
  - saving compiled procedure [131](#)
  - suppressing messages [1247](#)
- complement [1135](#)
- complex
  - arrays, creating [135](#), [138](#)
  - conjugate [142](#)
  - data type [135](#)
- COMPLEX function [135](#)
- COMPLEXARR function [138](#)
- Computer Graphics Metafile. *See* CGM
- concurrent processes [334](#)
- CONE function [139](#)
- CONGRID function [140](#)
- CONJ function [142](#)
- .CON [1268](#)
- contour plots
  - algorithms for drawing [149](#)
  - algorithms used to draw [1207](#)
  - cell drawing method [1207](#)
  - combining with
    - surfaces [1219](#)
    - surfaces and images [823](#)
  - creating [144](#), [147](#)
  - cubic spline, interpolation of [1220](#)
  - default number of levels [1211](#)
  - filled [147](#)
    - with color [151](#)
    - with pattern [1214](#)
  - gridding irregular data [387](#)
  - labeling [1201](#)
  - levels
    - color of [1200](#)
    - number of [1209](#), [1211](#)
  - line thickness of [1202](#)
  - line-following drawing method [1207](#)
  - linestyle of [1202](#)
  - maximum value to contour [1211](#)
  - scattered data [147](#)
  - shading [151](#)
  - sparse data [147](#)
- CONTOUR procedure [144](#)
- CONTOUR2 procedure [147](#)
- CONTOURFILL procedure [151](#)
- contraction [931](#)
- contrast, control [434](#)
- CONV\_FROM\_RECT function [158](#)
- CONV\_TO\_RECT function [163](#)
- CONVERT\_COORD procedure [156](#)
- converted [1046](#)
- converting
  - between graphics coordinate systems
    - [9](#), [158](#), [163](#), [680](#), [697](#)
  - color lists [677](#)
  - data to
    - See also* extracting data
    - byte type [97](#), [102](#)

- complex type [135](#)
- date/time [519](#), [783](#), [894](#), [973](#)
- double complex type [188](#)
- double-precision type [300](#)
- floating-point type [364](#)
- integer type [362](#)
- longword integer type [547](#)
- string type [872](#)
- Julian day number to PV-WAVE date/time [519](#)
- CONVOL function [160](#)
- convolution [160](#)
- coordinate systems
  - converting from one to another [9](#), [158](#), [163](#), [680](#), [697](#)
  - data [1206](#)
  - device [1206](#)
  - normalized [1212](#)
  - polar [1215](#)
  - reading the cursor position [175](#)
- copying
  - graphics, from window to clipboard [1070](#)
  - pixels [B-40](#), [B-49](#)
- CORRELATE function [166](#)
- correlation and regression analysis, Hilbert transform [432](#)
- correlation coefficient, computing for arrays [166](#)
- COS function [168](#)
- COSH function [169](#)
- cosine [168](#)
- COSINES function [170](#)
- count-intensity distributions [437](#)
- cpr files. *See* runtime mode
- CPROD function [171](#)
- CREATE\_HOLIDAYS procedure [171](#)
- CREATE\_WEEKENDS procedure [172](#)
- cross product, calculating [174](#)
- CROSSP function [174](#)
- CSV files, generating [251](#)
- cube, establishes frame of reference [1099](#), [1116](#)
- cubic splines
  - interpolation [862](#)
  - to smooth contours [1220](#)
- cursor
  - for Windows [B-40](#)
  - for X Windows system [B-62](#)
  - hot spot of [B-62](#)

- manipulating with images [947](#)
- position, reading [175](#)
- selecting default [B-40](#), [B-62](#)
- specifying bitmap for [B-62](#)
- specifying pattern of [B-40](#)
- system variable, !C [1233](#)
- CURSOR procedure [175](#)
- CURVATURES function [178](#)
- curve fitting
  - Gaussian [375](#)
  - least squares [913](#)
  - multiple linear regression [747](#)
  - non-linear least squares [179](#)
  - polynomial [691](#), [694](#)
  - singular value decomposition method [913](#)
  - to a surface [905](#)
- CURVEFIT function [179](#)
- cut-away volumes, defining [985](#)
- CYLINDER function [181](#)

## D

- damage repair of windows [B-2](#)
- data
  - 3D scaling [782](#)
  - building tables from [93](#)
  - column-oriented [210](#), [225](#), [245](#), [253](#)
  - connecting symbols with lines [1217](#)
  - defining a plotting symbol [1217](#)
  - dense points in gridding [347](#), [392](#)
  - fitting, cubic spline [862](#)
  - grouping tables [399](#)
  - importing tool [1174](#)
  - linear regression fit to [747](#)
  - magnetic tape storage [764](#)
  - maximum value to contour [1211](#)
  - points, connecting with lines [1217](#)
  - range [1250](#)
  - reduction before plotting [1213](#)
  - scaling to byte values [102](#)
  - skipping over [834](#)
  - smoothing of [838](#)
  - sorting tables [631](#)
  - sparse points in gridding [392](#)
  - symbols [1217](#)
- data area, PV-WAVE [1271](#)
- data types
  - of variable, determining [832](#)

- date/time data
  - array of, generating 309
  - converting to
    - double-precision variables 314
    - numerical data 319
    - string data 316
  - current system date and time 930
  - day of the year for each date 185
  - decrementing values 312
  - determining elapsed time 308
  - duration 308
  - incrementing values 303
  - printing values 311
  - removing holidays and weekends 304
- DAY\_NAME function 183
- DAY\_OF\_WEEK function 184
- DAY\_OF\_YEAR function 185
- DBLARR function 186
- DC\_allow\_chars 206, 221
- DC\_binary\_check 206, 221
- DC\_ERROR\_MSG function 191
- DC\_OPTIONS function 193
- DC\_READ\_24\_BIT function 196
- DC\_READ\_8\_BIT function 194
- DC\_READ\_CONTAINER function 199
- DC\_READ\_DIB function 201
- DC\_READ\_FIXED function 203
- DC\_READ\_FREE function 218
- DC\_READ\_TIFF function 231
- DC\_SCAN\_CONTAINER function 235
- DC\_WRITE\_24\_BIT function 238
- DC\_WRITE\_8\_BIT function 237
- DC\_WRITE\_DIB function 240
- DC\_WRITE\_FIXED function 242
- DC\_WRITE\_FREE function 250
- DC\_WRITE\_TIFF function 256
- DCINDGEN function 187
- DCL
  - logical names 798
  - symbols
    - defining 798
    - deleting 271
- DCOMPLEX function 188
- DCOMPLEXARR function 190
- deallocating
  - file units 367, 371
- decomposed color B-70
- DECW\$DISPLAY logical, for X Windows B-59
- DECwindows Motif B-59
- DEFINE\_KEY procedure 259
- DEFROI function 266
- DEFSYSV procedure 269
- degrees
  - convert to radians 1237
- DEL\_FILE procedure 272
- delaying program execution 1070
- DELETE\_SYMBOL procedure 271
- deleting
  - compiled functions from memory 273
  - compiled procedures from memory 275
  - DCL symbols 271
  - files 272
  - graphics 324
  - structure definitions 276
  - variables 277
  - VMS logical names 274
  - VMS symbols 271
- DELFUNC procedure 273
- DELLOG procedure 274
- DELPROC procedure 275
- DELSTRUCT procedure 276
- DELVAR procedure 277
- density function, calculating histogram 437
- DERIV function 279
- DERIVN function 281
- DETERM function 282
- determinant, calculating 279, 282
- device drivers
  - controlling output of 283
  - discussion of B-1
  - list of B-1
  - selecting 791
  - system variable 1234
- DEVICE procedure 283
- DIAG function 284
- DIB data
  - reading 201
  - reading into a graphics window 1148
  - writing from window to file 1156
  - writing to a file 240
- DICOM 285
- differentiation 281
- differentiation, numerical 279
- Digital terminals, generating output for B-34
- DIGITAL\_FILTER function 286

DILATE function [289](#)  
 dimensions of array, expanding [342](#)  
 DINDGEN function [293](#)  
 directory path  
     searching for procedures and functions [1245](#)  
 directory stack  
     popping directories off of [713](#)  
     printing out [716](#)  
     pushing [724](#)  
 directory, change working [107](#)  
 display  
     copy area from [B-61](#)  
     dimensions of [1236](#)  
     reading from [951](#)  
     tool [1172](#)  
 DIST function [294](#)  
 divisor [377](#)  
 DOC\_LIBRARY procedure [297](#)  
 documentation of user routines [297](#)  
     *See also* Users' Library  
 dollar sign [1266](#)  
 double complex  
     arrays, creating [188](#), [190](#)  
     data type [188](#)  
 DOUBLE function [300](#)  
 double-buffering [B-70](#)  
 double-precision  
     arrays, creating [186](#), [293](#)  
     data, converting to [300](#)  
     variables, converting date/time variables to [314](#)  
 drawing application, exchanging data with PV-WAVE [1149](#)  
 DROP\_EXEC\_ON\_SELECT procedure [302](#)  
 DT\_ADD function [303](#)  
 DT\_COMPRESS function [304](#)  
 DT\_DURATION function [308](#)  
 DT\_PRINT procedure [311](#)  
 DT\_SUBTRACT function [312](#)  
 DT\_TO\_SEC function [314](#)  
 DT\_TO\_STR procedure [316](#)  
 DT\_TO\_VAR procedure [319](#)  
 DTGEN function [309](#)

## E

eavesdrop mode, HPGL plotter output [B-11](#)  
 edge enhancement  
     Roberts method [769](#)  
     Sobel method [841](#)  
 eigenvalues and eigenvectors, determining [934](#)  
 8-bit image data  
     writing data to a file [237](#)  
 8-bit color [B-70](#)  
 EMF files  
     creating [1157](#)  
 empty output buffer [321](#), [367](#)  
 EMPTY procedure [321](#)  
 Encapsulated PostScript (EPS)  
     appearance of EPSI plots [B-23](#)  
     EPSI (Interchange Format) [B-23](#)  
     keyword [B-22](#)  
 end of file  
     testing for [323](#)  
     writing to tape [1073](#)  
 ENVIRONMENT function [322](#)  
 environment variables  
     adding [788](#)  
     changing [788](#)  
     \$DISPLAY for X Windows [B-59](#)  
     returning [378](#)  
 EOF function [323](#)  
     *See also* WEOF function  
 ERASE procedure [324](#)  
 erasing. *See* deleting  
 erf. *See* ERRORF  
 ERODE function [326](#)  
 error bars, plotting [331](#)  
 error handling  
     accumulated math error status [118](#)  
     for import/export (DC) functions [191](#)  
     function, evaluating [330](#)  
     input/output [614](#)  
     message  
         obtaining text of [1237](#)  
         prefix [1238](#)  
     plot truncated message [1216](#)  
     recovery from errors [613](#)  
     report level for DC functions [193](#)  
 ERRORF function [330](#)  
 ERRPLOT procedure [331](#)  
 EUCLIDEAN function [333](#)  
 EXEC\_ON\_SELECT procedure [334](#)  
 EXECUTE function [337](#)  
 executing  
     operating system commands [855](#),

- 859
- statements one at a time 337
- executive 1270
- executive commands
  - .CON 1268
  - .GO 1268
  - .LOCALS 1268
  - .RNEW 1269
  - .RUN
    - description 1269
    - examples 1270
    - l argument 1270
    - t argument 1270
  - .SIZE 1271
  - .SKIP 1271
  - .STEP 1272
  - table of 1267
  - See also* runtime mode
- EXIT procedure 340
- exiting
  - See also* aborting
  - PV-WAVE 340
- EXP function 341
- expand an array 342
- EXPAND function 342
- EXPON function 343
- exponential function, natural 341
- exponentiation 343
- Extended Metafile System (VMS) B-8
- extracting image plane B-52
- EXTREMA function 344

## F

- factor 345
- FACTOR function 345
- fast Fourier transform
  - function for 353
- FAST\_GRID2 function 346
- FAST\_GRID3 function 348
- FAST\_GRID4 function 351
- FFT function 353
- FILEPATH function 356
- files
  - allocating units 380
  - closing 122
  - deallocating LUNs 368
  - deleting 272
  - exporting VDA Tool 1168

- on magnetic tape 764
- opening 615, 621
- pointer, positioning 670
- printing 716
- skipping over 834
- testing for end of file 323
- unformatted data 1151
- filters
  - digital 286
  - mean smoothing 838
  - median smoothing 577
- FINDFILE function 358
- FINDGEN function 359
- FINITE function 360
- finite values, checking for 360
- FIPS code 562, 969
- fitting with Gaussian curve 375
- FIX function 362
- FLOAT function 364
- floating-point
  - arrays, creating 364, 366
  - type, converting to 364, 366
- floor 386
- FLTARR function 366
- FLUSH procedure 367
- flushing
  - output buffer 321, 367
- fonts
  - character size 1203, 1219
  - PostScript B-26
  - selection commands 1207
  - software, shown 1255
  - specifying hardware/software 1207, 1239
  - Windows system 1136
- force fields
  - plotting 664, 978, 982
- FREE\_LUN procedure 368
- FSTAT function 369
- FUNCT procedure 372
- function keys
  - defining 259
  - getting information about 506
- functions
  - See also* compiling; procedures; program
  - checking for keywords 520
  - deleting from memory 273
  - I/O errors in 614

## G

GAMMA function [374](#)  
GAUSSFIT function [375](#)  
Gaussian  
    curve fitting [375](#)  
    function, evaluating [376](#)  
    integral [376](#)  
GAUSSINT function [376](#)  
GCD function [377](#)  
GET\_KBRD function [379](#)  
GET\_LUN procedure [380](#)  
GET\_SYMBOL function [385](#)  
GETENV function [378](#)  
GETNCERR function [382](#)  
GETNCOPTS function [383](#)  
.GO [1268](#)  
Gouraud shading [702](#), [795–796](#)  
graphics  
    functions, interaction with write mask [B-50](#)  
graphics window  
    commands [B-45](#)  
graphs. *See* plotting  
GREAT\_INT function [386](#)  
greatest integer [386](#)  
grid [388](#)  
GRID function [387](#)  
GRID\_2D function [389](#)  
GRID\_3D function [391](#)  
GRID\_4D function [393](#)  
GRID\_SPHERE function [396](#)  
gridding  
    2D [346](#), [389](#)  
    3D [348](#), [391](#)  
    4D [351](#), [393](#)  
    dense data points [346](#), [348](#), [351](#)  
    sparse data points [389](#), [391](#), [393](#)  
    spheres [396](#)  
    summary of routines [16](#)  
GRIDN function [388](#)  
GROUP\_BY function [399](#)

## H

HAK procedure [404](#)  
HANNING function [405](#)  
hardcopy. *See* device drivers; printing  
hardware fonts. *See* fonts

hardware pixels [B-84](#)  
hardware polygon fill [B-10](#)  
HDF interface  
    base functions, defined [A-2](#)  
    convenience functions, defined [A-2](#)  
    documentation, NCSA [A-2](#)  
    example programs, location of [A-2](#)  
    help on functions [A-5](#)  
    initializing [A-3](#)  
    list of base functions [A-6](#)  
    overview [A-1](#)  
    starting [A-3](#)  
    testing [429](#)  
    using [A-3](#)  
HDF\_STARTUP [A-3](#)  
HDF\_TEST procedure [429](#)  
HDFGET24 function [407](#)  
HDFGETANN function [409](#)  
HDFGETFILEANN function [410](#)  
HDFGETNT function [412](#)  
HDFGETR8 function [414](#)  
HDFGETRANGE function [416](#)  
HDFGETSDS function [417](#)  
HDFLCT procedure [419](#)  
HDFPUT24 function [420](#)  
HDFPUTFILEANN function [422](#)  
HDFPUTR8 function [423](#)  
HDFPUTSDS function [425](#)  
HDFSCAN procedure [427](#)  
HDFSETNT function [428](#)  
HELP procedure [430](#), [506](#)  
help, online  
    documentation of user-written  
        routines [297](#)  
    getting [506](#)  
    HDF functions [A-5](#)  
    obtaining with INFO procedure [506](#)  
Hershey fonts [1207](#)  
Hewlett-Packard  
    Graphics Language plotters [B-8](#)  
    ink jet printers [B-14](#)  
    laser jet printers [B-14](#)  
    Printer Control Language printers [B-14](#)  
hexadecimal value, specifies color [B-79](#)  
hiding windows [1155](#)  
Hierarchical Data Format. *See* HDF interface  
highpass filters [286](#)

HILBERT function [432](#)  
 HIST\_EQUAL function [434](#)  
 HIST\_EQUAL\_CT procedure [437](#)  
 HISTN function [438](#)  
 histogram [438](#)  
     calculating density function [437, 440](#)  
     equalization [434](#)  
     of volumetric surface data [1100](#)  
     plotting [662](#)  
 HISTOGRAM function [440](#)  
 histogram plotting mode [1217](#)  
 HLS procedure [448](#)  
 holidays, removing from date/time variables  
     [304](#)  
 hot spot, of cursor [B-40](#)  
 HP VEE container file [199](#)  
 HPGL output [B-8–B-12](#)  
 HSV procedure [450](#)  
 HSV\_TO\_RGB procedure [451](#)  
 HTML Routines [18](#)  
 HTML\_BLOCK Procedure [453](#)  
 HTML\_CLOSE Procedure [454](#)  
 HTML\_HEADING Procedure [455](#)  
 HTML\_HIGHLIGHT Function [456](#)  
 HTML\_IMAGE Function [457](#)  
 HTML\_LINK Function [459](#)  
 HTML\_LIST Procedure [460](#)  
 HTML\_OPEN Procedure [463](#)  
 HTML\_PARAGRAPH Procedure [464](#)  
 HTML\_RULE Procedure [466](#)  
 HTML\_SAFE Function [466](#)  
 HTML\_TABLE Procedure [468](#)  
 HTML\_TEXT Procedure [470](#)  
 Hypertext Markup Language [18](#)

## I

image processing  
     calculating histograms [437](#)  
     convolution [160](#)  
     creating digital filters [286](#)  
     dilation operator [289](#)  
     edge enhancement [769, 841](#)  
     expanding [140, 745](#)  
     fast Fourier transform [353](#)  
     Hanning filter [405](#)  
     histogram equalization [434](#)  
     magnifying [160, 777, 952, 1192](#)  
     morphological dilation [289](#)

    morphological erosion [326](#)  
     polynomial warping [672, 673, 710](#)  
     profiles [718](#)  
     resizing images [743](#)  
     Roberts edge enhancement [769](#)  
     rotating [772, 775, 777](#)  
     selecting a region of interest [266](#)  
     shrinking [140](#)  
     smoothing [577, 838](#)  
     Sobel edge enhancement [841](#)  
     special effects [684, 700, B-69](#)  
     warping [710](#)  
     zooming [1192](#)

IMAGE\_CHECK function [473](#)  
 IMAGE\_COLOR\_QUANT procedure [474](#)  
 IMAGE\_CONT procedure [477](#)  
 IMAGE\_CREATE procedure [479](#)  
 IMAGE\_DISPLAY procedure [486](#)  
 IMAGE\_QUERY\_FILE function [488](#)  
 IMAGE\_READ procedure [492](#)  
 IMAGE\_WRITE procedure [495](#)

images

*See also* animation; image  
     processing  
         animation of [B-49](#)  
         combining with surface and contour  
             plots [823](#)  
         DIB data, reading [201](#)  
         direction of display (!Order) [1238](#)  
         8-bit, reading [194](#)  
         extracting plane [B-52, B-84](#)  
         interleaving [198, 234, 239](#)  
         optimizing transfer of [B-72](#)  
         overlying with contour plots [477](#)  
         PostScript display of [B-29](#)  
         reading  
             pixel values from [690](#)  
             skirt added to a surface [1127](#)  
             subscripts of pixels inside polygon  
                 region [690](#)  
             transposing of [936](#)  
             24-bit, reading [196](#)  
             24-bit, rendering [754](#)  
             value of individual pixels [1111](#)

IMAGINARY function [499](#)  
 IMG\_TRUE8 procedure [500](#)  
 INDEX\_AND function [502](#)  
 INDEX\_CONV function [503](#)  
 INDEX\_OR function [504](#)

- INDGEN function [505](#)
- INFO procedure
  - reference [506](#)
- INTARR function [509](#)
- integer
  - array, creating [509](#)
  - bit shifting [516](#)
  - converting an expression to [362](#)
  - finding nearest [598](#)
  - long. See longword
- integral of Gaussian [376](#)
- interapplication communication
  - C programs
    - calling from PV-WAVE [527](#)
    - communicating with [962](#)
  - calling PV-WAVE
    - from a C program [527](#)
  - linking
    - C code to PV-WAVE [527](#)
    - LINKLOAD function [527](#), [530](#)
  - sockets
    - routines [844–853](#)
- interleaving
  - 24-bit images [198](#), [234](#), [239](#)
- INTERPOL function [510](#)
- INTERPOLATE function [513](#)
- interpolation [513](#)
  - bilinear [82](#)
  - cubic spline [862](#)
  - linear [510](#)
  - linear, between colorable values [1087](#)
  - scattered data [513](#)
- intersection [1135](#)
- INTRP function [513](#)
- INVERT function [514](#)
- inverting pixels [B-51](#), [B-83](#)
- irregular data, gridding [387](#)
- ISASKEY function [515](#)
- ISHFT function [516](#)
- iso-surfaces
  - viewing interactively [1096](#)

## J

- JACOBIAN function [518](#)
- JOURNAL procedure [518](#)
- journaling
  - description of [518](#)
  - obtaining unit number of output [1237](#)

- JUL\_TO\_DT function [519](#)
- Julian day
  - converting to a date/time variable [519](#)
  - in !DT\_Base [785](#)

## K

- keyboard
  - defining keys [259](#)
  - getting input from [379](#)
  - interrupt [1268](#)
  - line editing, enabling [1237](#)
- KEYWORD\_SET function [520](#)
- keywords
  - checking for presence of [520](#)
  - graphics and plotting, usage [1197–1232](#)

## L

- landscape orientation [B-24](#)
- LaTeX documents
  - inserting plots [B-30](#)
  - using PostScript with [B-30](#), [B-32](#)
- latitude, finding on map [969](#)
- LCM function [522](#)
- least integer [837](#)
- least square
  - curve fitting [691](#), [710](#)
  - non-linear curve fitting [179](#)
  - problems, solving [911](#)
- Lee filter algorithm [523](#)
- LEEFILT function [523](#)
- LEGEND procedure [524](#)
- legend, adding to a plot [524](#)
- light source
  - modifying [795](#)
  - shading [702](#), [795](#)
- LINDGEN function [526](#)
- line
  - color of [1204](#)
  - connecting symbols with [1217](#)
  - drawing [664](#)
  - style of [1202](#), [1240](#)
  - thickness of [1223](#), [1244](#)
- linear
  - interpolation of a vector [510](#)

- least squares problems, solving 911
- regression, fit to data 747
- linear algebra
  - eigenvalues and eigenvectors 934
  - LU decomposition 549, 551
  - reducing matrices 938
  - solving equations 593, 910
  - solving matrices 938, 939
- LINKNLOAD function 527
- list
  - returning 534
- LIST function 533
- LISTARR function 534
- LN03 procedure 535
- LOAD\_HOLIDAYS procedure 538
- LOAD\_OPTION procedure 539
- LOAD\_WEEKENDS procedure 545
- LOADCT procedure 535
- LOADCT\_CUSTOM procedure 537
- LOADSTRINGS 542
- .LOCALS 1268
- logarithm
  - base 10 52
  - natural 51
- logarithmic
  - axes 651
  - plotting 651
  - scaling 651
- logical names
  - defining 789
  - VMS, DCL 941
  - VMS, deleting 274
- LONARR function 546
- LONG function 547
- longitude, finding on map 969
- longword
  - integer arrays, creating 526, 546
  - integer, converting to 547
- lookup table, color. *See* color tables
- lowpass filters 286
- LUBKSB procedure 549
- LUDCMP procedure 551
- LUNs
  - allocating 380
  - current output file 1235
  - deallocating 368
  - journal output 1237
  - waiting for input 785

## M

- magnetic tape
  - reading from 923
  - writing to 924
- magnifying images 777, 1192
- main
  - PV-WAVE directory 1236
- MAKE\_ARRAY function 553
- map datasets
  - subsetting 561
  - USGS Databank II 557
  - USGS Digital Line Graph 557
  - USGS Names 969
- MAP procedure 556
- map projections
  - PV-WAVE 560
- MAP\_CONTOUR procedure 565
- MAP\_PLOTS procedure 567
- MAP\_POLYFILL procedure 569
- MAP\_REVERSE procedure 571
- MAP\_VELOVECT procedure 572
- MAP\_XYOUTS procedure 574
- mapping, updated map dataset 561
- maps
  - annotating 574
  - colors 562
  - contours
    - filled 566
  - grid 557
- marker symbols
  - displaying in a volume 993
- masking
  - color B-82
- math errors
  - accumulated math error status 118
  - messages, issuing 582
- mathematical function
  - absolute value 46
  - arc-cosine 47
  - arcsine 55
  - arctangent 60
  - area of polygon 676
  - base 10 logarithm 52
  - Bessel I function 77
  - Bessel J function 78
  - Bessel Y function 80
  - bilinear interpolation 82
  - bit shifting 516

- boundary [88](#)
- checking for finite values [360](#)
- common divisor [377](#)
- compare [780](#)
- complex conjugate [142](#)
- convolution [160](#)
- correlation coefficient [166](#)
- cosine [168](#)
- cross product [174](#)
- cubic splines interpolation [862](#)
- derivative [279](#)
- determinant of matrix [282](#)
- error function [330](#)
- exponentiation [343](#)
- fast Fourier transform [353](#)
- GAMMA function [374](#)
- Gaussian integral [376](#)
- greatest integer [386](#)
- Hilbert transform [432](#)
- histogram [438](#)
- hyperbolic
  - cosine [169](#)
  - sine [831](#)
  - tangent [922](#)
- imaginary numbers [499](#)
- improve solution vector [593](#)
- least common multiple [522](#)
- least integer [837](#)
- list of [14–16](#)
- LU decomposition [549, 551](#)
- matrix
  - inversion [514](#)
  - reduction [938](#)
  - solutions [939](#)
- maximum value [575](#)
- mean [865](#)
- minimize [586](#)
- minimum value [584](#)
- moments [589](#)
- multiply [718](#)
- natural exponent [341](#)
- natural logarithm [51](#)
- neighbors [596](#)
- polynomial functions [672](#)
- polynomial roots [1194](#)
- prime factorization [345](#)
- primes [715](#)
- random numbers [737](#)
- resample array [758](#)
- sign of passed values [805](#)
- sine [829](#)
- singular value decomposition [911](#)
- solving simultaneous equations [910](#)
- square root [864](#)
- standard deviation [865](#)
- subset an array [835](#)
- tangent [921](#)
- vector, replicate [757](#)
- matrix
  - inverting [514](#)
  - printing
    - to specified file unit [669](#)
    - to standard output stream [667](#)
  - reading
    - from a file [768](#)
    - interactively [765](#)
  - reversing [762](#)
- MAX function [575](#)
- mean
  - of an array [865](#)
- median
  - filter [577](#)
  - value of array [577](#)
- MEDIAN function [577](#)
- memory
  - allocation [1271](#)
  - deleting
    - compiled functions [273](#)
    - compiled procedures [275](#)
    - structure definitions [276](#)
  - getting information about [506](#)
- menus, creating [1143](#)
- MESH function [580](#)
- mesh surfaces, drawing [902](#)
- message
  - See also* error handling
  - error [582](#)
  - error, setting level to report in DC functions [193](#)
  - for incomplete DC function [191](#)
  - prefix [1238](#)
- MESSAGE procedure [582](#)
- metafile
  - reading into a graphics window [1149](#)
  - writing from window to file [1157](#)
- Microsoft Paintbrush, exchanging data with PV-WAVE [1149](#)

Microsoft Word, importing CGM files into [594](#),  
[B-5](#)  
 MIN function [584](#)  
 minimize [586](#)  
 MINIMIZE function [586](#)  
 MODIFYCT procedure [587](#)  
 MOLEC function [588](#)  
 molecular model [588](#)  
 MOMENT function [589](#)  
 moments [589](#)  
 monochrome  
   displays [B-82](#)  
 MONTH\_NAME function [590](#)  
 morphologic  
   dilation operator [289](#)  
   erosion operator [326](#)  
 mouse  
   last button pushed [1237](#)  
   storing the button status [1238](#)  
   storing the position of [1238](#)  
   text editing functions [1131](#)  
 MOVIE procedure [591](#)  
 MPROVE procedure [593](#)  
 MSWORD\_CGM\_SETUP procedure [594](#)  
 multiple [522](#)  
 multiple plots [1240](#)  
 multiply [718](#)

## N

N\_ELEMENTS function [597](#)  
 N\_PARAMS function [601](#)  
 N\_TAGS function [602](#)  
 National Center for Supercomputer Applications (NCSA) [A-1](#)  
   *See also* HDF interface  
 National Imagery and Mapping Agency (NIMA) [561](#)  
 natural  
   exponential function [341](#)  
   logarithm [51](#)  
 Navigator  
   avoid DEVICE command [283](#)  
   24-bit display [283](#)  
 Navigator procedure [595](#)  
 neighbors [596](#)  
 NEIGHBORS function [596](#)  
 NINT function [598](#)  
 normal coordinate systems [1212](#)

normally distributed random numbers [736](#)  
 NORMALS function [600](#)

## O

ON\_ERROR procedure [604](#)  
 ON\_ERROR\_GOTO procedure [613](#)  
 ON\_IOERROR procedure [614](#)  
 OPENR procedure [615](#), [621](#)  
 OPENU procedure [615](#), [621](#)  
 OPENURL procedure [624](#)  
 OPENW procedure [615](#), [621](#)  
 OPLOT procedure [626](#)  
 OPLOTERR procedure [628](#)  
 OPTION\_IS\_LOADED function [630](#)  
 ORDER\_BY function [631](#)  
 output. *See* device drivers; input/output; printing; writing

## P

padding  
   volumes [995](#)  
 PADIT function [634](#)  
 PALETTE procedure [635](#)  
 palette, Windows [B-48](#)  
 PARAM\_PRESENT function [638](#)  
 parameters  
   checking for  
     number of [601](#)  
     presence of [638](#)  
 PARSEFILENAME procedure [640](#)  
 paste, graphics from clipboard [1145](#)  
 patterns. *See* polygon fill  
 PCL output [B-14](#)–[B-17](#)  
 PIE procedure [641](#)  
 PIE\_CHART procedure [646](#)  
 pixel map output [B-17](#)  
 pixels  
   copying [B-40](#), [B-49](#), [B-61](#), [B-75](#)  
   exact value of [1111](#)  
   hardware [B-84](#)  
   inverting region of [B-51](#), [B-83](#)  
   number per centimeter [1235](#)  
   reading from the display [738](#)  
 pixmaps  
   *See also* bitmaps  
   animating [1074](#), [1103](#)

- creating with WINDOW procedure [B-75](#)
- examples [B-76](#)
- wider than the physical width of the screen [B-77](#)
- with X Windows [B-4](#), [B-75](#)
- PLOT procedure [651](#)
- PLOT\_FIELD procedure [659](#)
- PLOT\_HISTOGRAM procedure [662](#)
- PLOT\_IO procedure [651](#)
- PLOT\_OI procedure [651](#)
- PLOT\_OO procedure [651](#)
- PLOTERR procedure [657](#)
- PLOTS procedure [664](#)
- plotters, HPGL [B-8](#)
- plotting
  - 2D array as 3D plot [929](#)
  - 2D graphs [651](#)
  - 2D tool [1179](#)
  - area, defining [793](#), [801](#)
  - bar [1160](#)
  - bar graphs [66](#), [73](#)
  - bar graphs, 3D [75](#)
  - bar3d [1162](#)
  - combination plots [823](#)
  - connecting symbols with lines [1217](#), [1243](#)
  - contour tool [1167](#)
  - error bars [331](#)
  - histogram [1170](#)
  - histogram style [1217](#)
  - keywords, list of [1197–1231](#)
  - line thickness [1223](#), [1244](#)
  - linear log scaling [651](#)
  - linestyles [1201](#)
  - margin around plot [1215](#)
  - margin for annotation [1249](#)
  - multiple plots [1240](#)
  - overplotting [626](#)
  - !P system variable [1238](#)
  - pie chart [1177](#)
  - pie charts [641](#), [646](#)
  - polar
    - coordinates [1215](#)
  - polygons
    - filling [683](#)
    - rendering [699](#)
  - position of plot in window [801](#), [1215](#), [1242](#)
  - range of data [1250](#)
  - routines, summary of [26–27](#)
  - speeding up with !P.Nsum [1242](#)
  - surface tool [1182](#)
  - symbols
    - creating new [967](#)
    - index of [1216](#), [1242](#)
    - specifying [1242](#)
  - vector fields from 3D arrays [975](#)
- pm driver [B-17](#)
- PM procedure [667](#)
- PMF procedure [669](#)
- !p.multi system variable [1240](#)
- POINT\_LUN procedure [670](#)
- polar
  - coordinates [1215](#)
  - plots [1215](#)
- POLY function [672](#)
- POLY\_2D function [673](#)
- POLY\_AREA function [676](#)
- POLY\_C\_CONV function [677](#)
- POLY\_COUNT function [679](#)
- POLY\_DEV function [680](#)
- POLY\_FIT function [691](#)
- POLY\_MERGE procedure [696](#)
- POLY\_NORM function [697](#)
- POLY\_PLOT procedure [699](#)
- POLY\_SPHERE procedure [705](#)
- POLY\_SURF procedure [708](#)
- POLY\_TRANS function [709](#)
- POLYFILL procedure [683](#)
- POLYFILLV function [690](#)
- POLYFITW function [694](#)
- polygon fill
  - 2D or 3D polygon [683](#)
  - color [1204](#)
  - hardware [B-10](#)
  - on maps [569](#)
  - pattern [1207](#), [1214](#)
- polygons
  - area of [676](#)
  - filling. *See* polygon fill
  - generating [27](#), [708](#)
  - lists, merging [696](#)
  - manipulating [28](#)
  - merging lists for rendering [696](#)
  - meshes [580](#)
  - plotting [699](#)
  - querying subscripts of internal pixels [690](#)
  - rendering [28](#), [699](#), [752](#)
  - returning

- list of colors for 677
- number contained in list 679
- shading 702
- polynomial
  - curve fitting 691, 694
  - functions, evaluating 672
  - warping of images 673, 710
- POLYSHADE function 702
- POLYWARP procedure 710
- POPD procedure 713
- portrait orientation B-25
- PostScript output B-19–B-33
- PRIME function 715
- prime numbers 715
- PRINT procedure 716
- PRINTD procedure 717
- Printer Control Language (PCL) output B-14
- PRINTF procedure 716
- printing
  - See also* fonts
  - ASCII output 716
  - CGM output B-4
  - contents of graphics window 1146
  - directories 717
  - files 716
  - HPGL output B-8
  - LN03 printer 535
  - PostScript printer B-20
  - values of date/time variables 311
  - variables 716
  - Windows 1137
- private colormaps B-72
- procedures
  - See also* functions; program
- processes
  - concurrent 334
  - spawning 855, 859
- product 718
- PRODUCT function 718
- product-moment correlation coefficient 166
- PROFILE function 718
- PROFILES procedure 720
- profiles, extracting from images 718
- program
  - See also* compiling; functions; procedures; programming
  - checking for positional parameters 601
  - deleting from memory 275
  - directory path for 1245

- files containing PV-WAVE
  - procedures 1270
- number of
  - non-keyword parameters 601
- pausing execution 1070
- stopping execution 867
- programming
  - create special effects with
    - graphics functions B-51
  - delaying program execution 1070
  - graphics functions B-82
  - menus, creating 1143
  - providing a GUI B-60
  - special effects
    - graphics functions B-83
    - write mask B-82
  - write mask B-82
- PROMPT procedure 722
- prompt, changing 1247
- PSEUDO procedure 723
- pseudo-color
  - 12-bit B-82
- PUSHD procedure 724
- PV-WAVE session
  - exiting 340
  - recording 518
  - restoring 759
  - saving 781

## Q

- QUERY\_TABLE function 726
- QUIT procedure 734
- quitting PV-WAVE 340, 734
- quotation marks 1266

## R

- radians, converting to degrees 1247
- random number
  - normal distribution 736
  - uniform distribution 737
- RANDOMN function 736
- RANDOMU function 737
- raster
  - graphics, colors B-81
- ray tracing
  - cone primitives 139

- cylinder primitives [181](#)
- mesh primitives [580](#)
- RENDER function [752](#)
- sphere primitives [860](#)
- summary of routines [30](#)
- volume data [1000](#)
- RDPIX procedure [738](#)
- READ procedure [739](#)
- READ\_XBM Procedure [742](#)
- READF procedure [739](#)
- reading
  - 24-bit image file [196](#)
  - 8-bit image data [194](#)
  - See also* input/output
  - ASCII data
    - from a file [203](#), [218](#), [739](#)
    - from standard input [739](#)
  - binary files [739](#)
  - cursor position [175](#)
  - DIB data [201](#)
  - DIB data into a graphics window [1148](#)
  - files [739](#)
  - fixed-format ASCII data [203](#)
  - freely-formatted ASCII data [218](#)
  - from magnetic tapes [923](#)
  - images from the display [951](#)
  - keyboard input [379](#)
  - metafile into a graphics window [1149](#)
  - TIFF files [231](#)
  - unformatted data [739](#)
  - waiting for input [785](#)
- READU procedure [739](#)
- REBIN function [743](#)
- recording a PV-WAVE session [518](#)
- REFORM function [745](#)
- region of interest, selecting [266](#)
- Regis output [B-34–B-35](#)
- REGRESS function [747](#)
- regression, fit to data [747](#)
- RENAME procedure [749](#)
- RENDER function [752](#)
- RENDER24 function [754](#)
- rendering
  - cone objects [139](#)
  - cylinder objects [181](#)
  - mesh objects [580](#)
  - polygons [28](#), [699](#)
  - ray-traced objects [752](#)
  - shaded surfaces [702](#), [806](#), [812](#)
  - sphere objects [860](#)
  - volumes [40](#), [815](#), [996](#), [1000](#)
- REPLICATE function [756](#)
- REPLV function [757](#)
- RESAMP function [758](#)
- resizing arrays or images [743](#)
- resource
  - loading [540](#)
  - loading strings [542](#)
- RESTORE procedure [759](#)
- RETAIL procedure [760](#)
- RETURN procedure [761](#)
- REVERSE function [762](#)
- REWIND procedure [764](#)
- RGB\_TO\_HSV procedure [764](#)
- RM procedure [765](#)
- RMF procedure [768](#)
- .RNEW [1269](#), [1270](#)
- Roberts edge enhancement [769](#)
- ROBERTS function [769](#)
- roots, finding complex polynomial [1194](#)
- ROT function [772](#)
- ROT\_INT function [777](#)
- ROTATE function [775](#)
- rotating
  - arrays or images [775](#)
  - corresponding to surface [902](#)
  - images [772](#), [775](#), [777](#)
- runtime mode
  - compiling procedures for [131](#)

## S

- SAME function [780](#)
- SAVE procedure [781](#)
- saving
  - compiled procedures [131](#)
  - PV-WAVE session [518](#), [781](#)
- SCALE3D procedure [782](#)
- scaling
  - 3D [782](#)
  - corresponding to surface [902](#)
  - images [952](#)
  - unit cube into viewing area [782](#)
- screen pixels, assigning color [B-69](#)
- searching
  - for text string [877](#)
- SEC\_TO\_DT function [783](#)

seconds, converting to date/time variables 783

SELECT\_READ\_LUN procedure 785

servers

- closing connections B-61
- X Window system B-58

session. *See* PV-WAVE session

SET\_PLOT procedure 791

SET\_SCREEN procedure 793

SET\_SHADING procedure 795

SET\_SYMBOL procedure 798

SET\_VIEW3D procedure 800

SET\_VIEWPORT procedure 801

SET\_XY procedure 803

SETDEMO procedure 786

SETENV procedure 788

SETLOG procedure 789

SETNCOPTS procedure 790

SETUP\_KEYS procedure 798

SGN function 805

SHADE\_SURF procedure 806

SHADE\_SURF\_IRR procedure 812

SHADE\_VOLUME procedure 815

shading

- contour plots 151
- Gouraud interpolation 796
- setting parameters 795
- surfaces 702, 801, 806, 812, 1127
- volumes 815

SHIF function 818

SHIFT function 819

SHOW\_OPTIONS procedure 825

SHOW3 procedure 823

SIGMA function 827

sign of passed values 805

signal processing

- convolution 160
- creating filters 286
- fast Fourier transform 353
- Hanning filter 405
- Hilbert filter 432
- histogram equalization 434
- Lee filter 523

simultaneous equations, solving 910

SIN function 829

SINDGEN function 830

sine

- hyperbolic 831
- trigonometric 829

singular value decomposition

- curve 910
- SVD function 911

SINH function 831

.SIZE 1271

SIZE function 832

.SKIP 1271

SKIPF procedure 834

SLICE function 835

SLICE\_VOL function 835

slicing

- interactively 985
- plane, defining 985
- volumes 835, 1113

SMALL\_INT function 837

SMOOTH function 838

smoothing

- boxcar 838
- contour plots 1220

Sobel edge enhancement 841

SOBEL function 841

socket, X11 786

SOCKET\_ACCEPT function 844

SOCKET\_CLOSE procedure 846

SOCKET\_CONNECT function 847

SOCKET\_GETPORT function 848

SOCKET\_INIT function 849

SOCKET\_READ function 851

SOCKET\_WRITE procedure 852

sockets. *See* interapplication communication

software fonts. *See* fonts

SORT function 854

sorting

- See also* subsetting
- array contents 854, 858
- tables 726

SORTN function 858

spatial transformation of images 673, 710

SPAWN procedure 855, 859

spawning a process 855, 859

special effects

- color B-83
- graphics functions and write mask B-82
- using graphics functions and write mask B-50
- Z-buffer use B-86

SPHERE function 860

spheres

- defining with SPHERE function [860](#)
  - generating [705](#)
  - gridding [396](#)
- SPLINE function [862](#)
- splines
  - cubic [862](#)
  - interpolation of contour plots [1220](#)
- SQRT function [864](#)
- square root, calculating [864](#)
- standard deviation, calculating [827](#), [865](#)
- statements
  - executing one at a time [337](#)
- STDEV function [865](#)
- .STEP [1272](#)
- STOP procedure [867](#)
- STR\_TO\_DT function [894](#)
- STRARR function [868](#)
- STRCOMPRESS function [869](#)
- STRETCH procedure [870](#)
- STRING function [872](#)
- string processing
  - See also* annotation; strings
  - compressing white space [869](#)
  - converting to lower case [880](#)
  - converting to upper case [900](#)
  - excess white space [869](#)
  - extracting substrings [886](#)
  - inserting substrings [889](#)
  - leading and trailing blanks [896](#)
  - locating substrings [887](#)
  - removing blanks [869](#)
- string resources [542](#)
- strings
  - arrays, creating [830](#), [868](#)
  - concatenating arrays to scalar [875](#)
  - database search [877](#)
  - matching [881](#)
  - splitting into tokens [890](#)
  - substituting [892](#)
- strip chart, creating [1119](#)
- STRJOIN function [875](#)
- STRLEN function [876](#)
- STRLOOKUP function [877](#)
- STRLOWCASE function [879](#)
- STRMATCH function [881](#)
- STRMESSAGE function [884](#)
- STRMID function [886](#)
- STRPOS function [887](#)
- STRPUT procedure [889](#)
- STRSPLIT function [890](#)
- STRSUBST function [892](#)
- STRTRIM function [896](#)
- STRUCTREF function [898](#)
- structures
  - date/time [244](#), [249](#), [318](#)
  - deleting [276](#)
  - getting information about [506](#)
  - list of references to, returning [898](#)
  - number of tags in [602](#)
  - tag names
    - array [920](#)
    - returning [920](#)
- STRUPCASE function [899](#)
- subscripts
  - \* (asterisk) operator [1265](#)
  - pixels inside a polygon region [690](#)
  - ranges
    - size using asterisk [1265](#)
- subsetting
  - cut-away views [985](#)
  - map datasets [561](#)
  - slicing 3D datasets [985](#), [1113](#)
  - tables [726](#)
- substrings. *See* string processing
- SUM function [901](#)
- surface plot
  - color of bottom [1199](#)
  - combining with image and contour [823](#), [1219](#)
  - creating [902](#), [1124](#)
  - curve fitting to [905](#)
  - horizontal lines only [1209](#)
  - interactive tool [1124](#)
  - iso-surface, viewing [1096](#)
  - lower surface only [1210](#)
  - overlying with contours [1218](#)
  - rendering of shaded [806](#), [812](#)
  - rotation
    - angle of X [1197](#)
    - angle of Z [1198](#)
  - saving 3D to 2D transformation
    - matrix [1218](#)
  - shaded [702](#), [1127](#)
  - skirt, adding [1127](#), [1219](#)
  - storing data in a file [809](#)
  - transformation matrix [902](#)
  - upper surface only [1224](#)
- SURFACE procedure [902](#)

SURFACE\_FIT function 905

SURFR procedure 907

SVBKSB procedure 910

SVD procedure 911

SVDFIT function 170, 913

symbols

colored, displayed in a volume 993

connecting with lines 1216

keyword 1242

marker 993

size of 1221

system variable 1216

user-defined 1216

VMS

defining 798

deleting 271

returnable 385

system variables

! character in name 1266

creating

date 930

time 915, 930

user-defined 269

list of 1233

setting to specific value 269

SYSTIME function 915

## T

T3D procedure 917

table tool 1184

tables

creating 93

determining unique values in 960

group by 399

sorting 726

sorting rows 631

subsetting with Where clause 726

unique elements of 960

tag

names in a structure 920

numbers of 602

Tag Image File Format. *See* TIFF

TAG\_NAMES function 920

TAN function 921

tangent

hyperbolic 922

trigonometric 921

TANH function 922

tape, magnetic

reading from 923

rewinding 764

skipping records or files 834

writing to 924, 1073

TAPRD procedure 923

TAPWRT procedure 924

TEK\_COLOR procedure 925

Tektronix

4115 device

color table 925

terminal family output B-36-B-38

SENSOR functions 927

tensor product 927

thickness

axis line 1251

characters 1239

lines 1244

THREED procedure 929

threshold

value of an iso-surface 1100

tick marks

centering Y label 1230

controlling length of 1224, 1245

extending away from the plot 1224

intervals

between marks 1229

setting number of 1253

linestyle 1208, 1226

number of minor marks 1250

suppressing minor marks 1250

TIFF

reading a file in 231

time

current 915

elapsed between dates 308

system 915

TODAY function 930

TOTAL function 931

town names 969

TQLI procedure 934

trace 931

transformation

affine 50

saving 1218

3D volumes 999

transformation matrices

3D points 709

4-by-4 709, 999, 1244

- centering the view [113](#)
- keyword discussion [1222](#)
- POLY\_TRANS function [709](#)
- set up 3D view [800](#)
- storing [917](#)
- SURFACE procedure [902](#)
- translation table
  - bypassing [B-61](#), [B-74](#)
- translucency, in images [700](#), [996](#)
- transparency, in images [684](#)
- TRANPOSE function [936](#)
- transposing
  - See also* rotating
  - arrays or images [775](#), [936](#)
- TRED2 procedure [938](#)
- TRIDAG procedure [939](#)
- trigonometric functions
  - arc-cosine [47](#)
  - arcsine [55](#)
  - arctangent [60](#)
  - cosine [168](#)
  - hyperbolic cosine [169](#)
  - hyperbolic sine [831](#)
  - hyperbolic tangent [922](#)
  - sine [829](#)
  - summary list of [32](#)
  - tangent [921](#)
- TRNLOG function [941](#)
- tuples, finding unique [959](#)
- TV procedure [943](#)
- TVCRS procedure [947](#)
- TVLCT procedure [949](#)
- TVRD function [951](#)
- TVSCL procedure [952](#)
- TVSIZE procedure [956](#)
- 24-bit color
  - differentiating data sets [B-81](#)
- 24-bit image data
  - rendering [754](#)
  - writing data to a file [238](#)
- types
  - data conversion to
    - bytes [97](#)
    - complex [135](#)
    - double complex [188](#)
    - double-precision [300](#)
    - floating-point [364](#)
    - integer [362](#)
    - longword integer [547](#)

- scaled byte [102](#)
- string [872](#)

## U

- uniformly distributed random numbers [737–738](#)
- union [1135](#)
- UNIQN function [959](#)
- UNIQUE function [960](#)
- UNIX operating system
  - calling from PV-WAVE [105](#)
  - commands from within PV-WAVE [855](#)
- UNIX\_LISTEN function [962](#)
- UNIX\_REPLY function [963](#)
- UNLOAD\_OPTION procedure [964](#)
- UPVAR procedure [965](#)
- USERSYM procedure [967](#)
- USGS\_NAMES function [969](#)

## V

- VAR\_MATCH function [971](#)
- VAR\_TO\_DT function [973](#)
- variables
  - See also* environment variables;
  - system variables
  - adding at \$MAIN\$ [49](#)
  - associated [58](#)
  - binding to different program level [965](#)
  - deleting [277](#)
  - determining
    - data type of [832](#)
  - exporting [1186](#)
  - number of structure tags [602](#)
  - printing [716](#)
  - structure tag names [920](#)
  - viewing [1186](#)
- VDA Tools
  - avoid DEVICE command [283](#)
  - 24-bit display [283](#)
- VDA Tools Utilities
  - messages
    - stored in resource files [540](#)
- vector
  - See also* arrays; linear algebra

- building tables from [93](#)
- cross product [174](#)
- deriving unique elements from [960](#)
- fields, plotting [664](#), [975](#), [978](#)
- grouping tables [399](#)
- linear interpolation of [510](#)
- replicate [757](#)
- reversing [762](#)
- solution, improving [593](#)
- sorting tables [631](#)
- string [877](#), [881](#), [900](#)
- VECTOR\_FIELD3 procedure [975](#)
- VEL procedure [978](#)
- VELOECT procedure [982](#)
- version number of PV-WAVE [1248](#)
- vertex lists
  - merging [696](#)
- video memory, of workstation [B-70](#)
- video modes, Windows [B-46](#)
- VIEWER procedure [985](#)
- viewport, defining [793](#), [801](#)
- Virtual Reality Modeling Language [39](#)
- visual classes
  - for X windows [B-68](#)
  - not inherited by PV-WAVE [B-78](#)
  - set for root window [B-78](#)
- visualization toolkit [4](#)
- VMS operating system
  - calling PV-WAVE [527](#)
  - CGM output, binary [B-8](#)
  - Extended Metafile System [B-8](#)
  - logical names, deleting [274](#)
  - symbols
    - defining [798](#)
    - deleting [271](#)
    - return value [385](#)
- VOL\_MARKER procedure [993](#)
- VOL\_PAD function [995](#)
- VOL\_REND function [996](#)
- VOL\_TRANS function [999](#)
- VOLUME function [1000](#)
- volumes
  - defining [985](#), [1000](#)
  - displaying markers in [993](#)
  - manipulating [39](#)
  - padding [995](#)
  - shaded [815](#)
  - slicing
    - with SLICE\_VOL function [835](#)
    - with VIEWER procedure [985](#)
    - transforming [999](#)
- volumetric surface data [1096](#), [1113](#)
- VRML Routines [39](#)
- VRML\_AXIS Procedure [1002](#)
- VRML\_CAMERA Procedure [1004](#)
- VRML\_CLOSE Procedure [1005](#)
- VRML\_CONE Procedure [1006](#)
- VRML\_CUBE Procedure [1009](#)
- VRML\_CYLINDER Procedure [1011](#)
- VRML\_LIGHT Procedure [1014](#)
- VRML\_LINE Procedure [1015](#)
- VRML\_OPEN Procedure [1017](#)
- VRML\_POLY Procedure [1018](#)
- VRML\_SPHERE Procedure [1020](#)
- VRML\_SPOTLIGHT Procedure [1022](#)
- VRML\_SURFACE Procedure [1024](#)
- VRML\_TEXT Procedure [1025](#)
- VT graphics terminals [B-34](#)
- vtkADDATTRIBUTE Procedure [1028](#)
- vtkAXES Procedure [1029](#)
- vtkCAMERA Procedure [1031](#)
- vtkCLOSE Procedure [1032](#)
- vtkCOLORBAR Procedure [1033](#)
- vtkCOMMAND Procedure [1034](#)
- vtkERASE Procedure [1035](#)
- vtkGRID Procedure [1036](#)
- vtkHEDGEHOG Procedure [1037](#)
- vtkINIT Procedure [1039](#)
- vtkLIGHT Procedure [1040](#)
- vtkPLOTS Procedure [1041](#)
- vtkPOLYDATA Procedure [1043](#)
- vtkPOLYSHADE Procedure [1044](#)
- vtkPPMREAD Function [1046](#)
- vtkPPMWRITE Procedure [1047](#)
- vtkRECTILINEARGRID Procedure [1048](#)
- vtkRENDERWINDOW Procedure [1049](#)
- vtkSCATTER Procedure [1050](#)
- vtkSLICEVOL Procedure [1053](#)
- vtkSTRUCTUREDGRID Procedure [1055](#)
- vtkSTRUCTUREDPOINTS Procedure [1056](#)
- vtkSURFACE Procedure [1057](#)
- vtkSURFGEN Procedure [1060](#)
- vtkTEXT Procedure [1061](#)
- vtkTVRD Function [1062](#)
- vtkUNSTRUCTUREDGRID Procedure [1063](#)
- vtkWDELETE Procedure [1064](#)

vtkWINDOW Procedure [1065](#)  
vtkWRITEVRML Procedure [1067](#)  
vtkWSET Procedure [1068](#)

## W

WAIT procedure [1070](#)  
waiting, in programs [1070](#)  
warping of images [673](#), [710](#)  
wavevars function  
    with LINKLOAD [529](#)  
WCOPY function [1070](#)  
WDELETE procedure [1072](#), [B-86](#)  
WEOF procedure [1073](#)  
WgAnimateTool procedure [1073](#)  
WgCbarTool procedure [1079](#)  
WgCeditTool procedure [1083](#)  
WgCtTool procedure [1092](#)  
WgIsoSurfTool procedure [1096](#)  
WgMovieTool procedure [1102](#)  
WgOrbit procedure [1108](#)  
WgSimageTool procedure [1109](#)  
WgSliceTool procedure [1113](#)  
WgStripTool procedure [1119](#)  
WgSurfaceTool procedure [1124](#)  
WgTextTool procedure [1130](#)  
WHERE function [1133](#)  
WHEREIN function [1135](#)  
white space, compressing [869](#)  
WIN32 driver  
    256 color mode [B-48](#)  
    color, use of [B-46](#)  
    graphics function codes [B-66](#)  
    video modes [B-46](#)  
WIN32\_PIC\_PRINTER [B-54](#)  
WIN32\_PICK\_FONT function [1136](#)  
WIN32\_PICK\_PRINTER function [1137](#), [B-54](#),  
    [B-57](#)  
window  
    background color of [1198](#), [1238](#)  
    commands [B-45](#)  
    creating [1138](#), [B-2](#)  
    current [1153](#)  
    damage repair [B-2](#)  
    deleting by ID number [B-86](#)  
    exposing [1155](#)  
    hiding [1155](#)  
    IDs (Windows) [B-52](#)  
    IDs (X Windows) [B-85](#)

margin around plot [1249](#)  
pasting into [1145](#)  
positioning plot in [793](#), [801](#)  
printing contents of [1146](#)  
reading  
    DIB data into [1148](#)  
    EMF file into [1149](#)  
selecting [1153](#)  
specifying plot coordinates [1215](#)  
with bitmap graphics [1138](#)  
write DIB data from [1156](#)  
write EMF data from [1157](#)  
X window state [B-67](#)  
WINDOW procedure [1138](#)  
window systems  
    backing store [B-2](#)  
    common features [B-2](#)  
    features of supported systems [B-2](#)  
    X Windows [B-58](#)  
Windows  
    bitmaps [B-49](#)  
    commands from within PV-WAVE  
        [859](#)  
    environment variables [788](#)  
    font [1136](#)  
    graphics window commands [B-45](#)  
    palette [B-48](#)  
    printer dialog [1137](#)  
    resizing graphics [B-44](#)  
    video modes [B-46](#)  
    window IDs [B-52](#)  
WMENU function [1143](#)  
WMF driver  
    24-bit color [B-54](#)  
    color correction [B-54](#)  
    description of [B-53](#)  
    keywords [B-55](#)  
    printing [B-53](#)  
World Databank II dataset [561](#)  
WPASTE function [1145](#)  
WPRINT [1146](#)  
WREAD\_DIB function [1148](#)  
WREAD\_META function [1149](#)  
write mask  
    creating special effects [B-82](#)  
    graphics function interaction [B-83](#)  
    interacts with selected graphics  
        function [B-51](#)  
    using to create special effects [B-50](#)

WRITE\_XBM procedure [1152](#)  
 WRITEU procedure [1151](#)  
 writing  
   *See also* input/output  
   ASCII data [242](#), [250](#), [716](#)  
   CSV data [247](#), [251](#)  
   DIB data [240](#)  
   DIB data from window to file [1156](#)  
   8-bit image data [237](#)  
   fixed format ASCII data [242](#)  
   flushing buffers [367](#)  
   metafile from window to file [1157](#)  
   TIFF image data to a file [256](#)  
   to tape (VMS) [924](#)  
   24-bit image data [238](#)  
   unformatted data [1151](#)  
 WSET procedure [1153](#)  
 WSHOW procedure [1155](#)  
 WWRITE\_DIB function [1156](#)  
 WWRITE\_META function [1157](#)  
 WzAnimate procedure [1159](#)  
 WzBar procedure [1160](#)  
 WzBar3D procedure [1162](#)  
 WzColorEdit procedure [1164](#)  
 WzContour procedure [1167](#)  
 WzExport procedure [1168](#)  
 WzHistogram procedure [1170](#)  
 WzImage procedure [1172](#)  
 WzImport procedure [1174](#)  
 WzMultiView procedure [1176](#)  
 WzPie procedure [1177](#)  
 WzPlot procedure [1179](#)  
 WzPreview procedure [1180](#)  
 WzSurface procedure [1182](#)  
 WzTable procedure [1184](#)  
 WzVariable procedure [1186](#)

## X

X server  
   closing connection [B-61](#)  
 X Window System  
   \$DISPLAY environment variable [B-59](#)  
   client [B-58](#)  
   color translation [B-74](#)  
   colormaps  
     private [B-72](#)  
     shared [B-71](#)  
   damage repair [B-2](#)

DECW\$DISPLAY logical [B-59](#)  
 DEVICE procedure [B-61](#)  
 IDs [B-85](#)  
 keywords [B-61](#)  
 overview [B-58](#)  
 pixmaps [B-75–B-76](#)  
 private colormaps [B-72](#)  
 providing a GUI for application [B-60](#)  
 servers [B-58](#)  
 shared colormaps [B-71](#)  
 X11 socket [786](#)  
 XOR operator [B-52](#), [B-84](#)  
 XYOUTS procedure [1189](#)

## Z

Z-buffer output  
   special effects [684](#), [B-86](#)  
   using [B-86](#)  
 ZOOM procedure [1192](#)  
 zooming  
   3D window, use in [113](#)  
   images [1192](#)  
   reference cube, use of [1098](#), [1115](#)  
 ZROOTS procedure [1194](#)