



PV-WAVE *Advantage*[™]

PV-WAVE Command Language[™]



Reference
Volume 1

Version 4.2 • • •
2860
June 1993



IMSL and Precision Visuals are now one.

Visual Numerics, Inc.

Corporate Headquarters

Suite 400, 9990 Richmond Avenue
Houston, Texas 77042
United States of America
713/784-3131
FAX: 713/781-9260

Boulder, Colorado

6230 Lookout Road
Boulder, Colorado 80301
United States of America
303/530-9000
FAX: 303/530-9329

France

33-1-42-94-19-65
FAX: 33-1-42-94-94-22
33-1-34-51-26-26
FAX: 33-1-34-51-97-69

Germany

49-211-367-7122
FAX: 49-211-367-7100

Japan

81-3-5689-7550
FAX: 81-3-5689-7553

United Kingdom

44-0753-790-600
FAX: 44-0753-790-601

Copyright ©1993 by Visual Numerics, Inc.

The information contained in this document is subject to change without notice.

VISUAL NUMERICS, INC., MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Visual Numerics, Inc., shall not be liable for errors contained herein or for incidental, consequential, or other indirect damages in connection with the furnishing, performance, or use of this material.

All rights are reserved. No part of this document may be photocopied or reproduced without the prior written consent of Visual Numerics, Inc.

Restricted Rights Legend

Use, duplication or disclosure by the US Government is subject to restrictions as set forth in FAR 52.227-19, subparagraph (c)(1)(ii) of DOD FAR SUPP 252.227-7013, or the equivalent government clause for other agencies.

Restricted Rights Notice: The version of PV-WAVE described in this document is sold under a per-machine license agreement. Its use, duplication, and disclosure are subject to the restrictions in the license agreement.

Contents Summary

Volume 1

Preface *iii*

Chapter 1: Functional Summary of Routines *1*

Chapter 2: Procedure and Function Reference (A–P) *29*

Multivolume Index *i*

Volume 2

***Chapter 2: Procedure and Function Reference
(Continued: Q–Z)*** *1*

Chapter 3: Graphics and Plotting Keywords *497*

Chapter 4: System Variables *537*

Chapter 5: Software Character Sets *563*

Appendix A: Picture Index *A-1*

Multivolume Index *i*

Preface

The *PV-WAVE Reference* is a two-volume set that describes the *PV-WAVE Advantage* and CL functions and procedures, keywords, and system variables.

This reference is part of a larger set of documentation; the entire set is shown in Figure I. Start with the *PV-WAVE Tutorial*, and then refer to this reference for details about the basic elements of the *PV-WAVE Command Language*. You will also want to refer to the *PV-WAVE User's Guide for Advantage and CL* and the *PV-WAVE Programmer's Guide for Advantage and CL* for detailed information.

For your convenience, all of the documents shown in Figure I are available online, as well as being available in a hardcopy format. Additional copies of the hardcopy documentation can also be ordered from Visual Numerics, Inc., by calling 800/447-7147.

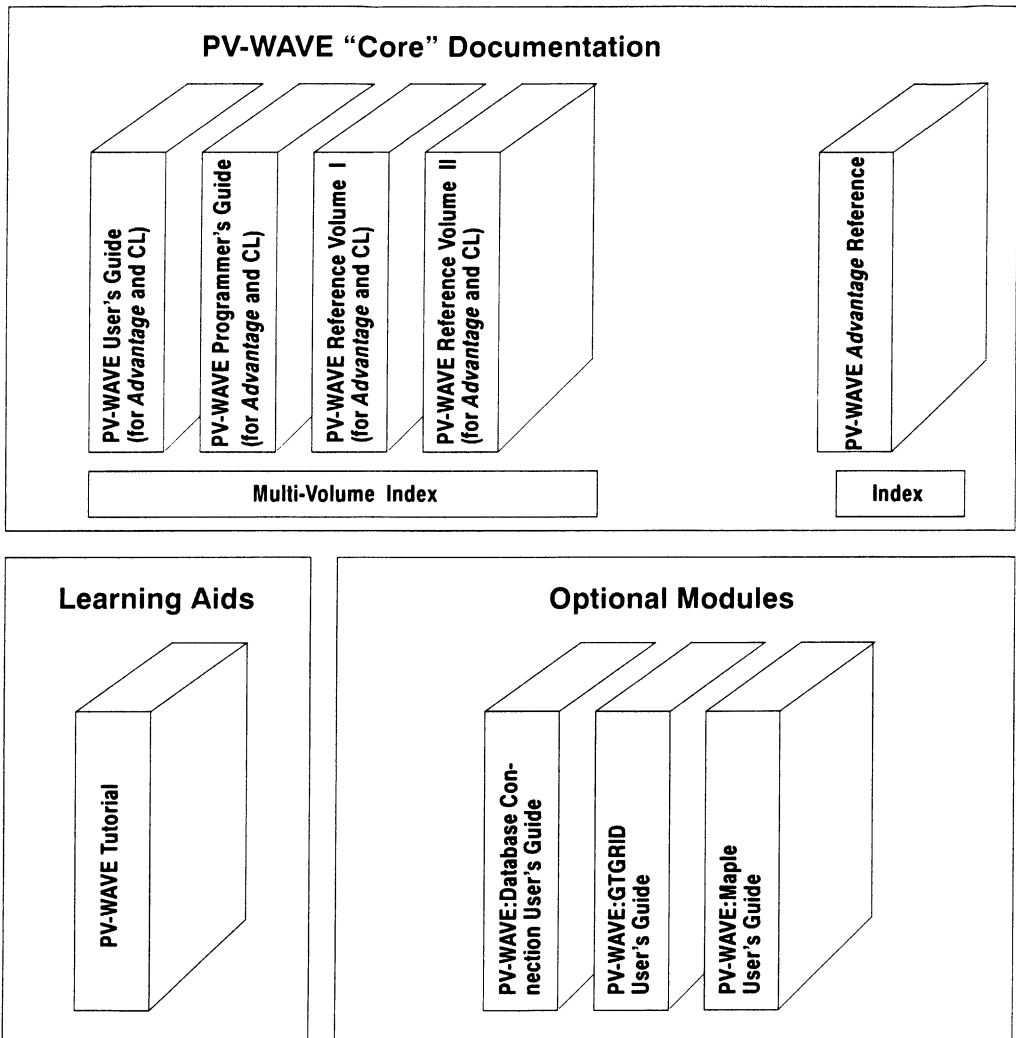


Figure I PV-WAVE documentation set; for more information about any one book you see shown here, refer to its preface, where the contents of each chapter are explained briefly. All documents are available both online and in a hardcopy format. Additional copies of the hardcopy documentation can be ordered by calling Visual Numerics, Inc., at 303/447-7147.

Contents of this Reference

Volume 1 Contains:

- **Preface** — Describes the contents of this guide, lists the typographical conventions used, explains how to use the PV-WAVE documentation set, and explains how to obtain customer support.
- **Chapter 1: Functional Summary of Routines** — A handy 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: A – P** — An alphabetically arranged reference for all PV-WAVE procedures and functions. Most descriptions include one or more examples and cross references to related information.
- **Appendix A: Picture Index** — A graphical index of the illustrations that appear in the PV-WAVE documentation set.
- **Index** — A multivolume index that includes references to the *PV-WAVE User's Guide*, *PV-WAVE Programmer's Guide*, as well as both volumes of the *Reference*.

Volume 2 Contains:

- **Preface** — Describes the contents of this guide, lists the typographical conventions used, explains how to use the PV-WAVE documentation set, and explains how to obtain customer support.
- **Chapter 2 (Continued: Q – Z)** — Procedure and Function Reference.
- **Chapter 3: Graphics and Plotting Keywords** — Describes the keywords that can be used with the PV-WAVE graphics and plotting system routines.

- **Chapter 4: System Variables** – Describes each of the PV-WAVE system variables.
- **Chapter 5: Software Character Sets** – Shows each of the software character sets supported by PV-WAVE.
- **Appendix A: Picture Index** – A graphical index of the illustrations that appear in the PV-WAVE documentation set.
- **Index** – A multivolume index that includes references to the *PV-WAVE User's Guide*, *PV-WAVE Programmer's Guide*, as well as both volumes of the *Reference*.

Typographical Conventions

The following typographical conventions are used in this guide:

- PV-WAVE code examples appear in this typeface. For example:

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

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

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

```
    This command plots air temperature data vs. sulphur dioxide concentration.
```

Comments are used often in this reference to explain code fragments and examples. Note that in actual PV-WAVE code, all comment lines must be preceded by a semicolon (;).

- PV-WAVE commands are not case sensitive. In this reference, 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 widget routines are shown in mixed case (WwMainMenu).
- A \$ at the end of a PV-WAVE line 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.

- The | symbol means “or” when used in a usage line. It is not to be typed. For example, in the following command:

```
result = QUERY_TABLE(table,  
' [Distinct] * | coli [alias] [, ..., coln [alias]] ...
```

the | means use either * or col_i [alias] [, ..., col_n [alias]], but not both.

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

Customer Support

If you have problems unlocking your software or running the license manager, you can talk to a Visual Numerics Customer Support Engineer. The Customer Support group researches and answers your questions about all Visual Numerics products.

Please be prepared to provide Customer Support with the following information when you call:

- The name and version number of the product. For example, **PV-WAVE 4.2** or **PV-WAVE P&C 2.0**.
- Your license number, or reference number if you are an Evaluation site.
- The type of system on which the software is being run. For example, Sun-4, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, SunOS 4.1.3.
- A detailed description of the problem.

The phone number for the Customer Support group is 303/530-5200.

Trademark Information

PostScript is a registered trademark of Adobe Systems, Inc.

QMS QUIC is a registered trademark of QMS, Inc.

HP Graphics Language, HP Printer Control Language, and HP LaserJet are trademarks of Hewlett-Packard Corporation.

Macintosh and PICT are registered trademarks of Apple Computer, Inc.

Open Windows and Sun Workstation are trademarks of Sun Microsystems, Inc.

TEKTRONIX 4510 Rasterizer is a registered trademark of Tektronix, Inc.

OPEN LOOK and UNIX are trademarks of UNIX System Laboratories, Inc.

PV-WAVE, PV-WAVE Command Language, PV-WAVE *Advantage*, and PV-WAVE P&C are trademarks of Visual Numerics, Inc.

OSF/Motif and Motif are trademarks of the Open Software Foundation, Inc.

X Window System is a trademark of the Massachusetts Institute of Technology

Functional Summary of Routines

This chapter lists the following groups of related routines:

- *Array Creation Routines* on page 3
- *Array Manipulation Routines* on page 3
- *Data Connection Routines* on page 4
- *Data Conversion Routines* on page 5
- *Data Extraction Routines* on page 5
- *Date/Time Functions* on page 6
- *File Manipulation Routines* on page 7
- *General Graphics Routines* on page 7
- *General Mathematical Functions* on page 9
- *Help and Information Routines* on page 11
- *Image Display Routines* on page 11
- *Image Processing Routines* on page 12
- *Input and Output Routines* on page 14
- *Operating System Access Routines* on page 15

- *Plotting Routines* on page 16
- *Programming Routines* on page 18
- *Session Routines* on page 19
- *Special Mathematical Functions* on page 19
- *String Processing Routines* on page 20
- *Table Manipulation Functions* on page 20
- *Transcendental Mathematical Functions* on page 20
- *Window Routines* on page 21
- *WAVE Widgets Routines* on page 24
- *Widget Toolbox Routines* on page 25
- *Concurrent Processing Routines* on page 27

Array Creation Routines

- BINDGEN**(dim₁ [, dim₂, ... , dim_n])
Returns a byte array with the specified dimensions, setting the contents of the result to increasing numbers starting at 0.
- BYTARR**(dim₁ [, dim₂, ... , dim_n])
Returns a byte vector or array.
- CINDGEN**(dim₁ [, dim₂, ... , dim_n])
Returns a complex single-precision floating-point array.
- COMPLEXARR**(dim₁ [, dim₂, ... , dim_n])
Returns a complex single-precision floating-point vector or array.
- DBLARR**(dim₁, ... , dim_n)
Returns a double-precision floating-point vector or array.
- DINDGEN**(dim₁, ..., dim_n)
Returns a double-precision floating-point array with the specified dimensions.
- FINDGEN**(dim₁, ..., dim_n)
Returns a single-precision floating-point array with the specified dimensions.
- FLTARR**(dim₁, ..., dim_n)
Returns a single-precision floating-point vector or array.
- INDGEN**(dim₁, ... , dim_n)
Returns an integer array with the specified dimensions.
- INTARR**(dim₁, ... , dim_n)
Returns an integer vector or array.
- LINDGEN**(dim₁, ... , dim_n)
Returns a longword integer array with the specified dimensions.

- LONARR**(dim₁, ... , dim_n)
Returns a longword integer vector or array.
- MAKE_ARRAY**([dim₁, ..., dim_n])
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, dim₁, ..., dim_n)
Forms an array with the given dimensions, filled with the specified scalar value.
- SINDGEN**(dim₁, ... , dim_n)
Returns a string array with the specified dimensions.
- STRARR**(dim₁, ... , dim_n)
Returns a string array.

Array Manipulation Routines

- 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.
- CORRELATE**(x, y)
Standard Library function that calculates a simple correlation coefficient for two arrays.

DETERM(array)

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

HISTOGRAM(array)

Returns the density function of an array.

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.

REBIN(array, dim₁, ..., dim_n)

Returns a vector or array resized to the given dimensions.

REFORM(array, dim₁, ... , dim_n)

Reformats an array without changing its values numerically.

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.

SHIFT(array, shift₁, ... , shift_n)

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.

SMOOTH(array, width)

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

SORT(array)

Sorts the contents of an array.

STDEV(array [, mean])

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

TOTAL(array)

Sums the elements of an input array.

TRANSPOSE(array)

Transposes the input array.

WHERE(array_expr [, count])

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

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_FIXED(filename, var_list)

Reads fixed-formatted ASCII data using a PV-WAVE CL 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_READ_8_BIT(filename, imgarr)
Reads an 8-bit image file.

DC_READ_24_BIT(filename, imgarr)
Reads a 24-bit image file.

DC_WRITE_FIXED(filename, var_list, format)
Writes the contents of one or more PV-WAVE CL 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 CL 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.

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.

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.

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.

STRING(expr₁, ... , expr_n)
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.

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 !Week-end_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 CL 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 CL Date/Time variables into string data.

DT_TO_STR, dt_var, [, dates] [, times]

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

DT_TO_VAR, dt_value

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

DTGEN(dt_start, dimension)

Returns an array of PV-WAVE CL 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 CL 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 CL Date/Time variables.

STR_TO_DT(date_strings [,time_strings])

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

FLUSH, unit₁, ..., unit_n

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

FREE_LUN, unit₁, ..., unit_n

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.

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.

File Manipulation Routines

CLOSE[, unit₁, ... , unit_n]

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.

General Graphics Routines

CURSOR, x, y [, wait]

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

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.

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.

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.

POLYCONTOUR, filename

Standard Library procedure that shades enclosed contours with specified colors.

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 CL graphics procedures.

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.

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(v₁, v₂)

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

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.

DETERM(array)

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

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(p₁, p₂)

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

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.)

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 [, dim₁, ... , dim_n])

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

RANDOMU(seed [, dim₁, ... , dim_n])

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.

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.

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.

Help and Information Routines

DOC_LIBRARY [, name]

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

INFO, expr₁, ... , expr_n

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

Image Display Routines

ADJCT

Standard Library procedure that lets you interactively change the range and lower limit of the current color table.

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, i₀, i₁, i₂, o₀, o₁, o₂, 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 CL color table.

MODIFYCT, table, name, red, green, blue

Standard Library procedure that lets you replace one of the PV-WAVE CL 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, v₁, v₂, v₃ [, start]

Loads the display color translation tables from the specified variables.

TVRD(x₀, y₀, n_x, n_y [, channel])

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

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

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.

Image Processing Routines

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(size_x, size_y [, xverts, yverts])

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

DIGITAL_FILTER(flow, fh_{igh}, 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.

MOVIE, images [, rate]

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

POLY_2D(array, coeff_x, coeff_y [, interp [, dim_x ,..., dim_y]])

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, dim₁, ..., dim_n)

Returns a vector or array resized to the given dimensions.

REFORM(array, dim₁, ... , dim_n)

Reformats an array without changing its values numerically.

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, shift₁, ... , shift_n)

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, variable₁, ... , variable_n

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 CL 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, expr₁, ... , expr_n

PRINT performs output to the standard output stream (PV-WAVE CL file unit -1).

PRINTF, unit, expr₁, ... , expr_n

PRINTF requires the output file unit to be specified.

READ, var₁, ..., var_n

Read input from the standard input stream into PV-WAVE CL variables.

READF, unit, var₁, ... , var_n

Read input from a file into PV-WAVE CL variables.

READU, unit, var₁, ... , var_n

READU reads binary (unformatted) input from a specified file. (No processing of any kind is done to the data.)

REWIND, unit

(VMS Only) Rewinds the tape on the designated PV-WAVE CL tape unit.

SKIPF, unit, files

SKIPF, unit, records, r

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

TAPRD, array, unit [, byte_reverse]

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

TAPWRT, array, unit [, byte_reverse]

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

WRITEU, unit, expr₁, ... , expr_n

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

DELETE_SYMBOL, name

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

DELLOG, logname

(VMS Only) Deletes a logical name.

ENVIRONMENT()

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

GETENV(name)

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

GET_SYMBOL(name)

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

LINKNLOAD(object, symbol [, param₁, ..., param_n])

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.

Operating System Access Routines

CALL_UNIX(p₁ [, p₂, ... , p₃₀])

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

CD [, directory]

Changes the current working directory.

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

SETLOG, logname, value
(VMS Only) Defines a logical name.

SET_SYMBOL, name, value
(VMS Only) Defines a DCL interpreter symbol for the current process.

SPAWN [, command [, result]]
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)
(VMS Only) Searches the VMS name tables for a specified logical name and returns the equivalence string(s) in a PV-WAVE CL variable.

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

Plotting Routines

AXIS [[[, x], y], z]
Draws an axis of the specified type and scale at a given position.

CONTOUR, z [, x, y]
Draws a contour plot from data stored in a rectangular array.

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.

OPLOT, x [, y]
Plots vector data over a previously drawn plot.

OPLOTERR, x, y, error [, psym]
Standard Library procedure that overplots symmetrical error bars on any plot already output to the display device.

PLOT, x [, y]
PLOT produces a simple XY plot.

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.

POLYCONTOUR, filename

Standard Library procedure that shades enclosed contours with specified colors.

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.

Programming Routines

BREAKPOINT, file, line

Lets you insert and remove breakpoints in programs for debugging.

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_i ,..., function_n

Deletes one or more compiled functions from memory.

DELPROC, procedure_i ,..., procedure_n

Deletes one or more compiled procedures from memory.

DELSTRUCT, structure_i ,..., structure_n

Deletes one or more named structure definitions from memory.

DELVAR, v₁, ... ,v_n

Deletes variables from the main program level.

EXIT

Exits PV-WAVE CL 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 non-zero value.

MESSAGE, text

Issues error and informational messages using the same mechanism employed by built-in PV-WAVE CL 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 CL 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 CL user-written procedure or function.

ON_IOERROR, label

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

RECALL

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.

SIZE(expr)

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

STOP [, expr₁, ..., expr_n]

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.

TVMENU(text [, title, x, y])

Standard Library procedure that creates an interactive menu.

WAIT, seconds

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

Session Routines

INFO, expr₁, ... , expr_n

Displays information on many aspects of the current PV-WAVE CL 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 CL objects saved in a file by the SAVE procedure.

SAVE [, var₁, ... , var_n]

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.

STRLEN(expr)

Returns the length of the input parameter.

STRLOWCASE(string)

Converts a copy of the input string to lowercase letters.

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.

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_i [alias], ..., var_n [alias] ')

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

QUERY_TABLE(table, ' [Distinct] * | col_i [alias] [, ..., col_n [alias]] [Where cond] [Group By col_g_i [,... col_g_n]] | [Order By col_o_i [direction][,...col_o_n [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.

Window Routines

TVMENU(text [, title, x, y])

Standard Library procedure that creates an interactive menu.

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.

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.

Coordinate Conversion Routines

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.

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).

GRID_SPHERE(points, grid_x, grid_y)

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

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(xtmp, ytmp, ztmp)

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

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

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.

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.

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.

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.)

Volume Manipulation Routines

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

WwButtonBox(parent, labels, callback)

Creates a horizontally or vertically oriented box containing push buttons.

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.

WwGetValue(widget)

Returns a specific value for a given 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.

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.

WwOptionMenu(parent, label, items)

Creates an option menu.

WwPopupMenu(parent, items)

Creates a popup menu.

WwRadioBox(parent, labels, callback)

Creates a box containing radio buttons.

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.

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.

WwToolBox(parent, labels, callback)

Creates an array of graphic buttons (icons).

Widget Toolbox Routines

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 (OLIT or 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.

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.

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.

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.

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.

EXEC_ON_SELECT, luns, commands

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

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.

Concurrent Processing Routines

SELECT_READ_LUN, luns

Waits for input on any list of logical unit numbers.

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 `wave/lib/std`.

Users' Library Routines

Additional routines that have been contributed by PV-WAVE users comprise the Users' Library. For a list of such routines distributed with your version of PV-WAVE, see the `wave/lib/user` subdirectory.

Users' Library routines are not covered in this chapter; use the documentation available in the `.pro` source file for each routine in the `wave/lib/user` subdirectory.

For more information, see *The Users' Library* on page 255 of 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 or string, the result is double-precision floating-point. All other data types produce a result with the same data type as x .

Examples

```
x = [-1, 2, 3, -4, 5]
PRINT, ABS(x)
    1    2    3    4    5

x = [4 + 3i]
PRINT, ABS(x)
    5.0
```

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 π .

Example

```
x = ACOS(1)
PRINT, x
0
```

See Also

COS

For a list of other transcendental functions, see *Transcendental Mathematical Functions* on page 20.

ADD_EXEC_ON_SELECT Procedure

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

SELECT_READ_LUN, EXEC_ON_SELECT,
DROP_EXEC_ON_SELECT

ADDSYSVAR Procedure

Standard Library procedure that lets you define new system variables. Should only be used with Version 1 of PV-WAVE.

Usage

ADDSYSVAR, *name*, *type* [, *string_length*]

Input Parameters

name — The name of the system variable. Must be a scalar string starting with the ! character.

type — The type of the system variable expressed as a one-character string. The following are valid data type values:

Value	Description
B	Byte
D	Double-precision floating-point
F	Floating-point
I	Integer
L	Longword
S	String

string_length — Not used. (This parameter is present only for compatibility with Version 1 of PV-WAVE.)

Keywords

None.

Discussion

ADDSYSVAR lets you define constants or strings that will be used repeatedly as system variables, making them easily accessible. This procedure takes the name and initialized values and passes them directly to the DEFSYSV procedure.

The initialized values are set as zero for the appropriate type (i.e., 0.0 for floating-point or 0L for long integer); if the type is string, they are set as the null string (' '). The system variable will then need to be assigned to the appropriate value.

Note 

ADDSYSVAR is present for backward compatibility and should be used only with code taken from Version 1 of PV-WAVE. Users of more recent versions of PV-WAVE should use the DEFSYSV procedure instead.

Example

```
ADDSYSVAR, '!cm', 'f'  
!cm = 2.54  
inches = 20.3  
value_in_cms = inches * !cm  
PRINT, value_in_cms  
51.5620
```

See Also

DEFSYSV

For more information, see Chapter 4, *System Variables*.

ADJCT Procedure

Standard Library procedure that lets you interactively change the range and lower limit of the current color table.

Usage

ADJCT

Parameters

None.

Keywords

None.

Discussion

ADJCT works only on displays with window systems. It creates an interactive window that lets you use the mouse to change the range and lower limit of the current color table. This window is shown in Figure 2-1.

The ADJCT window contains the following items:

- **Color Bar** — Displays the current color table.
- **Graph** — Displays the color intensity versus pixel value.
- **Control Button Area** — Contains buttons for manipulating the display (labeled Ramp, Segments, and Draw) and for getting Help.

To select a button, click on it with the left mouse button.

Click the **Help** button to display information about using the other buttons.

To exit ADJCT, click the right mouse button.

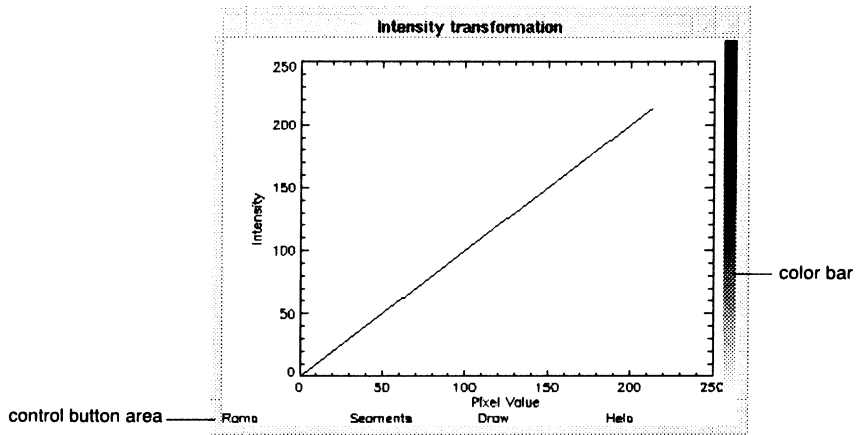


Figure 2-1 The ADJCT window lets you use the mouse to change the range and lower limit of the current color table.

Example

```

b = FINDGEN(200)
y = SIN(b/5)/EXP(b/50)
YY = y # y
LOADCT, 5
TVSCL, YY
ADJCT

```

— User modifies the color table and exits the procedure.

—

```

TVLCT, r, g, b, /Get
SAVE, Filename='test_colortable', r, g, b
LOADCT, 9
RESTORE, 'test_colortable'
TVLCT, r, g, b

```

See Also

C_EDIT, COLOR_EDIT, COLOR_PALETTE, HLS, HSV, LOADCT, MODIFYCT, PALETTE, PSEUDO, STRETCH, TVLCT, WgCbarTool, WgCeditTool, WgCtTool

For background information about color systems, see *Understanding Color Systems* on page 305 of the *PV-WAVE User's Guide*.

ALOG Function

Returns the natural logarithm of x .

Usage

$result = \text{ALOG}(x)$

Input Parameters

x — The expression that is evaluated. Must be > 0 . 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 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 *Transcendental Mathematical Functions* on page 20.

ALOG10 Function

Returns the logarithm to the base 10 of x .

Usage

$result = \text{ALOG10}(x)$

Input Parameters

x – The expression that is evaluated. Must be > 0 . 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 types are converted to single-precision floating-point and yield a floating-point result.

ALOG10 handles complex numbers in the following way:

$$\text{Alog10}(x) = \text{Complex}(\log_{10}(|x|), \arctan(x))$$

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 *Transcendental Mathematical Functions* on page 20.

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 *Transcendental Mathematical Functions* on page 20.

ASSOC Function

Associates an array structure with a file, allowing random access input and output.

Usage

```
result = ASSOC(unit, array_structure [, offset])
```

Input Parameters

unit — The PV-WAVE file unit to associate with *array_structure*.

array_structure — An expression that defines the data type and structure 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.



Offset is useful for skipping past descriptive header blocks in files.

Returned Value

result — A variable that associates the array structure with the file.

Keywords

None.

Discussion

ASSOC provides a basic method of random access input/output in PV-WAVE. 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.

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.  
TV, 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 *How Data is Transferred into Associated Variables* on page 214 of the *PV-WAVE Programmer's Guide*.

See Also

OPENR, OPENW

For background information, see Chapter 10, *Programming with PV-WAVE*, in the *PV-WAVE Programmer's Guide*.

ATAN Function

Calculates the arctangent of the input variable(s).

Usage

result = ATAN(*x* [, *y*])

Input Parameters

x – The tangent of the desired angle.

y – The divisor of *x*.

Returned Value

result – The arctangent of *x* (or optionally of *x/y*).

Keywords

None.

Discussion

ATAN returns the radian angle whose tangent is *x*. If the optional parameter *y* is used, then ATAN returns the radian angle for the tangent of *x/y*. The parameters for the ATAN function are the exact reverse of the usual definition of tangent as *y/x*.

The range of ATAN is between $-\pi/2$ and $\pi/2$ when *x* is the only parameter, and $-\pi$ and π when the *y* parameter is specified.

An “arithmetic error” warning is displayed if $x=y=0$.

Example

```
x = [0.1, 0.2, 0.3]
PRINT, ATAN(x, x)
      0.785398 0.785398 0.785398
```

See Also

ACOS, ASIN, COS, SIN, TAN

For a list of other transcendental functions, see *Transcendental Mathematical Functions* on page 20.

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. May be any type except string.

dim — 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 PV-WAVE 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* — Scalars giving the coordinates of the new AXIS.

Input 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 plotting keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords*.

Channel	Thick	XTitle	YType
Charsize	Tickformat	XType	
Charthick	Ticklen		ZAxis
Clip	Title	YAxis	ZCharsize
Color		YCharsize	ZGridstyle
Data	XAxis	YGridstyle	ZMargin
Device	XCharsize	YMargin	ZMinor
Font	XGridstyle	YMinor	ZRange
Gridstyle	XMargin	YNozero	ZStyle
Noclip	XMinor	YRange	ZTickformat
Nodata	XRange	YStyle	ZTicklen
Noerase	XStyle	YTickformat	ZTickname
Normal	XTickformat	YTicklen	ZTicks
Position	XTicklen	YTickname	ZTickv
Save	XTickname	YTicks	ZTitle
Subtitle	XTicks	YTickv	ZValue
T3d	XTickv	YTitle	

Output Keywords

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.

For more information and an illustration, see *Specifying the Location of the Plot* on page 85 of the *PV-WAVE User's Guide*.

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, see *Drawing Additional Axes on Plots* on page 87 of the *PV-WAVE User's Guide*.

BESELI Function

Calculates the Bessel I function for the input parameter.

Usage

result = BESELI(*x* [, *n*])

Input Parameters

x – The expression that is evaluated. Must be > 0.

n – The order to which the BESELI function is calculated. Must be ≥ 0. The default is 0.

Returned Value

result – The Bessel I function for *x*. Has 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 three 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. Must be > 0.

n — The order to which the BESELJ function is calculated. Must be ≥ 0. The default is 0.

Returned Value

result — The Bessel J function for *x*. It is floating-point, 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 three 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 .

Note 

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. Must be > 0.

n — The order to which the BESELY function is calculated. Must be ≥ 0. The default is 0.

Returned Value

result — The Bessel Y function for *x*. Has 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 $n = 0, 1, 2, \dots$



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. 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 below). 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 below). 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*₁ [, *dim*₂, ..., *dim*_{*n*})

Input Parameters

*dim*_{*i*} — 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:

$$\text{array}(i) = \text{BYTE}(i \text{ MOD } 256)$$

$$\text{for } i = 0, 1, \dots, \left(\prod_{j=1}^n D_j - 1 \right).$$

Keywords

None.

Discussion

Each element of the result array is set to the value of its one-dimensional subscript.

Example 1

```
a = BINDGEN(4, 2)
    Create 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

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 — The line number at which the breakpoint is to occur.

Input Keywords

Clear — Removes the breakpoint specified by *file* and/or *line*. Note that *file* is an optional parameter when used with the *Clear* keyword.

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 at line 3 in the current routine.

```
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`

For more information on the `.CON` executive command, see *Using .CON* on page 29 of the *PV-WAVE User'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_i — A vector (one-dimensional array) variable. If additional vectors are specified, they must contain the same number of elements as *var_i*. The input variable(s) can be of any PV-WAVE data type.

alias — 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.

Keywords

None.

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`.

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
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

Note 

The */Structure* keyword is used because tables are represented in PV-WAVE as an array of structures. For more information on this, see Chapter 8, *Creating and Querying Tables*, in the *PV-WAVE User's Guide*.

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

QUERY_TABLE, UNIQUE

For more information on BUILD_TABLE, see Chapter 8, *Creating and Querying Tables*, in the *PV-WAVE User's Guide*.

For information on reading data into PV-WAVE variables, see Chapter 8, *Working with Data Files*, in the *PV-WAVE Programmer's Guide*.

BYTARR Function

Returns a byte vector or array.

Usage

result = BYTARR(*dim*₁ [, *dim*₂, ... , *dim*_{*n*})

Input Parameters

*dim*_{*i*} — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result — A one-dimensional or multi-dimensional byte array.

Input 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*₁, ... , *dim*_{*n*}])

This form is used to extract data.

Input Parameters

expr — The expression to be converted, or from which to extract data.

offset — (Used to extract data only.) 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*_{*i*} — (Used to extract data only.) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result — If converting, returns a copy of *expr* converted to byte data type. The result has the same size and structure (scalar or array) as *expr*.

If extracting, returns a copy of only part of *expr* — the part that is defined by the *offset* and *dimension* parameters. The result has the size and structure of the specified *dimensions* and is of byte data type. If no *dimensions* are specified, the result is scalar.

Keywords

None.

Discussion – Conversion Usage

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 *Extracting Fields* on page 37 of the *PV-WAVE Programmer's Guide*.

BYTEORDER Procedure

Converts integers between host and network byte ordering. Can also be used to swap the order of bytes within both short and long integers.

Usage

BYTEORDER, *variable*₁, ... , *variable*_{*n*}

Input Parameters

*variable*_{*i*} — A variable (see Discussion below).

Output Parameters

*variable*_{*i*} — A variable (see Discussion below).

Input 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 (B₀,B₁,B₂,B₃), to (B₃, B₂,B₁, B₀).

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 *External Data Rep-*

resentation (XDR) Files on page 205 of 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_i* 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='(Z9)', 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='(Z9)', 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.

Input 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. If *Top* is not specified, 255 is used.

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

To scale an array of floats to byte values, you might enter the following:

```
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
    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 $0 < x < 255$.

```
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, BYTSCL(b, Min = 50, Max = 70), 1  
  Display the contrast enhanced image at right side of window.
```

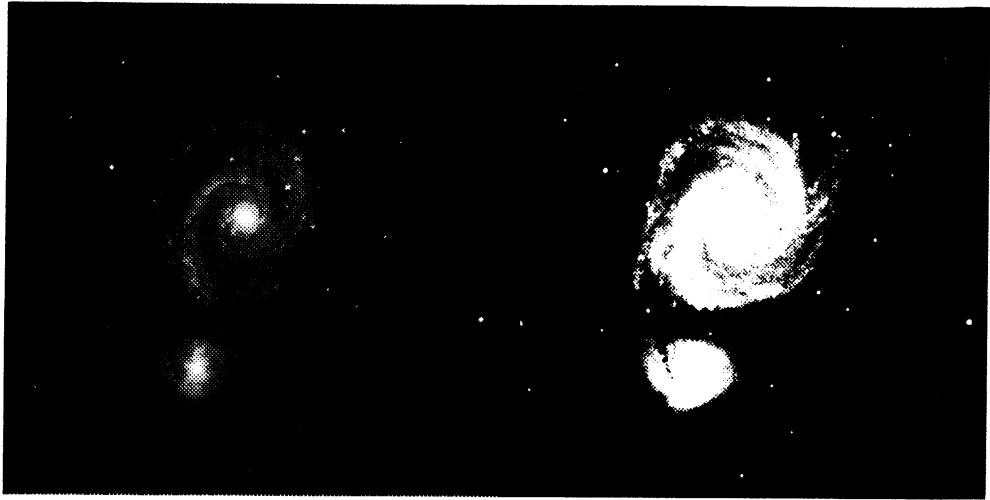


Figure 2-2 Galaxy image before (left) and after (right) contrast enhancement.

See Also

BYTE, BYTARR, BINDGEN

For more information on expressions, see *Type and Structure of Expressions* on page 34 of the *PV-WAVE Programmer's Guide*.

CALL_UNIX Function

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

Usage

result = CALL_UNIX(*p*₁ [, *p*₂, ... , *p*₃₀])

Input Parameters

*p*_{*i*} — A variable of any type. At least one parameter must be passed, but can be up to 30. 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. Cannot be -1, since -1 is reserved to indicate failure.

Input 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_smpl_reply`, or `w_cmpnd_reply` to send values and parameters back to PV-WAVE.

For information on these C routines, see Chapter 13, *Interapplication Communication*, in the *PV-WAVE Programmer'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

For examples, see Chapter 13, *Interapplication Communication*, in the *PV-WAVE Programmer's Guide*.

See Also

ON_IOERROR, UNIX_LISTEN, UNIX_REPLY

CD Procedure

Changes the current working directory.

Usage

CD [, *directory*]

Input Parameters

directory — If present, it should be 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.

Output Keywords

Current — Used to create 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 PUSHD, POPD, and PRINTD procedures, which maintain a directory stack and call CD to change directories, provide a convenient interface to CD.

Examples

On a UNIX system, to change the current working directory to /usr/home/mydata, enter:

```
WAVE> CD, '/usr/home/mydata'
```

On a VMS system, to change the current working directory to SYS\$SYSDEVICE:[MYDATA], enter:

```
WAVE> CD, 'SYS$SYSDEVICE:[MYDATA]'
```

On either system, to move to the home directory, enter:

```
WAVE> CD, ''
```

See Also

!Dir, !Path, FILEPATH, PUSHD, POPD, PRINTD

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 – 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)
```

Input 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 2-3.

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.

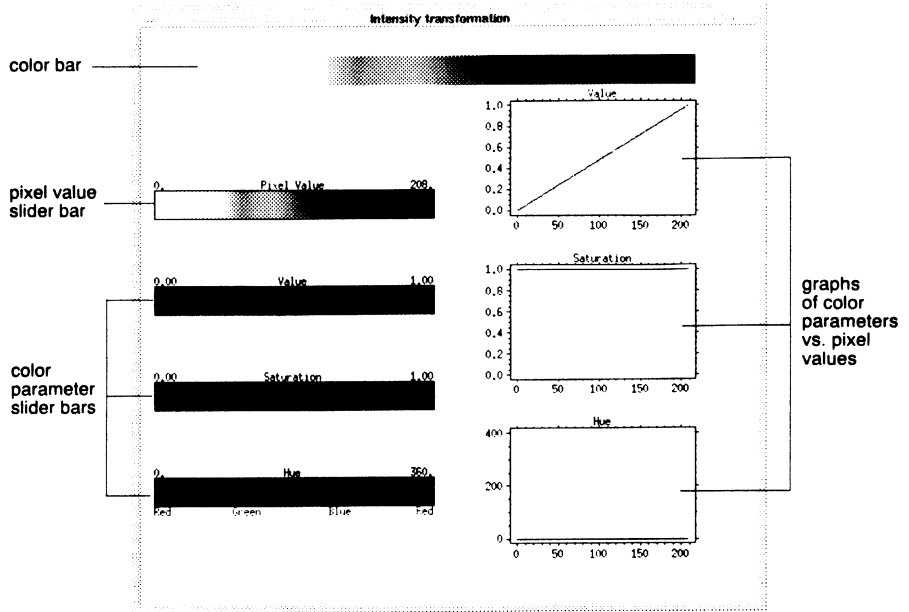


Figure 2-3 The C_EDIT window lets you use the mouse to create a new color table based on either the HLS or HSV color system.

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. (Note that 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.

Note that 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

ADJCT, COLOR_CONVERT, COLOR_EDIT,
COLOR_PALETTE, HLS, HSV, LOADCT, MODIFYCT,
PALETTE, PSEUDO, STRETCH, TVLCT, WgCbarTool,
WgCeditTool, WgCtTool

For background information about color systems, see *Understanding Color Systems* on page 305 of 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.

Input Keywords

Ax — The angle, in degrees, at which to rotate the data around the X axis. The default is -60.0.

Note

The data is rotated *Az* degrees about the Z axis first, *Ay* degrees about the Y axis second, and *Ax* degrees about the X axis last.

Ay — The angle, in degrees, at which to rotate the data around the Y axis. The default is 0.0.

Az — The angle, in degrees, at which to rotate the data around the Z axis. The default is 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 PV-WAVE plot window in device coordinates. The default is 640.

Winy — The Y size of the PV-WAVE plot window in device coordinates. The default is 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

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*.)

Examples

```
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 = BYTSC(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 in \$WAVE_DIR/demo/ar1: grid_demo4, grid_demo5, sphere_demo1, sphere_demo2, sphere_demo3, vol_demo2, vol_demo3, and vol_demo4.

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.

CHECK_MATH Function

Returns and clears the accumulated math error status.

Usage

```
result = CHECK_MATH([print_flag, message_inhibit])
```

Input Parameters

print_flag – 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 – 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.)



On machines that do not implement the IEEE standard for floating-point math, CHECK_MATH does not properly maintain an accumulated error status.

Input 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 error 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]
    Second element should cause a divide-by-zero error.

junkstatus = CHECK_MATH(1, 0, Trap=1)
    Clear 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, see *Detection of Math Errors* on page 263 of the *PV-WAVE Programmer's Guide*. See also the section *Hardware-dependent Math Error Handling* on page 268 of the *PV-WAVE Programmer's Guide*.

CINDGEN Function

Returns a complex single-precision floating-point array.

Usage

```
result = CINDGEN(dim1 [, dim2, ..., dimn])
```

Input Parameters

dim_i — The dimensions of the result. May be any scalar expression. 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:

$$\text{array}(i) = \text{COMPLEX}(i, 0)$$

$$\text{for } i = 0, 1, \dots, \left(\prod_{j=1}^n D_j - 1 \right)$$

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, DINDGEN,
FINDGEN, INDGEN, LINDGEN, SINDGEN

CLOSE Procedure

Closes the specified file units.

Usage

CLOSE[, *unit*₁, ... , *unit*_{*n*}]

Input Parameters

*unit*_{*i*} – The file units to close.

Input 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

GET_LUN, FREE_LUN, OPENR, OPENU, OPENW,
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.

Input Keywords

One of the following *keywords* is required:

HLS_RGB — Convert from HLS (Hue, Lightness, Saturation) to RGB (Red, Green, Blue).

HSV_RGB — Convert from HSV (Hue, Saturation, Value) to RGB.

RGB_HLS — Convert from RGB to HLS.

RGB_HSV — Convert from RGB to HSV.

Discussion

RGB 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 background information about color systems, see *Understanding Color Systems* on page 305 of 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 — 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)
```

Input 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

COLOR_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 2-3.

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.



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.

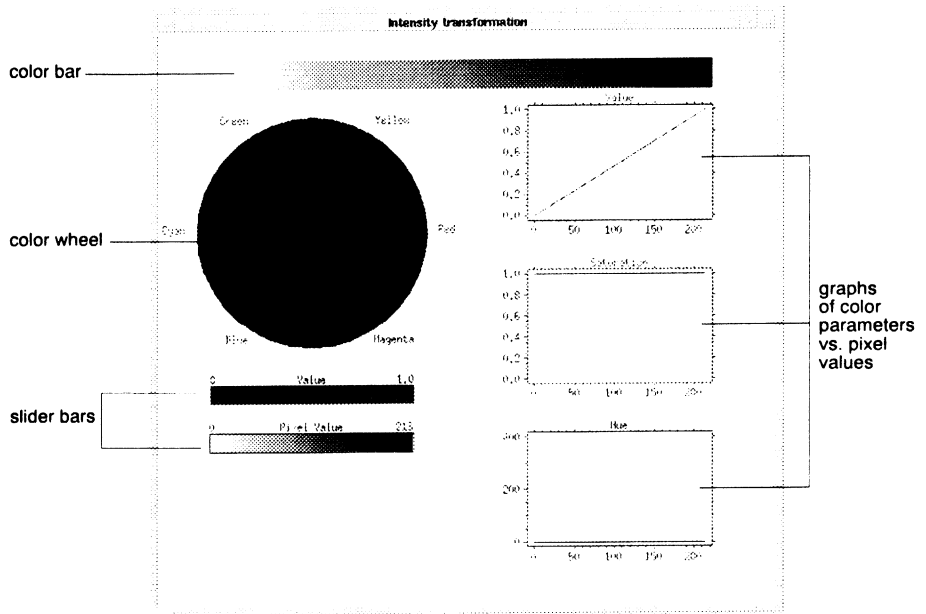


Figure 2-4 The COLOR_EDIT window lets you use the mouse to create a new color table based on either the HLS or HSV color system.

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. (Note that 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.

Note that 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 84.

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

ADJCT, C_EDIT, COLOR_CONVERT, COLOR_PALETTE, HLS, HSV, LOADCT, MODIFYCT, PALETTE, PSEUDO, STRETCH, TVLCT, WgCbarTool, WgCeditTool, WgCtTool

For background information about color systems, see *Understanding Color Systems* on page 305 of 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 is shown in Figure 2-5.

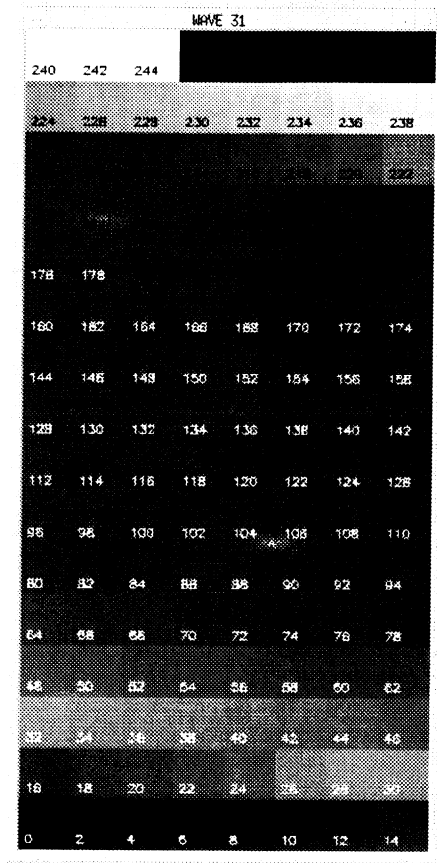


Figure 2-5 The COLOR_PALETTE window displays every other color in the current color table, along with the corresponding numerical value or color table index. The five black cells in the upper-right corner of the window represent colors that are not available to PV-WAVE because they have been reserved by some other 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 *Determining the Number of Available Plot Colors* on page 322 of the *PV-WAVE User's Guide*.

COMPILE Procedure

Saves compiled user-written procedures and functions in a file.

Usage

COMPILE, *routine*₁ [, ..., *routine*_{*n*}]

Input Parameters

*routine*_{*i*} — A string containing the name of the compiled function or procedure that you want to save.

Input 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 that support PV-WAVE.

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.

Saved compiled routines can be executed directly from the operating system level. 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 on a UNIX system, enter:

```
setenv WAVE_FEATURE_TYPE RT
```

On a VMS system, enter:

```
DEFINE WAVE_FEATURE_TYPE RT
```

Then, you can execute compiled PV-WAVE 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.

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, 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.

```
WAVE> .RUN log_plot
      Compile the procedure.

WAVE> COMPILE, 'log_plot'
      Save the compiled procedure.

WAVE> $!s log_plot*
log_plot.cpr  log_plot.pro
      The file log_plot.cpr is created. This file contains the compiled
      procedure.

WAVE> DELPROC, 'log_plot'
      Delete the log_plot procedure that is currently in memory.

WAVE> log_plot
      Run the compiled procedure log_plot.cpr.

WAVE> EXIT
      Exit PV-WAVE, and return to the operating system prompt.

% wave -r log_plot
      Run 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 command:

```
WAVE> .RUN comtool
        Compile the application.
```

```
WAVE> COMPILE, 'ComTool', 'CbuttonCB', $  
      'CommandOK', 'CommandDone', $  
      Filename = 'comtool'
```

Save the compiled procedures in the file comtool.cpr.

```
WAVE> $!s comtool*
```

```
comtool.cpr  comtool.pro
```

The file comtool.cpr is created. This file contains the compiled procedures.

```
WAVE> EXIT
```

```
% wave -r comtool
```

Run 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

SAVE, RESTORE

See also *Using PV-WAVE in Runtime Mode* on page 41 of the *PV-WAVE User's Guide* for more information.

COMPLEX Function

Converts an expression to complex data type.

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

Usage

$result = \text{COMPLEX}(real [, imaginary])$

This form is used to convert data.

$result = \text{COMPLEX}(expr, offset, dim_1 [, dim_2, \dots, dim_n])$

This form is used to extract data.

Input Parameters

real – (Used to convert data only.) Scalar or array to be used as the real part of the complex result.

imaginary – (Used to convert data only.) 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.

expr – (Used to extract data only.) The expression to be converted, or from which to extract data.

offset – (Used to extract data only.) The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

dim_i – (Used to extract data only.) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result – If converting, *result* is of type complex with the size and structure of *result* determined by the size and structure of *real* and *imaginary*. If either or both of the parameters are arrays, the result will be an array, following the same rules as standard PV-WAVE operators.

If extracting, *result* is of type complex with the size and structure of *result* determined by the *dimension* parameters.

Keywords

None.

Discussion – Conversion Usage

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 be jumped 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 *Extracting Fields* on page 37 of the *PV-WAVE Programmer's Guide*.

COMPLEXARR Function

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

Usage

result = COMPLEXARR(*dim*₁ [, *dim*₂, ... , *dim*_{*n*}])

Input Parameters

*dim*_{*i*} – 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.

Input Keywords

Nozero – Normally, COMPLEXARR sets every element of the result to zero. If *Nozero* is nonzero, this zeroing is not performed, thereby causing COMPLEXARR to execute faster.

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.

Input 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 *Defining Color and Shading* on page 198 of the *PV-WAVE User's Guide*.

Decal — A 2D array of bytes whose elements correspond to indices into the arrays of material properties. For more information, see the section *Decals* on page 201 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 *Ambient Component* on page 199 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 *Diffuse Component* on page 198 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 *Transmission Component* on page 199 of the *PV-WAVE User's Guide*.

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 $Z = -0.5$. The default is `Radius=0.0`.

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* on page 201 of the *PV-WAVE User's Guide*.

Discussion

A CONE is used by the RENDER function to render conic objects, such as caps on axes. It is defined by default to be centered at the origin with a height of 1.0, and to have 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.

Examples

```
T3D, /Reset, Rotate=[90, 0., 0]
c = CONE(Radius=0.33, Transform=!P.T)
TV, RENDER(c)
```

See Also

CYLINDER, MESH, RENDER, SPHERE, VOLUME

For more information, see *Ray-tracing Rendering* on page 196 of 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.

Input 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 *Efficiency and Accuracy of Interpolation* on page 170 of the *PV-WAVE User's Guide*.

Example 1

Here's what a mandril image looks like before and after using the following commands:

```
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
```

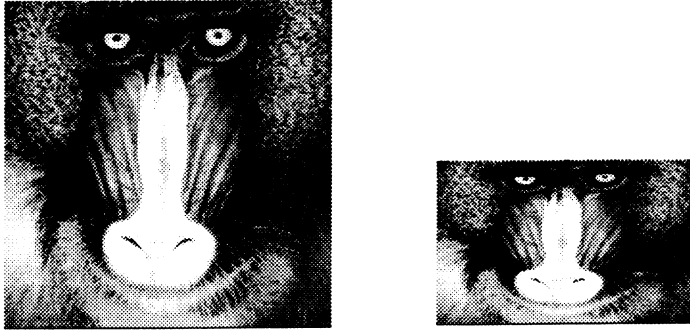


Figure 2-6 CONGRID has been used to shrink this 512-by-512 mandril image to one measuring 400-by-256.

Example 2

```
x = DIST(100)
new_x = CONGRID(x, 500, 200)
TVSCL, x
ERASE
TVSCL, new_x
```

See Also

REBIN, BILINEAR

CONJ Function

Returns the complex conjugate of the input variable.

Usage

result = CONJ(*x*)

Input Parameters

x — The variable that is evaluated. Can be a scalar or an array.

Returned Value

result — The complex conjugate of *x*.

Keywords

None.

Discussion

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 the input value *x* is not complex, then CONJ converts it to complex data type, with the real part described by *x* and the imaginary part set to 0.

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, IMAGINARY

CONTOUR Procedure

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

Usage

```
CONTOUR, z [, x, y]
```

Input Parameters

z — A two-dimensional array containing the values that make up the contour surface.

x — A vector or two-dimensional array specifying the *x* coordinates for the contour surface.

y — A vector or two-dimensional array specifying the *Y* coordinates for the contour surface.

Input Keywords

CONTOUR keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords*.

Background	Noclip	XStyle	YTitle
Channel	Nodata	XTickformat	YType
Charsize	Noerase	XTicklen	
Charthick	Normal	XTickname	ZAxis
Clip	Overplot	XTicks	ZCharsize
Color	Path_Filename	XTickv	ZGridstyle
C_Annotation	Position	XTitle	ZMargin
C_Charsize	Spline	XType	ZMinor
C_Colors	Subtitle		ZRange
C_Labels	T3d	YCharsize	ZStyle
C_Linestyle	Thick	YGridstyle	ZTickformat
C_Thick	Tickformat	YMargin	ZTicklen
Data	Ticklen	YMinor	ZTickname
Device	Title	YRange	ZTicks
Follow		YStyle	ZTickv
Font	XCharsize	YTickformat	ZTitle
Gridstyle	XGridstyle	YTicklen	ZValue
Levels	XMargin	YTickname	
Max_Value	XMinor	YTicks	
NLevels	XRange	YTickv	

Discussion

If x and y 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 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}).

If y is a vector, each element of y specifies the Y coordinate for a row of z . 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}).

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_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 Advantage 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.
```

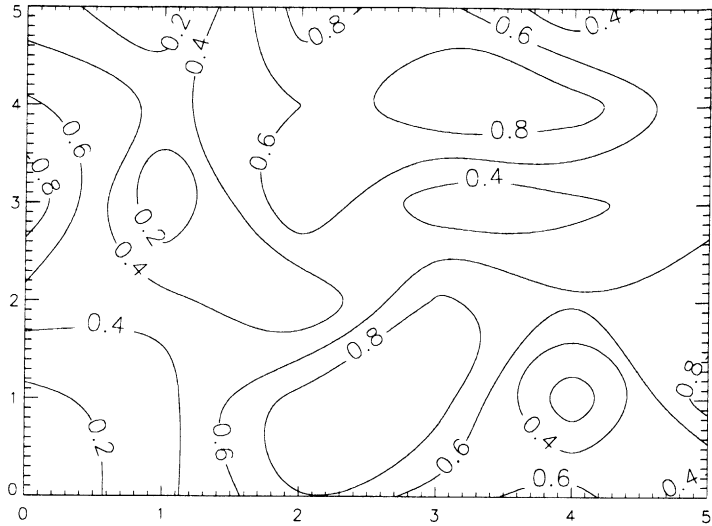


Figure 2-7 Contour plot of random data.

See Also

IMAGE_CONT, SURFACE, SHOW3

For more information, see Chapter 4, *Displaying 3D Data*, 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.

Input 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 *Coordinate Conversion* on page 193 of the *PV-WAVE User's Guide*.

CONVOL Function

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

Usage

```
result = CONVOL(array, kernel [, scale_factor])
```

Input Parameters

array – The array to be convolved. Can be of any data type except string.

kernel – The array used to convolve each value in *array*. Must have dimensions smaller than those of *array*. 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 – A scaling factor that reduces each output value by the specified factor (the default value is 1.0). Can be used with integer and byte type data only.

Returned Value

result – The convolved array. It is of the same data type and dimensions as *array*.

Input Keywords

Center – Specifies how the kernel is to be centered:

- If nonzero or omitted, centers the kernel over each array data point.
- If explicitly set to zero, centers the kernel in the strict mathematical sense.

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 PV-WAVE functions, such as DIGITAL_FILTER, SMOOTH, and SHIFT.



When using CONVOL with image data, make sure the data has been first converted to floating-point type.

The *kernel* 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, thereby aligning 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 $t \geq m - 1$, 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

Here is what a 512-by-512 mandril image looks like before and after applying the CONVOL function. For this example, the following parameters were used:

```
result = CONVOL(mandril_img, kernel, /Center)
```

where kernel is a 3-by-3 array with 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.

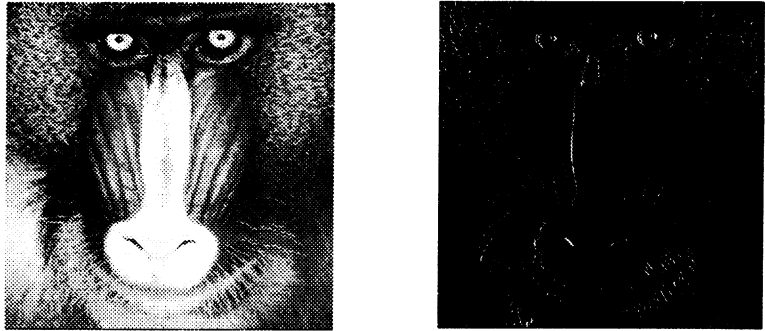


Figure 2-8 The CONVOL function has been used to enhance the edges of this 512-by-512 mandril image. In other words, after CONVOL is applied, the dark colors change quickly to light ones.

See Also

DIGITAL_FILTER, ROBERTS, SHIFT, SMOOTH, SOBEL

For more information on displaying images, see *Image Sharpening* on page 160 of the *PV-WAVE User's Guide* and *Image Smoothing* on page 158 of the *PV-WAVE User's Guide*.

For a signal processing example, see the DIGITAL_FILTERS *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.

Input 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 `$WAVE_DIR/demo/ar1`

See Also

`CONV_FROM_RECT`

For more information, see *Coordinate Conversion* on page 193 of 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. 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. 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* on page 20.

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:

$$\cosh(x) = (e^x + e^{-x})/2$$

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* on page 20.

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

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.

Example 1

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, LOAD_HOLIDAYS,
DT_COMPRESS, 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 sub-string 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 1

```
CREATE_WEEKENDS, 'Sat'
      CREATE_WEEKENDS 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

DT_COMPRESS, CREATE_HOLIDAYS,
LOAD_WEEKENDS

CROSSP Function

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

Usage

$result = CROSSP(v_1, v_2)$

Input Parameters

v_1 — The first operand of the cross product. Must be a three-element vector.

v_2 — 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_1 and v_2 .

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

$$v_1 \times v_2 = (b_1 c_2 - b_2 c_1)i - (c_1 a_2 - c_2 a_1)j + (a_1 b_2 - a_2 b_1)k$$

Example

```
v1 = [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 — An integer specifying when CURSOR returns. This parameter may be used interchangeably with the keywords listed below that specify the type of wait.

Value	Keyword	Action
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.

Note 

Not all modes of waiting 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.

For device-specific information, see Appendix A, *Output Devices and Window Systems*, in the *PV-WAVE User's Guide*.

Output Parameters

x — A named variable to receive the cursor's current column.

y — A named variable to receive the cursor's current row.

Input 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: Using PLOTS Interactively

In this example, PLOTS and CURSOR are used in a loop to build a simple sketch pad. While the cursor is in the graphics window and a button is held down, CURSOR returns the device coordinates of the cursor. Procedure PLOTS 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 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

!Mouse, !Err, TVCRS

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 X (independent) coordinates of the input data points. There are *n* elements in the vector.

y — A vector containing the Y (dependent) coordinates of the input data points. Must have the same number of elements as *x*.

wt — The vector of weighting factors for determining the weighting of the least-squares fit (see Discussion below). Must be the same size as *x*.

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 — 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 the FUNCT procedure.

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 $wt = 1/(Std\ dev\ of\ y_i)$
 - ✓ 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.

Input 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 *Defining Color and Shading* on page 198 of the *PV-WAVE User's Guide*.

Decal — A 2D array of bytes whose elements correspond to indices into the arrays of material properties. For more information, see the section *Decals* on page 201 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 *Ambient Component* on page 199 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 *Diffuse Component* on page 198 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 *Transmission Component* on page 199 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* on page 201 of the *PV-WAVE User's Guide*.

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.

Examples

```
T3D, /Reset, Rotate=[90, 0., 0]  
c = CYLINDER(Transform=!P.T)  
TV, RENDER (c)
```

See Also

CONE, MESH, RENDER, SPHERE, VOLUME

For more information, see *Ray-tracing Rendering* on page 196 of 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 PV-WAVE 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.

Examples

```
date = TODAY( )  
day = DAY_NAME(date)  
PRINT, day  
Monday
```

See Also

DAY_OF_YEAR, DAY_OF_WEEK, MONTH_NAME

For more information, see Chapter 7, *Working with Date/Time Data*, in 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 PV-WAVE Date/Time variable.

Returned Value

result — The day of the week expressed as an integer. Day 0 is Sunday and day six is Saturday.

Keywords

None.

Examples

Assume that you have a PV-WAVE 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, see Chapter 7, *Working with Date/Time Data*, in 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 PV-WAVE 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).

Examples

```
today = TODAY()  
Create a PV-WAVE Date/Time variable.  
  
daynumber = DAY_OF_YEAR(today)
```

PRINT, daynumber

106

See Also

DAY_OF_WEEK, DAY_NAME, MONTH_NAME

For more information, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

DBLARR Function

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

Usage

result = DBLARR(*dim*₁, ... , *dim*_{*n*})

Input Parameters

*dim*_{*i*} – 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.

Input Keywords

Nozero – Normally, DBLARR sets every element of the result to zero. If *Nozero* is nonzero, this zeroing is not performed, causing DBLARR to execute faster.

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

DINDGEN, DOUBLE, BYTARR, COMPLEXARR, FLTARR, INTARR, LONARR, MAKE_ARRAY

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 text 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.
    TV, 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 VMS).

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_FIXED Function

Reads fixed-formatted ASCII data using a PV-WAVE 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 PV-WAVE 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 255. 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_READ_FIXED; expected values are:

- < 0 Indicates an error, such as an invalid filename or an I/O error.
- 0 Indicates a successful read.

Input Keywords

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.

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 165.

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.

Row — A flag that signifies *filename* is a row-organized file. If neither *Row* nor *Column* is present, *Row* is the default.

Column — A flag that signifies *filename* is a column-organized file.

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 186.

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*.

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 175.

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 170.

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 171.

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 173.

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 *Example 6* on page 174. To see a complete list of date/time templates, see *Transferring Date/Time Data* on page 168 of the *PV-WAVE Programmer's Guide*.

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 PV-WAVE structures.

How the Data is Transferred into Variables

As many as 255 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 187 (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.



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, *FORTRAN and C Format Strings*, 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 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 Appendix A, *FORTRAN and C Format Strings in the 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 173.



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 170, or DC_READ_FREE, *Example 4* on page 187.

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 *Row-Oriented ASCII Data Files* on page 146 of 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 *Column-Oriented ASCII Data Files* on page 144 of 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 PV-WAVE variables:

Table 2-1: Reading Data from a Column-Oriented File

Dimensions of Variable	How Data is Read From the File (If Variable is Pre-dimensioned)
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*.

Caution

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', /Col, $
    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 PV-WAVE 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, /Col, $
    Filter=['/', ':', ','], $
    Format="(F7.2, 8X, F6.4, 3X)", $
    Ignore["$BLANK_LINES"])
```

reads data from the file `yields.doc` and places the data into four PV-WAVE 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)', /Col)
```

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 171, 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], /Col)
```

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], /Col)
```

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.)

Note 

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 PV-WAVE 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(/Col, "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 PV-WAVE 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)", /Col)
```

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, /Col, $
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:

1070.0	9007.97	1100.0	1250.0	850.5	2010.0
5000.0	3050.0	1044.12	3500.0	6031.0	905.0
415.0	5200.0	1300.10	350.0	745.0	3000.0
200.0	3100.0	8100.0	7050.0	6780.0	2310.25
950.0	1050.0	1350.0	410.0	797.0	200.36
2600.0	2000.0	1500.0	2000.0	1000.0	400.0
1000.0	9000.0	1100.0	2091.0	3440.10	2000.37
5000.0	3000.0	1000.01	3500.0	6000.0	900.12

The following PV-WAVE 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 PV-WAVE 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_READ_FREE`, `DC_WRITE_FIXED`, `DC_ERROR_MSG`

See *Explicitly Formatted Input and Output* on page 165 of the *PV-WAVE Programmer's Guide* for more information about fixed format I/O in PV-WAVE.

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 PV-WAVE 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 255. 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_READ_FREE; expected values are:

- < 0 Indicates an error, such as an invalid filename or an I/O error.
- 0 Indicates a successful read.

Input Keywords

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.

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.

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 181.

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.

Row — A flag that signifies *filename* is a row-organized file. If neither *Row* nor *Column* is present, *Row* is the default.

Column — A flag that signifies *filename* is a column-organized file.

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*.

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 185.

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 182.

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 171, or DC_READ_FREE, *Example 4* on page 187.

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 187.

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 189. To see a complete list of date/time templates, see *Transferring Date/Time Data* on page 168 of the *PV-WAVE User's Guide*.

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 PV-WAVE structures.

How the Data is Transferred into Variables

As many as 255 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 187.

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 187.



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 187.



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.

Caution

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 *Row-Oriented ASCII Data Files* on page 146 of 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 *Column-Oriented ASCII Data Files* on page 144 of the *PV-WAVE Programmer's Guide*.

For more information about how column-oriented data in a file is read into multi-dimensional PV-WAVE variables, see *Multi-dimensional Variables* on page 169.

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, $
/Col, 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, /Col, 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, $  
/Col, 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, $
    /Col, Delim=[';', '/'], Filter=['','], $
    Resize=[1, 2], Vals_Per_Rec=2)
```

result in:

```
gap =  $\begin{bmatrix} 5992 & 17121 & 0 \\ 8348 & 17562 & 0 \\ 5672 & 19451 & 0 \\ 5459 & 18659 & 0 \\ 7088 & 17052 & 0 \\ 8541 & 13437 & 0 \\ 6362 & 15894 & 0 \\ 8992 & 17509 & 0 \\ 7785 & 14796 & 0 \end{bmatrix}$ 
```

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 PV-WAVE 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, /Col, 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`.



Note For more information about the internal organization of the !DT system structure, see *The PV-WAVE Date/Time Structure* on page 225 of the *PV-WAVE User'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, /Col, 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_READ_FIXED, DC_WRITE_FREE, DC_ERROR_MSG

See *Free Format Input and Output* on page 159 of the *PV-WAVE Programmer's Guide* for more information about free format I/O in PV-WAVE.

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.

Output 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 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.

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`. `xdim` and `ydim` are computed by taking the square root of the number of bytes in the file, since 200-by-350 is not one of the “common” image sizes that `DC_READ_8_BIT` checks for.

See Also

`DC_WRITE_8_BIT`, `DC_ERROR_MSG`

See *Input and Output of Image Data* on page 189 of the *PV-WAVE Programmer's Guide* for more information about 8-bit (binary) data.

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.

Input 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.

Output Keywords

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:

Image Width	Image Height	Image Depth
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`:

Interleaving Method	Dimensions of Image Variable
Pixel Interleaving	Dimension <code>imgarr</code> as $3 \times w \times h$, where w and h are the width and length of the image in pixels.
Image Interleaving	Dimension <code>imgarr</code> as $w \times h \times 3$, where w and h 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_WRITE_24_BIT, DC_ERROR_MSG

See *Input and Output of Image Data* on page 189 of the *PV-WAVE Programmer's Guide* for more information about 24-bit (binary) data and for more information about image interleaving options.

DC_READ_TIFF Function

Reads a Tag Image File Format (TIFF) file.

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 PV-WAVE 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. Note that variables of type structure are not supported.

Returned Value

status — The 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.

Input Keywords

Imgnum — The number of the image to read from the file. If not provided, the first image (image number 0) is read.

Output Keywords

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.

Imagewidth — The TIFF image width. If present, the TIFF image width is returned. *Imagewidth* is returned as a long integer value.

Imagelength — The TIFF image length. If present, the TIFF image length is returned. *Imagelength* is returned as a long integer value.

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.

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

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.

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

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
- 32773 PackBits

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*
- 1 Bilevel/Grayscale*
- 2 Full RGB color
- 3 Palette color
- 4 Transparency mask†

* If *PhotometricInterpretation*=0, 0 is imaged as white, and $2^{**BitsPerSample}-1$ is imaged as black.

If *PhotometricInterpretation*=1, 0 is imaged as black, and $2^{**BitsPerSample}-1$ is imaged as white.

† 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.

Note 

The first four classes of TIFF images are explained in more detail in *TIFF Conformance Levels* on page 192 of 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 *Image Interleaving* on page 193 of the *PV-WAVE Programmer's Guide*.

Note 

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 *Image Interleaving* on page 193 of the *PV-WAVE Programmer's Guide*.



Note

Compressed TIFF images are uncompressed before being transferred to the named PV-WAVE 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 PV-WAVE 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_WRITE_TIFF`, `DC_ERROR_MSG`

See *TIFF Image Data* on page 191 of the *PV-WAVE Programmer's Guide* for more information about TIFF image I/O.

DC_WRITE_FIXED Function

Writes the contents of one or more PV-WAVE variables (in ASCII fixed format) to a file using a format that you specify.

Usage

```
status = DC_WRITE_FIXED(filename, var_list, format)
```

Input Parameters

filename — A string containing the pathname and filename of the file where the data will be stored.

var_list — The list of PV-WAVE 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.

Input Keywords

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.

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.

Column — A flag that signifies *filename* is to be written as a column-organized file.

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*.

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 218. To see a complete list of date/time templates, see *Transferring Date/Time Data* on page 168 of the *PV-WAVE Programmer's Guide*.

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 PV-WAVE structures.

How the Data is Written to the File

As many as 255 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 207.

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 has been written successfully (prior to the error) is left intact in the file.



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 *Appendix A, FORTRAN and C Format Strings*, in the *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 *Appendix A, FORTRAN and C Format Strings*, in the *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 *Row-Oriented ASCII Data Files* on page 146 of the *PV-WAVE Programmer's Guide*.

Writing Column-Oriented Data

The following table shows how PV-WAVE variables of any dimensions are stored in a columnar format:

Table 2-2: Writing Data to a Column-Organized File

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 *Column-Oriented ASCII Data Files* on page 144 of 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', /Col, $
    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", $
    /Col)
```

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.



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 Separated 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 Date/Time Structure* on page 225 of the *PV-WAVE User'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 PV-WAVE command:

```
status = DC_WRITE_FIXED("thymus.dat", $
                        date, float, /Col, 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 PV-WAVE 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 218.

Example 5

Suppose you have a number of PV-WAVE 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, /Col, $
    Format="%s %s %5.2f %i %i %5.2f %s")
```

Note

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 PV-WAVE's 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, /Col, 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_WRITE_FREE, DC_READ_FIXED, DC_ERROR_MSG

See *Explicitly Formatted Input and Output* on page 165 of 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 PV-WAVE 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 PV-WAVE 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.

Input Keywords

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.

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.

Column — A flag that signifies *filename* is to be written as a column-organized file.

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*.

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 *Example 4* on page 218. To see a complete list of date/time templates, see *Transferring Date/Time Data* on page 168 of the *PV-WAVE Programmer's Guide*.

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 PV-WAVE structures.

How the Data is Written to the File

As many as 255 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 207.

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 PV-WAVE data types are stored using the default formats shown in the following table:

Table 2-3: Output Formats Used by DC_WRITE_FREE

Data Type	Output Formats used by DC_WRITE_FREE
Byte	I4
Integer	I8
Long Integer	I12
Float	G13.6
Double	G16.8
Complex	('(', G13.6, ',', G13.6, ')')
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 *Row-Oriented ASCII Data Files* on page 146 of the *PV-WAVE Programmer's Guide*.

Writing Column-Oriented Files

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 207.

If you are interested in an illustration demonstrating what column-oriented data can look like inside a file, see *Column-Oriented ASCII Data Files* on page 144 of 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('intfile.dat', bogus, $
                      Delim=[','], /Col)
```


creates a file `intfile.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 PV-WAVE 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='/', /Col)
```

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 PV-WAVE date/time structure as a series of integer and floating-point values.

The variable `date` is a PV-WAVE 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 Date/Time Structure* on page 225 of the *PV-WAVE User'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:

Number	Template Description
1	MM/DD/YY (/ = delimiter)
-1	HH:MM:SS (: = delimiter)

If you enter the following PV-WAVE command:

```
status = DC_WRITE_FREE("thymus.dat", date, $  
float, date, /Col, 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_READ_FREE, DC_WRITE_FIXED, DC_ERROR_MSG

See *Explicitly Formatted Input and Output* on page 165 of the *PV-WAVE Programmer's Guide* for more information about free format I/O in PV-WAVE.

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 2-dimensional 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 2-dimensional 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 by PV-WAVE Point & Click, or with the PV-WAVE function DC_READ_8_BIT.

See Also

DC_READ_8_BIT, DC_WRITE_24_BIT, DC_ERROR_MSG

See *Input and Output of Image Data* on page 189 of 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 3-dimensional 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.

Input 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 3-dimensional 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:

<u>Interleaving Method</u>	<u>Dimensions of Image Variable</u>
Pixel (RGB triplets)	$3 \times w \times h$, where w and h are the width and length of the image in pixels.
Image (separate planes)	$w \times h \times 3$, where w and h are the width and length of the image in pixels.

The difference between pixel-interleaved and image-interleaved data is discussed in *Image Interleaving* on page 193 of 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 by PV-WAVE Point & Click, or with the PV-WAVE function DC_READ_24_BIT.

See Also

DC_READ_24_BIT, DC_WRITE_8_BIT, DC_ERROR_MSG

See *Input and Output of Image Data* on page 189 of the *PV-WAVE Programmer's Guide* for more information about 24-bit (binary) data.

DC_WRITE_TIFF Function

Writes image data to a file using the Tag Image File Format (TIFF) format.

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-dimensional 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.

Input Keywords

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.

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 *TIFF Conformance Levels* on page 192 of 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'
```

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.



Note

If *imgarr* is not a 2- or 3-dimensional 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-dimensional 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 *Image Interleaving* on page 193 of 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_READ_TIFF`, `DC_ERROR_MSG`

See *TIFF Image Data* on page 191 of the *PV-WAVE Programmer's Guide* for more information about TIFF image I/O.

DEFINE_KEY Procedure

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

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. Key names are handled differently in UNIX and VMS:

- 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. Table 2-4 on page 229 describes the standard key definitions; however, available function keys and the escape sequences they send vary from keyboard to keyboard.

Note that the SETUP_KEYS procedure should be used once at the beginning of the session to enter the keys for the current keyboard.

- Under VMS, key names are defined by the Screen Management utility (SMG). Table 2-5 on page 231 describes some of these keys. For a complete description, see the *VMS RTL Screen Management (SMG\$) Manual*.

value — 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.

Input Keywords

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_Line – (UNIX only) Programs *key* to delete all characters to the left of the current 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 228 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.

Under UNIX, the default match key is the up caret “^” key when pressed in column 1.

Under VMS, 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> that is added to the end of *value*.

Defining New UNIX Function Keys

Under UNIX, PV-WAVE can handle arbitrary function keys. Standard UNIX key definitions are listed in Table 2-4.

Table 2-4: UNIX Line Editing Keys

Editing Key	Function
<Control> <A>	Move cursor to start of line.
<Control> 	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.

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
```

Note that it is convenient to include commonly used key definitions in a startup file so that they will always be available. For a discussion of startup files, see *WAVE_STARTUP: Using a Startup Command File* on page 48 of the *PV-WAVE User's Guide*.

Defining New VMS Function Keys

Under VMS, PV-WAVE uses the SMG screen-management package, which ensures that PV-WAVE command editing will behave in the standard VMS 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 Table 2-5 below.

Table 2-5: VMS Line Editing Keys

Key Name	Comment
<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.

See Also

SETUP_KEYS, SETDEMO

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_x*, *size_y* [, *xverts*, *yverts*])

Input Parameters

size_x — The size of the image, in pixels, in the X direction.

size_y — The size of the image, in pixels, in the Y direction.

Output Parameters

xverts — The X coordinates of the vertices enclosing the region.

yverts — The Y coordinates of the vertices enclosing the region.

Returned Value

result — A vector containing the subscripts of the pixels inside the region.

Input Keywords

Noregion — If nonzero, inhibits the return of the pixel subscripts.

Zoom — The zoom factor to be used for displaying the image. If omitted, 1 is assumed.

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.

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 2-9 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.

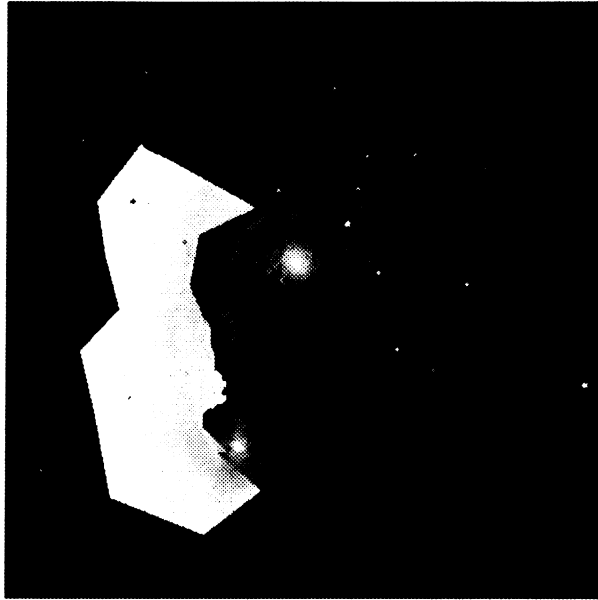


Figure 2-10 Galaxy image with region of interest inverted.

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 – 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 the main program level only. They cannot be defined from within a procedure or function.

Example

This example uses procedure DEFSYSV to create read-only system variables to hold Euler's 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  
. .  
. .
```

See Also

ADDSYSVAR

DELETE_SYMBOL Procedure

(VMS Only) 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.

Input Keywords

Type – Indicates the VMS 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, SET_SYMBOL, SETLOG,
TRNLOG

DELFUNC Procedure

Deletes one or more compiled functions from memory.

Usage

DELFUNC, *function*₁ ,..., *function*_n

Input Parameters

*function*_i — A string containing the name of a compiled function to be deleted.

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 the compiled FILEPATH and DIST functions from memory.

See Also

DELSTRUCT, DELPROC, DELVAR

DELLOG Procedure

(VMS Only) Deletes a logical name.

Usage

DELLOG, *logname*

Input Parameters

logname — A scalar string containing the name of the logical to be deleted.

Input 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, SET_SYMBOL,
SETLOG, TRNLOG

DELPROC Procedure

Deletes one or more compiled procedures from memory.

Usage

DELPROC, *procedure*₁ ,..., *procedure*_{*n*}

Input Parameters

*procedure*_{*i*} — 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

DELSTRUCT, DELFUNC, DELVAR

DELSTRUCT Procedure

Deletes one or more named structure definitions from memory.

Usage

DELSTRUCT, {*structure*₁} ,..., {*structure*_{*n*}}

Input Parameters

*structure*_{*i*} — 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.

Input Keywords

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.

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.



You cannot delete structure definitions that are system structure definitions, such as !Axis and !Plot, or any structures that begin with !. Therefore, if you want to create a new structure that cannot be deleted, begin its name with !.

Example

```
WAVE> x = {struct1, a:float(0)}
```

The user changes his or her mind about what type x.a should be.

```
WAVE> DELVAR, x
```

```
WAVE> DELSTRUCT, {struct1}
```

```
WAVE> x = {struct1, a:double(0)}
```

See Also

STRUCTREF, DELPROC, DELFUNC, DELVAR

For more information on structures, see Chapter 6, *Working with Structures*, in the *PV-WAVE Programmer's Guide*.

DELVAR Procedure

Deletes variables from the main program level.

Usage

DELVAR, v_1, \dots, v_n

Input Parameters

v_i – One or more named variables to be deleted.

Keywords

None.

Discussion

The following restrictions on the use of DELVAR apply:

- DELVAR may only be called from the main program level.
- Each time DELVAR is called, the main program is erased. Variables that are not deleted remain unchanged.
- DELVAR forces the end of the current procedure.

Note



DELVAR does not free the memory used by the variable to the operating system. To release memory to the operating system, you must exit PV-WAVE.

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 *Running Out of Virtual Memory?* on page 284 of the *PV-WAVE Programmer's Guide*.

DERIV Function

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

Usage

$result = \text{DERIV}([x,] y)$

Input Parameters

x — The vector of X (independent) 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 Y (dependent) 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

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

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.

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 A, Output Devices and Window Systems*, in the *PV-WAVE User's Guide*.

Discussion

The PV-WAVE 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

!D, SET_PLOT

DIGITAL_FILTER Function

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

Usage

$result = \text{DIGITAL_FILTER}(flow, fhigh, gibbs, nterm)$

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.



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:

$$Coeff = \text{DIGITAL_FILTER}(flow, fhigh, gibbs, nterm)$$

$$Filtered_Signal = \text{CONVOL}(input_signal, coeff)$$

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*₀, *y*₀])

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 – The X coordinates of *structure*'s origin.

*y*₀ – The Y coordinates of *structure*'s origin.

Returned Value

result – The dilated image.

Input 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 ori-

gin 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).

Note 

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 2-11 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

$result = DINDGEN(dim_1, \dots, dim_n)$

Input Parameters

dim_i — 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:

$$array(i) = DOUBLE(i), \text{ for } i = 0, 1, \dots, \left(\prod_{j=1}^n D_j - 1 \right)$$

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, FINDGEN, INDGEN, LINDGEN,
SINDGEN, DBLARR

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*)

Input Parameters

n — The size of the resulting 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:

$$\text{result}(i,j) = \sqrt{F(i)^2 + F(j)^2}$$

where

$$F(x) = x \text{ if } 0 \leq x \leq n/2$$

or

$$F(x) = n - x \text{ if } x > n/2$$

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.

Example 1

```
mandril = BYTARR(512,512)
OPENR, unit, !Data_dir + 'mandril.img', $
  /Get_lun
READU, unit, mandril
FREE_LUN, unit
```

Read the mandril 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 mandril 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 below.)

filt_image = FFT(FFT(mandrill, -1) * filter, 1)
    Filter the image by transforming it to the frequency domain, mul-
    tiplying 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

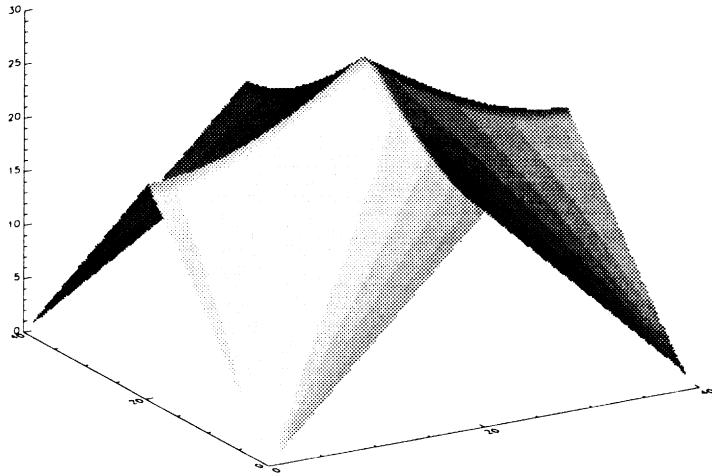
Use these commands:

```

testarr = DIST(40)
CONTOUR, testarr
SURFACE, testarr
LOADCT, 7
SHADE_SURF, testarr
testing = DIST(200)
TVSCL, testing

```

to create the following surface of an array:



See Also

For more information, see *Frequency Domain Techniques* on page 162 of the *PV-WAVE User's Guide*.

DOC_LIBRARY Procedure

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

Input Keywords

Directory — (UNIX only) The name of the directory to search. If this keyword is omitted, the current directory and !Path are used.

File — (VMS 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 — (VMS 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 in PV-WAVE.)

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, 'adjct'
```

On the screen, display the header for the adjct.pro procedure using the default search path.

```
DOC_LIBRARY, 'adjct', $
Print='cat > adjct_header'
```

Print the header for adjct.pro to the file adjct_header.

```
DOC_LIBRARY, '*',directory=$
'/usr/local/pvi/wave/lib/std', $
Print='cat > user_lib_headers'
```

Print the headers for all the files located in the Users' Library.

VMS Examples

```
DOC_LIBRARY, 'adjct'
```

On the screen, display the header for the adjct.pro procedure using the default search path.

```
DOC_LIBRARY, 'adjct', /File
```

Print the header for adjct.pro to the file userlib.doc.

See Also

INFO, !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*₁ [, ..., *dim*_{*n*}])

This form is used to extract data.

Input Parameters

expr — The expression to be converted, or from which to extract data.

offset — (Used to extract data only.) The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim*_{*i*} — (Used to extract data only.) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result — (For data conversion.) A copy of *expr* converted to double-precision floating-point data type. If no *dim*_{*i*} parameters are specified, *result* has the same structure as *expr*.

(For data extraction.) 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.

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 `PV-WAVE` 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, FIX, FLOAT, LONG, DBLARR

For more information on using this function to extract data, see *Extracting Fields* on page 37 of the *PV-WAVE Programmer's Guide*.

DROP_EXEC_ON_SELECT Procedure

Drops a single item from the EXEC_ON_SELECT list.

Usage

DROP_EXEC_ON_SELECT, *lun*

Input Parameters

lun – Logical unit number.

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

SELECT_READ_LUN, EXEC_ON_SELECT,
ADD_EXEC_ON_SELECT

DT_ADD Function

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

Usage

result = DT_ADD(*dt_var*)

Input Parameters

dt_var – The original PV-WAVE Date/Time variable or array of variables.

Returned Value

result – A PV-WAVE Date/Time variable incremented by the specified amount.

Input Keywords

Compress – If present and nonzero, excludes predefined weekends and holidays from the result. The default is no compression (0).

Output Keywords

Year – Specifies an offset value in years.

Month – Specifies an offset value in months.

Day – Specifies an offset value in days.

Hour – Specifies an offset value in hours.

Minute – Specifies an offset value in minutes.

Second – Specifies an offset value in seconds.

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 PV-WAVE 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 PV-WAVE Date/Time variable `dtarray` contains two dates.

```
dtarray1 = DT_ADD(dtarray, /Day)  
    Create a new PV-WAVE Date/Time variable dtarray1 that con-  
    tains two dates with one day added to each date.
```

```
DT_PRINT, dtarray1  
03/18/1992  
04/19/1993
```

See Also

`DT_SUBTRACT`, `DT_DURATION`

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.



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 PV-WAVE 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 = cmarray(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)) $
  cmarray(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, `cmarray` 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 PV=WAVE 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, Date_Fmt=1)
```

Create a PV=WAVE 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, see Chapter 7, *Working with Date/Time Data*, in 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.

Input 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, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

DTGEN Function

Returns a PV-WAVE 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 — 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.

Input Keywords

Year — Specifies an offset value in years.

Month — Specifies an offset value in months.

Day — Specifies an offset value in days.

Hour — Specifies an offset value in hours.

Minute — Specifies an offset value in minutes.

Second — Specifies an offset value in seconds.

Compress — If present and nonzero, excludes predefined weekends and holidays from the result. The default is no compression (0).

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 PV-WAVE Date/Time structures for consecutive years.

```
date1 = TODAY( )
```

Create a PV-WAVE Date/Time variable containing the current date.

```
date2 = DTGEN(date1, 4, /Year)
```

Use DTGEN to create a new PV-WAVE 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 PV-WAVE Date/Time structures for every other month of a year.

```
date = VAR_TO_DT(1992, 1, 1)
```

Create a PV-WAVE Date/Time variable for January, 1992.

```
date1 = DTGEN(date, 6, Month=2)
```

Create an array variable containing PV-WAVE 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, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

DT_PRINT Procedure

Standard Library procedure that prints the values in PV-WAVE Date/Time variables in a readable format.

Usage

DT_PRINT, *dt_var*

Input Parameters

dt_var — A PV-WAVE 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

`!Date_Separator, !Time_Separator`

For more information, see Chapter 7, *Working with Date/Time Data*, in 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 PV-WAVE Date/Time variable or array of variables.

Returned Value

result – A PV-WAVE Date/Time variable decremented by the specified amount.

Input Keywords

Compress – If present and nonzero, excludes predefined weekends and holidays from the result. The default is no compression (0).

Output Keywords

Year – Specifies an offset value in years.

Month – Specifies an offset value in months.

Day – Specifies an offset value in days.

Hour – Specifies an offset value in hours.

Minute – Specifies an offset value in minutes.

Second – Specifies an offset value in seconds.



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 PV=WAVE Date/Time variable containing a Date/Time.

```
dtvar1= DT_SUBTRACT(dtvar, Year=4)
```

Create a new PV=WAVE Date/Time variable by subtracting 4 years from dtvar.

```
PRINT, dtvar1
```

```
{ 1988 3 17 9 30 54.0000 86017.396 0}
```

Display the new PV=WAVE Date/Time variable.

Example 2

This example shows how to add one day to a Date/Time variable containing two PV-WAVE 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 PV-WAVE Date/Time variable dtarray contains two dates.

```
dtarray1 = DT_SUBTRACT(dtarray, /Day)
                    Create a new PV-WAVE Date/Time variable dtarray1 that con-
                    tains 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, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

DT_TO_SEC Function

Standard Library function that converts a PV-WAVE 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 PV-WAVE 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:

Table 2-6: Valid Date Formats

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 Chapter 7, *Working with Date/Time Data*, in 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 `dtarray` that contains the following PV-WAVE 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(dtarray)
PRINT, seconds
7.5589031e+09 7.5904391e+09 7.5355751e+09
```

Example 2

Assume that you have created the array `dtarray` that contains the following PV-WAVE 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(dtarray, $
              Base='1-1-70', Date_Fmt=1)
PRINT, seconds
7.0332296e+08 7.3485896e+08 7.6639496e+08
```

See Also

`SEC_TO_DT`, `DT_TO_VAR`, `DT_TO_STR`, `!DT_Base`

For more information, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

DT_TO_STR Procedure

Converts PV-WAVE 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 – A variable containing the date strings extracted from the PV-WAVE Date/Time variable.

times – A variable containing the time strings extracted from PV-WAVE 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:

Table 2-7: Valid Date Formats

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:

Table 2-8: Valid Time Formats

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.

Note 

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 PV-WAVE 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 PV=WAVE 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

STR_TO_DT, DT_TO_VAR, DT_TO_SEC, !Time_Separator,
!Date_Separator

For more information, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

DT_TO_VAR Procedure

Standard Library procedure that converts a PV-WAVE Date/Time variable to regular numerical data.

Usage

DT_TO_VAR, dt_var

Input Parameters

dt_var — A PV-WAVE Date/Time variable.

Output 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 PV-WAVE 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

`VAR_TO_DT`, `DT_TO_STR`, `DT_TO_SEC`

For more information, see Chapter 7, *Working with Date/Time Data*, in 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 Only) Returns a string array containing all the UNIX environment strings for the PV-WAVE process.

Usage

```
result = ENVIRONMENT( )
```

Parameters

None.

Returned Value

A string array containing all the UNIX environment strings for the PV-WAVE process. Each element of the result contains one environment string.

Keywords

None.

Discussion

Like any UNIX process, PV-WAVE inherits its environment from its parent process, which is usually the shell.

Example

```
p = ENVIRONMENT( )  
PRINT, p
```

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

Under VMS, 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 *Opening and Closing Files* on page 138 of the *PV-WAVE Programmer's Guide*.

ERASE Procedure

Erases the display surface of the currently active window.

Usage

ERASE [, *background_color*]

Input Parameters

background_color — The background color index.



Not all devices support this parameter. Workstations and display terminals, such as X workstations and Tektronix terminals, generally do, while some hardcopy devices, such as HPGL plotters, do not.

Input 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

WAVE> ERASE

Erase the display surface for the current window and use the value in !P.Background to set the background color.

Example 2

WAVE> COLOR_PALETTE

Create a color palette so the color table can be easily viewed.

WAVE> LOADCT, 2

Load in color table 2, GRN-RED-BLU-WHT, as it has distinctive colors.

WAVE> WINDOW, 1

Create window 1.

WAVE> ERASE

Erase it using !P.Background.

WAVE> WINDOW, 2

Create window 2.

WAVE> ERASE, 22

Erase it, setting the background color to 22 (lime green).

WAVE> WINDOW, 1

Switch back to window 1.

WAVE> ERASE, 64

Reset the background color to 64 (bright red).

WAVE> WINDOW, 2

Switch back to window 2.

WAVE> !P.Background=180

Explicitly set the background color to 180 (lavender).

WAVE> 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_0 , y_0])

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_0 – The X coordinates of *structure*'s origin.

y_0 – The Y coordinates of *structure*'s origin.

Returned Value

result – The eroded image.

Input 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 ori-

gin 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).

Note 

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.



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 \ominus 11 =	1100
1000	0000
0000	0000

Example 2

Here is 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.

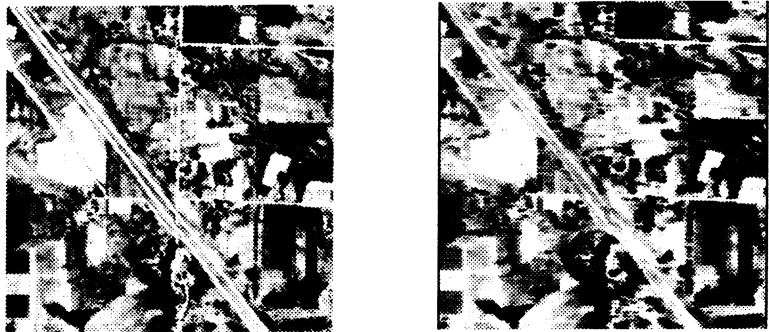


Figure 2-12 The ERODE function has been used to “wear away” the visual elements of this 512-by-512 aerial image.

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:

$$\text{erf}(0) = 0$$

$$\text{erf}(\infty) = 1$$

$$\operatorname{erf}(-x) = -\operatorname{erf}(x)$$

It is related to the incomplete gamma function by:

$$\operatorname{erf}(x) = \Gamma(1/2, x^2)$$

where $x \geq 0$.

See Also

GAMMA

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 — 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.

Input 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. They extend 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 PV-WAVE 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 plot below:

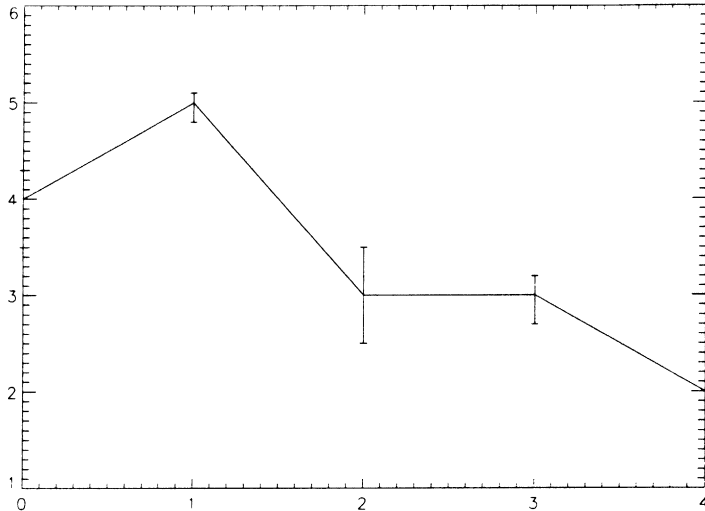


Figure 2-13 In this example, asymmetrical error estimates have been constrained by using ERRPLOT's *low* and *high* parameters.

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:

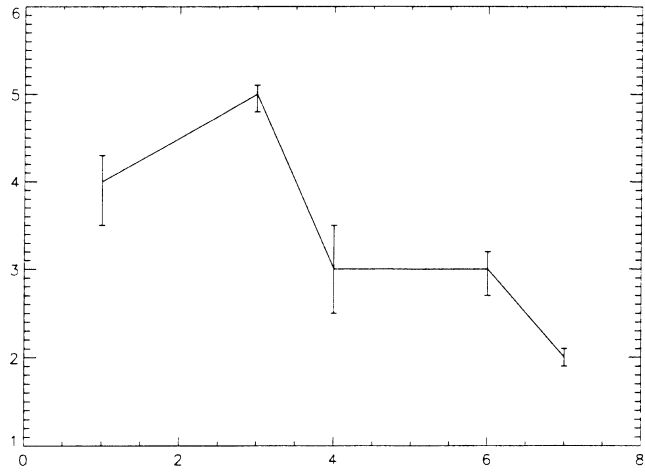


Figure 2-14 In this example, error bars have been plotted over a vector containing specific points along the X axis.

See Also

O PLOT, O PLOTERR, PLOT, PLOTERR

EXEC_ON_SELECT Procedure

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*.

Input 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.

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
        code = 0L
        READU, lun, code
        PRINT, 'SERVER1', code, lun

    END

        PRO SERVER2, top, data, nparams, id, lun
        code = 0L
        READU, lun, code
        PRINT, 'SERVER2', code, lun

    END

        PRO SERVER3, top, data, nparams, id, lun
        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

```

SELECT_READ_LUN, ADD_EXEC_ON_SELECT,
DROP_EXEC_ON_SELECT

```

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 (\$) or period (.) 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.

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 vari-
    able pp. This procedure prints the table.

    IF val EQ 0 THEN BEGIN
        PRINT, "***Error in execution of " + $
        "printing procedure! ***"
    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 PV-WAVE 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 *Executing One or More Statements* on page 273 of the *PV-WAVE Programmer's Guide*.

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:

$$exp(x) = complex(e^r \cos(i), e^r \sin(i))$$

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 20.

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.

Input 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.

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 neighbor-
    hood and number of iterations.

!Y.Range = [0.0, 0.0]
    Reset the Y axis range to the default value.

END

```

See Also

GRID_2D, GRID_3D, GRID_4D, GRID_SPHERE,
FAST_GRID3, FAST_GRID4

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.

Input 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.)



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

GRID_2D, GRID_3D, GRID_4D, GRID_SPHERE,
FAST_GRID2, FAST_GRID4

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.

Input 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.

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.

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.

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

GRID_2D, GRID_3D, GRID_4D, GRID_SPHERE,
FAST_GRID3

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 for the input variable.

Usage

result = FFT(*array*, *direction*)

Input Parameters

array — The array for which the FFT or reverse FFT is computed. The size of each dimension may be any 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

None.

Discussion

The Fourier transform of a scaled-time function is defined by:

$$Fw = Ffat = \int_{-\infty}^{\infty} fate^{-jw't} dt$$

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 PV-WAVE functions.

The FFT function is used to transform the image into the frequency domain. For the example shown in Figure 2-15, 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:

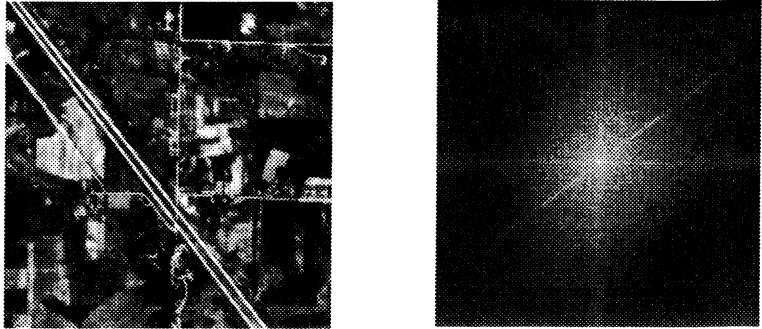


Figure 2-15 PV-WAVE makes it easy to generate the Fourier spectrum for any image. Note that the diagonal, vertical, and horizontal lines in the Fourier spectrum correspond to the roads in the original 512-by-512 image, but are perpendicular to them; this is because of the 90-degree phase shift that occurs 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 Figure 2-15.

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 *Frequency Domain Techniques* on page 162 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 PV-WAVE 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*.

Input Keywords

Subdirectory — The name of the subdirectory in the PV-WAVE distribution area in which *filename* is located.

Output Keywords

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 both UNIX and VMS and will open files in the PV-WAVE distribution.

UNIX Examples

```
PRINT, filepath('wvstartup')
      /usr/local/pvi/wave/wvstartup

full_name = FILEPATH('errplot', $
      Subdirectory='lib/std')
PRINT, full_name
      /usr/local/pvi/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
```

See Also

FINDFILE, !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.

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.

Output 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.

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*.

UNDER VMS, 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('*.*', 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*₁, ..., *dim*_{*n*})

Input Parameters

*dim*_{*i*} — 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:

$$\text{array}(i) = \text{FLOAT}(i), \text{ for } i = 0, 1, \dots, \left(\prod_{j=1}^n D_j - 1 \right)$$

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 complex, single- 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

`ON_ERROR`, `ON_IOERROR`, `CHECK_MATH`

For more information, 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*₁ [, ..., *dim*_{*n*}])

This form is used to extract data.

Input Parameters

expr — The expression to be converted, or from which to extract data.

offset — (Used to extract data only.) The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim*_{*i*} — (Used to extract data only.) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result — (For data conversion.) A copy of *expr* converted to integer data type. If no *dim*_{*i*} parameters are specified, *result* has the same structure as *expr*.

(For data extraction.) If *offset* is used, FIX does not convert *result*, but allows fields of data extracted from *expr* to be treated as integer data.

Keywords

None.

Discussion – Conversion Usage



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 (6 6 0 0 0)` 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

In this example, `FIX` is used in two ways. 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)
```

Notice that the floating-point numbers in a were truncated by `FIX`.

```
PRINT, b
```

```
0    1    2    3    4    5
```

```
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
          4          5

```

See Also

BYTE, COMPLEX, DOUBLE, FLOAT, LONG

For more information on using this function to extract data, see *Extracting Fields* on page 37 of 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*₁ [, ..., *dim*_{*n*}])

This form is used to extract data.

Input Parameters

expr — The expression to be converted, or from which to extract data.

offset — (Used to extract data only.) The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

dim_i — (Used to extract data only.) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result — (For data conversion.) A copy of *expr* converted to single-precision floating-point data type. If no *dim_i* parameters are specified, *result* has the same size and number of dimensions as *expr*.

(For data extraction.) 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.

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 *Extracting Fields* on page 37 of the *PV-WAVE Programmer's Guide*.

FLTARR Function

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

Usage

result = FLTARR(*dim*₁, ..., *dim*_{*n*})

Input Parameters

*dim*_{*i*} — 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.

Input Keywords

Nozero — Normally, FLTARR sets every element of *result* to zero. If *Nozero* is nonzero, this zeroing is not performed, thereby causing FLTARR to execute faster.

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*₁, ..., *unit*_{*n*}

Input Parameters

*unit*_{*i*} — The PV-WAVE 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*₁, ..., *unit*_{*n*}

Input Parameters

*unit*_{*i*} – The PV-WAVE 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 *Logical Unit Numbers (LUNs)* on page 141 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 PV-WAVE 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 PV-WAVE 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 will not contain 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 PV-WAVE data items transferred in the last I/O operation on the unit. This is set by the following PV-WAVE 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 — VMS-specific record length, in bytes. This field is always zero in UNIX.

Example 2

The following **PV-WAVE** 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 PV-WAVE 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

POINT_LUN, GET_LUN, FREE_LUN, CLOSE, OPENR, OPENU, OPENW

For more information, see *Getting Information About Files* on page 220 of the *PV-WAVE Programmer's Guide*.

For background information, see *Logical Unit Numbers (LUNs)* on page 141 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 below.

funcval — The value of the function, described in the Discussion section below, at each $x(i)$.

Output Parameters

pder — 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^{th} point with respect to the j^{th} parameter.

Keywords

None.

Discussion

FUNCT 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 overflow 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 — A six-element vector with the coefficients A_0 through A_5 of the equation described in the Discussion section below.

Returned Value

result — A real vector containing the dependent Y values of the fitted function.

Keywords

None.

Discussion

GAUSSFIT fits $y = F(x)$, where:

$$F(x) = A_0 * \text{EXP}(-z^2/2) + A_3 + A_4x + A_5x^2$$

and where

$$z = (x - A_1) / (A_2)$$

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, POLY_FIT, GAUSSINT

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:

$$\text{Gaussint}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{(-t^2)/2} dt$$

See Also

ERRORF, GAMMA, GAUSSFIT

GETENV Function

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

Usage

result = GETENV(*name*)

Input Parameters

name — The scalar for which an equivalence string from the environment is desired.

Returned Value

result — The equivalence string for *name*. If *name* does not exist in the environment, returns a null string.

Keywords

None.

Discussion — VMS Usage

VMS 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 VMS:

- 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.

Example 1

This PV-WAVE command prints information about the environment:

```
PRINT, 'Current shell is: ', GETENV('SHELL')
```

Example 2

This example shows how to read data from a file in a manner that will work on either UNIX or VMS 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
```

See Also

ENVIRONMENT, SETENV

GET_KBRD Function

Returns the next character available from standard input (PV-WAVE 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 PV-WAVE 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

FREE_LUN, POINT_LUN, OPENR, OPENU, OPENW,
READF, WRITEU, CLOSE, ON_IOERROR

For background information, see *Logical Unit Numbers (LUNs)* on page 141 of the *PV-WAVE Programmer's Guide*.

GET_SYMBOL Function

(VMS Only) Returns the value of a VMS 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 a VMS Digital Command Language interpreter symbol. If the symbol is undefined, the null string is returned.

Output Keywords

Type — Indicates in which VMS 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

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.

Input Keywords

Nghbr — The number of neighboring data points to be used in the gridding algorithm. Must be in the range of {3 ... 25}. The default is 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

GRID accepts irregularly-spaced data from three variables; each variable is a vector that corresponds to the X, Y, or Z direction. All three variables must be the same size. GRID returns an evenly-spaced two-dimensional array.

Specifying the number of rows and columns in the resulting array is optional. However, keep the following guidelines in mind:

- If *nx* is not specified, but *ny* is specified, then *nx* is equal to *ny*.
- If *ny* is not specified, but *nx* is specified, then *ny* is equal to *nx*.
- If neither *nx* nor *ny* is specified, then *nx* and *ny* default to 20.

Increasing the value of the *Nghbr* keyword increases the accuracy of the GRID result, but will take longer to compute.

GRID can be used to prepare irregularly-spaced data so that it can be successfully displayed using either the SURFACE or CONTOUR procedures. The X and Y vectors contain the independent data, and the Z vector contains the dependent data that determines the height of the surface or the spacing of the contour lines.

There are several gridding algorithms that grid irregularly spaced data. The algorithm used by GRID works well on some types of data, but not for all.

Note

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.

Example

```
z = MAKE_ARRAY(100, /Index)
x = SIN(z/5) / EXP(z/40)
y = COS(z/5) / EXP(z/40)
SURFACE, DIST(2), XRange=[MIN(x), MAX(x)], $
    Yrange=[MIN(y), MAX(y)], ZRange=[MIN(z), $
    MAX(z)], /Nodata, /Save
```

```
PLOTS, x, y, z, /T3D
result = GRID(x, y, z, Nx=100, Ny=100)
LOADCT, 3
SURFACE, result
SHADE_SURF, result
```

See Also

CONTOUR, SURFACE

Source code for additional gridding routines in the Advanced Rendering Library is located in the directory `wave/demo/ar1`. For more information, view the `.pro` files for the routines.

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.

Input Keywords

Order — The order of the weighting function to use for neighborhood averaging. The default is 2. Points are weighted by the function:

$$w = 1.0 / (\text{dist} \wedge \text{Order})$$

where *dist* is the distance to the point.

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.

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

GRID_3D, GRID_4D, GRID_SPHERE, FAST_GRID2,
FAST_GRID3, FAST_GRID4

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.

Input Keywords

Order — The order of the weighting function to use for neighborhood averaging. The default is 2. Points are weighted by the function:

$$w = 1.0 / (\text{dist} \wedge \text{Order})$$

where *dist* is the distance to the point.

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

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

GRID_2D, GRID_4D, GRID_SPHERE, FAST_GRID2
FAST_GRID3, FAST_GRID4

For information on the optional software package for advanced gridding, PV-WAVE:GTGRID, contact your Visual Numerics account representative.

GRID_4D Function

Returns a gridded, 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.

Input Keywords

Order — The order of the weighting function to use for neighborhood averaging. The default is 2. Points are weighted by the function:

$$w = 1.0 / (\text{dist} \wedge \text{Order})$$

where *dist* is the distance to the point.

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.

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.

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.

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 `$WAVE_DIR/demo/ar1`.

See Also

GRID_2D, GRID_3D, GRID_SPHERE, FAST_GRID2, FAST_GRID3, FAST_GRID4

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.

Input 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. The default is 2. Points are weighted by the function:

$$w = 1.0 / (\text{dist} \wedge \text{Order})$$

where *dist* is the distance to the point.

Radius — The radius of the sphere to grid on. The minimum allowable radius is 0.5; the default is 1.0.

With a smaller radius, the data points are closer together and more smoothing occurs. A larger radius causes less smoothing.

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.

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 π is mapped to the right 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.

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.

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 `$WAVE_DIR/demo/applications/arl/examples`.

See Also

`POLY_SPHERE`, `GRID_2D`, `GRID_3D`, `GRID_4D`,
`FAST_GRID2`, `FAST_GRID3`, `FAST_GRID4`

For information on the optional software package for advanced gridding, `PV-WAVE:GTGRID`, contact your Visual Numerics account representative.

HAK Procedure

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

Usage

HAK

Parameters

None.

Input 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, a
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 — 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 is given by the same equation as above, except i is replaced with $i * j$, where i and j are the subscripts for the rows and columns, respectively.

Example

```
OPENR, unit, !Data_dir + 'mandril.img', $
  /Get_lun
  Open the PV=WAVE mandril demo 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.

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 — 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.

Note

The HANNING 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.

O PLOT, 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.

O PLOT, 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.

Input 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.

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.

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.

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 mapping 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 the PV-WAVE wave/data directory. The commands shown in this example produce the image on the right below:

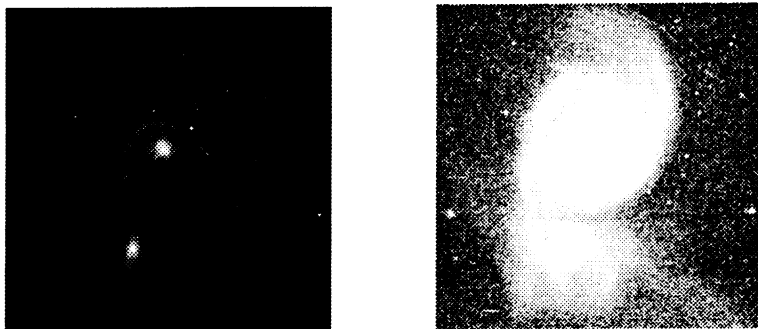


Figure 2-16 The HIST_EQUAL function has been used to make the visual elements of this 512-by-512 galaxy image more pronounced.

```
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 number (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

HISTOGRAM, HIST_EQUAL_CT

For more information, see *Histogram Equalization* on page 155 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 — 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 contained in the PV-WAVE wave/data 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'
TV, 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 *Histogram Equalization* on page 155 of the *PV-WAVE User's Guide*.

HISTOGRAM Function

Returns the density function of an array.

Usage

result = HISTOGRAM(*array*)

Input Parameters

array — The vector or array for which the density function will be computed.

Returned Value

result — A longword vector equal to the density function of the input *array*.

Input Keywords

Binsize — The size of the bin (i.e., the range of values to consider as having a single value). If not specified, a value of 1 is used.

Max — The maximum value to consider. If not specified, *array* is searched for its largest value.

Min — The minimum value to consider. If not specified and *array* is a byte data type, 0 is used. If not specified and *array* is one of the other data types, *array* is searched for its smallest value.

Discussion

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 $\{0 \dots n-1\}$. Then H_v (the result of the HISTOGRAM function) is given by:

$$H_v = \sum_{i=0}^{n-1} P(F_i, v)$$

where

$$v = 0, 1, 2, \dots, \left\lceil \frac{Max - Min}{Binsize} \right\rceil$$

and $P(F_i, v) = 1$ when

$$v \leq (F_i - Min) / (Binsize < v + 1)$$

and $P(F_i, v) = 0$ otherwise.

Caution

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 Chapter 11, *Tips for Efficient Programming*, in 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 be 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.
```

```
TV, g
  Display the original image.
```

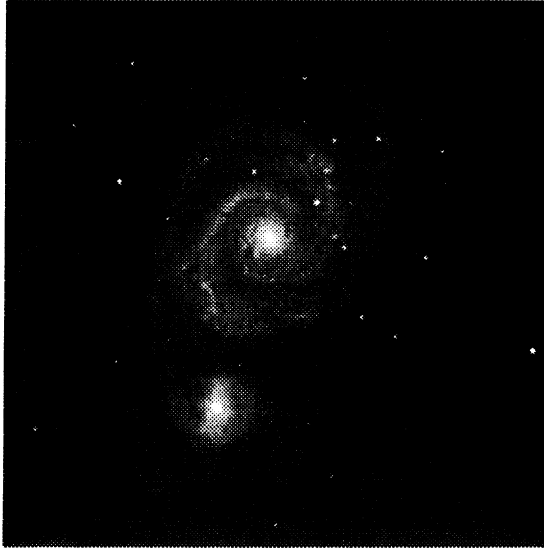


Figure 2-17 Original galaxy image.

```
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 func-
    tion 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.
```

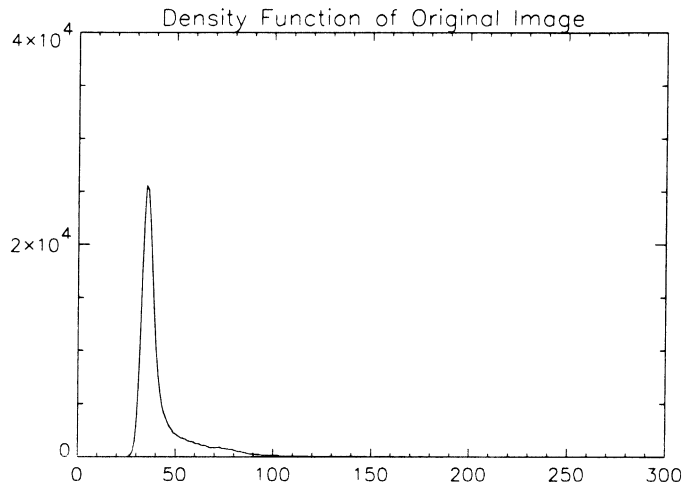



Figure 2-18 Plot of density function of original galaxy image.

```
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.

```
TV, g2
```

Display the equalized image.

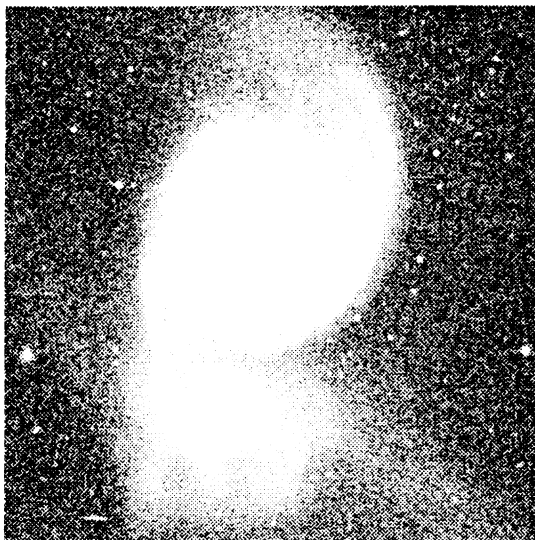


Figure 2-19 Histogram equalized galaxy image.

```
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.

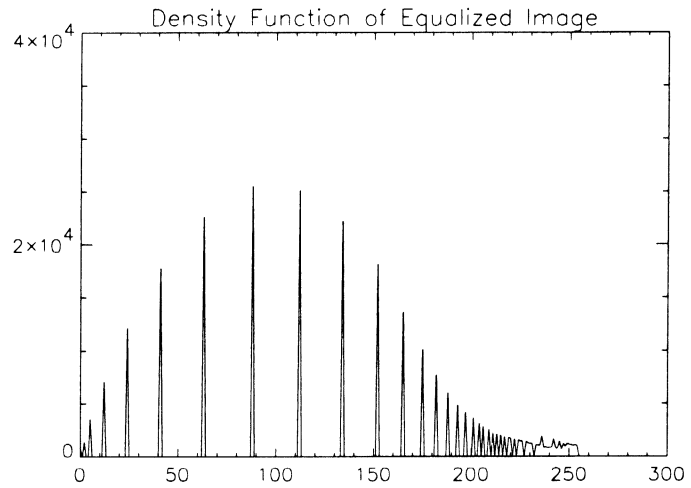


Figure 2-20 Plot of density function of equalized galaxy image.

See Also

HIST_EQUAL, 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 — An optional 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

HSV, HLS, COLOR_CONVERT, COLOR_EDIT, LOADCT, MODIFYCT, RGB_TO_HSV, TVLCT, WgCeditTool, WgCtTool

For more information, see *The HSV and HLS Color Systems* on page 308 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 — An optional 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

HSV, HLS, COLOR_CONVERT, COLOR_EDIT, LOADCT, MODIFYCT, RGB_TO_HSV, TVLCT, WgCeditTool, WgCfTool

For more information, see *The HSV and HLS Color Systems* on page 308 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

HSV, HLS, COLOR_CONVERT, COLOR_EDIT, LOADCT, MODIFYCT, RGB_TO_HSV, TVLCT, WgCeditTool, WgCtTool

For more information, see *The HSV and HLS Color Systems* on page 308 of the *PV-WAVE User's Guide*.

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.

Input 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.
```

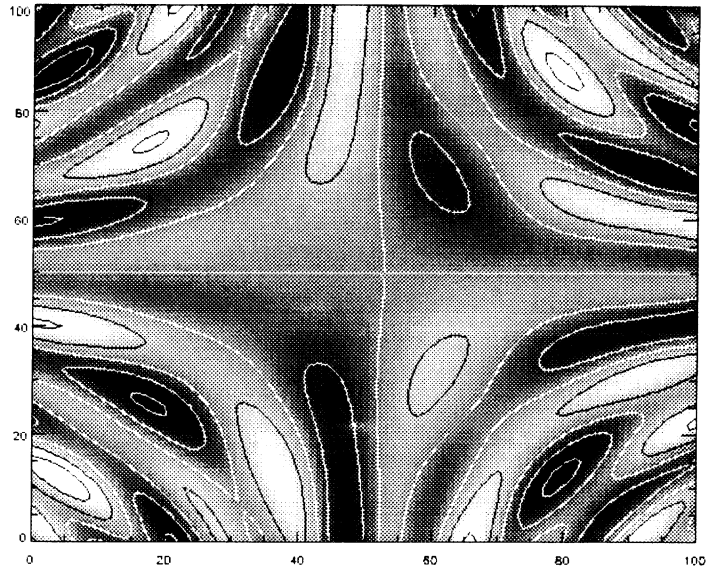


Figure 2-21 Image and contour plot of
 $f(x,y) = x \sin (y) + y \cos (x) - 10 \sin (xy/4) .$

See Also

CONTOUR, SHOW3, TV, TVSCL

For details on methods of interpolation, see *Efficiency and Accuracy of Interpolation* on page 170 of the *PV-WAVE Programmer's Guide*.

IMAGINARY Function

Returns the imaginary part of a complex number.

Usage

```
result = IMAGINARY(complex_expr)
```

Input Parameters

complex_expr — A complex number (one with both a real and imaginary part). Can be of any dimension.

Returned Value

result — The imaginary part as a single-precision floating-point value. It is of the same dimension as *complex_expr*.

Keywords

None.

Discussion

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

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 below).

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.



Note

For correct appearance, *rgb_img* should be displayed in a PV-WAVE window with 256 colors allocated to it. 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
```



Tip

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.

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

```

To see an example using the same data, except displayed in true 24-bit color on a 24-bit X workstation, see *Displaying Images on Monochrome Devices* on page 148 of the *PV-WAVE User's Guide*.

See Also

LOADCT

For a comparison of pseudo- and true-color images, see *Not all Color Images are True-color Images* on page 146 of the *PV-WAVE User's Guide*.

INDGEN Function

Returns an integer array with the specified dimensions.

Usage

$result = \text{INDGEN}(dim_1, \dots, dim_n)$

Input Parameters

dim_i — 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:

$$array(i) = i, \text{ for } i = 0, 1, \dots, \left(\prod_{j=1}^n D_j - 1 \right)$$

Keywords

None.

Example

This example creates a 4-by-2 integer array.

```
a = INDGEN(4, 2)
    Create 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*₁, ... , *expr*_{*n*}

Input Parameters

*expr*_{*i*} — 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.

Output Keywords

Breakpoints — Displays the breakpoint table containing the program module and location of each breakpoint.

Calls — Stores the procedure call stack, in the form of a string array, in a specified variable.

Each string element contains the name of the program module, source file name, and line number. Array element zero contains the information about the caller of INFO, element one contains information about its caller, and so on.

Calls is useful for programs that require traceback information.

Device — If present and nonzero, displays information about the currently selected graphics device.

Files – Displays information about file units:

- 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.

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.

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.

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.

Structures – Displays information on the structure of variables:

- 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.

Traceback — Displays the current nesting of procedures and functions. Input parameters are ignored.

Userstruct — Displays information on the regular user-defined structures (structures that do not begin with '!'). This keyword is a subset of the *Structures* keyword.

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 Chapter 14, *Getting Session Information*, in the *PV-WAVE Programmer's Guide*.

INTARR Function

Returns an integer vector or array.

Usage

```
result = INTARR(dim1, ... , dimn)
```

Input Parameters

dim_i — 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_i*.

Input Keywords

Nozero — Normally, INTARR sets every element of the result to zero. If *Nozero* is nonzero, this zeroing is not performed, thereby causing INTARR to execute faster.

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

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.

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

$result$ – For a regular grid, a vector containing n points interpolated from vector v . For an irregular grid, 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:

$$\text{result}(i) = v(j) + (j - \text{FIX}(j)) (v(j + 1) - v(j))$$

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:

$$\text{abscissa_value}(i) = \text{FLOAT}(i) / m$$

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:

$$\text{result}(i) = v(j) + (j - \text{FIX}(j)) (v(j + 1) - v(j))$$

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

INVERT Function

Returns an inverted copy of a square array.

Usage

```
result = INVERT(array [, status])
```

Input Parameters

array — A two-dimensional square array. May be of any data type except string.

Output Parameters

status — 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

ISHFT Function

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

Usage

result = ISHFT(*p*₁, *p*₂)

Input Parameters

*p*₁ — 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)
      Divide each element of A by 4.
```

```
0   4   8   12  16
0   1   2   3   4
```

```
PRINT, a, ISHFT(a, 1)
      Multiply each element of A by 2.
```

```
0   4   8   12  16
0   8  16  24  32
```

See Also

SHIFT

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 — 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

None.

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

JUL_TO_DT Function

Converts a Julian day number to a PV-WAVE 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 PV-WAVE Date/Time variable containing the converted data.

Keywords

None.

Discussion

The Julian day number is based on the date September 14, 1752.

Example

```
dt = JUL_TO_DT(87507)
    Converts the Julian day 87507 into a PV-WAVE Date/Time variable.
print, dt
    { 1992 4 15 0 0 0.00000 87507.000 0 }
```

See Also

VAR_TO_DT, STR_TO_DT, SEC_TO_DT

For more information, see Chapter 7, *Working with Date/Time Data*, in 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 in PV-WAVE:

```
IF KEYWORD_SET(x) THEN PRINT, $
  'It is set' ELSE PRINT, 'Not set'
```

You will see:

```
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 would see:

```
WAVE> x = 2
WAVE> 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:

```
WAVE> .run SQUARE_IT.PRO
WAVE> y = SQUARE_IT(5)
WAVE> PRINT, y
      25
WAVE> y = SQUARE_IT(10, /Print_Flag)
      100
```

See Also

`N_ELEMENTS`, `N_PARAMS`

LEEFILT Function

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

Usage

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

Input Parameters

image — The array to be smoothed. Can be a one- or two-dimensional array.

n — 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*. The default value for *n* is 5.

sigma — The estimate of the standard deviation. The default value is 5. Must be a positive value.

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

None.

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 vectors 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. May be characters or numbers.

col — 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 — 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 — 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 — 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 — 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 — 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

XYOUTS, PLOT

For more information, see Chapter 4, *System Variables*.

LINDGEN Function

Returns a longword integer array with the specified dimensions.

Usage

result = LINDGEN(*dim*₁, ... , *dim*_{*n*})

Input Parameters

*dim*_{*i*} — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be 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 the following:

$$\text{array}(i) = \text{LONG}(i), \text{ for } i = 0, 1, \dots, \left(\prod_{j=1}^n D_j - 1 \right)$$

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,
SINDGEN, INTARR

LINKNLOAD Function

Provides simplified access to external routines in shareable images.

Usage

result = LINKNLOAD(*object*, *symbol* [, *param*₁, ..., *param*_{*n*}])

Input Parameters

object — A string specifying the filename, optionally including file path, of the sharable object file to be linked and loaded.

symbol — A string specifying the function name symbol entry point to be invoked in the shared object file.

Output Parameters

*param*_{*i*} — The data to be passed as a parameter to the function. Any PV-WAVE 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

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.

S_Value — Specifies that the returned scalar is of type string.

Discussion

LINKNLOAD provides a simple, yet powerful, mechanism for dynamically invoking your own code from PV-WAVE on the following operating systems that support dynamic linking: AIX 3.1, HPUX 8.0, SunOS 4.1, and VMS 5.0.

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.



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.

Notes and Cautions

For AIX 3.1, 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. In addition, for AIX, `wavevars` may not be called in a C function called from PV-WAVE via LINKNLOAD.

LINKNLOAD is not available for operating systems that do not support dynamic linking, such as DEC ULTRIX and SGI IRIX.

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.

LINKNLOAD only works directly for C. To call a FORTRAN routine, you must first write a simple C function that calls your FORTRAN code, then call the C function from PV-WAVE.

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 being a snap-shot 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 C function (or C wrapper) match what the C function (or C wrapper) 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.

Example

The following PV-WAVE code demonstrates how to invoke a C function using LINKNLOAD. For more information on compiling shareable objects, see *Using LINKNLOAD to Call External Programs* on page 319 of the *PV-WAVE Programmer'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'
```

See Also

SIZE

LN03 Procedure

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 — The name of the file that will contain the LN03 graphics output.

Keywords

None.

Discussion

You may also use the DEC SIXEL device driver for LN03 graphics output (the LN03 procedure was created before this driver was available). For details, see the section *SIXEL Output* on page A-58 in Appendix A, *Output Devices and Window Systems*, in the *PV-WAVE User's Guide*.

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 PV-WAVE color table.

Usage

LOADCT [, *table_number*]

Input Parameters

table_number — A number between 0 and 15; each number is associated with a predefined color table.

Input Keywords

Silent — If present and nonzero, suppresses the message indicating that the color table is being loaded.

Discussion

Predefined color tables are distributed with PV-WAVE in the file `colors.tbl`. There are 16 predefined color tables, with indices ranging from 0 to 15, as shown in Table 2-9.

To see a menu listing of these color tables, call LOADCT with no parameters.

Table 2-9: Standard PV-WAVE Color Tables

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 *Experimenting with Different Color Tables* on page 310 of the *PV-WAVE User's Guide*.

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 PV-WAVE Date/Time routines.

Note that 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.

```
!holiday_list(0).RECALC = byte(1)
```

Manually set the Recalc field of the structure to 1. This field contains a byte value. See the Caution below.

```
LOAD_HOLIDAYS
```

Run LOAD_HOLIDAYS so the new holiday value will take effect.

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.

See Also

LOAD_WEEKENDS, CREATE_HOLIDAYS,
CREATE_WEEKENDS

For more information, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

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 PV-WAVE Date/Time routines.

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

LOAD_HOLIDAYS, CREATE_HOLIDAYS,
CREATE_WEEKENDS

For more information, see Chapter 7, *Working with Date/Time Data*, in the *PV-WAVE User's Guide*.

LONARR Function

Returns a longword integer vector or array.

Usage

result = LONARR(*dim*₁, ... , *dim*_{*n*})

Input Parameters

*dim*_{*i*} — 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.

Input Keywords

Nozero — Normally, LONARR sets every element of the result to zero. If *Nozero* is nonzero, this zeroing is not performed, thereby causing LONARR to execute faster.

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 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*₁ [, ... , *dim*_{*n*}])

This form is used to extract data.

Input Parameters

expr — The expression to be converted, or from which to extract data.

offset — (Used to extract data only.) The offset, in bytes, from the beginning of *expr* to where the extraction is to begin.

*dim*_{*i*} — (Used to extract data only.) The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result — (For data conversion.) A copy of *expr* converted to longword integer data type. If no *dim*_{*i*} parameters are specified, *result* has the same structure as *expr*.

(For data extraction.) If *offset* is used, LONG does not convert *result*, but allows fields of data extracted from *expr* to be treated as longword integer data.

Keywords

None.

Discussion – Conversion Usage



Caution

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 $A = 2.0 \wedge 31 + 2$. 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 1.50000 2.50000 3.50000 4.50000
5.50000
```

```
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 using this function to extract data, see *Extracting Fields* on page 37 of the *PV-WAVE Programmer's Guide*.

LUBKSB Procedure

Solves the set of n linear equations $\mathbf{Ax} = \mathbf{b}$.
(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.
 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 - 2z = 3
    -x + y - 5z = 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 an n -by- n matrix, a , with the LU decomposition of a row-wise permutation of itself.

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 $Ax = B$ 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(*[dim₁, ... , dim_n]*)

Input Parameters

dim_i — The dimensions of the result. May be any scalar expression. Up to eight dimensions may be specified.

Returned Value

result — An array of specified type, dimensions, and initialization.

Input 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.

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 floating-point
7	String

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 that 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

SIZE, BYTARR, DBLARR, COMPLEXARR, FLTARR, INTARR, LONARR, BINDGEN, CINDGEN, DINDGEN, FINDGEN, INDGEN, LINDGEN, SINDGEN

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 — 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*.

Input Keywords

Min — Used to specify a variable to hold the value of the minimum array element.



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
```

See Also

!C, AVG, 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 — 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, *result* is returned as the average of the two middle values.



The *Average* keyword is ignored when filtering a byte array.

Discussion

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.

```
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.

```
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
```

```
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.

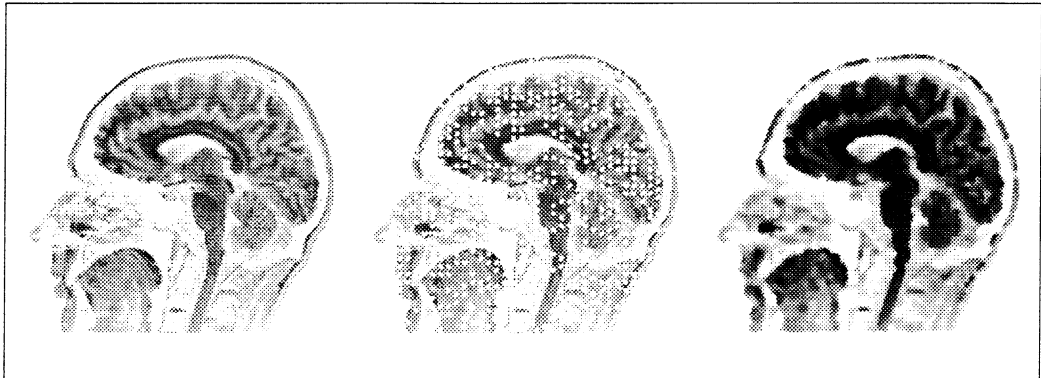


Figure 2-22 Original image (left), corrupted image (center), and median filtered image (right).

See Also

LEEFILT, SMOOTH, AVG, MAX, MIN

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 *Vertex Lists and Polygon Lists* on page 187 of the *PV-WAVE User's Guide*.

Returned Value

result — A structure that defines an object consisting of multiple polygons.

Input 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 *Defining Color and Shading* on page 198 of the *PV-WAVE User's Guide*.

Decal — A 2D array of bytes whose elements correspond to indices into the arrays of material properties. For more information, see the section *Decals* on page 201 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 *Ambient Component* on page 199 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 *Diffuse Component* on page 198 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 *Transmission Component* on page 199 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 *Defining Object Material Properties* on page 200 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* on page 201 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 that 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 *Ray-tracing Rendering* on page 196 of the *PV-WAVE User's Guide*.

For details on the POLYSHADE and SHADE_VOLUME routines, see their descriptions in the *PV-WAVE Reference*.

MESSAGE Procedure

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

Usage

MESSAGE, *text*

Input Parameters

text — A text string containing the message.

Input 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.

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.

Output Keywords

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.

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 */Continue* or */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 — 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*.

Input Keywords

Max — Used to specify a variable to hold value of the largest 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, 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
```

See Also

AVG, MAX, MEDIAN

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

None.

Discussion



Note

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 *Loading Your Own Color Tables: TVLCT* on page 313 of the *PV-WAVE User's Guide*.

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 PV-WAVE 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 PV-WAVE Date/Time data for today's date.

```
m = MONTH_NAME(dttoday)
PRINT, m
    April
```

See Also

DAY_NAME, DAY_OF_WEEK, !Month_Names,
!Day_Names

For more information, see Chapter 7, *Working with Date/Time Data*, in 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 — The initial rate, in approximate frames per second. If *rate* is omitted, the inter-frame delay is set at 0.01 second.

Input 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, x , of a linear set of equations, $Ax = b$. (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 $Ax = b$.

$alud$ — 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, it contains the improved solution.

Keywords

None.

See Also

LUDCMP, LUBKSB

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.

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)
    Display the number of elements in a.

6

DELVAR, a
    Delete the variable a.

INFO, a
VARIABLE    UNDEFINED = <Undefined>

PRINT, N_ELEMENTS(a)
0
```

See Also

`N_TAGS`, `N_PARAMS`, `SIZE`, `TAG_NAMES`

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.

Description

N_PARAMS is used to determine if user-written procedures or functions were called with positional (non-keyword) parameters.

See Also

N_TAGS, N_ELEMENTS, SIZE, TAG_NAMES

For more information on functions and procedures, see Chapter 9, *Writing Procedures and Functions*, in the *PV-WAVE Programmer's Guide*.

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
      Create the structure.
```

```
** Structure EXAMPLE, 3 tags, 32 length:
```

```
T1 INT          Array(3)
T2 FLOAT        7.00000
T3 STRUCT       -> FIELD3 Array(1)
```

```
PRINT, N_TAGS(b)
      Display the number of tags in b.
```

```
3
```

```
PRINT, N_TAGS(b.t1)
      Display the number of tags in each field of b.
```

```
0
```

```
PRINT, N_TAGS(b.t2)
      0
```

```
PRINT, N_TAGS(b.t3)
      2
```

See Also

`STRUCTREF`, `TAG_NAMES`, `SIZE`, `N_ELEMENTS`,
`N_PARAMS`

ON_ERROR Procedure

Determines the action taken when an error is detected inside a PV-WAVE 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

None.

Example

In this example, a procedure named PROC1 calls a procedure, PROC2, which 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 non-existent procedure, PAINT, being called. The ON_ERROR procedure, which is called from PROC2, is passed different argument 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, 10 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, OPENR, OPENU, OPENW, READ, RETALL, RETURN, STOP, WRITEU

For more information, see *Error Handling in Procedures* on page 250 of the *PV-WAVE Programmer's Guide*.

Additional information can be found in *Description of Error Handling Routines* on page 258 of the *PV-WAVE Programmer's Guide*.

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`, `OPENR`, `OPENU`, `OPENW`, `READ`, `RETALL`, `RETURN`, `STOP`, `WRITEU`

Additional information can be found in *Description of Error Handling Routines* on page 258 of the *PV-WAVE Programmer's Guide*.

For more information on system variables, see Chapter 4, *System Variables*.

For more information on statement labels, see *Statement Labels* on page 52 of the *PV-WAVE Programmer's Guide*.

For background information, see Chapter 8, *Working with Data Files*, in the *PV-WAVE Programmer's Guide*.

OPEN Procedures (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 VMS, 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 VMS 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 VMS.
- Under VMS, 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 — (VMS 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.

Input 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 — (VMS 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 — (VMS 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.

Extendsize — (VMS 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, VMS 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 — (VMS 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 — (VMS 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 — (VMS 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 — (VMS only) Specifies that the file has indexed organization.

List — (VMS 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 VMS, 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, 'text.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 — (VMS only) Specifies that explicit carriage control will be used when you create a new file. This means that VMS 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 — (VMS only) If present, sends the file to `SYS$PRINT` (the default system printer) when it is closed.

Segmented — (VMS only) Specifies that the file has VMS FORTRAN-style segmented records. Segmented records allow logical records to exist with record sizes that exceed the maximum possible physical record sizes supported by VMS. Segmented

record files are useful primarily for passing data between FORTRAN and PV-WAVE programs.

Shared – (VMS 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, VMS 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 – (VMS only) Specifies that the file will be opened in stream mode.

Submit – (VMS 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 – (VMS 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 – (VMS 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 VMS, 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 VMS, allowed only with stream files.) 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.

Output Keywords

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 mes-
    sage to the standard error file (logical unit -2).
```

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.
```

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 *External Data Representation (XDR) Files* on page 205 of the *PV-WAVE Programmer's Guide*.

OPLLOT Procedure

Plots vector data over a previously drawn plot.

Usage

OPLLOT, 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 — A vector. If two parameters are supplied, y is plotted as a function of x .

Keywords

OPLLOT keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords*.

Background	Psym	XTicks	YTitle
Channel	Subtitle	XTickv	YType
Charsize	Symsize	XTitle	
Charthick	T3d	XType	ZCharsize
Clip	Thick		ZGridstyle
Color	Tickformat	YCharsize	ZMargin
Data	Ticklen	YGridstyle	ZMinor
Device	Title	YMargin	ZRange
Font		YMinor	ZStyle
Gridstyle	XCharsize	YNozero	ZTickformat
Linestyle	XGridstyle	YRange	ZTicklen
Noclip	XMargin	YStyle	ZTickname
Nodata	XMinor	YTickformat	ZTicks
Noerase	XRange	YTicklen	ZTickv
Normal	XStyle	YTickname	ZTitle
Nsum	XTickformat	YTicks	ZValue
Polar	XTicklen	YTickv	
Position	XTickname		

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 Advantage 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.

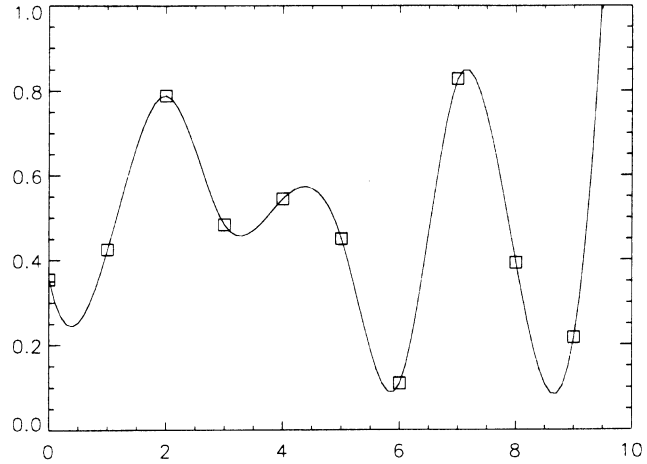


Figure 2-23 Overplotting a plot.

See Also

PLOT, PLOTERR, OPLOTERR

For more information, see Chapter 3, *Displaying 2D Data*, in the *PV-WAVE User's Guide*.

OPLOTERR Procedure

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

Usage

`OPLOTERR, 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 — Specifies the plotting symbol to use. It corresponds to the PV-WAVE 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
OPLOTERR, x, y, error, psym
```

This produces the plot below:

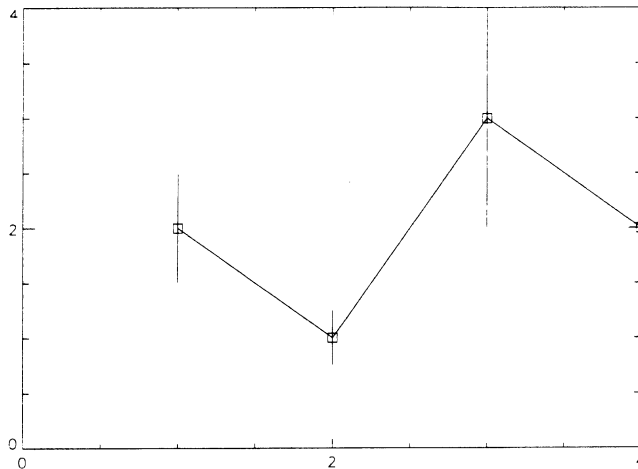


Figure 2-24 In this example, OPLOTERR was used to plot error bars over the x and y vectors, using the square symbol at the data values.

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 PV-WAVE Advantage 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.

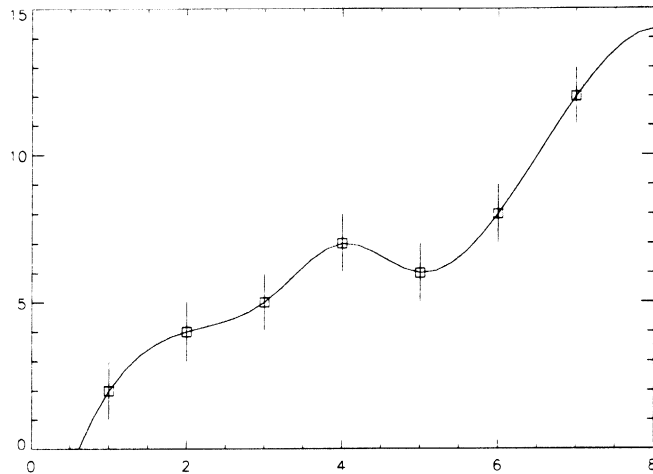


Figure 2-25 Scattered data interpolant with overplotted data points and symmetric error bars.

See Also

ERRPLOT, PLOTERR, !P.Psym

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 — 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 2-26.

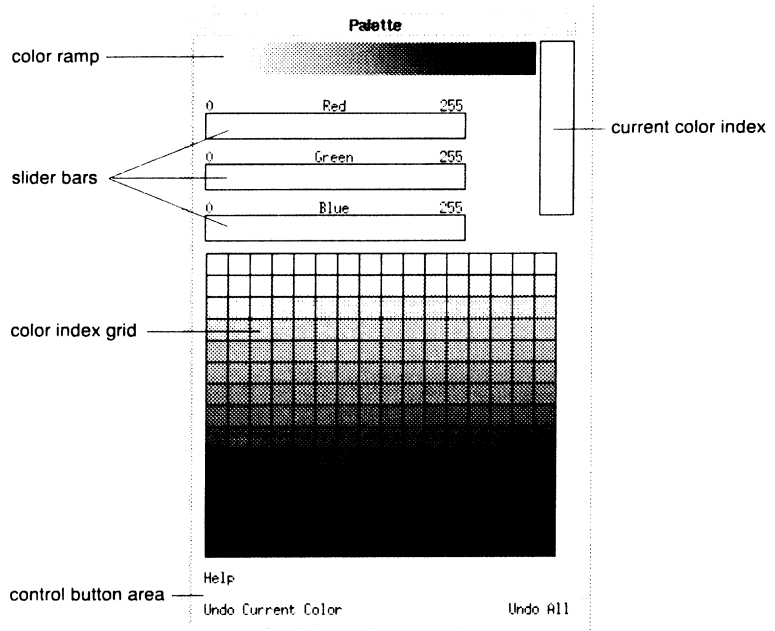


Figure 2-26 The PALETTE window lets you use the mouse to create a new color table interactively.

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
```

— User modifies the color table and exits the procedure. —

```
SAVE, filename='my_colortable', rgb_array
LOADCT, 2
RESTORE, 'ry_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
```

— 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, MODIFYCT,
PALETTE, WgCeditTool

For more ideas about what can be done with color tables, see *Modifying the Color Tables* on page 314 of the *PV-WAVE User's Guide*.

PLOT Procedures **(PLOT, PLOT_IO, PLOT_OI, and 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 — A vector. If two parameters are supplied, y is plotted as a function of x .

Keywords

Valid keywords for the four PLOT procedures are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords*.

Background	Normal	XRange	YTickname
Box	Nsum	XStyle	YTicks
Channel	Polar	XTickformat	YTickv
Charsize	Position	XTicklen	YTitle
Charthick	Psym	XTickname	YType
Clip	Start_Level	XTicks	
Color	Subtitle	XTickv	ZCharsize
Compress	Symsize	XTitle	ZGridstyle
Data	T3d	XType	ZMargin
Device	Thick		ZMinor
DT_Range	Tickformat	YCharsize	ZRange
Font	Ticklen	YGridstyle	ZStyle
Gridstyle	Title	YMargin	ZTickformat
Linestyle	Week_Boundary	YMinor	ZTicklen
Max_Levels		YNozero	ZTickname
Month_Abbr	XCharsize	YRange	ZTicks
Noclip	XGridstyle	YStyle	ZTickv
Nodata	XMargin	YTickformat	ZTitle
Noerase	XMinor	YTicklen	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.

```
O PLOT, 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_00, 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 Advantage 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.

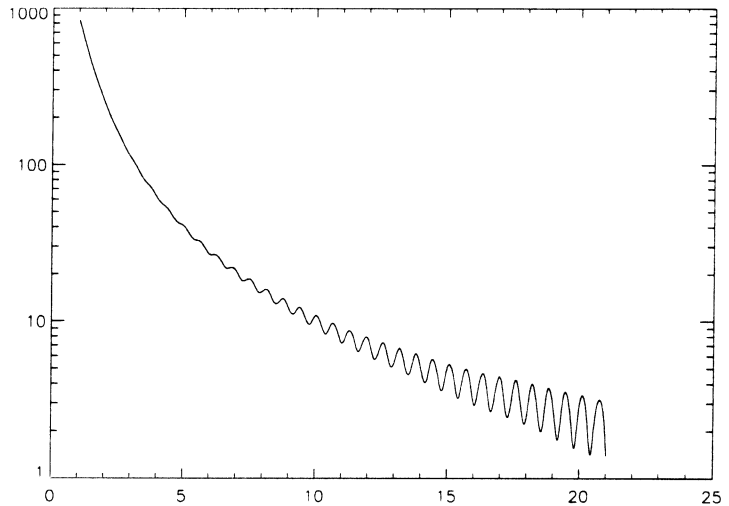


Figure 2-27 PLOT_IO plot of $f(x) = 1000 \sin(1/x^2) + \cos(10x)$.

Example 4

This example uses the PV-WAVE Advantage 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.

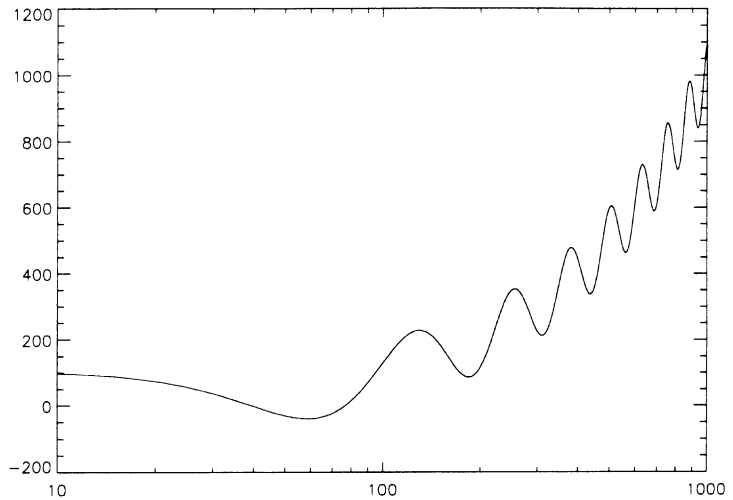


Figure 2-28 PLOT_OI plot of $f(x) = x + 100 \cos(0.05x)$.

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 axis.

```
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.

```
OPLLOT, 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.

ylabel = STRARR(6)

FOR i = 0, 5 DO ylabel(i)='$' + $
    STRTRIM(string(100L * i * 2), 2)
    Create labels on the Y axis that start with a '$'.

PLOT, x, y, /Month_abbrev, /Box$
    Title='Test Date Plot', YRange=[0, 1000], $
    YStyle=1, YTickname=ylabel, 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 3, *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 — 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.

Input 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 PV-WAVE 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:

```
OXPLOT, y
```

This produces the plot below:

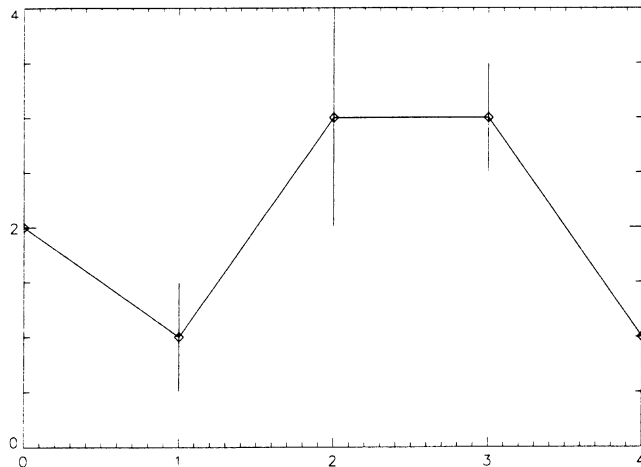


Figure 2-29 In this example, PLOTERR was first used to plot data points with their accompanying symmetrical error bars, and then OXPLOT was used to overplot a line through the error bar.

See Also

ERRPLOT, OXPLOT, OXPLOTERR, PLOT

PLOT_FIELD Procedure

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

Usage

PLOT_FIELD, u , v

Input Parameters

u – A two-dimensional array giving the field vector at each point along the X axis.

v – A two-dimensional array giving the field vector at each point along the Y axis.

Input 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. The default is 1.

Length – The length of the longest field vector, expressed as a fraction of the plotting area. The default is 0.1.

N – The number of arrows to draw. The default is 200.

Title – A string containing the title of the plot. The default is “Velocity Field”.

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 PV-WAVE 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
```

Display the velocity field with default values.

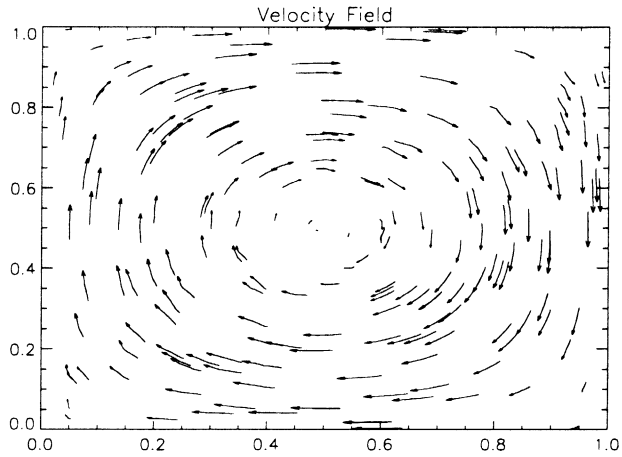


Figure 2-30 Velocity field displayed with default values.

```
PLOT_FIELD, u, v, N=400
```

Display the velocity field using 400 arrows.

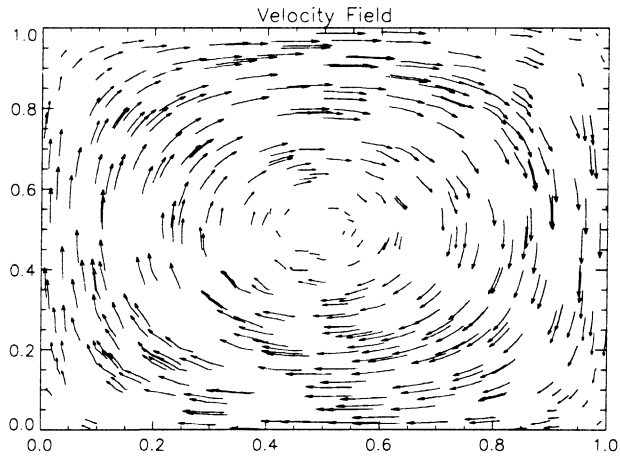


Figure 2-31 Velocity field displayed with 400 arrows.

```
PLOT_FIELD, u, v, Aspect=.7, Length=.4, N=20, $
Title='Example of PLOT_FIELD'
```

Display the velocity field with individual modifications.

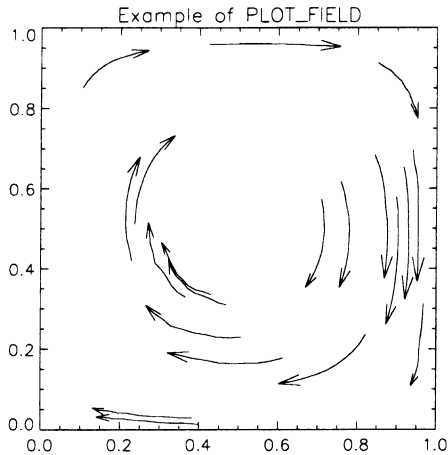


Figure 2-32 Velocity field modified with various keywords.

See Also

VEL, VELOVECT

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 — A vector parameter providing the Y coordinates of the points to be connected.

z — 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

PLOTS keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords*.

Channel	Fill_Pattern	Orientation	T3d
Clip	Linestyle	Pattern	Thick
Color	Line_Fill	Psym	Z
Data	Noclip	Spacing	
Device	Normal	Symsize	

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
```

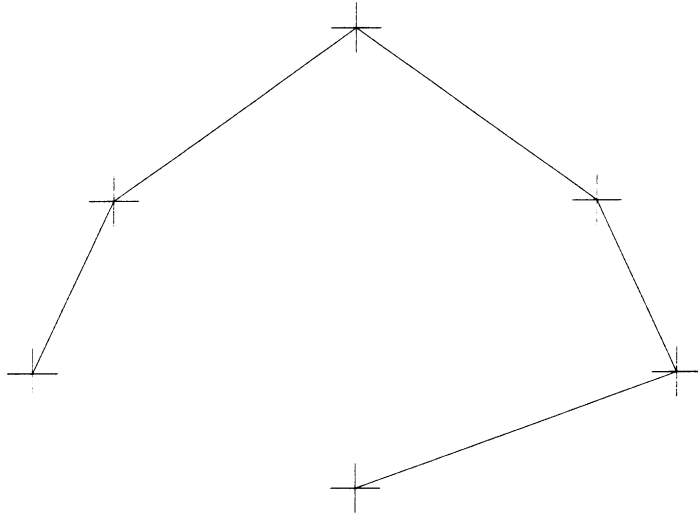


Figure 2-33 Connected lines drawn with PLOTS.

Example 2

This example plots a curve in \mathbb{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
```

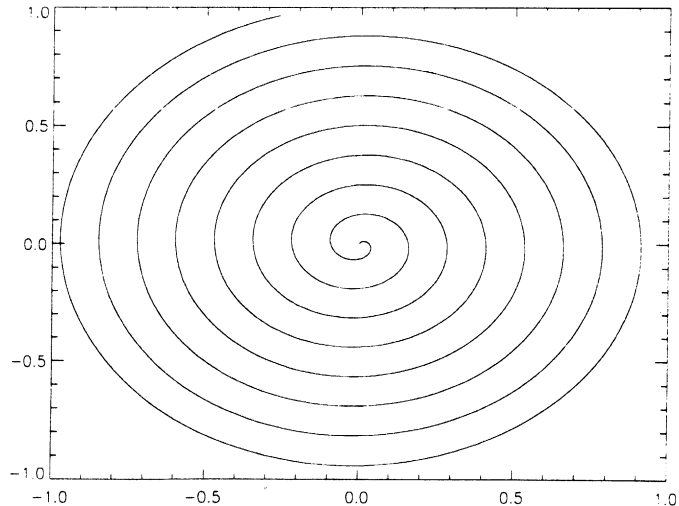


Figure 2-34 Plot of curve in \mathbb{R}^2 using PLOTS.

Example 3: Line Plot in \mathbb{R}^3

This example plots a curve in \mathbb{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
```

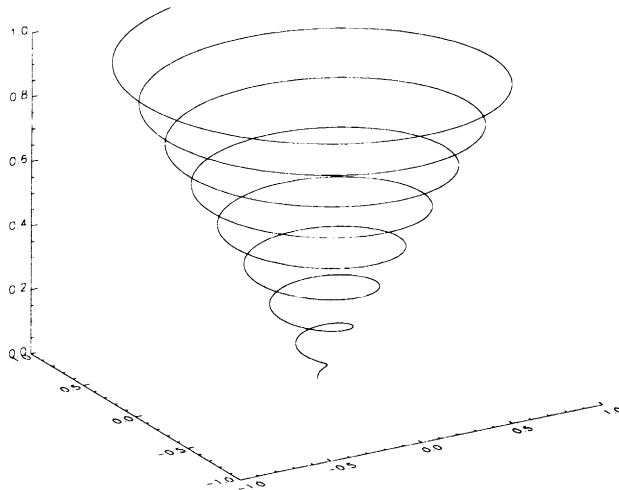


Figure 2-35 Line plot in \mathbb{R}^3 .

See Also

PLOT, OPLOT

For additional examples, see *Getting Input from the Cursor* on page 90 of the *PV-WAVE User's Guide* and *Procedure Used to Draw a House* on page 122 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, $expr_1, \dots, expr_n$

Input Parameters

$expr_i$ — Expression to be output.

Input Keywords

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.

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.

Description

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 2×3 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  $2 \times 3$  matrix.
```

```
row 0: 11 22 33
```

```
row 1: 40 50 60
```

```
PM, a
```

```
    Output the matrix to standard output.
```

```
11.0000      22.0000      33.0000
```

```
40.0000      50.0000      60.0000
```

See Also

PMF, RM, RMF

See *Matrices* on page 93 of the *PV-WAVE Programmer's Guide* for more information.

PMF Procedure

Performs formatted output of matrices to a specified file unit.

Usage

PMF, *unit*, *expr*₁, ..., *expr*_n

Input Parameters

unit — File unit to which the output is sent.

*expr*_i — Expression to be output.

Input Keywords

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.

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.

Description

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 2×3 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
```

```
  Output the matrix to standard output.
```

```
      11.0000      22.0000      33.0000
```

```
      40.0000      50.0000      60.0000
```

See Also

PM, RM, RMF

See *Matrices* on page 93 of the *PV-WAVE Programmer's Guide* 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 PV-WAVE file unit (logical unit number) for which the file position will be set.

position – A positive integer specifying the position of the file pointer as a byte offset from the start of the file.

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.

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.'
    Retrieve and display the current position within ptlun.dat.

Current offset into ptlun.dat is 200 bytes.

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 file unit number.

```

See Also

FREE_LUN, GET_LUN, OPENR, OPENU, OPENW, READ, WRITEU, FSTAT

For more information, see *Positioning File Pointers* on page 219 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 *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_FIT, POLYFITW, POLY_2D

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.

Examples

```
x = [1, 3, 2]
y = [1, 1, 4]
area = POLY_AREA(x, y)
PRINT, area
    3.00000

x = [2, 4, 4, 2]
y = [1, 1, 2, 2]
PRINT, POLY_AREA(x, y)
    2.00000
```

See Also

POLYCONTOUR, 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 *Vertex Lists and Polygon Lists* on page 187 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.

Examples

```
PRO sphere_demo2
```

This program displays data warped onto an irregular sphere.

```

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.

TV, 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 to plot in.

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 example, see the `poly_demo1` demonstration program in `$WAVE_DIR/demo/ar1`.

See Also

POLY_PLOT

For details on the POLYSHADE and SHADE_VOLUME routines, see their descriptions in the *PV-WAVE Reference*.

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_x — The X coefficients (see Discussion below).

coeff_y — The Y coefficients (see Discussion below).

interp — 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_x — If present, specifies the number of columns in the resulting array. Otherwise, the output has the same number of columns as *array*.

dim_y — 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*.

Output 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:


$$x' = a(x,y) = \sum_{i=0}^n \sum_{j=0}^n \text{coeff}_{x,i,j} x^i y^j$$


$$y' = b(x,y) = \sum_{i=0}^n \sum_{j=0}^n \text{coeff}_{y,i,j} x^i y^j$$

where n is the degree of the polynomials that are being used to produce the warping, and coeff_x and coeff_y are arrays containing the polynomial coefficient. Each array must contain $(n + 1)^2$ elements.

For example, for a linear transformation, coeff_x and coeff_y must contain four elements and may be either a 2-by-2 array or a 4-element vector. $Cx_{i,j}$ contains the coefficient used to determine x' , and is the weight of the term $x^i y^j$.

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.

Note  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 .

Example

Some simple linear (degree one) transformations are shown below:

$P_{0,0}$	$P_{1,0}$	$P_{0,1}$	$P_{1,1}$	$Q_{0,0}$	$Q_{1,0}$	$Q_{0,1}$	$Q_{1,1}$	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
<i>z</i>	0	0	0	0	0	0	0	Shift left by <i>z</i> pixels
0	1	0	0	0	0	1	0	Transpose

See Also

BILINEAR, POLY, POLY_FIT, POLYWARP

For details on interpolation methods, see *Efficiency and Accuracy of Interpolation* on page 170 of the *PV-WAVE User's Guide*.

POLYCONTOUR Procedure

Standard Library procedure that shades enclosed contours with specified colors.

Usage

POLYCONTOUR, *filename*

Input Parameters

filename – The name of the file containing the contour paths. This file must have been created with a CONTOUR call using the *Path_Filename* keyword.

Input Keywords

Color_Index – If present, must be set to 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, then *Color_Index* equals $\text{INDGEN}(25) + 1$.

Delete_File – If present and nonzero, deletes *filename* after POLYCONTOUR has finished processing.

Pattern – A three-dimensional array containing the patterns used to fill the various contour levels. Each pattern must be an n -by- m rectangular array of pixels of any size. (See the description of the *Pattern* graphics keyword in Chapter 3, *Graphics and Plotting Keywords*, for an example.)

If there are NP number of patterns specified, then *Pattern* will be dimensioned (n, m, NP) . The patterns will be used in order to fill the various contour levels. If there are more levels than patterns, the patterns will be cyclically repeated.

Discussion

POLYCONTOUR creates a contour plot with the area in between the contour lines filled with a solid color or a user-defined pattern. It uses the POLYFILL procedure with normalized coordinates to fill the contours on the display.

All contours must be closed before calling POLYCONTOUR, as it does not draw open contours. If not, an error message is printed and the procedure stops.



Tip

To eliminate open contours, surround the original array with a one-element border on all sides. These border elements must have a value smaller than or equal to the minimum data array value. The following code fragment demonstrates this tip:

```
new_a = REPLICATE(MIN(a), n+2, m+2)
    Make the background.

new_a(1, 1) = a
    Insert original data.

CONTOUR, new_a, Path='filename'
    Create contour path file.

POLYCONTOUR, 'filename'
    Fill contours with default colors.
```

Example

This example creates a contour plot of the PV-WAVE 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.

c_pikes = FLTARR(62, 42)
c_pikes(1, 1) = pikes
    Close any open contours.
```

```
TEK_COLOR
```

Load a color table.

```
CONTOUR, c_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.

```
POLYCONTOUR, 'path.dat'
```

Display the contour plot with contours filled with solid colors.

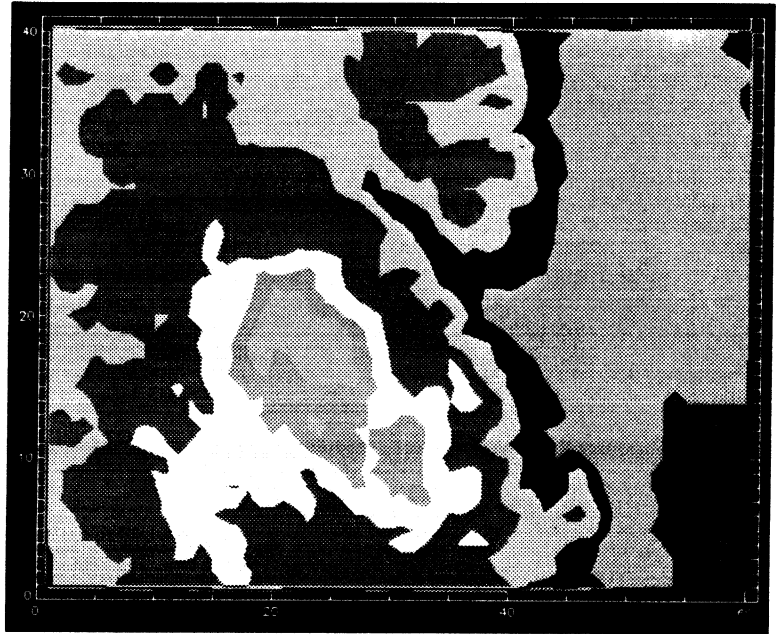


Figure 2-36 Contour plot filled with solid colors.

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.

POLYCONTOUR, 'path.dat', Pattern=pat3d, $
/Delete_file
    Display the contour plot with the contours lines filled with
    the pattern.

```

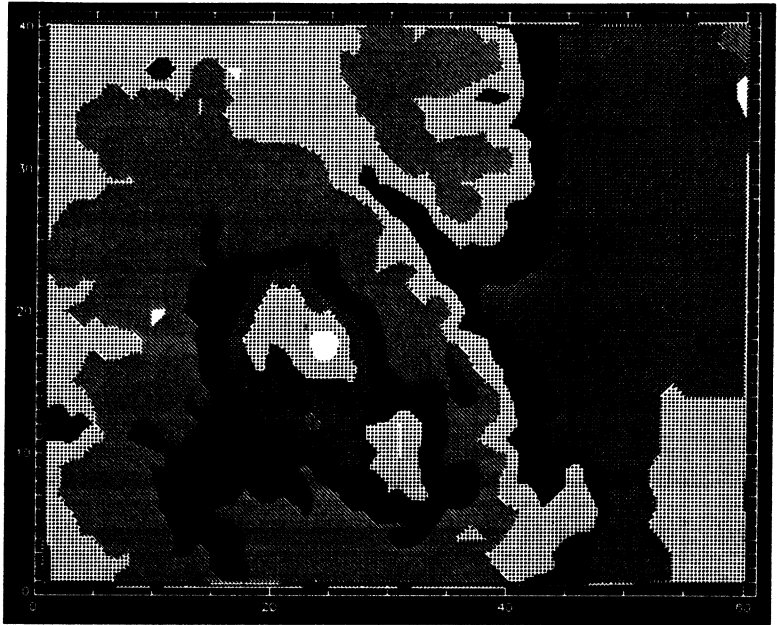


Figure 2-37 Pattern-filled contours.

Tip 

Instead of using POLYCONTOUR to create color-filled contour plots, a similar result can be achieved by loading a color table with the TEK_COLOR procedure 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 above example could be displayed using the commands:

```
TEK_COLOR
pikes=REBIN(pikes, 600, 400)
TV, BYTSCL(pikes, Top=10)
```

Here are some of the advantages to using this technique, rather than the POLYCONTOUR procedure, to create color-filled contour plots:

- You have easy access to the image processing routines that let you quickly analyze your data (these routines include DEFROI, HISTOGRAM, and PROFILES).
- You don't have to worry about open contours.
- You don't have to create a temporary file in your directory.

See Also

CONTOUR, POLYFILL, TEK_COLOR

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 *Vertex Lists and Polygon Lists* on page 187 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.

Examples

```
PRO sphere_demo2
```

This program displays data warped onto an irregular sphere.

```

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.

TV, 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 to plot in.

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 example, see the `poly_demo1` demonstration program in `$WAVE_DIR/demo/ar1`.

See Also

POLY_PLOT

For details on the POLYSHADE and SHADE_VOLUME routines, see their descriptions in the *PV-WAVE Reference*.

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 *Vertex Lists and Polygon Lists* on page 187 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.

Examples

See the Examples section in the description of the POLY_C_CONV routine.

For another example, see the `sphere_demo3` demonstration program in `$WAVE_DIR/demo/ar1`.

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* — The maximum X and Y dimension, respectively, in device coordinates. Usually, this is set to the X and Y size of the current PV-WAVE window.

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.

Examples

```
PRO poly_demo1
```

This program displays a perspective view of a surface from a viewpoint within the data.

```
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, see the following demonstration programs in `$WAVE_DIR/demo/ar1:sphere_demo2`, `sphere_demo3`, and `vol_demo4`.

See Also

POLY_NORM, POLY_TRANS

For more information, see *Coordinate Conversion* on page 193 of the *PV-WAVE User's Guide* and *Three Graphics Coordinate Systems* on page 59 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 — A vector parameter providing the Y coordinates of the points to be connected.

z — 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

POLYFILL keywords are listed below. For a description of each keyword, see Chapter 3, *Graphics and Plotting Keywords*.

Channel	Fill_Pattern	Pattern	T3d
Clip	Linestyle	Psym	Thick
Color	Line_Fill	Spacing	
Data	Noclip	Symsize	
Device	Normal		

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 *Z-buffer Output* on page 99 of the *PV-WAVE User's Guide*.

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 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 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 c.

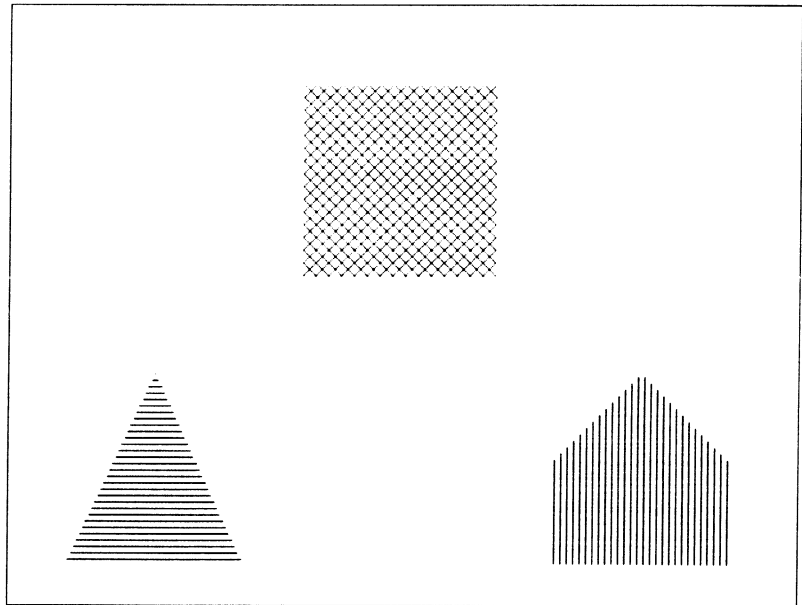


Figure 2-38 Pattern-filled polygons.

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, as shown below.

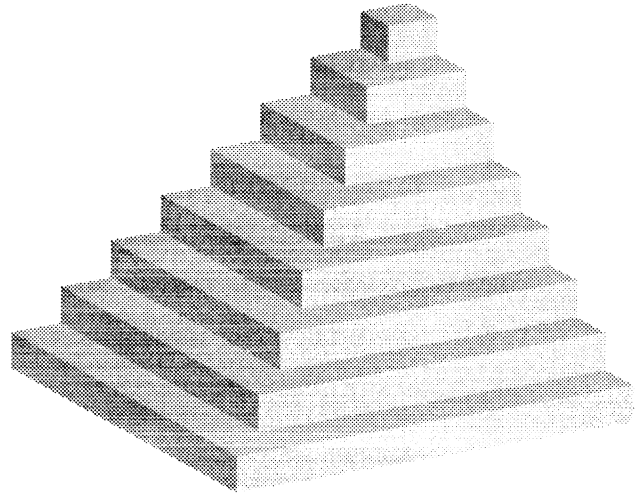


Figure 2-39 Example of three-dimensional polygon filling.

See Also

POLY_AREA, POLYCONTOUR

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 — 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 X (independent) coordinates of the data.

y — The vector containing the Y (dependent) 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 — The vector containing the calculated Y values.

ybd — The vector containing the error estimate of each point. (The error estimate is equal to one standard deviation.)

sig — The standard deviation of the function, expressed in the units of the Y direction.

mat — 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:

$$f(x) = result(3)x^3 + result(2)x^2 + result(1)x + result(0)$$

The *yfit* 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.
```

```
O PLOT, 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

Additional curve fitting algorithms are available with *PV-WAVE Advantage*.

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 X (independent) coordinates of the data.

y – The vector containing the Y (dependent) 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*.

deg – The degree of the polynomial to be fitted to the data.

Output Parameters

yft – The vector containing the calculated Y values.

ybd – The vector containing the error estimate of each point. (The error estimate is equal to one standard deviation.)

sig – The standard deviation of the function, expressed in the units of the Y direction.

mat – 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 149.

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 *Vertex Lists and Polygon Lists* on page 187 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.

Input 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.

Fill1 — A vector containing the fill colors for the first polygon list.

Fill2 — A vector containing the fill colors for the second polygon 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.

Output Keywords

Edge_List — The edge colors for the merged list.

Fill_List — The fill colors for the merged 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.

Examples

Spee the sphere_demo3 demonstration program in \$WAVE_DIR/demo/ar1.

See Also

POLY_PLOT

For details on the POLYSHADE routine, see its description in the *PV-WAVE Reference*.

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 the *PV-WAVE Reference*.)

Examples

See the Examples section in the description of the POLY_C_CONV routine.

For other examples, see the following demonstration programs in \$WAVE_DIR/demo/ar1: poly_demo1, sphere_demo3, and vol_demo4.

See Also

POLY_DEV, POLY_TRANS

For more information, see *Coordinate Conversion* on page 193 of the *PV-WAVE User's Guide* and *Three Graphics Coordinate Systems* on page 59 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.



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 *Vertex Lists and Polygon Lists* on page 187 of the *PV-WAVE User's Guide*.

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.

Input 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 PV-WAVE window.
- If *Image* is present, then no polygon edges are plotted and the *winx* and *winy* parameters are ignored.

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.

Output Keywords

Image — On output, contains the original image with the polygons plotted on it. This image may then be displayed using TV or other similar routines.

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 Examples section in the description of the POLY_DEV routine.

For other examples, see the following demonstration programs in \$WAVE_DIR/demo/ar1: sphere_demo2, sphere_demo3, and vol_demo4.

See Also

POLY_NORM, POLY_TRANS, POLY_DEV,
POLY_C_CONV

For details on the POLYSHADE routine, see its description in the *PV-WAVE Reference*.

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*₀, *i*₁, ..., *i*_{*n*-1}], 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.

Input 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.



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.

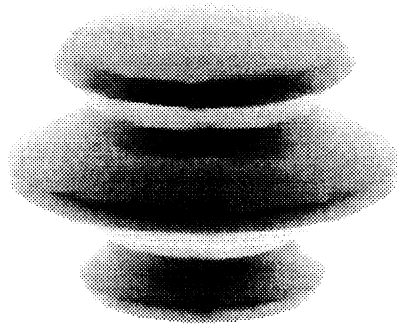


Figure 2-40 Isosurface at level 7 of volume dataset from example.

See Also

!P.T, SET_SHADING

For additional information on defining a coordinate system, see Chapter 4, *Displaying 3D Data*, in the *PV-WAVE User's Guide*.

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 *Vertex Lists and Polygon Lists* on page 187 of the *PV-WAVE User's Guide*.

Input Keywords

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).

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 π is mapped to the right 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).

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).

Degrees — If *Degrees* is present and nonzero, then the values for *XMin*, *XMax*, *YMin*, and *YMax* are in degrees instead of radians.

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
```

Examples

```
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 following demonstration programs in \$WAVE_DIR/demo/ar1: grid_demo5, sphere_demo2, and sphere_demo3.

See Also

GRID_SPHERE, POLY_SURF

For details on the POLYSHADE routine, see its description in the *PV-WAVE Reference*.

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 (see Discussion below).

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 (see Discussion below).

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 procedures. For more information, see *Vertex Lists and Polygon Lists* on page 187 of the *PV-WAVE User's Guide*.

Note

Examples

See the Examples 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 the section *Overview of Transformation Matrices* in Chapter 5, *Displaying Images*, in the *PV-WAVE User's Guide*.

Examples

See the Examples section in the description of the POLY_DEV routine.

For other examples, see the following demonstration programs in \$WAVE_DIR/demo/ar1: sphere_demo2, sphere_demo3, and vol_demo4.

See Also

POLY_DEV, POLY_NORM

For more information, see *Coordinate Conversion* on page 193 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 X independent coordinates. Must have the same number of points as *xd*.

yin — The vector containing the Y independent 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 $A_y = ym$.



Tip *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 4, *Displaying 3D Data*, in the *PV-WAVE User's Guide*.

For additional information, see *Geometric Transformations* on page 167 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 PUSH/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, PUSH is used to change the current working directory to the /sub1/data (UNIX) or [.sub1.data] (VMS) subdirectories under 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.

PRINTD

Display the current working directory and the directory stack.

PUSHHD, 'sub1/data'

Push the current working directory onto the directory stack, and move to the subdirectory /sub1/data (UNIX) or [.sub1.data] (VMS).

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.

Under VMS, a path specification that is consistent with that used while at the VMS command line prompt can be used in the argument to PUSHHD.

PUSHHD, '[.sub1.data]'

Push the current working directory onto the directory stack, and move to the subdirectory [.sub1.data] (VMS).

See Also

PUSHHD, PRINTD, CD

PRINT, PRINTF Procedures

Perform output of ASCII data:

PRINT performs output to the standard output stream (PV-WAVE file unit -1).

PRINTF requires the output file unit to be specified.

Usage

PRINT, *expr*₁, ... , *expr*_{*n*}

PRINTF, *unit*, *expr*₁, ... , *expr*_{*n*}

Input Parameters

unit — The file unit to which the output will be sent.

*expr*_{*i*} — The expressions to be output.

Input 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 Appendix A, *FORTRAN and C Format Strings*, 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 Table 8-7 on page 164 of the *PV-WAVE User'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 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

PUSHD, POPD, CD

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*.

Input Keywords

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.

Nomark — If set to 1, inhibits marking the selected line on the image display.

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.

Input 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 cerebral image found in the PV-WAVE directory `wave/data`, display this image in the current window and then 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.  
  
TV, cereb  
    Display the image.  
  
PROFILES, cereb  
    Display the profile window.
```

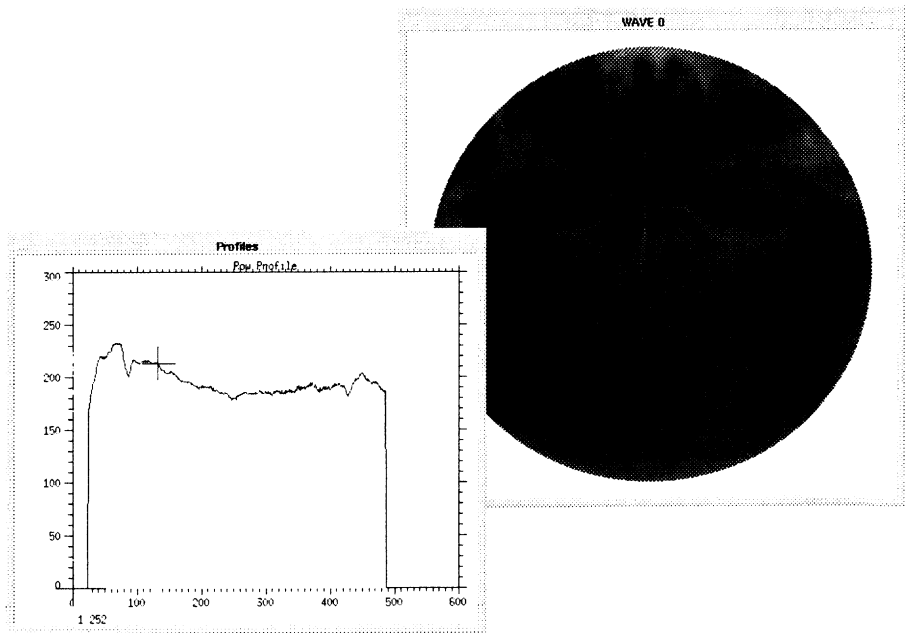


Figure 2-41 Profile plot taken from the image.

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

!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 — 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 *Understanding Color Systems* on page 305 of the *PV-WAVE User's Guide*.

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, pushes the current directory onto the stack. The current directory remains the working directory.
- 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

`POPD`, `PRINTD`, `CD`

Multivolume Index

This is a multivolume index that includes references to the *PV-WAVE User's Guide* (UG), *PV-WAVE Programmer's Guide* (PG), and both volumes of the *PV-WAVE Reference* (R1 and R2).

Symbols

! character UG-34
operator PG-46
\$ character UG-34
& character UG-35
' character UG-34
" character UG-35
() operator PG-44
* character UG-36
* operator PG-45, PG-85
*, in array notation PG-85
+ operator PG-45, PG-123
- operator PG-45
. character UG-35
/ operator PG-45
: character UG-36
; character UG-34
 after @ symbol UG-20
< operator PG-46
= operator PG-44
> operator PG-46
@ character UG-36
^ operator PG-46

A

A format code PG-A-9, PG-A-11
aborting
 plots UG-62
 PV-WAVE UG-15–UG-16
ABS function R1-30, PG-21
absolute value R1-30, PG-21
access mode, for VMS PG-225

accessing data in PV-WAVE PG-350
accumulated math error status R1-90,
 PG-263
ACOS function R1-31
actual parameters PG-234, PG-236
ADD_EXEC_ON_SELECT procedure
 R1-33
addition operator (+)
 examples PG-45
 for concatenating strings PG-123
 operator precedence PG-32
ADDSYSVAR procedure R1-34
ADJCT procedure R1-36, UG-331
allocating colors UG-312
ALOG function R1-38
ALOG10 function R1-40
ambient component of color UG-199
ampersand
 in command files UG-21
 separating multiple statements
 UG-35
AND operator
 description of PG-48
 operator precedence PG-33
 truth table PG-43
animation
 controlling the pace R2-280,
 R2-315
 cycling through images R2-280
 double-buffering on 24-bit displays
 UG-A-79
 of data in pixmaps R2-278, R2-316

- of images R2-277, R2-315
- of images using pixmaps UG-A-85
- annotation
 - alignment of the text R2-497
 - axes tick marks R2-527, R2-531, R2-535, R2-559
 - centering of text R2-497
 - character size R2-501, R2-518, R2-543, R2-554
 - color of R2-504
 - contour plots R2-499
 - controlling tick marks of UG-78
 - display without data R2-511
 - margin for R2-525, R2-530, R2-534
 - on axes R2-524, R2-529, R2-533
 - plots R2-490, UG-68–UG-69
 - positioning text, example UG-91
 - three-dimensional orientation of text R2-521
 - width of text string R2-524
 - with hardware fonts UG-291
 - with software fonts UG-294, UG-302
- X axis R2-520, R2-527
- Y axis R2-531
- Z axis R2-535
- application programming
 - create special effects with
 - graphics functions UG-A-94
 - the write mask UG-A-94
 - look-and-feel of GUI PG-411, UG-51
 - not enough colors available UG-A-81
 - providing a GUI UG-A-70
 - setting resources PG-465
 - unexpected colors UG-A-96
 - WAVE Widgets PG-412, PG-414
 - Widget Toolbox PG-479, PG-494
- arc-cosine R1-31
- arcsine R1-41
- arctangent R1-45
- area of polygon R1-522
- arithmetic errors, checking for PG-263, PG-265
- arithmetic operations, overflow condition PG-19
- arrays
 - * in notation PG-85
 - accessing
 - by memory order PG-281, PG-282
 - large arrays PG-281
 - arbitrary type, creating R1-449
 - assignment statements PG-55–PG-56
 - average of R1-46
 - building tables from R1-63
 - calculating determinant of R1-248
 - columns in PG-47
 - combining PG-88–PG-91
 - concatenation operators PG-8, PG-47
 - contouring 2D UG-94
 - correlation coefficient of R1-133
 - creating
 - dynamically R1-449
 - from unique elements R2-230
 - with arbitrary initialization R2-31
 - day of the year for each date in a date/time variable R1-155
 - definition of PG-4
 - degenerate dimensions PG-86
 - double-precision type R1-156
 - elements, number of R1-469
 - eliminating unnecessary dimensions PG-83
 - extracting fields from PG-37
 - finding the maximum value R1-452
 - floating type R1-340, R1-343
 - functions to create from various data types PG-39
 - in structures PG-112, PG-114
 - indexed by rows and columns PG-81
 - integer type, creating R1-409, R1-413
 - linear log scaling R1-497
 - locating non-zero elements in R2-354

- log-linear scaling R1-497
- log-log scaling R1-497
- longword type, creating R1-442
- masking PG-49
- mean of R2-136
- median value of R1-454
- minimum value of R1-462
- number of elements R1-469
- reading
 - data into PG-180, PG-181
 - into from display R2-223, UG-142
 - with READF PG-181
- referencing a non-existent element PG-83
- reformatting R2-24
- replicating values of R2-31
- resizing of R1-125, R2-21
- rotating R2-48
- rows in PG-47
- scaling to byte values R1-73
- setting default range of R2-81
- shifting elements of R2-99
- size and type of R2-114
- smoothing of R2-119
- sorting contents of R2-125
- standard deviation of R2-136
- storing elements with array subscripts PG-91
- string array with names of days of the month R1-465
- string array with names of days of the week R1-153
- string, creating R2-111, R2-139
- subarrays PG-86
- subscripts
 - of other arrays PG-87–PG-88
 - of points inside polygon region R1-542, R1-550
 - storing elements PG-91
- subsetting PG-42
- sum elements of R2-200
- transposition of R2-204
- transposition of rows and columns PG-47
- ASCII
 - files
 - formatted PG-144
 - reading R1-161, R1-177
 - row-oriented PG-146
 - fixed format PG-155, PG-157
 - free format PG-155–PG-156
 - free format data, writing to a file R1-203, R1-212
 - input/output
 - comparison with binary PG-151
 - procedures PG-155
- ASIN function R1-41
- assignment operator (=) PG-44
- assignment statements
 - description of PG-9
 - examples PG-53–PG-54
 - form 4 PG-56
 - four forms of PG-53
 - using array subscripts with
 - form 2 PG-55
 - form 4 PG-57
 - with associated variables PG-59
- ASSOC function R1-43
 - description PG-154
 - example of PG-213, PG-214
 - exporting image data with PG-191
 - general form PG-212
 - in relation to UNIX FORTRAN programs PG-218
 - offset parameter PG-212, PG-216
 - use of PG-152
- associated variables R1-43
 - See also ASSOC function
 - description PG-24
 - in assignment statements PG-59
- asterisk, for subscript ranges UG-36
- at (@) character UG-36
- ATAN function R1-45
- attachments of widgets PG-422–PG-424
- attributes
 - See also resource file
 - of an expression PG-272

- of variables PG-24
- VMS record PG-226–PG-227
- average, boxcar R2-119
- AVG function R1-46
- axes
 - See also AXIS procedure
 - See also tick marks
 - adding to plots UG-87
 - additional R1-48
 - annotation of tick marks R2-527, R2-531, R2-535, R2-559
 - box style R2-526, R2-531, R2-534
 - color of R2-504
 - controlling
 - appearance R2-554
 - scaling R2-554
 - converting between data and normalized coordinates R2-557
 - coordinate systems for UG-59, UG-87, UG-89
 - data range of R2-526, R2-530, R2-534
 - data values of tick marks R2-560
 - date/time UG-221, UG-245
 - drawing and annotating UG-87
 - drawn without data R2-511
 - endpoints R2-560
 - exchange of UG-126
 - inhibiting drawing of UG-88
 - intervals R2-525
 - length of tick marks R2-522–R2-535, R2-559
 - linear R2-560
 - logarithmic R2-529, R2-532, R2-536
 - scaling R2-560, UG-84
 - multiple R1-48, UG-68, UG-87
 - output axis range R2-554
 - outward pointing tick marks R2-522
 - positioning of UG-85
 - range
 - input range R2-556
 - max and min of last plot R2-554
 - specification R2-526–R2-534, UG-65
 - specifying data to plot R2-556
 - saving the scaling parameters R2-517
 - scaling of R2-557, UG-63, UG-84
 - setting
 - style R2-557
 - thickness R2-558
 - size of annotation R2-524, R2-529, R2-533
 - styles of UG-64
 - suppressing of R2-526
 - title of R2-529, R2-532, R2-535, R2-560
- AXIS procedure R1-48, UG-87–UG-89

B

- background color
 - LJ-250 output UG-A-23
 - PCL output UG-A-27
 - plotting background color R2-498
 - PostScript output UG-A-40
 - setting R2-543
 - SIXEL output UG-A-61
- backing store, for windows UG-A-7
- back-substitution for linear equations R2-177
- bandpass filters R1-250, UG-164
- bar charts UG-76–UG-79
- bar, menu PG-426
- base 10 logarithm R1-40
- basis function, example of R1-137
- batch files
 - See command files
- BEGIN statement PG-60
- BESEL function R1-51
- BESELJ function R1-53
- BESELY function R1-55
- BILINEAR function R1-57
- bilinear interpolation R1-57, UG-141, UG-170

- binary
 - data PG-152
 - advantages and disadvantages of PG-151
 - input/output procedures PG-188
 - input/output, comparison with ASCII PG-151
 - reading FORTRAN files PG-224
 - transferring PG-154
 - transferring strings PG-197
 - UNIX vs. FORTRAN PG-199
 - files
 - comparison to human-readable files PG-151
 - efficiency of PG-188
 - record length PG-148
 - record-oriented PG-149
 - VMS PG-149
 - input/output
 - advantages and disadvantages of PG-151
 - procedures PG-154
 - routines PG-154
 - transfer of string variables PG-197
- BINDGEN function R1-60
- bit shifting operation R1-418
- bitmap, used for cursor pattern UG-A-72
- blank, removing from strings R2-140
- block mode file access (VMS) PG-225
- blocking window
 - dialog box PG-453
 - file selection box PG-457
 - popup message PG-449
- blocks of statements
 - BEGIN and END identifiers PG-60
 - definition PG-10
 - description of PG-60
 - END identifier PG-61
 - with IF statements PG-72
- Boolean operators
 - AND PG-43, PG-48
 - applying to integers PG-43
 - applying to non-integers PG-43
 - NOT PG-43, PG-50
 - operator precedence PG-32
 - OR PG-43, PG-50
 - table of PG-7, PG-43
 - XOR PG-50
- Bourne shell PG-293, PG-299
- box style axes R2-526, R2-531, R2-534
- BREAKPOINT procedure R1-61
- buffer, flushing output R1-294, R1-344
- BUILD_TABLE function R1-63, UG-267
- bulletin board widget PG-421
- Butterworth filters UG-163–UG-164
- button box widget R2-406, PG-433
- buttons (widgets)
 - active PG-429
 - iconic PG-430, PG-433
 - in dialogs PG-454
 - in messages PG-451
 - menu toggle PG-430
 - radio PG-436
 - sensitivity of PG-470
 - toolbox PG-433
- BYTARR function R1-66
- byte
 - See also BYTE function
 - arrays, creating R1-66
 - data
 - a basic data type PG-3, PG-17
 - converting to characters PG-37, PG-125
 - reading from XDR files PG-206–PG-207
 - scaling R1-73
 - shown in figure PG-146
 - type
 - converting to R1-68
 - extracting from R1-68
- BYTE function
 - description R1-68
 - for type conversion PG-37
 - using with STRING function PG-199
 - with single string argument PG-126
- BYTEORDER procedure R1-71
- BYTSCL function R1-73, UG-154

C

- C format strings
 - conversion characters PG-A-23
 - for data export PG-A-24
 - for data import PG-A-22
 - when to use PG-167
- C functions
 - CALL_WAVE PG-373
 - loading a UT_VAR PG-387, PG-392
 - used with CALL_UNIX PG-364
 - w_cmpnd_reply PG-368
 - w_get_par PG-366
 - w_listen PG-365
 - w_send_reply PG-367
 - w_smp1_reply PG-367
 - wavevars PG-321, PG-335
- C programs
 - accessing
 - from PV-WAVE with SPAWN PG-312
 - PV-WAVE's variable data space from PG-350—PG-351
 - as a client PG-382
 - as a server PG-380
 - CALL_WAVE as a client PG-382
 - calling
 - from within PV-WAVE PG-310
 - PV-WAVE (cwavec) PG-337
 - PV-WAVE (wavecmd) PG-316
 - using LINKNLOAD function R1-429
 - communicating with R2-232
 - creating XDR files PG-207—PG-208
 - ending PV-WAVE with waveterm command PG-315
 - linking to PV-WAVE PG-316
 - under UNIX PG-347
 - under VMS PG-349
 - reading files with PG-196
 - running PV-WAVE from PG-335, PG-342
 - sending commands to PV-WAVE PG-315
 - SETBUF function PG-310—PG-311
 - writing to PV-WAVE PG-194
 - C shell PG-299
 - C_EDIT procedure R1-81, UG-331
 - CALL_UNIX function R1-77, PG-359—PG-381
 - CALL_WAVE PG-371—PG-382
 - callback
 - adding to a Widget Toolbox widget PG-485
 - definition of PG-413, PG-485
 - for menu button PG-428
 - list PG-485
 - parameters, Motif PG-C-3—PG-C-6
 - parameters, OLIT PG-C-6—PG-C-12
 - procedures R1-311
 - registering R1-309
 - reason PG-485
 - scrolling list PG-447
 - calling PV-WAVE
 - cwavec routine PG-333, PG-340
 - cwavefor routine PG-333, PG-340
 - from a C program R1-429, PG-316
 - from a FORTRAN program (wavecmd) PG-318
 - from a statically linked program PG-333
 - from an external program PG-313
 - in a UNIX unidirectional environments PG-313
 - callwave object module PG-313
 - carriage control
 - VMS FORTRAN, table of PG-227
 - VMS output PG-226
 - case folding PG-127
 - case selector expression PG-62
 - CASE statement PG-10, PG-62
 - CD procedure R1-79, PG-302
 - CeditTool procedure R2-294
 - CENTER_VIEW procedure R1-86, UG-194
 - CGM output UG-A-9—UG-A-12
 - changing the PV-WAVE prompt R2-552

- characters
 - changing case of R2-150, R2-166, PG-127
 - concatenation PG-123
 - converting
 - from byte data PG-125
 - parameters to R2-144, PG-124
 - extracting R2-154, PG-131
 - inserting R2-157, PG-131
 - locating substring R2-154, PG-131
 - non-printable, specifying PG-22
 - non-string arguments to string routines PG-133
 - obtaining length of strings R2-148, PG-130
 - removing white space R2-140, R2-162, PG-128
 - semicolon UG-34
 - setting to lowercase R2-150
 - setting to uppercase R2-166
 - size of R2-501, R2-518, R2-543
 - axis notation R2-501, R2-518, R2-524, R2-529, R2-533, R2-554
 - software UG-291
 - special UG-33
 - string operations supported PG-121
 - thickness of R2-543
 - unprintable PG-125
 - vector-drawn UG-291
 - working with PG-121
- CHEBYSHEV function R1-89
- CHECK_MATH function R1-90, PG-36, PG-263, PG-265
- checking for
 - keywords PG-269
 - number of elements PG-270
 - overflow in integer conversions PG-264
 - parameters in procedures and functions PG-269
 - positional parameters PG-269
 - presence of keywords PG-271
 - size and type of parameters PG-272
 - validity of operands PG-264
- child processes PG-309—PG-310
 - spawning R2-126
- CINDGEN function R1-93
- clear output buffer R1-294, R1-344
- clear screen of graphics device R1-298
- client
 - C program as PG-382
 - definition of PG-359
 - in X Window Systems UG-A-69
 - linking with PV-WAVE PG-361
 - PV-WAVE as (CALL_UNIX) PG-362, PG-377—PG-381, PG-387
- clipping
 - graphics output R2-503
 - of graphics window R2-543
 - rectangle R2-503
 - strings R2-158
 - suppressing R2-511, R2-546
- CLOSE procedure R1-94
- closing
 - files R1-94
 - DC (Data Connection) functions PG-140
 - description PG-140
 - LUNs PG-140
 - graphics output file UG-A-5
 - widget hierarchy PG-468, PG-471, PG-485
- colon (:)
 - format code PG-A-10, PG-A-12
 - use in PV-WAVE UG-35
- color
 - See also color tables, colormaps
 - See also TEK_COLOR procedure as 6-digit hexadecimal value UG-A-91
 - can be viewed in bar R2-284, UG-317, UG-331
 - characteristics, determining UG-A-82
 - contours, filling UG-109
 - decomposed into red, green, and blue UG-A-79

- editing interactively R2-290, R2-292, R2-301, UG-316, UG-331
- histogram equalizing R1-383
- images PG-189
- in resource file PG-415
- in TIFF files PG-192
- indices
 - changing default UG-57
 - definition of UG-57, UG-305
 - displaying UG-332
 - for Tektronix 4510 rasterizer UG-A-62
 - interpretation of UG-58
- list of PG-463
- lists, converting R1-535
- making color table with HLS or HSV system UG-331
- map
 - See colormap, color tables
- model
 - See color systems
- number available on graphics device R2-538, UG-322
- obtaining tables from common block UG-321
- of
 - annotation R2-504
 - axes R2-504
 - contour levels R2-500
 - data R2-504
 - elements inside plots UG-311, UG-321
 - graphics output R2-504
 - surface bottom R2-499
- on 24-bit display UG-A-79
- on 8-bit display UG-A-79
- PostScript devices UG-A-39
- pseudo PG-189
- raster graphics UG-A-92
- represented by shades of gray PG-189
- reserving for other applications UG-A-84
- rotate R2-286, R2-296, R2-303
- running out of UG-A-81
- setting background R2-498, R2-543
- setting in WAVE Widgets PG-463
- specifying by name UG-A-92
- translation
 - See also color table
 - in X Windows UG-A-84
 - table UG-308, UG-A-71, UG-A-78, UG-A-85
- true-color UG-147
- unexpected UG-A-96
- vector graphics UG-311, UG-A-92
- with monochrome devices UG-310
- X Windows UG-A-77
- color bar
 - purpose of R2-284, UG-317, UG-331
 - shown in figure R2-298
- COLOR field of !P, changing of UG-A-66
- color systems
 - definition UG-305
 - HLS UG-309
 - HSV UG-309
 - RGB UG-306
- color tables
 - adding new UG-330
 - based on
 - HLS system UG-329—UG-330
 - HSV system UG-329
 - changing predefined UG-330
 - controlling contrast UG-331
 - copying R2-67, UG-328
 - creating R1-81, R1-95, R1-97, UG-149, UG-329
 - creating interactively R1-81, R1-97, R1-396, R1-398, R1-400, R1-494, R1-588
 - custom, creating R2-293
 - discussion of UG-145, UG-310
 - editing UG-149
 - from Standard Library procedures UG-329
 - interactively R2-290, R2-292, R2-301, UG-316, UG-331
 - expanding R2-142

- histogram equalizing R1-383, R1-386, UG-329
 - indices R2-294
 - listed in table UG-312
 - loading R2-219, UG-311, UG-329
 - from colors.tbl file UG-145
 - from variables UG-145, UG-311, UG-330
 - into device UG-145
 - predefined tables R1-435
 - lookup table UG-308, UG-A-79
 - modifying R1-36, R1-464, UG-330
 - to enhance contrast UG-155
 - obtaining UG-321
 - predefined UG-145, UG-311, UG-329
 - replacing R1-464
 - reversing R2-302, UG-320
 - rotating R2-302, UG-320
 - saving in TIFF file PG-192
 - stretching R2-142, R2-302, R2-303, UG-320, UG-330
 - supplied with PV-WAVE R1-435, R1-464, UG-145, UG-311, UG-329
 - switching between devices UG-327
 - Tektronix 4115 color table R2-195
 - utility widgets, shown in figure R2-298
- color wheel, shown in figure R2-298
- COLOR_CONVERT procedure R1-95
- COLOR_EDIT procedure R1-97, UG-331
- COLOR_PALETTE procedure R1-101, UG-332
- colormap
 - See also color tables
 - compared to color table UG-A-78
 - how PV-WAVE
 - chooses one UG-A-80
 - obtains UG-A-83
 - private, advantages, disadvantages UG-A-82
 - shared, advantages, disadvantages UG-A-81
- column-oriented
 - ASCII files PG-144
 - data
 - in arrays PG-47
 - reading R1-168, R1-184, R1-207, R1-216
 - transposing with rows PG-47
 - FORTRAN write PG-181
- combining
 - image and contour plots UG-100
 - images with three-dimensional graphics UG-129—UG-130
 - surface and contour plots UG-125
- command files
 - description of UG-19
 - difference from main programs UG-25
 - examples UG-20
 - executing UG-19
 - at startup UG-13, UG-48
 - guidelines for creating UG-21
 - terminating execution of UG-20
 - three methods of running UG-19
 - using ampersand and dollar sign with UG-21
- command interpreter, VMS PG-299
- command line
 - editing PG-401
 - entering interactive commands UG-17
 - keys UG-33
- command recall
 - description of UG-32
 - using function keys UG-33
 - with INFO procedure PG-401
- command widget R2-410, PG-459
- comment field
 - in statements PG-52
 - semicolon UG-34
- common block
 - colors UG-321
 - COMMON block procedure
 - examples PG-249
 - COMMON statement PG-63
 - creating common variables for procedures PG-249

- definition PG-12
- not needed with user data PG-467
- use of variables PG-63
- used to pass client data PG-486–PG-489
- communicating with
 - a child process PG-309
 - operating systems from PV-WAVE PG-291
- compilation
 - automatic PG-69
 - of one or more statements R1-313
 - run-time PG-273
- COMPILE procedure R1-104
- compiler messages, suppressing R2-553
- compiling
 - .RUN with filename PG-239
 - automatic PG-239, UG-22
 - compiler messages, suppressing PG-29
 - external programs PG-313
 - procedures and functions PG-239
 - saving compiled procedure R1-104
 - system limits PG-241
 - under UNIX for LINKLOAD PG-322, PG-326, PG-328
 - with EXECUTE function PG-273
- complex
 - arrays, creating R1-109, R1-112
 - conjugate R1-118
 - constants PG-20
 - data type R1-109, PG-3, PG-18
 - converting to R1-109
 - shown in figure PG-146
 - numbers PG-21
 - type arrays, creating R1-109
 - variables, importing data into PG-160
- COMPLEX function R1-109, PG-37
- COMPLEXARR function R1-112
- compressing date/time variables R1-272
- compression, of TIFF files PG-192
- Computer Graphics Metafile
 - See CGM
- concatenating
 - arrays PG-47
 - characters PG-123
 - strings PG-122–PG-123
 - text PG-123
- concurrent processes R1-309
- CONE function R1-113, UG-196
- conformance levels, for TIFF PG-192
- CONGRID function
 - description of R1-115
 - using to magnify or minify an image UG-141
- CONJ function R1-118
- connecting data points with lines UG-70
- constant shading UG-131
- constants
 - complex PG-20
 - double-precision PG-20
 - examples PG-19
 - floating-point PG-20
 - hexadecimal PG-18
 - integer PG-18
 - numeric PG-4
 - octal PG-18
 - ranges of PG-19
 - string PG-4, PG-21
 - using correct data types PG-280
- .CONTINUE command UG-16
- continuing program execution UG-15
- contour plots
 - 2D arrays UG-94
 - adding a Z axis R2-533
 - algorithms used to draw R2-506, UG-95
 - annotation of R2-499
 - cell drawing method R2-506
 - closing open contours with arrays UG-110
 - color of contour levels R2-500
 - coloring of UG-109
 - combining with
 - PLOT procedure UG-124
 - SURFACE procedure UG-125
 - combining with surface plots and images R2-104
 - creating R1-119

- cubic spline interpolation of R2-519
- default number of levels R2-511
- enhancing UG-97
- examples UG-94, UG-96–UG-99, UG-102, UG-105
- filling with color R1-529, UG-109
- filling with pattern R2-513
- follow method algorithm UG-95
- gridding irregular data R1-362
- labeling R2-501, UG-105
- levels to contour R2-508, R2-511
- line thickness of R2-504
- line-following drawing method R2-506
- linestyle of R2-502
- maximum value to contour R2-510
- overlying
 - an image UG-100
 - with scalable pixel devices UG-102
- placement of Z axis R2-533
- shading of R1-529
- size of annotation R2-500
- smoothing UG-108
- superimposing on surface plots R2-517
- CONTOUR procedure R1-119
- contrast
 - control UG-331
 - enhancement UG-153, UG-330
- control points, using to correct linear distortion UG-170
- Control-\, aborting PV-WAVE UG-15
- Control-C
 - interrupting PV-WAVE UG-15
 - using to abort plots UG-62
- Control-D, exiting PV-WAVE on a VMS system UG-14
- controls box, see sliders
- Control-Y UG-16, UG-34
- Control-Z
 - exiting
 - PV-WAVE on a VMS system UG-14
 - suspending PV-WAVE on a UNIX system UG-14
- CONV_FROM_RECT function R1-123, UG-193
- CONV_TO_RECT function R1-129, UG-193
- convenience widgets (Motif) PG-B-3
- conversion characters
 - C code PG-A-23–PG-A-25
 - C export PG-A-25
 - C import PG-A-23
 - FORTRAN PG-A-8
- conversion functions PG-36
- converting
 - between graphics coordinate systems R1-21, R1-123, R1-129, R1-538, R1-559, UG-60, UG-193
 - color lists R1-524, R1-535
 - data to
 - See also extracting data
 - byte type R1-68, R1-73
 - complex type R1-109
 - date/time R1-421, R2-58, R2-159, R2-238, UG-223, UG-229
 - double-precision type R1-267
 - floating-point type R1-340
 - integer type R1-338
 - longword integer type R1-442
 - string type R2-144
 - strings PG-124
 - date/time variables to
 - double-precision variables R1-286
 - numerical data R1-292
 - string data R1-289
 - strings for tables UG-282
 - degrees to radians PG-27
- DT_TO_SEC function UG-258
- DT_TO_STR procedure UG-256
- DT_TO_VAR procedure UG-257
- JUL_TO_DT function UG-233
- Julian day number to PV-WAVE date/time R1-421
- mixed data types in expressions PG-281
- radians to degrees PG-27

- scalars to date/time variables
 - R2-238
- SEC_TO_DT function UG-233
- STR_TO_DT function UG-229, UG-231
- string data to date/time variables
 - R2-159
- strings to
 - lower case PG-127
 - upper and lower case PG-122
 - upper case PG-127
- three-dimensional to
 - two-dimensional coordinates UG-119
- VAR_TO_DT function UG-232
- CONVOL function R1-125, UG-159, UG-162
- convolution R1-125, UG-159
- coordinate systems
 - constructing 3D UG-121
 - converting from one to another
 - R1-21, R1-123, R1-129, R1-538, R1-559, UG-60, UG-193
 - data R2-505
 - device R2-505
 - graphics UG-59, UG-61
 - homogeneous UG-115
 - in PV-WAVE UG-59
 - normalized R2-512
 - polar R2-515, UG-89
 - reading the cursor position R1-143, UG-90
 - right-handed UG-116
 - screen display UG-138
- copying
 - color tables R2-67, UG-328
 - pixels UG-A-72, UG-A-85
- CORRELATE function R1-133
- correlation and regression analysis, Hilbert transform R1-381
- correlation coefficient, computing for arrays R1-133
- COS function R1-135
- COSH function R1-136
- COSINES function R1-137
- count-intensity distributions R1-386
- CREATE_HOLIDAYS procedure
 - R1-138, UG-239–UG-240
- CREATE_WEEKENDS procedure
 - R1-140, UG-241
- cross product, calculating R1-142
- CROSSP function R1-142
- CSV files, generating R1-213
- cube, establishes frame of reference
 - R2-311, R2-332
- cubic splines
 - interpolation R2-132
 - to smooth contours R2-519, UG-108
- cursor
 - controlling position of with TVCRS UG-143
 - for X Windows system UG-A-72
 - hot spot of UG-A-73
 - manipulating with images R2-217
 - position, reading R1-143
 - positioning text with UG-91
 - selecting default UG-A-72
 - specifying pattern of UG-A-72
 - system variable, !C R2-537
- CURSOR procedure R1-143, UG-91
- curve fitting
 - Gaussian R1-353
 - least squares fitting R2-181
 - multiple linear regression R2-26
 - non-linear least squares R1-147
 - polynomial R1-552, R1-555
 - singular value decomposition
 - method R2-181
 - to a surface R2-172
- CURVEFIT function R1-147
- custom color tables, creating R2-293
- cut-away volumes, defining R2-253
- cwavec routine PG-333–PG-340
- cwavefor routine PG-333–PG-340
 - calling PV-WAVE PG-334
- CYLINDER function R1-150, UG-197

D

- !D system variables, fields of R2-537–R2-540
- damage repair of windows UG-A-7
- data
 - See also data files
 - 3D graphics UG-115
 - accessing with wavevars PG-350
 - adding to existing plots UG-66
 - annotating R2-490
 - building tables from R1-63
 - changing to another coordinate system UG-59, UG-119
 - color of R2-504
 - column-oriented R1-168, R1-184, R1-207, R1-216
 - converting
 - byte type to characters PG-125
 - coordinates UG-193
 - date/time variables to
 - double-precision variables R1-286
 - date/time variables to numerical data R1-292
 - date/time variables to strings R1-289
 - into date/time UG-229
 - scalars to date/time variables R2-238
 - coordinate systems R2-505, UG-59, UG-61
 - dense R1-320
 - drop-outs PG-58
 - export
 - with C format strings PG-A-22, PG-A-24
 - with FORTRAN format strings PG-A-22
 - extracting
 - See extracting data
 - fitting, cubic spline R2-132
 - fixed format I/O PG-152, PG-165, PG-175
 - floating-point, with format reversion PG-A-6
 - formatting
 - into strings PG-124, PG-A-1–PG-A-2
 - with STRING function PG-199
 - free format PG-139
 - ASCII I/O PG-159, PG-161, PG-164
 - output PG-164
 - gridding irregular R1-362
 - importing
 - from more than one file UG-187
 - with C format strings PG-A-22
 - with FORTRAN format strings PG-A-22
 - input/output
 - from a file PG-12
 - of unformatted string variables PG-197
 - irregular R1-362
 - linear regression fit to R2-26
 - logarithmic scaling UG-84
 - magnetic tape storage R2-38
 - manipulation UG-8, UG-191
 - maximum value to contour R2-510
 - range R2-526, R2-530, R2-534, R2-556
 - reading
 - 8-bit image PG-189
 - for date/time UG-228
 - from magnetic tapes R2-193
 - into arrays PG-180, PG-181
 - tables of formatted data PG-175
 - unformatted R2-17, PG-193
 - record-oriented PG-150
 - reduction before plotting R2-512
 - row-oriented PG-146
 - scaling to byte values R1-73
 - skipping over R2-115
 - smoothing of R2-119
 - sorting tables of formatted data PG-175
 - sparse R1-368

- three-dimensional scaling R2-57
- TIFF format PG-191
- transfer
 - using FORTRAN or C formats PG-165
 - with XDR files PG-206
- transformation
 - by T3D procedure R2-185, UG-116
 - rotating R2-48, R2-51, UG-117
 - scaling UG-117
- type conversion PG-125
- types of PG-17
 - XDR routines for PG-210
- unformatted PG-193
- user-defined coordinate system UG-121
- writing
 - date/time UG-255
 - to a file PG-12
 - to magnetic tape R2-194
 - unformatted PG-193
 - unformatted output R2-361
 - with output formats PG-164
- data files
 - CSV format R1-213, PG-154
 - logical records PG-147
 - reading
 - 24-bit image data R1-195
 - 8-bit image data R1-192
 - ASCII data from a specific file R2-17
 - ASCII data from the standard input stream R2-17
 - binary data from a specific file R2-17
 - fixed-format ASCII data R1-161
 - freely-formatted ASCII data R1-177
 - TIFF R1-198
 - row-oriented ASCII PG-146
 - writing
 - 24-bit image data R1-221
 - 8-bit image data R1-219
 - ASCII free format data R1-203
 - fixed format ASCII data R1-203
 - TIFF R1-221, R1-223
 - to a file in ASCII free format R1-212
- data types
 - byte PG-3
 - combining in arrays PG-39–PG-40
 - complex PG-3
 - constants PG-17
 - double precision PG-3
 - floating point PG-3
 - integer PG-3
 - longword PG-3
 - mixed, in
 - expressions PG-35
 - relational operators PG-42
 - seven basics PG-34
 - string PG-3
 - structure PG-3
 - variables PG-24
- date/time
 - creating empty variables UG-227
 - data
 - adding R1-270
 - compressing R1-270, R1-272
 - converting to UG-229
 - converting to double-precision variables R1-286
 - converting to numerical data R1-292
 - converting to string data R1-289
 - converting to strings for tables UG-282
 - decrementing values R1-283
 - determining elapsed time R1-277
 - duration R1-277
 - formats UG-230
 - generating day of the year for each date R1-155
 - holidays UG-239
 - in tables UG-281
 - incrementing values R1-270
 - plotting UG-243–UG-253, UG-267

- printing values R1-282
- reading into PV-WAVE PG-170
- removing holidays and week-ends R1-272
- subtracting R1-283
- templates for transferring PG-169
- transfer with
 - DC_READ_FIXED PG-174
- transferring with templates PG-168
- using Julian day for tables UG-283
- using to generate specific arrays R1-279
- weekends UG-241
- writing to a file UG-255
- DC_WRITE functions UG-255
- description of UG-221
- excluding days UG-239
- Julian day UG-225
- reading data UG-228
- recalc flag UG-226
- structures
 - elements of UG-225
 - importing data into PG-162
- templates PG-169
- variables
 - creating from scalars R2-238
 - with current system date and time R2-199
- DAY_NAME function R1-153, UG-259
- DAY_OF_WEEK function R1-154, UG-260
- DAY_OF_YEAR function R1-155, UG-261
- DBLARR function R1-156
- DC (Data Connection) functions
 - advantages of PG-153
 - description of PG-13
 - for simplified data connection PG-153
 - opening, closing files PG-140
 - table of PG-14
- DC_ERROR_MSG function R1-157
- DC_OPTIONS function R1-159
- DC_READ_24_BIT function R1-195, PG-154
- DC_READ_8_BIT function R1-192, PG-154
- DC_READ_FIXED function R1-161, PG-157, PG-174–PG-184, PG-187, PG-A-8, UG-252
- DC_READ_FREE function R1-177, PG-136, PG-156, UG-246–UG-248
- DC_READ_TIFF function R1-198, PG-154
- DC_WRITE_24_BIT function R1-221, PG-154
- DC_WRITE_8_BIT function R1-219, PG-154, PG-191
- DC_WRITE_FIXED function R1-203, PG-157
- DC_WRITE_FREE function R1-212, PG-137, PG-156
- DC_WRITE_TIFF function R1-223, PG-154, PG-191
- DCL
 - logical names R2-74
 - symbols
 - defining R2-74
 - deleting R1-238
- deallocating
 - file units R1-344
 - example R1-348, PG-222
 - LUNs PG-141, PG-143
- DEC
 - printers UG-A-58
 - terminals, generating output for UG-A-54
- decals, definition of UG-200–UG-201
- declaring variables PG-5
- decomposed color UG-A-79
- DECStation 3100, error handling PG-268
- DECW\$DISPLAY logical, for X Windows UG-A-69
- DECwindows Motif UG-A-69
- DEFINE_KEY procedure R1-226
- DEFROI function R1-232
- DEFSYSV procedure R1-236

degrees
 convert from radians PG-27
 convert to radians R2-541, PG-27

delaying program execution R2-274

DELETE_SYMBOL procedure R1-238, PG-297

deleting
 compiled functions from memory R1-239
 compiled procedures from memory R1-241
 DCL symbols R1-238
 ERASE procedure R1-298
 variables R1-244
 VMS logical names R1-240
 VMS symbols R1-238

DELFUNC procedure R1-239

DELLOG procedure R1-240, PG-296

DELPROC procedure R1-241

DELSTRUCT procedure R1-242, PG-104

DELVAR procedure R1-244, PG-104

demonstration gallery UG-179

demonstration programs UG-183

dense data R1-320

density function, calculating histogram R1-386, UG-329

DERIV function R1-246

Desc qualifier, for sorting columns UG-276

DETERM function R1-248

determinant, calculating R1-246, R1-248

device
 controlling output of R1-249
 coordinate systems UG-59, UG-61
 current parameters R2-537
 name of R2-539
 selecting output from R2-66

device drivers
 general information about UG-A-1
 getting information about PG-398
 selecting output from UG-A-2
 table of supported UG-A-2

DEVICE procedure R1-249, UG-A-4

dialog box R2-419, PG-453–PG-454

differentiation, numerical R1-246

diffuse component of color, for RENDER function UG-198

digital gradient function PG-237

DIGITAL_FILTER function R1-250

DILATE function R1-254

dimensions of expressions, determining PG-272

DINDGEN function R1-259

!Dir system variable R2-540

directory path
 !Path UG-20
 searching for procedures and functions R2-550

directory stack
 popping directories off of R1-578
 printing out R1-580
 pushing R1-590

directory, changing to R1-79

disappearing variables PG-251, PG-258

display
 color, control of UG-145, UG-310
 dithering UG-148
 of images R2-225
 of multiple plots on a page UG-83
 reading from R2-223, UG-142

\$DISPLAY environment variable, for X Windows UG-A-69

!Display_Size system variable R2-540

DIST function R1-260, UG-163

Distinct qualifier, for removing duplicate rows in tables UG-272

distortion, linear UG-170

dithering, different methods compared UG-148

division operator (/) PG-32

DOC_LIBRARY procedure R1-264

documentation of user routines R1-264

dollar sign
 as continuation character UG-34
 format code PG-A-10, PG-A-12
 in command files UG-21

DOUBLE function R1-267

double-buffering UG-A-79

- double-precision
 - arrays, creating R1-156, R1-259
 - constants PG-20
 - data type PG-17
 - data, converting to R1-267
 - variables, converting date/time variables to R1-286
- !Dpi system variable R2-540
- drawing area R2-422, PG-439
- driver
 - default settings
 - SIXEL UG-A-58
 - See device drivers
- DROP_EXEC_ON_SELECT procedure R1-269
- DT_ADD function R1-270, UG-237
- DT_COMPRESS function R1-272, UG-242
- DT_DURATION function R1-277, UG-239
- DT_PRINT procedure R1-282, UG-261
- DT_SUBTRACT function R1-283, UG-238
- DT_TO_SEC function R1-286, UG-258
- DT_TO_STR procedure R1-289, UG-256
- DT_TO_VAR procedure R1-292, UG-257
- DTGEN function R1-279
- !Dtor system variable R2-541
- dynamic
 - data types PG-35
 - structures PG-35

E

- eavesdrop mode, HPGL plotter output UG-A-18
- edge enhancement
 - Roberts method R2-45
 - Sobel method R2-122
- !Edit_Input system variable R2-541
- editing, command line UG-33

- editor
 - See also text widget
 - creating text PG-445
- eigenvalues and eigenvectors, determining R2-201
- 8-bit image data
 - how stored PG-189
 - writing data to a file R1-219
- 8-bit color UG-A-79
- elements
 - definition of PG-4
 - finding the number of R1-469
- ELSE
 - in CASE statement PG-62
 - in IF statement PG-71
- empty output buffer R1-294, R1-344
- EMPTY procedure R1-294
- Encapsulated PostScript
 - Interchange Format UG-A-34
 - keyword UG-A-33
 - with Microsoft Word UG-A-46
- END identifier PG-60–PG-61
- end of file
 - testing for R1-296
 - when applied to a pipe PG-310
 - writing to tape R2-276
- ENDCASE, end statement PG-62
- ENDIF, end statement PG-72
- ending PV-WAVE sessions PG-315, PG-335, PG-341
- ENDREPEAT, end statement PG-61
- ENDWHILE, end statement PG-61
- enhancing contrast, of images R1-383
- environment
 - after an error PG-250
 - manipulating R1-77
 - modifying UG-44
- ENVIRONMENT function R1-295, PG-294
- environment variables
 - \$DISPLAY for X Windows UG-A-69
 - adding R2-64, PG-293
 - changing R2-64, PG-292
 - DELETE_SYMBOL procedure (VMS) PG-297
 - DELLOG procedure (VMS) PG-296

ENVIRONMENT function (UNIX)
 PG-294
 GETENV function (UNIX) PG-294
 obtaining
 all R1-295
 values of R1-356, PG-294
 SET_SYMBOL procedure (VMS)
 PG-296
 SETENV procedure (UNIX) PG-293
 SETLOG procedure (VMS) PG-295
 translating PG-294
 TRNLOG function (VMS) PG-295
 UNIX PG-292
 VMS PG-295–PG-296
 WAVE_DEVICE UG-44, UG-45
 WAVE_DIR UG-45
 WAVE_STARTUP UG-48, UG-51
 EOF
 function R1-296, PG-171, PG-219,
 PG-310
 statement, example of PG-78
 testing for PG-219
 WEOF procedure, writing to end of
 mark on tape PG-229
 writing to tape R2-276
 EQ operator
 description of PG-48
 operator precedence PG-33
 equality
 See EQ operator
 ERASE procedure R1-298
 erasing
 screen of graphics device R1-298
 variables UG-27
 ERODE function R1-300
 !Err system variable R2-541
 !Err_String system variable R2-541
 error bars, plotting R1-305
 error handling
 "Plot truncated" message R2-515
 See also math errors
 arithmetic PG-263
 categories of PG-257, PG-258
 CHECK_MATH function R1-90,
 PG-263
 DECStation 3100 PG-268
 during program execution PG-258
 enabling/disabling math traps
 PG-266
 example of checking for PG-267
 floating-point PG-263
 for I/O PG-140
 function, evaluating R1-304
 input/output R1-479, PG-259
 list of procedures PG-257
 machine dependent handling
 PG-268
 math
 See math errors
 mechanism in procedures and func-
 tions PG-250
 MESSAGE procedure PG-251
 messages PG-261–PG-262
 prefix for R2-542
 setting report level for "DC" func-
 tions R1-159
 obtaining text of message R2-541
 ON_ERROR procedure PG-258
 ON_IOERROR procedure PG-259
 options for recovery PG-258
 overflow PG-264
 recovery from PG-250, PG-259
 running SunOS Version 4 PG-268
 Sun-3 and Sun-4 PG-268
 VMS PG-268
 ERRORF function R1-304
 ERRPLOT procedure R1-305
 escape sequences
 creating PG-22
 SIXEL output UG-A-61
 event handler
 definition R2-367, PG-487
 registering R2-367, PG-487
 event loop
 WAVE Widgets PG-471
 Widget Toolbox PG-494
 exclamation point UG-34
 excluding days from date/time variables
 UG-239
 exclusive OR operator PG-50
 EXEC_ON_SELECT procedure R1-309

EXECUTE function R1-313, PG-243–
PG-244, PG-273

executing

- command files UG-19
- existing functions and procedures
UG-23
- interactive programs UG-25
- main programs UG-24
- operating system commands
R2-126
- statements one at a time R1-313

executive commands

- ..LOCALS compiler directive
PG-244
- .CON UG-15, UG-29
- .GO UG-29
- .LOCALS PG-243, UG-31
- .RNEW UG-28, UG-29
- .RUN
 - compiling functions and proce-
dures PG-239
 - description UG-27
 - examples UG-29
 - l argument UG-28
 - syntax PG-239
 - t argument UG-28
- .SIZE PG-242–PG-243, UG-31
- .SKIP UG-30
- .STEP UG-30
- definition of UG-26
- table of UG-27

EXIT procedure R1-316, PG-335,
PG-341, UG-14

exiting

- an application
 - See closing widget hierarchy
- PV-WAVE
 - EXIT procedure, description
R1-316
 - on a UNIX system UG-14
 - on a VMS system UG-14

EXP function R1-317

expanding

- color tables R2-142
- images R2-21, R2-493
- explicitly formatted
 - See fixed format
- exponential function, natural R1-317
- exponentiation operator (^)
 - examples PG-46
 - operator precedence PG-32
- exporting data PG-191
- exposing windows R2-363
- expressions
 - data type and structure evaluation
PG-34
 - description of PG-6
 - efficiency of evaluation PG-279
 - evaluation of PG-35
 - finding attributes of PG-272
 - forcing to specific data types PG-36
 - general description PG-31
 - invariant PG-281
 - mixed data types in expressions
PG-35
 - overview of PG-6
 - structure of PG-34, PG-39
 - type of PG-34
 - with arrays PG-40
 - with mixed data types PG-281
- Extended Metafile System UG-A-13
- extracting data
 - See also converting data to
double-precision floating-point data
type R1-267
 - from variables PG-37, PG-38
 - Julian day numbers R1-421
 - longword integer type R1-442
 - string type R2-144
- extracting image plane UG-A-95

F

false

- definitions for different data types
PG-71
- representation of PG-42

Fast Fourier Transform R1-328

FAST_GRID2 function R1-319, UG-191

FAST_GRID3 function R1-322, UG-191

FAST_GRID4 function R1-325, UG-191

FFT function
 applied to images UG-162
 description R1-328

fields
 definition of PG-101
 extraction and conversion of PG-37
 reference to PG-108
 syntax PG-108

file selection box (widget) R2-426,
 PG-456–PG-458

FILEPATH function R1-331

files
 access mode, VMS PG-225
 allocating units R1-359, PG-139,
 PG-143
 closing R1-94, PG-140
 column-oriented ASCII PG-144
 compression of TIFF files PG-192
 CSV format PG-154
 deallocating LUNs R1-345,
 PG-141, PG-143
 formatting, codes for PG-A-9–
 PG-A-25
 free format ASCII I/O PG-156
 FREE_LUN procedure, deallocating
 files PG-143
 getting information about PG-220,
 PG-399
 header, example PG-A-3
 I/O with structures PG-116
 in UNIX PG-223
 indexed files for VMS PG-228
 inputting data from PG-12
 locating on disk PG-218
 logical records, changing size of
 PG-147
 logical units PG-138
 on magnetic tape R2-38
 opening R1-480, PG-138–PG-139
 organization of VMS PG-224
 organization options PG-144
 pointer, positioning R1-519,
 PG-219
 portable binary PG-205
 printing R1-580

procedures for fixed format I/O
 PG-165

reading R2-17
 C programs PG-196
 DC_READ_FIXED PG-177–
 PG-184, PG-187, PG-A-8
 FORTRAN binary data with
 PV-WAVE PG-224
 one input character R1-358
 READF PG-175–PG-185
 READU PG-195, PG-203
 unformatted R2-17

record-oriented binary files PG-149

row-oriented PG-146

skipping over R2-115

standard input, output and error
 PG-140

startup command UG-48

stopping batch R2-138

testing for end of file R1-296,
 PG-219

units, closing R1-94

VMS
 attributes PG-227
 fixed-length record format
 PG-217, PG-226
 record oriented PG-226
 when to open PG-140
 writing
 8-bit image data to R1-219,
 R1-221
 ASCII free format to R1-203,
 R1-212
 data to PG-12
 unformatted output R2-361
 with UNIX FORTRAN programs
 PG-200
 with WRITEU PG-195

XDR PG-205

filling
 contour plots UG-110
 pattern R2-506

filters
 2D UG-163
 bandpass UG-164
 Butterworth UG-164

- creating digital filters R1-250
- exponential high or low pass UG-164
- for
 - images UG-162–UG-163
 - mean smoothing R2-119
 - median smoothing R1-454
 - signals or images R1-260
- high pass UG-160, UG-164
- low pass UG-164
- Roberts UG-160
- Sobel UG-160
- FINDFILE function R1-333, PG-218
- FINDGEN function R1-335
- FINITE function R1-336, PG-36, PG-264
- finite values, checking for R1-336
- fitting with Gaussian curve R1-353
- FIX function
 - description R1-338
 - example PG-36–PG-37
- fixed format PG-156
 - ASCII I/O PG-155, PG-157
 - comparison with free format PG-152
 - data, examples of reading PG-176
 - I/O PG-165, PG-175
- fixed-length record format, VMS PG-217, PG-226
- FLOAT function R1-340, PG-37
 - example PG-21
- floating-point
 - arrays, creating R1-340, R1-343
 - constants PG-20
 - data type PG-3, PG-17
 - errors PG-263
 - format code defaults PG-A-14
 - type, converting to R1-340, R1-343
- Floyd-Steinberg dithering UG-148
- FLTARR function R1-343
- flush output buffer R1-294, R1-344
- FLUSH procedure R1-344, PG-158, PG-219
- flushing file units PG-218
- Font field of !P UG-293
- fonts
 - 3D transformations UG-292
 - appearance (hardware vs. software) UG-291
 - changing UG-295
 - character size R2-501, R2-518
 - computer time for drawing UG-293
 - flexibility (hardware vs. software) UG-293
 - formatting commands UG-293
 - hardware R2-507
 - hardware vs. software UG-291
 - Hershey character sets UG-291
 - index of
 - graphics text font R2-544
 - hardware font R2-544
 - list of PG-464
 - portability (hardware vs. software) UG-292
 - positioning commands UG-301
 - PostScript
 - See PostScript fonts
 - selecting R2-507
 - to annotate plots UG-68
 - selection commands UG-295
 - setting in WAVE Widgets PG-464
 - software R2-507, R2-563
 - specifying R2-507
 - vector-drawn R2-507, UG-291
 - with graphic routines R2-507
 - with plotting routines R2-507
- FOR statement
 - definition PG-9
 - examples PG-65, PG-66
 - (variable increment) PG-67
 - explicit increment PG-67
 - format with a block statement PG-60
 - two forms of PG-65
- force fields
 - examples PG-65
 - increment parameter PG-65
 - plotting R1-510, R2-244, R2-248
 - simple PG-65

- form layout (widgets)
 - See also layout (WAVE widgets) PG-422
- formal parameters
 - copying actual parameters into PG-236
 - definition of PG-235
- format
 - C and FORTRAN for data transfer PG-165
 - for
 - DC (Data Connection) functions PG-169
 - writing data PG-164
 - interpreting format strings PG-166
 - tick labels UG-81–UG-82
- format codes
 - C PG-A-22
 - examples of integer output PG-A-17
 - floating-point
 - defaults PG-A-14
 - output PG-A-15
 - FORTRAN PG-A-9–PG-A-21
 - group repeat specifications PG-A-7
 - integer defaults PG-A-16
- format reversion PG-167, PG-A-5
- format strings
 - C, for data import and export PG-A-22
 - description of PG-A-1
 - FORTRAN PG-185, PG-A-22
 - when to use PG-A-2
- formatted data
 - commands for software fonts UG-293
 - rules for PG-159, PG-165
 - strings PG-124
 - structures for I/O PG-116
 - using STRING function PG-187
- FORTRAN
 - binary data, reading with PV-WAVE PG-224
 - ending PV-WAVE with waveterm PG-315
 - format codes PG-A-9–PG-A-21
 - format strings PG-A-8
 - reading multiple array elements PG-185
 - programs
 - calling PV-WAVE (wavcmd) PG-318
 - linking to PV-WAVE under UNIX PG-348
 - reading
 - data in relation to UNIX PG-199
 - multiple array elements PG-185
 - sending commands to PV-WAVE
 - wavcmd PG-315
 - writing data to PV-WAVE PG-181–PG-185
- 4D gridding R1-325, R1-369, UG-191
- Fourier
 - spectrum, 2D UG-165
 - transform
 - See FFT function
- free format
 - ASCII I/O PG-155–PG-156
 - input PG-159–PG-160
 - output PG-164
- FREE_LUN procedure R1-345, PG-143–PG-144
- frequency
 - domain techniques UG-162
 - image UG-163
- FSTAT function R1-346, PG-220–PG-221
- FUNCT procedure R1-350
- FUNCTION definition statement PG-235
- function keys
 - defining R1-226
 - equating to character strings UG-52
 - for line editing UG-33
 - getting information about R1-410, PG-400
- functions
 - actual parameters PG-234
 - calling PG-11
 - calling mechanism PG-248

- checking for
 - keyword parameters R1-422
 - number of elements R1-469
 - parameters PG-269
 - positional parameters R1-471
- compiling PG-239
 - automatically PG-69, PG-239
 - with .RUN and filename PG-239
 - with interactive mode PG-240
- conditions for automatic compilation PG-234
- copying actual parameters into formal parameters PG-236
- creating PG-15
 - interactively UG-26
 - with a text editor UG-22
- definition of PG-233
- definition statement of FUNCTION PG-235
- deleting from memory R1-239
- determining number of parameters PG-269
- directory path for R2-550, PG-28
- documentation of user R1-264
- error handling PG-250
- formal parameters PG-235
- I/O errors in R1-479
- I/O for simplified data connection PG-13
- information, obtaining with INFO procedure PG-401
- keyword parameters PG-68, PG-74
- libraries of VMS PG-251
- library UG-23
- maximum size of code PG-397
- number of parameters required PG-236
- overview of PG-2, PG-14
- parameters PG-74, PG-234
 - passing mechanism PG-246
 - positional PG-68, PG-74, PG-235
- program control routines PG-273
- recursion PG-248
- required components of PG-237

- search path UG-46–UG-50
- syntax for
 - calling PG-15
 - defining PG-15
- type conversion PG-36
- user-defined PG-241

G

- GAMMA function R1-352
- GAUSSFIT function R1-353
- Gaussian
 - curve fitting R1-353
 - function, evaluating R1-355
 - integral R1-355
- GAUSSINT function R1-355
- GE operator
 - description of PG-49
 - for masking arrays PG-49
 - operator precedence PG-33
- geometric transformations UG-167, UG-168
- GET_KBRD function R1-358, PG-154, PG-222, PG-223
- GET_LUN procedure R1-359, PG-143
- GET_SYMBOL function R1-361
- GETENV function R1-356, PG-294
- GOTO statement PG-10, PG-52, PG-70
- Gouraud shading R1-564, R2-70–R2-71, UG-131
- graphical user interface
 - See GUI
- graphics
 - 2D array as 3D plot R2-197
 - 3D to 2D transformations R2-549
 - bar charts UG-76
 - changing graphics device UG-45
 - clipping R2-503
 - output R2-503
 - window R2-543
 - combining
 - CONTOUR and PLOT procedures UG-127
 - CONTOUR and SURFACE procedures UG-126

- connecting symbols with lines
 - R2-548
- converting from 3D to 2D coordinates
 - UG-119
- coordinate systems for
 - UG-59,
 - UG-116
- current device parameters
 - R2-537,
 - PG-398
- cursor
 - R1-143, UG-90
- device driver
 - changing
 - UG-45
 - controlling
 - R1-249
 - size of display
 - R2-539
 - supported by PV-WAVE
 - UG-A-2
- establishing a 3D coordinate system
 - UG-121
- exposing and hiding windows
 - R2-363
- extracting image profiles
 - R1-583
- keywords
 - R2-497
- line thickness
 - R2-522, R2-549
- multiple graphs on a page
 - R2-545
- output device
 - configuring with DEVICE
 - UG-A-4
 - selecting with SET_PLOT
 - UG-A-3
- overlying an image on a contour
 - R1-402, UG-100
- polygon
 - filling
 - R1-542, UG-74
 - shading
 - R1-564
- positioning in window
 - R2-547
- procedures for plotting data
 - UG-55
- reading
 - cursor position
 - R1-143
 - values from cursor
 - R2-15
- shaded surfaces
 - R2-83, R2-91,
 - UG-131
- specifying range of data to plot
 - R2-556
- speeding up plotting
 - R2-547
- subtitle
 - R2-549
- symbols
 - R2-547
- text
 - See annotation
- thickness of lines
 - R2-522
- 3D
 - UG-115
- three-dimensional scaling
 - R2-57
- tick marks
 - See tick
- title
 - R2-550
- transformation matrices
 - R2-185,
 - UG-116
- window display
 - UG-A-6
- graphics window (drawing area)
 - PG-439
- gray
 - levels
 - dithering
 - UG-148
 - transformations
 - UG-152
 - scales
 - PG-189, PG-192
- greater than
 - See GT operator
- greater than or equal
 - See GE operator
- GRID function
 - R1-362
- GRID_2D function
 - R1-364, UG-191
- GRID_3D function
 - R1-367, UG-191
- GRID_4D function
 - R1-369, UG-191
- GRID_SPHERE function
 - R1-373,
 - UG-192
- gridding
 - 2D
 - R1-319, R1-364
 - 3D
 - R1-322, R1-367
 - 4D
 - R1-325, R1-369
 - definition of
 - UG-191
 - spheres
 - R1-373
 - summary of routines
 - R1-22
 - table of
 - UG-184
 - with dense data points
 - R1-319,
 - R1-322, R1-325
 - with sparse data points
 - R1-364,
 - R1-367, R1-369
- grids
 - making plots with
 - R1-362, UG-81
 - plotting keyword for
 - UG-78
- Group By clause
 - UG-273–UG-275
- group repeat specifications
 - PG-A-6–
 - PG-A-7
- groups of statements
 - PG-60

GT operator
description of PG-49
operator precedence PG-33
GUI
methods of creating PG-408
selecting look-and-feel PG-411,
UG-51

H

HAK procedure R1-377
handler, event R2-367, PG-487
HANNING function R1-378
hardcopy devices
See output devices
hardware fonts
See fonts
hardware pixels UG-A-96
hardware polygon fill UG-A-18
help
See also information
documentation of user-written rou-
tines R1-264
getting R1-410
HELP procedure R1-410
Hershey fonts R2-507, UG-291
Hewlett-Packard
Graphics Language plotters
UG-A-13
ink jet printers UG-A-23
laser jet printers UG-A-23
Printer Control Language printers
See PCL output
hexadecimal characters, for representing
non-printable characters PG-23
hexadecimal value, specifies color
UG-A-91
hide a widget PG-469
hiding windows R2-363
hierarchy of operators PG-32
high pass filters R1-250, UG-160,
UG-164
HILBERT function R1-381
HIST function, example of PG-66
HIST_EQUAL function R1-383, UG-158

HIST_EQUAL_CT procedure R1-386,
UG-156
histogram
calculating density function R1-386,
UG-329
equalization R1-383, UG-155
HISTOGRAM function R1-388,
UG-155
mode UG-70
of volumetric surface data R2-312
HLS
color model UG-308, UG-309
compared to HSV UG-308
procedure R1-396, UG-329
!Holiday_List system variable R1-437
holidays, removing from date/time vari-
ables with DT_COMPRESS
R1-272
homogeneous coordinate systems
UG-115
HPGL output UG-A-13—UG-A-19
HSV
color model UG-308, UG-309
compared to HLS UG-308
procedure R1-398, UG-329
HSV_TO_RGB procedure R1-400
hyperbolic
cosine R1-136
sine R2-112
tangent R2-191

I

icons
on menu PG-431
on tool box PG-433
turning windows into R2-363
IF statement PG-70—PG-72
avoiding PG-276, PG-277
definition PG-9
image processing
See also images
calculating histograms R1-386
convolution R1-125
creating digital filters R1-250

edge enhancement R2-45, R2-122
 Fast Fourier Transform R1-328
 Hanning filter R1-378
 histogram equalization R1-383
 magnifying images R1-125, R2-21
 minimizing images R2-21
 morphological dilation R1-254
 morphological erosion R1-300
 polynomial warping R1-526,
 R1-574
 Roberts edge enhancement R2-45
 rotating images R2-48, R2-51
 selecting a region of interest
 R1-232
 smoothing R1-454, R2-119
 Sobel edge enhancement R2-122
 write mask UG-328
 IMAGE_CONT procedure R1-402
 images
 See also color, image processing
 24-bit, reading R1-195
 8-bit, reading R1-192
 animating R2-277, R2-315
 annotation of R2-490
 as pixels PG-192
 combining
 CONTOUR and SURFACE pro-
 cedures UG-126
 with surface and contour plots
 R2-104
 with three-dimensional graph-
 ics UG-129—UG-130
 contrast
 control UG-331
 enhancement UG-330
 convolution of R1-125, UG-159
 data
 how stored PG-189
 interleaving PG-190, PG-193
 output PG-191
 pixels PG-193
 reading block of from tape
 PG-231
 reading with associated variable
 method PG-213
 definition of UG-135
 dilation operator R1-254
 direction of display (!Order) R2-542
 display
 order UG-138
 routines UG-4, UG-136
 TVRD function R2-223
 without scaling intensity
 R2-212
 dithering UG-148
 8-bit format PG-189
 erosion operator R1-300
 expanding R1-115, R2-24, R2-36
 extracting plane UG-A-95
 extracting profiles from R1-583
 Fast Fourier Transform R1-328
 filtering UG-162
 frequency domain techniques
 UG-162
 geometric transformations UG-167
 interleaving R1-197, R1-201,
 R1-222
 interpolation of UG-170
 magnifying R2-24, R2-36, R2-54,
 R2-225, R2-493, UG-140
 modifying intensities of UG-152
 morphologic dilation of R1-254
 morphologic erosion of R1-300
 optimizing transfer of UG-A-82
 orientation of UG-138
 overlying with contour plots
 R1-402, UG-100
 placing the cursor in UG-143
 polynomial warping of R1-521,
 UG-168
 position of on screen UG-138
 PostScript display of UG-A-39
 profiles R1-583
 reading
 from display device UG-142
 pixel values from R1-550
 reformatting of R1-574
 resizing of R2-225
 Roberts edge enhancement R2-45
 rotating R2-54, UG-168
 routines used to display UG-135
 scaling to bytes UG-154

- sharpening UG-160
- shrinking R1-115
 - or expanding R2-21, UG-140
- size of display UG-139
- smoothing R2-119
- Sobel edge enhancement of R2-122
- spatial warping R1-574
- special effects R1-543, R1-561, UG-328, UG-A-78
- subscripts of pixels inside polygon region R1-550
- transformation matrices UG-116
- transposing of R2-204
- true-color UG-146–UG-147
- 24-bit format PG-189
- value of individual pixels R2-327
- warping UG-168
- with contour plots UG-100
- zooming of R2-493
- IMAGINARY function R1-405
 - example PG-21
- IMG_TRUE8 procedure R1-406
- In operator UG-280
- include files for widgets PG-495
- including program files UG-33
- inclusive OR operator PG-50
- increment parameter
 - error if equal to zero PG-67
 - in FOR statement PG-67
- index number, color index UG-306, UG-322
- indexed files, VMS PG-228
- INDGEN function
 - description R1-409
 - using to make color tables UG-313
- indices of colors in color table R2-294
- indirect compilation UG-21
- Infinity PG-263
- INFO procedure
 - description of R1-410, PG-395, UG-16
 - getting information about files PG-220
 - viewing table structure UG-268
- information, obtaining with INFO procedure R1-410
- informational procedures, list of PG-257
- initializing WAVE Widgets PG-413
- input/output R1-479
 - advantages and disadvantages of each type PG-152
- ASCII PG-151
- ASSOC with unformatted data PG-212
- binary data PG-151, PG-188
- C binary data PG-194
- C-generated XDR data PG-207
- date/time data PG-162, PG-170
- DC (Data Connection) functions PG-140, PG-153
- efficiency of associated variables PG-212
- error handling R1-479, PG-140, PG-259
- fixed formats PG-152, PG-165
- flushing file units PG-218
- format reversion PG-167
- free format R1-580, R2-17, PG-159, PG-164
- from
 - keyboard R1-358, PG-222
 - word-processing application PG-175
- image data PG-189
- into
 - complex variables PG-160
 - structures PG-161
- overview of PG-12
- portable binary PG-206
- procedures
 - for ASCII data PG-155
 - for binary data PG-154
- READF procedure PG-175
- reading records with multiple array elements PG-180–PG-182, PG-185
- rules for
 - binary transfer of string variables PG-197
 - fixed format PG-156

- formatted data PG-159, PG-165
- record-oriented binary files PG-149
- selecting type of PG-138
- strings with structures PG-117
- structures PG-116
- types of PG-151
- unformatted R2-17
- using UNIX pipes PG-310
- VMS binary files PG-149
- waiting for input R2-61
- when to open PG-140
- XDR files PG-205
- INTARR function R1-413
- integer
 - bit shifting R1-418
 - constants
 - examples PG-19
 - range of PG-19
 - syntax of PG-18
 - conversions, errors in PG-264
 - data
 - shown in figure PG-146
 - type PG-3, PG-17
 - writing with format reversion PG-A-5—PG-A-6
 - format defaults PG-A-16
 - output, for format codes PG-A-17
 - syntax of constants PG-18
 - type array, creating R1-413
 - type, converting to R1-338
- Integral of Gaussian R1-355
- intensities, modifying images UG-152
- intensity, in 24-bit color UG-A-91
- interapplication communication
 - bidirectional and unidirectional PG-307
 - calling PV-WAVE from a C program PG-337
 - methods of PG-305
 - using SPAWN PG-309
 - with a child process PG-309
 - with RPCs (remote procedure calls) PG-359
- interleaving
 - description of PG-190
 - image data PG-190, PG-193
 - in 24-bit images R1-197, R1-201, R1-222
 - pixels PG-192
- INTERPOL function R1-414
- interpolated shading UG-131
- interpolation
 - bilinear R1-57
 - cubic spline R2-132
 - linear, between colortable values R2-294
- interprocess communication
 - See interapplication communication
- invariant expressions, removing from loops PG-281
- INVERT function R1-416
- inverting pixels UG-A-95
- irregular data, gridding R1-362
- ISHFT function R1-418
- iso-surfaces
 - example of UG-212, UG-214, UG-216
 - viewing interactively R2-307

J

- JOURNAL procedure R1-419
- !Journal system variable R2-542
- journaling R1-419
 - description of UG-36
 - examples UG-38
 - in relation to PRINTF UG-37
 - obtaining unit number of output R2-542, PG-28
- JUL_TO_DT function R1-421
 - description of UG-233
 - example of UG-234, UG-253
- Julian day
 - converting to a date/time variable R1-421
 - description of UG-225
 - in !DT_Base R2-60
 - using with tables UG-283

K

- keyboard
 - accelerators UG-52
 - defining keys R1-226, UG-52
 - getting input from R1-358, PG-222
 - interrupt UG-15, UG-29
 - key definitions PG-400
 - line editing, enabling R2-541, PG-28
 - using for command recall UG-32
- KEYWORD_SET function R1-422, PG-238, PG-239, PG-269
- keywords
 - checking for presence of PG-271
 - description of UG-23
 - examples PG-74
 - with functions PG-238
 - parameters
 - abbreviating PG-74
 - advantages of PG-75
 - and functions PG-68
 - checking for presence of R1-422, PG-239, PG-269
 - definition of PG-74, PG-235
 - graphics and plotting routines R2-497
 - passing of PG-234, PG-235
 - using the Keyword construct PG-74, PG-235
 - relationship to system variables UG-24, UG-57
- Korn shell PG-299

L

- labels, destinations of GOTO statements PG-52
- Lambertian
 - ambient component UG-199
 - diffuse component UG-198
 - transmission component UG-199
- landscape orientation UG-A-34

- LaTeX documents
 - inserting plots UG-A-40
 - using PostScript with UG-A-40, UG-A-43
- layout (WAVE Widgets)
 - arranging a PG-422
 - example PG-417, PG-420—PG-421
 - form PG-422
 - row/column PG-420
- LE operator
 - description of PG-49
 - operator precedence PG-33
- least square
 - curve fitting R1-552, R1-574, UG-72
 - non-linear curve fitting R1-147
 - problems, solving R2-179
- Lee filter algorithm R1-424
- LEEFILT function R1-424
- LEGEND procedure R1-426
- legend, adding to a plot R1-426
- less than
 - See LT operator
- less than or equal
 - See LE operator
- libraries
 - creating and revising for VMS PG-253
 - PV-WAVE Users' PG-255
 - searching (VMS) PG-252
 - Standard PG-255
- light source
 - lighting model, for RENDER function UG-197
 - modifying R2-70
 - shading R1-564, R2-70, UG-131
 - setting parameters for R2-70
- LINDGEN function R1-427
- line
 - color of R2-504
 - connecting symbols with UG-72
 - drawing R1-510, UG-121
 - fitting, example using POLY_FIT UG-72
 - linestyle index R2-545

- style of R2-502, R2-545
- thickness of R2-522, R2-549
- linear
 - algebra
 - eigenvalues and eigenvectors R2-201
 - LU decomposition R1-445, R1-447
 - reducing matrices R2-207
 - rules PG-46
 - solving matrices R2-207, R2-208
 - axes, specifying R2-560
 - distortion UG-170
 - equations
 - solution vector R1-468
 - solving R2-177
 - least squares problems, solving R2-179
 - regression, fit to data R2-26
- linking
 - applications to PV-WAVE PG-346
 - C code to PV-WAVE R1-429
 - C program under
 - UNIX PG-347
 - VMS PG-349
 - client with PV-WAVE PG-361
 - FORTRAN applications under
 - UNIX PG-348
 - LINKNLOAD function
 - accessing external functions R1-433, PG-323, PG-327
 - calling external programs PG-319
 - compiling under UNIX PG-322, PG-326, PG-328
 - description R1-429, PG-319
 - example R1-431, PG-321, PG-324
 - operating systems supported under PG-320
- list, scrolling
 - See scrolling list
- LJ-250 output UG-A-20—UG-A-23
- LN03 procedure R1-434
- LOAD_HOLIDAYS procedure R1-437, UG-240
- LOAD_WEEKENDS procedure R1-439, UG-241, UG-242
- LOADCT procedure R1-435, UG-145, UG-311, UG-329
 - description UG-136
- local variables, definition of PG-237
- log scaling, plotting R1-497
- logarithm
 - base 10 R1-40
 - natural R1-38
- logarithmic
 - axes R1-497
 - specifying R2-560
 - plotting R1-497
 - scaling R1-497
 - keywords UG-84
 - PLOT_IO procedure UG-85
- logical names
 - defining R2-65
 - VMS, DCL R2-209
 - VMS, deleting R1-240
- logical operators
 - evaluating PG-42
 - representation of PG-42
 - used with arrays PG-42
- logical records PG-147
- logical unit number
 - allocating R1-359
 - See LUNs
- LONARR function R1-440
- LONG function R1-442
- longword
 - data type PG-17
 - integer arrays, creating R1-427, R1-440
 - integer, converting to R1-442
- look-and-feel, of GUI PG-411, UG-51
- lookup table, color UG-308, UG-A-79
 - See also color tables
- loop, event
 - See event loop
- low pass filters UG-159, UG-164
 - creating R1-250

- LT operator
 - description of PG-49
 - operator precedence PG-33
- LUBKSB procedure R1-445
- LUDCMP procedure R1-447
- LUNs
 - closing PG-140
 - deallocating R1-345, PG-141
 - description PG-138
 - for
 - opening files PG-139
 - standard I/O PG-140
 - getting information using FSTAT PG-220
 - of current output file R2-539
 - of journal output R2-542, PG-28
 - operating system dependencies PG-142
 - range of PG-141
 - general use PG-143
 - reserved numbers PG-140, PG-141
 - standard
 - error output PG-142
 - input PG-141
 - output PG-141
 - used by
 - FREE_LUN PG-143
 - GET_LUN PG-143
 - with
 - UNIX PG-142
 - VMS PG-142

M

- Macintosh computers
 - See PICT output
- magnetic tape
 - accessing under VMS PG-229
 - mounting a tape drive (VMS) PG-230
 - reading from R2-193, PG-231
 - REWIND procedure PG-229
 - SKIPF procedure PG-229

- skipping
 - backward on a tape PG-231
 - forward on the tape (VMS) PG-230
- TAPRD procedure PG-229
- TAPWRT procedure PG-229
- WEOF procedure PG-229
 - writing to R2-194, PG-229
- magnifying images R2-21, R2-36, R2-54, R2-493, UG-140
- main programs
 - definition of UG-24
 - difference from command files UG-25
 - executing UG-24
- main PV-WAVE directory R2-540, PG-27
- main window
 - See shell window
- MAKE_ARRAY function R1-449
- makefile, using PG-346
- management, of widgets PG-484
- mapping, of widget PG-484
- margin, around plot R2-555
- marker symbols
 - displaying in a volume R2-263
 - for data points UG-72
 - user-defined UG-73
- masking
 - arrays PG-49
 - unsharp UG-161
- math errors
 - See also error handling
 - accumulated R1-90
 - accumulated math error status PG-263
 - CHECK_MATH function PG-265
 - detection of PG-263
 - hardware-dependent PG-268
 - procedures for controlling PG-258
- math messages, issuing R1-460
- math traps, enabling/disabling PG-266
- mathematical function
 - abbreviated list of UG-9
 - absolute value R1-30
 - arc-cosine R1-31

arcsine R1-41
 arctangent R1-45
 area of polygon R1-522
 base 10 logarithm R1-40
 Bessel I function R1-51
 Bessel J function R1-53
 Bessel Y function R1-55
 bilinear interpolation R1-57
 bit shifting R1-418
 checking for finite values R1-336
 complex conjugate R1-118
 convolution R1-125
 correlation coefficient R1-133
 cosine R1-135
 cross product R1-142
 cubic splines, interpolation R2-132
 derivative R1-246
 determinant of matrix R1-248
 error function R1-304
 Fast Fourier Transform R1-328
 GAMMA function R1-352
 Gaussian integral R1-355
 Hilbert transform R1-381
 hyperbolic
 cosine R1-136
 sine R2-112
 tangent R2-191
 imaginary numbers R1-405
 improve solution vector R1-468
 LU decomposition R1-445, R1-447
 matrix
 reduction R2-207
 solutions R2-208
 maximum value R1-452
 mean R2-136
 minimum value R1-462
 natural exponent R1-317
 natural logarithm R1-38
 polynomial functions R1-521
 polynomial roots R2-495
 random numbers R2-13
 sine R2-109
 singular value decomposition
 R2-179
 solving R2-177
 square root R2-134
 standard deviation R2-136
 tangent R2-190
 mathematical morphology UG-173
 matrices
 reading and printing PG-93
 reading interactively R2-40
 transformation, See transformation
 matrices
 using subscripts with PG-82
 matrix
 expressions PG-99
 multiplication PG-94
 print to specified file unit R1-517
 print to standard output stream
 R1-515
 reading from a file R2-43, PG-96
 subarrays PG-98
 matrix inversion
 REVERSE function R2-36
 SVD procedure R2-179
 matrix multiplication operator (#)
 examples PG-46
 operator precedence PG-32
 MAX function R1-452
 !Max_Levels system variable R2-551
 maximum operator (>)
 examples PG-46
 operator precedence PG-32
 mean of an array R2-136
 mean smoothing UG-159
 median
 filter R1-454
 value of array R1-454
 MEDIAN function R1-454, UG-159
 median smoothing UG-159
 memory
 allocation PG-244
 deleting
 compiled functions from
 R1-239
 compiled procedures from
 R1-241
 structure definitions R1-242
 getting information about R1-410

- order and arrays PG-281, PG-282
- virtual
 - See virtual memory
- menus
 - bar PG-426
 - callback PG-428
 - creating R2-221, R2-359
 - creating and handling PG-425
 - defining text PG-429
 - example PG-431
 - option PG-428
 - popup PG-427
 - separators PG-430
 - using unnamed structure to define text PG-429
- MESH function
 - description of R1-457
 - example of UG-204, UG-206
- mesh surfaces, drawing R2-168, UG-112
- message
 - See also error handling
 - error, setting level to report in "DC" functions R1-159
 - for incomplete "DC" function R1-157
 - issuing error R1-460
 - popup message PG-449
 - suppressing compiler messages PG-29
- MESSAGE procedure R1-460, PG-251, PG-261–PG-262
- Microsoft Word, inserting PV-WAVE plots into UG-A-45–UG-A-47
- MIN function R1-462
- minimizing images UG-140
- minimum operator (<)
 - examples PG-46
 - operator precedence PG-32
- minimum value of array R1-462
- mixed data types PG-35, PG-42, PG-281
- MOD operator
 - description of PG-49
 - operator precedence PG-32
- MODIFYCT procedure R1-464, UG-330
- modulo operator PG-49
- monochrome
 - devices UG-148
 - displays UG-A-94
 - dithering UG-148
- MONTH_NAME function R1-465, UG-260
- morphologic
 - dilation operator R1-254
 - erosion operator R1-300
- morphology, mathematical UG-173
- Motif GUI PG-479
- mouse
 - accesses text editing functions R2-351
 - positions slice through data R2-332
 - rotates color table R2-286
 - use to rotate and flip surface R2-344
- !Mouse system variable R2-542
- MOVIE procedure R1-466
- MPROVE procedure R1-468
- !Msg_Prefix system variable R2-542
- multidimensional arrays, using subscripts PG-82
- multiple
 - array elements, reading records with PG-181–PG-182, PG-185
 - plots UG-83
 - statements, separating with ampersand UG-35
- multiplication operator (*) PG-32

N

- N_ELEMENTS function R1-469, PG-270
- N_PARAMS function R1-471, PG-269
- N_TAGS function R1-472, PG-119
- NaN, not a number PG-263
- natural
 - exponential function R1-317
 - logarithm R1-38

- NE operator
 - description of PG-50
 - operator precedence PG-33
- nearest neighbor method UG-170
- nested procedures, getting information on PG-403
- nonblocking window
 - dialog box PG-453
 - popup message PG-449
- non-printable characters
 - examples PG-23
 - specifying PG-22
- normal coordinate systems UG-60, UG-61
- normalized coordinates R2-512
- not equal
 - See NE operator
- NOT operator
 - description of PG-50
 - example PG-43
 - operator precedence PG-32
- number of elements in expression PG-270
- numeric
 - constants PG-4
 - errors
 - machine specifics PG-268
 - trapping PG-263
 - operators, table of PG-7

O

- object modules, callwave, compiling PG-313
- octal characters, for representing non-printable characters PG-23
- offset parameter PG-216, PG-217
- ON_ERROR procedure
 - description R1-474, PG-258
 - examples PG-250
 - table of values PG-259
- ON_IOERROR procedure
 - description R1-479, PG-258
 - uses of PG-259
- OPEN procedures PG-139
- opening files
 - DC (Data Connection) functions PG-140
 - LUNs PG-138
 - OPEN procedures R1-480, PG-139
 - when to open PG-140
 - XDR PG-205
 - OPENR procedure R1-480
 - OPENU procedure R1-480
 - OPENW procedure R1-480
 - OpenWindows GUI PG-479
 - operands, checking validity of PG-264
 - operating system
 - accessing using SPAWN PG-297
 - communicating with from PV-WAVE PG-291
 - UNIX, calling from PV-WAVE R1-77
 - operator precedence
 - examples PG-33
 - table of PG-32
 - operators PG-47
 - addition (+) PG-45, PG-123
 - AND PG-48
 - array concatenation, table of PG-8
 - assignment (=) PG-44
 - Boolean, table of PG-7
 - division PG-45
 - EQ PG-48
 - evaluating PG-42
 - exponentiation (^) PG-46
 - GE PG-49
 - general description PG-31
 - grouping PG-44
 - GT PG-49
 - hierarchy of PG-32
 - infix notation PG-2
 - LE PG-49
 - LT PG-49
 - matrix multiplication (#) PG-46
 - maximum (>) PG-46
 - minimum (<) PG-46
 - MOD PG-49
 - multiplication PG-45
 - NE PG-50
 - NOT PG-50
 - numeric, table of PG-7

- OR PG-50
- overview of PG-6
- parentheses () PG-44
- precedence PG-32
- relational, table of PG-8
- string, table of PG-8
- subtraction (-) PG-45
- table of
 - Boolean PG-43
 - relational PG-41
- using with arrays PG-42
- XOR PG-50
- OPLOT procedure R1-488, UG-61, UG-66, UG-68, UG-252
- OPLOTERR procedure R1-491
- option menu PG-428
- OR operator
 - description of PG-50
 - operator precedence PG-33
 - truth table PG-43
- Order By clause UG-276
- !Order system variable R2-542
- ordered dithering UG-148
- output
 - 24-bit image data R1-221
 - 8-bit image data R1-219
 - See also device drivers
 - ASCII free format data R1-203, R1-212
 - binary string variables PG-197
 - capturing with SPAWN procedure PG-301
 - clipping graphics R2-503
 - color PostScript
 - See color PostScript devices
 - controlling graphics device R1-249
 - file, sending to plotter or printer UG-A-5
 - flushing buffers R1-344
 - formats, for writing data PG-164
 - generated by SPAWN command PG-301
 - hardcopy
 - See device drivers
 - image data PG-191
 - procedures, table of PG-13
 - selecting device R2-66
 - steps to produce UG-A-3
 - TIFF image data R1-221, R1-223
 - to a file, basic steps PG-12
 - using UNIX pipes PG-310
 - writing unformatted data R2-361
- overflow
 - checking for in integer conversion PG-264
 - in type conversions PG-36
 - with integer arithmetic PG-19
- overlying
 - image and contour plots R1-402
 - images and graphics UG-100
 - using devices with
 - fixed pixels UG-103–UG-104
 - scalable pixels UG-102
- overplotting data R1-488, UG-56, UG-66

P

- !P system variable, fields of R2-543–R2-550
- padding
 - bytes PG-117
 - volumes R2-266
- page faults PG-283
- page file quota PG-287
- palette
 - color PG-192
 - of color cells R2-293
- PALETTE procedure R1-494, UG-332
- parameters
 - See also keywords
 - actual PG-234
 - changing for graphics device R2-537
 - checking for PG-269
 - copying PG-236
 - correspondence between formal and actual PG-235–PG-236
 - determining number of PG-269
 - for procedures and functions PG-74
 - formal PG-235

keyword PG-74
 number of non-keyword in routines
 R1-471
 number required in routines PG-236
 passing PG-75, PG-246
 by reference PG-246
 by value PG-246
 mechanism PG-111, PG-234,
 PG-246
 positional, definition of PG-235
 parent widget, manages child PG-484
 parentheses operator() PG-44
 !Path system variable R2-550
 patterns
 See polygon fill, pattern
 PCL output UG-A-23–UG-A-27
 percent sign (%), meaning of PG-A-6,
 PG-A-22
 period symbol UG-35
 Pgfquo quota PG-287
 physical memory, need for PG-283
 physical record, definition of PG-147
 !Pi system variable R2-552
 PICT output UG-A-27–UG-A-30
 pipes, UNIX, description of PG-310
 pixel interleaving PG-190, PG-192
 pixels
 copying UG-A-72, UG-A-85
 data PG-192
 easy way to ascertain exact value
 R2-327
 hardware UG-A-96
 inverting region of UG-A-95
 number per centimeter on graphics
 device R2-539
 palette color PG-192
 reading back UG-142
 scalable UG-139
 values, reading from an image
 R2-15
 pixmaps
 animating R2-278, R2-316
 creating with WINDOW procedure
 UG-A-86
 examples UG-A-86
 wider than the physical width of the
 screen UG-A-87
 with X Windows UG-A-8, UG-A-85
 plot
 area, defining R2-67, R2-79
 axis format R2-526
 combining images and contours
 UG-100
 data window UG-85
 inhibiting clipping of R2-546
 line styles R2-502
 margin for annotation R2-525,
 R2-530, R2-534
 multiple plots on a page R2-545
 position of in window R2-547
 positioning in display window R2-79
 region UG-85
 specifying
 margin around R2-515,
 R2-555
 range of data to plot R2-526,
 R2-530, R2-534, R2-556
 symbol index R2-516, R2-547
 PLOT procedure
 2D graphs R1-497
 basic description of UG-61
 date/time examples UG-244,
 UG-246, UG-249
 for tables UG-286–UG-287
 PLOT_FIELD procedure R1-507
 PLOT_IO procedure UG-84–UG-85
 2D graphs R1-497
 PLOT_OI procedure UG-61
 2D graphs R1-497
 PLOT_OO procedure
 2D graphs R1-497
 example of UG-82
 PLOTERR procedure R1-505
 PLOTS procedure R1-510
 plotters, HPGL UG-A-13
 plotting
 See also annotation, axes, color,
 ticks, image processing
 !P system variable R2-543, PG-27
 2D array as 3D plot R2-197
 3D data UG-93

- 3D orientation of text R2-521
- 3D to 2D transformations R2-549
- adding
 - axes UG-87
- bar charts UG-76
- changing coordinate systems
 - UG-59
- clipping
 - output R2-503
 - window R2-543
- combination plots R2-104
- connecting symbols with lines
 - R2-548
- contour plots R1-119, R2-506
- controlling
 - placement of cursor R2-217
- converting from 3D to 2D coordinates
 - UG-119
- coordinate systems UG-59,
 - UG-115
- creating menus R2-221
- date/time data UG-243—UG-253
 - examples UG-243—UG-253
 - in tables UG-267
- defining a plotting symbol R2-235,
 - UG-72
- display of images R2-212, R2-225
- enhanced contour plots UG-99
- error bars R1-305
- exchange of axes UG-126
- exposing and hiding windows
 - R2-363
- filling contour plots UG-109
- flushing output buffer R1-294
- font selection R2-507
- ganging UG-83
- highlighting with POLYFILL UG-74
- histogram UG-70
- input from the cursor UG-90
- irregular polygon fill R1-542
- keywords R2-497
- line thickness R2-522
- linear log scaling R1-497
- location of plot in window R2-548,
 - UG-85
- logarithmic
 - axes R2-533
 - scaling UG-84
- main title R2-523
- marker symbols UG-72
- multiple UG-83
- number of points to sum R2-547
- overlying contour plots and images
 - UG-100
- overplotting R1-488, UG-56,
 - UG-66
- polar
 - coordinates R2-515
 - plots UG-89
- polygon filling R1-542, UG-74
- polygons R1-560
- position of graphics cursor R1-143,
 - UG-90
- region filling UG-74
- routines, summary of UG-56
- scaling XY UG-63—UG-64
- smoothing contour plots UG-108
- speeding up R2-547
- surface shading parameters
 - UG-132
- surfaces UG-112
- symbols R2-547, UG-72
- system variables UG-55
- tables UG-286—UG-287
 - with a date/time axis UG-284
- thickness of lines R2-549
- three-dimensional graphics UG-115
- transformation matrices UG-116
- user-defined symbols UG-73
- vector fields from 3D arrays R2-240
- window display UG-A-6
- windows
 - creating R2-356
 - deleting R2-275
- POINT_LUN procedure
 - description R1-519, PG-219
 - example PG-197
 - used to access VMS files PG-225

- pointer
 - function
 - example PG-488
 - WtPointer R2-390
 - into files R1-519
- polar
 - coordinates R2-515, UG-89
 - plots R2-515, UG-89
- POLY function R1-521
- POLY_2D function R1-526, UG-103, UG-168, UG-172
- POLY_AREA function R1-522
- POLY_C_CONV function R1-524, R1-535, UG-192
- POLY_COUNT function R1-537, UG-192
- POLY_DEV function R1-538, UG-194
- POLY_FIT function R1-552, UG-72
- POLY_MERGE procedure R1-557, UG-192
- POLY_NORM function R1-559, UG-194
- POLY_PLOT procedure R1-560, UG-195
- POLY_SPHERE procedure R1-568, UG-190
- POLY_SURF procedure R1-571, UG-189
- POLY_TRANS function R1-573, UG-192, UG-194
- POLYCONTOUR procedure
 - description R1-529
 - example of UG-110
 - filling closed contours R2-514, UG-109
 - syntax UG-109
- POLYFILL procedure
 - description R1-542
 - example UG-74, UG-76
 - fill pattern R2-506, R2-514
- POLYFILLV function R1-550
- POLYFITW function R1-555
- polygon fill
 - 2D or 3D polygon R1-542
 - color R2-504
 - example UG-74
 - hardware UG-A-18
 - pattern R2-506, R2-514
- polygon lists
 - description of UG-187
 - merging R1-557
- polygonal meshes
 - defining with MESH function R1-457
 - example of UG-204–UG-205
- polygons
 - calculating area of R1-522
 - filling
 - See polygon fill
 - generating R1-22, R1-571, UG-187
 - manipulating R1-23, UG-192
 - merging lists for rendering R1-557
 - plotting R1-560
 - querying subscripts of pixels inside R1-550
 - rendering R1-23, R1-560, R2-28, UG-176, UG-195
 - table of UG-183
 - returning
 - list of colors for R1-524, R1-535
 - number contained in list R1-537
 - shading R1-564
- polylines UG-121
- polynomial
 - curve fitting R1-552, R1-555
 - functions, evaluating R1-521
 - warping of images R1-526, R1-574
- POLYSHADE function R1-564, UG-195
- POLYWARP procedure R1-574, UG-173
- POPD procedure R1-578, PG-303
- popup menu
 - avoid using with text widget R2-462
 - creating PG-427
 - defining menu items PG-429
 - description R2-461
 - example PG-427
- popup message PG-449, PG-451
- popup widget PG-419, PG-449, PG-453, PG-457, PG-459

- portable data, XDR PG-205–PG-206
- portrait orientation UG-A-36
- position, of plot R2-515, UG-85
- positional parameters
 - checking for PG-269
 - examples PG-235
 - with functions PG-238
- positioning
 - file pointer PG-219
 - images on the display UG-138
 - text in PostScript UG-A-39
- PostScript output UG-A-31–UG-A-48
- PRINT procedure
 - description R1-580, PG-156
 - for tables with columns UG-285
 - for writing structures PG-116
- PRINTD procedure R1-582, PG-303
- Printer Control Language output
 - See PCL output
- printers supported by PV-WAVE UG-A-2
- PRINTF procedure
 - description R1-580, PG-156
 - for writing structures PG-116
 - in relation to journaling UG-37
- printing
 - files R1-580
 - graphics output UG-A-1
 - LN03 printer R1-434
 - tables with column titles UG-285
 - to LJ-250 printer UG-A-20
 - to PostScript printer UG-A-32
 - values of date/time variables R1-282
 - variables R1-580
- private colormaps UG-A-81
- PRO statement
 - format of PG-76
 - syntax PG-235
- Procedure Call statement PG-73
- procedures
 - action to take on error R1-474
 - actual parameters PG-234
 - automatic compiling, conditions for PG-69
 - call statement PG-76
 - calling PG-11
 - call stack PG-262
 - mechanism PG-248
- checking for
 - number of elements R1-469, PG-270
 - parameters R1-422, R1-472, PG-269
- compiling
 - automatically PG-69, PG-234, PG-239
 - three methods of PG-239
 - with .RUN and filename PG-239
 - with interactive mode PG-240
- creating PG-15
 - with a text editor UG-22
- definition of PG-11, PG-233
- definition statement of PRO PG-76, PG-235
- deleting from memory R1-241
- determining number of parameters PG-269
- directory path for R2-550, PG-28
- documentation of user-written routines R1-264
- error handling PG-250, PG-257–PG-258
- formal parameters PG-235
- getting a list of those compiled PG-401
- help with R1-410
- I/O errors in R1-479
- information on nested PG-403
- keyword parameters PG-74
- libraries of VMS PG-251
- library UG-23
- maximum size of code PG-397
- number of
 - non-keyword parameters R1-471, PG-269
 - parameters required PG-236
- parameters PG-74, PG-234
 - passing mechanism PG-75, PG-246
 - positional PG-235

- program control PG-257, PG-273
- recursion PG-248
- required components of PG-237
- search path UG-20, UG-46
- stopping execution R2-138
- syntax for
 - calling PG-15
 - creating PG-15
 - user-written PG-69
 - various sources of PG-73
- process, spawning R2-126, PG-297, PG-298
- processes, concurrent R1-309
- processing images
 - See image processing
- product-moment correlation coefficient R1-133
- PROFILE function R1-583
- PROFILES procedure R1-585
- profiles, extracting from images R1-583
- program
 - calling from within PV-WAVE PG-310
 - code area full PG-242
 - control routines PG-257, PG-273
 - creating and running UG-22
 - data area full PG-243
 - file search method UG-48
 - files containing PV-WAVE procedures UG-28
 - format of UG-21, UG-24
 - help with nested PG-403
 - increasing speed of PG-276
 - linking to PV-WAVE PG-310
 - listings UG-28
 - main PV-WAVE UG-24
 - maximum size of code PG-397
 - pausing execution R2-274
 - preparing and running UG-12
 - running as batch UG-19
 - stopping R2-138
 - submitting to Standard Library PG-255

- programming
 - accessing large arrays PG-281
 - avoid IF statements PG-276
 - code size PG-397
 - commenting programs PG-52
 - delaying program execution R2-274
 - format strings PG-A-1—PG-A-2
 - menus, creating R2-221, R2-359
 - minimize virtual memory allocation PG-288
 - running out of virtual memory PG-284
 - tips PG-498
 - and cautions PG-472
 - for writing efficient code PG-275
 - on loops PG-281
 - use correct type of constants PG-280
 - use optimized system routines PG-280
 - use vector and array data operations PG-278
 - use virtual memory efficiently PG-285
 - using arrays efficiently PG-279
 - writing efficient programs PG-275
- prompt
 - changing R2-552, UG-52
 - PV-WAVE prompt string UG-12
 - PROMPT procedure R1-587
 - !Prompt system variable R2-552
 - PSEUDO procedure R1-588, UG-330
 - Pseudo_Color, 12-bit UG-A-94
 - pseudo-color PG-189
 - compared to true-color UG-146
 - images, PostScript UG-146
 - PUSHD procedure R1-590, PG-303
 - PV-WAVE session
 - exiting R1-316
 - getting information about PG-395
 - recording R1-419
 - restoring R2-33
 - saving R2-56

Q

QMS QUIC output UG-A-48–UG-A-51
quadric animation, example of UG-207
QUERY_TABLE function
 combining multiple clauses UG-280
 description R2-1, UG-7, UG-263
 Distinct qualifier UG-272
 examples PG-179, UG-265
 features UG-270
 Group By clause UG-273
 In operator UG-280
 passing variable parameters
 UG-279
 rearranging a table UG-271
 renaming columns UG-272
 sorting with Order By clause
 UG-276
 syntax UG-271
 Where clause UG-277
quick.mk makefile PG-346
!Quiet system variable R2-553
QUIT procedure R2-11, UG-14
quitting PV-WAVE R1-316, R2-11,
 UG-13
quotas
 Pgfquo PG-287
 Wsquo PG-287
quotation marks UG-35
quoted string format code PG-A-10,
 PG-A-19

R

!Radeg system variable R2-553
radians
 converting from degrees PG-27
 converting to degrees R2-553,
 PG-27
radio button box widget PG-436
random file access PG-225
random number, uniformly distributed
 R2-13
RANDOMN function R2-12
RANDOMU function R2-13

range
 setting default for axes R2-81
 subscript, for selecting a subarray
 PG-84
raster graphics, colors UG-A-92
raster images UG-135
rasterizer, Tektronix 4510 UG-A-61
ray tracing
 cone primitives R1-113
 cylinder primitives R1-150
 definition of UG-197
 description of UG-175, UG-196
 mesh primitives R1-457
 RENDER function R2-28
 sphere primitives R2-129
 summary of routines R1-23
 viewing demonstration programs
 UG-177
 volume data R2-272
RDPIX procedure R2-15
READ procedure R2-17, PG-156,
 PG-160–PG-161
READF procedure
 description PG-156
 example PG-176
 with STR_TO_DT function
 PG-170
 for fixed format PG-175
 for row-oriented FORTRAN write
 PG-183
 reading
 ASCII files R2-17
reading
 24-bit image file R1-195
 8-bit image data R1-192, PG-189
 ASCII files R1-161, R1-177, R2-17
 binary files between different sys-
 tems PG-205
 binary input R2-17
 byte data from an XDR file
 PG-206–PG-207
 C-generated XDR data PG-207–
 PG-208
 cursor position R1-143, UG-90

- data
 - date/time UG-228
 - files R2-17
 - from a file PG-12
 - from a word-processing application PG-175
 - from multiple array elements PG-180
- DC_READ routines PG-A-8
- files, using C programs PG-196
- fixed-formatted ASCII files R1-161
- freely-formatted ASCII files R1-177
- from magnetic tapes R2-193
 - TAPRD PG-229
- from the display device UG-142
- images from the display R2-223
- multiple array elements with FORTRAN format string PG-185
- READF PG-175—PG-176, PG-181, PG-183, PG-185
 - for arrays PG-181
- READU PG-195, PG-203
- records with multiple array elements PG-180, PG-182, PG-185
- TIFF files R1-198
- unformatted input R2-17
- XDR files with READU procedure PG-209

READU procedure

- binary files PG-195
- description PG-154
- reading binary input R2-17
- syntax PG-193
- with Segmented keyword PG-203
- XDR files PG-209

realize, widget hierarchy PG-484

REBIN function

- decrease sampling with UG-98
- description R2-21
- resampling original image with UG-103
- using to magnify or minify an image UG-141

recalc flag, in IDT structure UG-226

record attributes of VMS files PG-226

recording a PV-WAVE session R1-419

record-oriented

- binary files PG-149
- data, transferring in VMS files PG-150

records

- definition of PG-147
- extracting fields from PG-37
- fixed length format (VMS) PG-217, PG-226
- length of PG-147
- multiple array elements PG-180, PG-181, PG-182, PG-185

recovering from errors PG-258

- PV-WAVE session R2-33

rectangular surfaces UG-189

recursion PG-248

reference, parameter passing by PG-246

REFORM function R2-24

region of interest, selecting R1-232

Regis output UG-A-54—UG-A-56

register, event handler PG-487

REGRESS function R2-26

regression, fit to data R2-26

relational operators

- descriptions PG-48—PG-50
- evaluating with operands of mixed data types PG-42
- table of PG-8, PG-41
- using with arrays PG-42

Remote Procedure Call

See RPC

RENDER function

- color, defining for UG-198
- cone objects R1-113, UG-196
- cylinder objects R1-150, UG-197
- default view UG-202
- defining material properties of objects UG-200
- description R2-28, UG-196, UG-203
- example of UG-204—UG-218
- invoking UG-203
- lighting model UG-197
- mesh objects R1-457, UG-197
- sphere objects R2-129, UG-197

- syntax UG-203
 - volume objects R2-272, UG-197
- rendering
 - images, displaying UG-219
 - iso-surfaces UG-216
 - polygons R1-23, R1-560, UG-183, UG-195
 - process of UG-185
 - ray-traced objects R2-28, UG-196
 - shaded surfaces R1-564, R2-83, R2-91
 - volumes R1-24, R2-96, R2-267, UG-182–UG-183, UG-195
 - with standard techniques UG-195
- REPEAT statement PG-10, PG-77
- REPLICATE function
 - description R2-31
 - description of PG-114
 - examples PG-195
- reserved LUNs
 - description PG-140–PG-141
 - operating system dependencies PG-142
- reserved words PG-25
- reserving colors for other applications use UG-A-84
- resizing arrays or images R2-21
- resource
 - data type of value PG-483
 - definition of PG-483
 - deriving name of PG-483
 - file PG-415, PG-482
 - setting for Widget Toolbox PG-483
- resource file
 - how to use PG-464
- RESTORE procedure R2-33, UG-40–UG-41
- RECALL procedure R2-34, PG-251
- RETURN procedure R2-35, PG-68, PG-76, PG-236, PG-251
- reversal of rows and columns PG-47
- REVERSE function R2-36
- reversing
 - current color table R2-302, UG-320
 - direction of vector R2-36
- reversion, format PG-167, PG-A-5
- REWIND procedure R2-38, PG-229
- RGB
 - color system UG-306
 - triplets PG-193
- RGB_TO_HSV procedure R2-39
- right-handed coordinate system UG-116
- RMS files, reading images PG-217
- Roberts edge enhancement R2-45
- ROBERTS function R2-45, UG-160
- root window
 - relationship to visual class UG-A-90
 - See shell window
- ROT function R2-48
- ROT_INT function R2-54
- ROTATE function
 - description R2-51
 - examples UG-138
 - syntax UG-168
- rotating
 - arrays or images R2-51
 - corresponding to surface R2-168
 - current color table R2-302, UG-320
 - data UG-117
 - images R2-48, R2-51, R2-54
- row-oriented
 - ASCII data PG-146
 - data, shown in figure PG-146
 - FORTRAN write PG-183
- rows
 - in arrays PG-47
 - removing duplicate from a table UG-270
 - transposing with columns PG-47
- RPC
 - description of PG-359
 - example, CALL_UNIX PG-377–PG-387
 - synchronization of processes PG-360
- running
 - See executing
- run-time compilation of statements PG-273

S

- sampled images UG-135
- SAVE procedure R2-56, UG-40
- saving
 - a PV-WAVE session R1-419, R2-56
 - compiled procedures R1-104
 - TIFF data PG-191
- scalable pixels UG-102, UG-139
- scalars
 - combining with subscript array, ranges PG-91
 - converting to date/time variables R2-238
 - definition of PG-4, PG-24
 - in relation to arrays PG-290
 - subscripting PG-84
- scale unit cube into viewing area R2-57
- SCALE3D procedure R2-57, UG-123
- scaling
 - corresponding to surface R2-168
 - data UG-117
 - images R2-225
 - input images with BYTSCL function UG-154
 - logarithmic UG-84
 - plots UG-63–UG-64
 - three-dimensional R2-57
 - Y axis with YNozero UG-64
- screen pixels, assigning color UG-A-78
- scroll bars
 - on drawing area PG-439
 - ScrollBar callback parameters (Motif) PG-C-5
 - scrollbar callback parameters (OLIT) PG-C-8
 - scrolling list PG-446
 - text widget PG-443
 - used to view
 - image R2-326
 - text R2-351
- scrolling list
 - callback PG-447
 - creating PG-446
 - description R2-439
 - example PG-448
 - multiple selection mode PG-447
 - single selection mode PG-447
- searching VMS libraries PG-252
- SEC_TO_DT function R2-58, UG-233
- seconds, converting to date/time variables R2-58
- semicolon after @ symbol UG-20
- sensitivity, of widgets PG-470
- servers
 - C program as PG-380
 - closing connections UG-A-72
 - definition of PG-359
 - example program(test_server.c) PG-369
 - in X Window systems UG-A-69
 - linking with PV-WAVE PG-361
 - UNIX_REPLY function PG-375
 - using PV-WAVE as PG-371–PG-373, PG-392–PG-394
- session
 - See PV-WAVE session
- SET_PLOT procedure
 - description R2-66
- SET_SCREEN procedure R2-67
- SET_SHADING procedure R2-70, UG-132
- SET_SYMBOL procedure R2-74, PG-296
- SET_VIEW3D procedure R2-77, UG-194
- SET_VIEWPORT procedure R2-79
- SET_XY procedure R2-81
- SETBUF function
 - example PG-311
 - with child program PG-310
- SETDEMO procedure R2-62
- setenv command
 - for WAVE_PATH UG-47
 - for WAVE_STARTUP UG-49
 - with WAVE_DEVICE UG-45
- SETENV procedure R2-64, PG-293
- SETLOG procedure R2-65, PG-295
- SETUP_KEYS procedure R2-75

- SHADE_SURF procedure
 - description R2-83, UG-131
 - examples UG-133
- SHADE_SURF_IRR procedure R2-91
- SHADE_VOLUME procedure
 - description R2-96, UG-190
 - example of UG-206
- shading
 - constant intensity R2-72
 - contour plots R1-529
 - examples UG-133
 - Gouraud R2-71
 - light source UG-131
 - methods UG-131
 - setting parameters UG-132
 - setting parameters for shading R2-70
 - surfaces R1-564, R2-83, R2-91, R2-345, UG-131
 - setting parameters R2-79
 - volumes R2-96
- shared colormaps UG-A-80—UG-A-81
- shared library, io.so R2-62
- sharpening, of images UG-160
- shell processes PG-299—PG-300
- shell window
 - creating PG-416, PG-482
 - with initial layout PG-414
 - destroying PG-468
 - managing PG-484
 - realizing PG-484
 - unmanaging PG-484
- SHELL, UNIX environment variable PG-293
- SHIFT function R2-99
- shifting elements of arrays R2-99
- show a widget PG-469
- SHOW3 procedure R2-104, UG-129
- shrinking images R1-115, R2-21, R2-36
- SIGMA function
 - computing standard deviation of array R2-107
 - description R2-107
- signal processing
 - convolution R1-125
 - creating filters R1-250
 - Fast Fourier Transform R1-328
 - Hanning filter R1-378
 - histogram equalization R1-383
 - Lee filter R1-424
- simultaneous equations, solution of R2-177
- SIN function R2-109
- SINDGEN function R2-111
- sine function R2-109
- sine, hyperbolic R2-112
- single step command UG-27, UG-30
- single-precision floating-point data
 - shown in figure PG-146
- singular value decomposition
 - curve R2-177
 - SVD function R2-179
- SINH function R2-112
- SIXEL output UG-A-58—UG-A-61
- SIZE function
 - description R2-114, PG-272
 - examples PG-272
 - table of type codes PG-272
- size of arrays, determining R2-114, PG-270
- SKIPF procedure R2-115, PG-229
- skirt
 - adding to surface R2-345
 - shown in figure R2-346
- SLICE_VOL function R2-117, UG-179, UG-193
- slicing
 - plane, defining R2-253
 - volumes R2-117, R2-329
- sliders
 - and text input field PG-437
 - creating PG-437
 - description R2-414
 - example PG-438
 - orientation R2-415
 - using PG-437
- SMOOTH function R2-119, UG-158
- smoothing
 - boxcar R2-119
 - contour plots R2-519
 - contours, using Spline keyword UG-108

- mean UG-159
- median UG-159
- of images UG-158–UG-159
- Sobel edge enhancement R2-122
- SOBEL function R2-122, UG-160
- software fonts
 - index of R2-544
 - See fonts
- solids, shading R1-564
- SORT function R2-125, UG-290
- sorting
 - array contents R2-125
 - SORT function UG-290
 - tables, using QUERY_TABLE
 - R2-1, PG-179–PG-180, UG-276
- spaces in statements PG-52
- sparse data R1-368
- spatial transformation of images
 - R1-526, R1-574
- SPAWN procedure
 - accessing C programs from
 - PV-WAVE PG-312
 - avoiding shell processes PG-300
 - calling without arguments PG-298
 - capturing output PG-301
 - communicating with a child process
 - PG-310
 - description R2-126
 - example of PG-298, PG-312
 - interapplication communication
 - PG-309
 - non-interactive use PG-299
 - syntax PG-297
 - when to use PG-307
- spawning a process R2-126
- special characters, list of UG-33
- special effects
 - created with color UG-328
 - using Z-buffer UG-A-99
 - with color UG-A-94
- SPHERE function
 - description of R2-129
 - example of UG-207
- spheres
 - defining with SPHERE function
 - R2-129, UG-197
 - generating R1-568
 - gridding R1-373
- spherical
 - gridding UG-192
 - surfaces UG-190
- SPLINE function R2-132
- splines
 - cubic R2-132
 - interpolation of contour plots
 - R2-519
- SQL, description of UG-264
- SQRT function R2-134
- square root, calculating R2-134
- stacking plots UG-83
- standard deviation, calculating R2-136
- standard error output
 - LUNs PG-142
 - reserved LUNs PG-140
- standard input
 - LUNs PG-141
 - reserved LUNs PG-140
- Standard Library
 - location of PG-16, PG-255, UG-23
 - submitting programs PG-255
 - suggestions for writing routines
 - PG-256
- standard output
 - LUNs PG-141
 - reserved LUNs PG-140
- starting PV-WAVE
 - description UG-12
 - executing a command file UG-13
 - from an external program PG-314
 - interactive UG-12
- startup file
 - for UNIX UG-49
 - for VMS UG-50
- statements
 - assignment PG-9, PG-53
 - blocks of PG-10, PG-60
 - can change data types PG-55
 - CASE PG-10, PG-62
 - COMMON block PG-12, PG-63

- compiling and executing with
 - EXECUTE function PG-273
- components of PG-52
- examples of assignment PG-54
- executing one at a time R1-313
- FOR PG-9
- function call PG-11
- function definition PG-11
- GOTO PG-10, PG-70
- groups of PG-60
- IF PG-9, PG-70, PG-71
- labels PG-52
- list of 12 types PG-51
- procedure
 - calls PG-11, PG-73
 - definition PG-11, PG-76
- REPEAT PG-10, PG-77
- spaces in PG-52
- tabs in PG-52
- types of PG-9, PG-51
- WHILE PG-10, PG-78
- statically linked programs PG-333
- STDEV function
 - computing standard deviation R2-136
 - description R2-136
- stdio facility PG-310
- STOP procedure R2-138
- stopping PV-WAVE UG-13
- storing image data PG-189
- STR_TO_DT function
 - description R2-159, UG-229, UG-231
 - example PG-171, UG-232
 - templates UG-231
- STRARR function R2-139
- STRCOMPRESS function R2-140
- stream mode files, VMS PG-226
- STRETCH procedure R2-142, UG-155, UG-330
- stretching the current color table R2-302, UG-320
- string data
 - a basic type PG-3
 - shown in figure PG-146
- STRING function
 - description R2-144
 - example PG-37, PG-124
 - formatting data PG-187
 - syntax PG-124
 - with
 - BYTE function PG-199
 - single byte argument PG-125
 - structures PG-118
- string processing PG-121
 - compressing white space R2-140
 - converting
 - lower to upper case R2-167
 - upper to lower case R2-150
 - extracting substrings R2-154
 - inserting substrings R2-157
 - locating substrings R2-156
 - removing
 - blanks R2-140
 - excess white space R2-141
 - leading and trailing blanks R2-162
- string variables
 - binary transfer of PG-197
 - determining number of characters to transfer PG-197
 - passing into QUERY_TABLE UG-279
- strings
 - array arguments PG-133
 - arrays, creating R2-111, R2-139
 - changing case PG-127
 - concatenating PG-123
 - constants PG-4, PG-21
 - definition of PG-121
 - determining length of PG-122
 - examples of string constants PG-21
 - extracting substrings PG-131
 - formatting PG-122
 - FORTRAN and C formats PG-166
 - importing with free format PG-160
 - initializing to a known length PG-199
 - input/output with structures PG-117
 - inserting substrings PG-131

- length issues with structures
 - PG-118
- locating substrings PG-131
- non-string arguments PG-133
- obtaining length of PG-130
- operations supported PG-122
- operators, table of PG-8
- reading with Format keyword
 - PG-166
- removing white space PG-122,
 - PG-128
- STRCOMPRESS function PG-122
- STRING function PG-122
- string operations supported PG-121
- STRLEN function PG-122
- STRLOWCASE function PG-122
- STRTRIM function PG-122
- STRUPCASE function PG-127
- substrings PG-131
- transferring as binary data PG-197
- working with PG-121
- writing to a file PG-164
- strip chart
 - controlling the pace R2-339
 - displaying R2-336
- STRLEN function
 - description R2-148
 - determining string length PG-122
 - example PG-130
 - output PG-197
 - syntax PG-130
- STRLOWCASE function
 - converting to lower case PG-122
 - description R2-150
 - example PG-127
 - syntax PG-127
- STRMESSAGE function R2-152
- STRMID function R2-154, PG-132
- STRPOS function R2-156, PG-131
 - example PG-131
- STRPUT procedure R2-157, PG-132
- STRTRIM function R2-162
 - examples PG-128
 - removing white space PG-122
 - syntax PG-128
- STRUCTREF function PG-104
 - description R2-164
- structure references
 - examples PG-109
 - nesting of PG-108
 - syntax PG-109
- structures
 - See also unnamed structures
 - advanced usage PG-119
 - arrays PG-112, PG-114
 - creating unnamed PG-105
 - data type PG-3
 - date/time R1-204, R1-211, R1-290,
 - UG-225
 - defining PG-102
 - definition of PG-101
 - deleting PG-102
 - expressions PG-39
 - formatted input/output PG-116
 - getting information about R1-410,
 - PG-110, PG-402
 - importing data into PG-161
 - in relation to tables UG-287–
 - UG-288
 - input and output PG-116
 - introduction to PG-101
 - listing references to R2-164
 - number of tags in R1-472
 - N_TAGS function PG-119
 - parameter passing PG-111
 - PRINT and PRINTF PG-116
 - processing with tag numbers
 - PG-119
 - reference to a field PG-108
 - references to R2-164
 - removing definitions from memory
 - R1-242
 - REPLICATE function PG-114
 - scope of PG-105
 - storing into array fields PG-112–
 - PG-113
 - string
 - input/output PG-117
 - length issues PG-118
 - structure references PG-109
 - subscripted references PG-108

- tag names R2-188
- TAG_NAMES function PG-119
- unformatted input/output PG-117
- using INFO PG-110
- STRUPCASE function R2-166, PG-127
- Style field of !X, !Y, !Z UG-64
- style, of axes R2-526, R2-531, R2-534, R2-557
- subarrays, structure of PG-86
- submatrices PG-98
- subscripts
 - * operator PG-85
 - combining arrays with PG-89–PG-90
 - notation for columns and rows PG-81
 - of pixels inside polygon region R1-550
 - ranges PG-56, PG-84
 - 4 types PG-84–PG-85
 - table of PG-86
 - using asterisk UG-36
 - reference syntax PG-82
 - storing elements in arrays PG-91
 - subscript arrays PG-53–PG-57
 - using arrays as PG-57
 - with
 - matrices PG-82
 - multidimensional arrays PG-82
 - scalars PG-84
 - structures PG-108
- subsetting
 - data
 - SORT function UG-290
 - Where clause UG-277
 - tables R2-1, UG-277
- substrings
 - extracting PG-131
 - from strings R2-154
 - inserting into strings R2-157–R2-158, PG-122, PG-131
 - locating in strings R2-156, PG-131
- subtraction operator (–)
 - examples PG-45
 - operator precedence PG-32
- SunOS Version 4, error handling PG-268
- superimposing
 - See overlaying
- surface data
 - in a file R2-87
 - volumetric R2-307, R2-329
- surface plot
 - adding skirt R2-345
 - angle of X rotation R2-497
 - angle of Z rotation R2-497
 - changing interactively R2-344
 - color of bottom R2-499
 - combining with image and contour plots R2-104
 - curve fitting to R2-172
 - display 3D array as R2-168
 - displaying data as R2-342
 - draw
 - a skirt on R2-518
 - horizontal lines only R2-508
 - lower surface only R2-509
 - upper surface only R2-523
 - easy way to change appearance R2-342, R2-344
 - example of combining with contour plot R2-517
 - placement of Z axis R2-533
 - rendering of shaded R1-564, R2-83, R2-91
 - rotating data UG-117
 - saving the 3D to 2D transformation R2-517
 - shaded R1-564, R2-345
 - shown with and without a skirt R2-346
 - superimposed with contours R2-517, UG-125
- SURFACE procedure
 - combining with CONTOUR procedure UG-125, UG-127
 - description R2-168, UG-93, UG-112
 - examples UG-112, UG-114, UG-119

- list of keywords UG-113
- transformation matrix R2-169
 - example UG-119
- SURFACE_FIT function R2-172
- SURFR procedure R2-174
- suspending PV-WAVE
 - on a UNIX system UG-14
 - on a VMS system UG-14
- SVBKS procedure R2-177
- SVD procedure R2-179
- SVDFIT function
 - description R2-181
 - example of basis function for R1-137
- swap area, increasing PG-285
- symbol
 - displaying in a volume R2-263
 - plotting R2-547
 - VMS, defining R2-74
 - VMS, returnable R1-361
- symbols
 - connecting with lines R2-516, UG-72
 - marker UG-72, UG-73
 - plotting R2-516
 - size of plotting R2-520
 - user-defined R2-516, UG-73
 - VMS, deleting R1-238
- SYSGEN parameters
 - VirtualPageCount PG-286
 - Wsmax PG-286
- system
 - color tables UG-320
 - date, creating variable with R2-199
 - functions, using for efficient programming PG-280
 - time, creating variable with R2-184, R2-199
- system limits and compiler PG-241
- system variables
 - adding R1-236
 - creating R1-236
 - definition of PG-26
 - description of PG-5, UG-24
 - exclamation point UG-34
 - for tick label formats UG-82

- getting information about PG-403
- listing their values PG-403
- passing of PG-246
- relationship to keywords UG-24, UG-57
- summary of PG-27
- systems, window
 - See window systems
- SYSTIME function R2-184

T

- T3D procedure UG-116
 - data transformation UG-116, UG-118
 - description R2-185
 - examples UG-119
 - setting up data for viewing UG-194
- table functions
 - description of UG-263
 - overview UG-264
- table widget, creating PG-462
- tables
 - creating with BUILD_TABLE function R1-63, UG-266–UG-270
 - date/time data UG-281
 - determining unique values R2-230
 - GUI tool R2-472
 - in relation to structures UG-287–UG-288
 - plotting UG-286–UG-287
 - date/time axis UG-284
 - printing with column titles UG-285
 - rearranging with QUERY_TABLE UG-271
 - renaming columns UG-272
 - sorting R2-1
 - columns in descending order UG-276
 - with QUERY_TABLE function PG-179–PG-180
 - subsetting R2-1
 - with Where clause UG-277

- unique elements of R2-230
- viewing with INFO procedure UG-268
- tabs in statements PG-52
- Tag Image File Format
 - See TIFF
- TAG_NAMES function R2-188, PG-119
- tags
 - definition of PG-101
 - names, of structures R2-188
 - number of R1-472
 - reference to structure PG-108
- TAN function R2-190
- tangent R2-190
 - hyperbolic R2-191
- TANH function R2-191
- tape, magnetic
 - accessing under VMS PG-229
 - mounting a tape drive (VMS) PG-230
 - reading block of image data PG-231
 - REWIND procedure PG-229
 - rewinding R2-38
 - SKIPF procedure PG-229
 - skipping
 - backward on a tape PG-231
 - forward on the tape (VMS) PG-230
 - TAPRD procedure PG-229
 - TAPWRT procedure PG-229
 - WEOF procedure PG-229
 - writing to PG-229
- TAPRD procedure R2-193, PG-229
- TAPWRT procedure R2-194, PG-229
- TEK_COLOR procedure R2-195, UG-326, UG-330
- Tektronix 4115 device, mimicking colors UG-330
- Tektronix 4510 rasterizer UG-A-61–UG-A-64
- Tektronix terminal
 - color table R2-195
- Tektronix terminal output UG-A-65–UG-A-68
- terminating, execution of a command file UG-20
- text
 - 3D orientation of R2-521
 - See also annotation, fonts
 - alignment R2-497
 - angle of R2-513
 - centered R2-497
 - changing case R2-150, R2-166, PG-127
 - character size R2-501, R2-518
 - color of R2-504
 - concatenation PG-123
 - converted from byte data PG-125
 - converting parameters to R2-168, PG-124
 - editor programs UG-22
 - extracting PG-131
 - inserting PG-131
 - locating substrings PG-131
 - non-string arguments to string routines PG-133
 - obtaining length of strings R2-148, PG-130
 - orientation of R2-513
 - positioning commands, for Post-Script UG-A-39
 - removing white space PG-128
 - size of R2-501, R2-518
 - string operations supported PG-121
 - suppressing clipping R2-511
 - vector-drawn UG-291
 - viewed in scrolling window R2-349, R2-351
 - width of R2-522
 - working with PG-121
 - writing to the display R2-490
- text widget
 - and popup menus R2-462
 - creating PG-443
 - description R2-481
 - multi-line PG-445
 - scroll bars PG-443
 - single line
 - editable PG-444
 - read-only PG-443

- thickness
 - of axis line R2-558
 - of characters R2-543
- THREED procedure R2-197
- three-dimensional
 - coordinate systems
 - converting to two-dimensional coordinate systems UG-119
 - user-defined UG-121
 - data, See three-dimensional data
 - graphics
 - combining with images UG-129–UG-130
 - drawing UG-115
 - rotating data UG-117
 - scaling data UG-117
 - shaded surfaces R1-564, R2-83, R2-91, R2-185
 - SHOW3 procedure R2-104
 - translating data UG-117
 - gridding R1-322, R1-367, UG-191
 - transformations
 - of text UG-292
 - with two-dimensional procedures UG-124
 - volumes UG-190
- three-dimensional data
 - contouring R1-119, UG-94
 - displayed as surface R2-168
 - MOVIE procedure R1-466
 - plotting UG-93
 - viewing as iso-surface R2-307
- threshold
 - dithering UG-148
 - value of an iso-surface R2-312
- thresholding of images UG-152
- tick
 - intervals, setting number of UG-66
 - label format UG-81–UG-82
 - marks
 - annotation of R2-527, R2-531, R2-535
 - controlling length of R2-522, R2-550, UG-78
 - extending away from the plot R2-522, UG-78
 - format of R2-527
 - intervals between R2-528, R2-532, R2-535
 - intervals, setting number of R2-560
 - linestyle of R2-507, R2-525, R2-529, UG-78
 - number of minor marks R2-525, R2-530, R2-534, R2-556, UG-78
 - producing non-linear marks UG-78
 - suppressing minor marks R2-556
 - values of R2-526, R2-530, R2-534
- TIFF
 - conformance levels PG-192
 - data
 - saving in TIFF format PG-191
 - writing image data to a file R1-221, R1-223
 - writing with DC_WRITE_TIFF PG-191
 - files
 - compression of PG-192
 - reading R1-198
 - reading a file in R1-198
 - writing image data to a file R1-221, R1-223
 - time
 - current R2-184
 - elapsed between dates R1-277
 - formats, STR_TO_DT function UG-231
 - timer, adding PG-489
 - tips, programming PG-472, PG-498
 - title
 - See also annotation
 - adding to two-dimensional plots UG-64
 - of axes R2-529, R2-532, R2-535, R2-560
 - of plot R2-523, R2-550
 - setting size of R2-501, R2-543
 - TODAY function R2-199, UG-259

- toggle button, menu PG-430
- tool box R2-485, PG-433–PG-435
- top-level window
 - See shell window
- TOTAL function
 - description R2-200
 - example of PG-280
 - use of PG-47
- total, of array elements R2-200
- TQLI procedure R2-201
- traceback information PG-262
- transferring
 - binary data PG-154
 - binary string data PG-197
 - complex data, with XDR routines PG-210
 - data with
 - C or FORTRAN formats PG-165
 - READU, WRITEU PG-194
 - XDR files PG-206
 - date/time data PG-168
 - images in server UG-A-82
- transformation
 - 3D volumes R2-270
 - geometric UG-167–UG-168
 - gray level UG-152
 - matrix, See transformation matrices
 - saving R2-517
- transformation matrices R2-521
 - 3D points R1-573
 - 4-by-4 R2-270, R2-549
 - description UG-116
 - rotating data UG-117
 - scaling data UG-117
 - Transform keyword for RENDER
 - function UG-202
 - translating data UG-117
 - with POLY_TRANS function R1-573
 - with SURFACE procedure UG-119
 - with T3D procedure UG-118
- translation table
 - bypassing UG-A-71, UG-A-85
 - color UG-308
- translucency, in images R1-561, R2-267
- transmission component of color, for
 - RENDER function UG-199
- transparency, in images R1-543
- TRANSPOSE function R2-204, PG-47
- transposing
 - arrays or images R2-51, R2-204
 - rows and columns PG-47
- TRED2 procedure R2-207
- TRIDAG procedure R2-208
- TRNLOG function R2-209, PG-295
- true
 - definition for IF statement PG-71
 - definitions for different data types PG-71
 - representation of PG-42
- true-color
 - compared to pseudo-color UG-146
 - images
 - definition UG-147
 - generating R1-406
- truth table
 - for AND operator PG-43
 - for OR operator PG-43
 - for XOR operator PG-43
- TV procedure
 - default coordinates for UG-59
 - description R2-212, UG-135
- TVCRS procedure R2-217
 - description UG-136, UG-143
 - example UG-143
- TVLCT procedure
 - description R2-219, UG-136, UG-311
 - examples UG-325
 - uses of UG-145
- TVMENU function PG-473
- TVRD function
 - description R2-223, UG-136, UG-142
 - examples UG-142
- TVSCL procedure
 - description R2-225, UG-136
 - examples UG-153
- 24-bit image data
 - writing data to a file R1-221
 - how stored PG-189

- 24-bit color, using color to differentiate data sets UG-A-93
- two-dimensional
 - coordinate systems, converting from three-dimensional coordinate systems UG-119
 - data, summary of plotting routines UG-56
 - gridding R1-319, R1-364
- 2D gridding UG-191
- type conversion
 - byte R1-68
 - complex R1-109
 - double-precision R1-267
 - floating-point R1-340
 - functions
 - examples PG-37
 - overflow conditions PG-36
 - syntax PG-38
 - table of PG-36
 - integer R1-338
 - longword integer R1-442
 - scaled byte R1-73
 - string R2-144
- type of variable, determining R2-114, PG-272
- types
 - See data types

U

- undefined results
 - Infinity PG-263
 - NaN PG-263
- undefined variables, checking for PG-270
- unexpected colors UG-A-96
- unformatted data
 - advantages and disadvantages of PG-151
 - FORTRAN generated, reading in UNIX environment PG-199
 - problems reading PG-205
 - reading with associated variable method PG-212

- routines for transferring PG-156
- VMS FORTRAN generated, example of reading PG-202
- unformatted I/O
 - description PG-159
 - with associated variables PG-212
 - with structures PG-117
- unformatted string variables, I/O PG-160
- uniformly distributed random numbers R2-13
- UNIQUE function R2-230, UG-7, UG-264
- unique values, determining R2-230
- UNIX
 - avoiding shells with SPAWN PG-300
 - calling from PV-WAVE R1-77
 - commands from within PV-WAVE R2-17, R2-33, R2-56, R2-64, R2-114, R2-126, PG-293–PG-297
 - CALL_UNIX PG-359
 - Control\ UG-15
 - compiling for LINKNLOAD PG-322, PG-326, PG-328
 - description of files in PG-223
 - environment variable
 - adding R2-64
 - changing R2-64
 - inspecting R1-295
 - translation R1-356
 - WAVE_DEVICE UG-45
 - WAVE_DIR UG-45
 - WAVE_PATH UG-47
 - WAVE_STARTUP UG-49
 - environment, description of PG-292
 - FORTRAN programs
 - in relation to ASSOC function PG-218
 - writing to PV-WAVE PG-200
- linking
 - C program to PV-WAVE PG-347
 - FORTRAN applications to PV-WAVE PG-348
- offset parameter PG-216

- pipe, description of PG-310
- reading data in relation to FORTRAN PG-199
- reserved LUNs PG-142
- sending output file to printer or plotter UG-A-6
- shells PG-299
- virtual memory PG-285
- UNIX_LISTEN function R2-232
 - definition PG-371
 - example PG-392
 - PV-WAVE as a server PG-375
 - PV-WAVE as a server, example PG-392
 - used with CALL_WAVE PG-371
- UNIX_REPLY function R2-234
 - example of PG-392, PG-394
 - PV-WAVE as a server PG-375, PG-392
 - used with CALL_WAVE PG-371
- unmanage, shell widget PG-484
- unnamed structures
 - and variable-length fields PG-106
 - creating PG-105
 - internal names for PG-107
 - scope of PG-105
 - syntax of PG-106
 - to define menu items PG-429
 - uses for PG-105
 - specify resources PG-483
- unsharp masking UG-161
- updating a file, using OPENU PG-137
- user data, defining and passing PG-467
- user interface, graphical
 - See GUI
- Users' Library
 - documentation for PG-256
 - location of in PV-WAVE PG-255
 - submitting programs to PG-255
 - support for routines in PG-256
- USERSYM procedure R2-235, UG-73
- UT_VAR PG-387, PG-392

V

- VAR_TO_DT function
 - description of R2-238, UG-232
 - example of UG-232, UG-244–UG-252
- variable length record format files, VMS PG-226
- variables
 - associated R1-43, PG-24
 - in assignment statements PG-59
 - attributes PG-23
 - checking for undefined PG-270
 - complex, importing data PG-160
 - creating
 - empty ones with date/time type UG-227
 - new system variables R1-236
 - data types PG-24
 - declaring PG-5
 - definition of PG-4, PG-5, PG-23
 - deleting R1-244
 - determining number of elements in PG-270
 - disappearing PG-251, PG-258
 - environment variables R1-410
 - erasing UG-27
 - forcing to specific data type PG-36
 - in common blocks PG-63
 - invalid names PG-26
 - local (definition of) PG-237
 - naming conventions PG-25
 - number of structure tags R1-472
 - obtaining environment variables R1-295
 - passing PG-467
 - printing R1-580
 - reading R1-580
 - into from display device UG-142
 - size and type information PG-24, PG-272
 - storing in structures PG-112
 - structure of PG-24

- structure tag names R2-188
- system
 - See system variables
 - types of PG-24
 - valid names PG-26
 - virtual memory PG-284
- VAX/VMS Extended Metafile System
 - UG-A-13
- vector
 - cross product R1-142
 - definition of PG-4, PG-24
 - fields, plotting R1-510, R2-244
 - from 3D arrays R2-240
 - of strings R2-149, R2-151, R2-167
 - reversing R2-36
 - solution, improving R1-468
 - subscripts PG-86
- vector graphics colors UG-311, UG-A-92
- VECTOR_FIELD3 procedure R2-240, UG-195
- vector-drawn text
 - See text
- vectors
 - BUILD_TABLE function UG-269
 - building tables from R1-63
 - definition of PG-24
 - deriving unique elements from R2-230
 - using as subscripts to other arrays PG-88
- VEL procedure R2-244
- VELOVECT procedure R2-248
- Ventura Publisher, inserting PV-WAVE
 - plots into UG-A-47–UG-A-48
- version number of PV-WAVE R2-553, PG-29
- !Version system variable R2-553
- vertex lists
 - description of UG-187
 - merging R1-557
- video memory, of workstation UG-A-79
- View Control window R2-259
- View Orientation window R2-260
- view setup
 - 3D view R2-77
 - centering data R1-86
 - defining R2-253
 - description of UG-194
 - summary of routines R1-23
- VIEWER procedure R2-253, UG-179, UG-194
- viewport, defining R2-67, R2-79
- virtual memory
 - description PG-283
 - in relation to
 - PV-WAVE PG-283
 - swap area PG-285
 - minimizing use of PG-288
 - under UNIX PG-285
 - variable assignments PG-284
 - VMS PG-286
- VirtualPageCount parameter PG-286
- visual class(es)
 - for X windows UG-A-77
 - not inherited by PV-WAVE UG-A-90
- visual data analysis, importance of color UG-310
- VMS
 - access mode PG-225
 - accessing magnetic tape PG-229
 - binary CGM output UG-A-13
 - binary files PG-149
 - calling PV-WAVE R1-429
 - command interpreter PG-299
 - commands from within PV-WAVE PG-295–PG-297
 - environment variables PG-295–PG-296, UG-46
 - error handling PG-268
 - file
 - access PG-225
 - attributes PG-227
 - organization PG-224
 - formal parameters for procedures and functions PG-236
 - FORTRAN programs, writing to a file PG-202
 - libraries PG-251, PG-253

- linking a C program to PV-WAVE
 - PG-349
- logical names, deleting R1-240
- mounting a tape drive PG-230
- offset parameter PG-217
- record attributes PG-226
- reserved LUNs PG-142
- RMS files, reading images PG-217
- searching libraries PG-252
- sending output file to printer or plotter
 - UG-A-6
- specific information on data files
 - PG-224
- stream mode files PG-226
- symbols
 - defining R2-74
 - deleting R1-238
 - return value R1-361
- transferring record-oriented data
 - PG-150
- variable length format PG-226
- virtual memory PG-286
- working set size PG-286
- VOL_MARKER procedure R2-263, UG-195
- VOL_PAD function R2-266, UG-192
- VOL_REND function R2-267, UG-195
- VOL_TRANS function R2-270, UG-193
- VOLUME function
 - description of R2-272
 - example of UG-211–UG-218
- volumes
 - defining R2-253
 - with VOLUME function R2-272, UG-197
 - displaying markers in R2-263
 - generating UG-187
 - manipulating R1-24, UG-192
 - padding R2-266
 - rendering R1-24, R2-28, R2-240, R2-263, R2-267, UG-183–UG-195
 - shaded R2-96
 - slicing
 - example of UG-210 with SLICE_VOL function R2-117
 - transforming R2-270
 - volumetric surface data R2-307, R2-329
 - VT graphics terminals UG-A-54

W

- w_cmpnd_reply function PG-364, PG-368
- w_get_par function PG-366
- w_listen function PG-365
- w_send_reply function PG-364, PG-367
- w_simpl_reply function PG-364
- w_smpl_reply function PG-367
- WAIT procedure R2-274
- waiting, in programs R2-274
- warping of images R1-526, R1-574, UG-168
- WAVE Widgets
 - application example PG-473
 - arranging PG-420
 - basic steps in creating PG-413
 - combining with Widget Toolbox PG-480
 - creating PG-416
 - event loop PG-471
 - getting values PG-466
 - hiding PG-469
 - initializing PG-414
 - introduction PG-410
 - list of PG-418
 - location of PG-411
 - passing user-defined data PG-467
 - sensitivity PG-470
 - setting
 - colors PG-463
 - fonts PG-464
 - values PG-466
 - showing PG-469
 - submitting to Users' Library PG-411
 - widget hierarchy PG-416
- WAVE_DEVICE UG-44–UG-45

WAVE_DIR UG-45–UG-46
 WAVE_PATH UG-46–UG-47
 WAVE_STARTUP UG-49–UG-50
 wavecmd routine
 description of PG-313
 examples PG-315, PG-316,
 PG-318
 when to use PG-307
 with FORTRAN programs PG-315
 waveinit routine PG-313–PG-314
 wavestartup file UG-49
 waveterm routine PG-313, PG-315
 wavevars function
 accessing data in PV-WAVE
 PG-350
 description of PG-321, PG-335
 examples PG-354–PG-358
 parameters PG-350
 syntax PG-321, PG-350
 WDELETE procedure R2-275, UG-A-98
 WEOF procedure R2-276, PG-229
 WgAnimateTool procedure
 description R2-277
 event handling R2-281
 WgCbarTool procedure
 description R2-284
 event handling R2-287
 stores colors in common block
 R2-288
 WgCeditTool procedure
 benefits of R2-291
 compared to WgCtTool procedure
 R2-291
 description R2-290
 event handling R2-299
 stores colors in common block
 R2-293
 utility widgets, shown in figure
 R2-298
 WgCtTool procedure
 compared to WgCeditTool R2-302
 description R2-301
 event handling R2-305
 stores colors in common block
 R2-303
 WgIsoSurfTool procedure
 description R2-307
 event handling R2-312
 WgMovieTool procedure
 benefits of R2-319
 description R2-315
 event handling R2-320
 WgSimAgeTool procedure
 description R2-323
 event handling R2-327
 WgSliceTool procedure
 description R2-329
 event handling R2-334
 shown in figure R2-332
 slices three-dimensional data
 R2-329
 WgStripTool procedure
 description R2-336
 event handling R2-340
 WgSurfaceTool procedure
 benefits of R2-344
 description R2-342
 event handling R2-347
 WgTextTool procedure
 description R2-349
 event handling R2-352
 Where clause UG-277
 WHERE function R2-354, PG-55,
 PG-58, PG-277, UG-A-23
 WHILE statement PG-10, PG-78
 white space
 compressing R2-141
 removing R2-140, R2-162
 removing from strings PG-122,
 PG-128
 widget
 class PG-482
 definition of PG-410, PG-482
 destroy PG-485
 hierarchy
 See widget hierarchy
 ID (handle) PG-418, PG-482
 input focus of PG-484
 managing PG-484
 mapped with an X window PG-484
 realizing PG-484

- resources PG-483
- resources, retrieving R2-376
- sensitivity PG-470
- setting resources PG-465
- unmanaging PG-484
- widget hierarchy
 - close PG-485
 - definition PG-416
 - displaying PG-471
- Widget Toolbox
 - adding
 - callbacks PG-485
 - event handlers R2-367, PG-487
 - timers PG-490
 - combining with WAVE Widgets PG-480
 - creating widgets PG-482
 - cursors PG-D1–PG-D5
 - destroying widgets PG-484
 - displaying widgets PG-484
 - example program PG-495
 - include files PG-495
 - initializing PG-481
 - managing widgets PG-484
 - resources PG-483
 - running an application PG-494
- window
 - creating R2-356
 - current R2-362
 - damage repair UG-A-7
 - deleting by ID number UG-A-98
 - exposing and hiding R2-363
 - looks dark and unreadable UG-A-82
 - margin around plot R2-525, R2-530, R2-534
 - menus, creating R2-221, R2-359
 - positioning plot in R2-67, R2-79
 - selecting R2-362
 - specifying plot coordinates R2-515
 - turning into icon R2-363
- window index, used with drawing area PG-439
- WINDOW procedure R2-356
- window systems
 - backing store UG-A-7
 - common features UG-A-6
 - defining with WAVE_DEVICE UG-44–UG-45
 - features of those supported UG-A-6
 - retained UG-A-7
 - table of UG-A-2
 - X Windows UG-53, UG-A-69
- WMENU function R2-359
- word-processing application, reading data from PG-175
- words, reserved PG-25
- working set
 - description PG-283
 - quota PG-287
 - VMS PG-286
- write mask
 - interacts with selected graphics function UG-A-95
 - using to create special effects UG-328, UG-A-94
- WRITEU procedure
 - description R2-361, PG-154
 - example PG-195
 - syntax PG-193
 - transferring strings with PG-197
- writing
 - 24-bit image data to a file R1-221
 - 8-bit image data to a file R1-219
 - ASCII free format data to a file R1-203, R1-212
 - data, output formats PG-164
 - date/time data UG-255
 - integer data, using format reversion PG-A-5–PG-A-6
 - strings to a file PG-164
 - TIFF image data to a file R1-223
 - to a data file, basic steps PG-12
 - to end of file on tape (VMS) PG-229
 - to tape (VMS) PG-229
 - using
 - UNIX FORTRAN programs PG-200
 - VMS FORTRAN PG-202
 - WRITEU PG-195

WSET procedure R2-362
 WSHOW procedure R2-363
 Wsquo quota PG-287
 WtAddCallback function R2-364,
 PG-485
 WtAddHandler function R2-367,
 PG-487
 WtClose function R2-369, PG-485
 WtCreate function R2-371, PG-480,
 PG-482
 WtCursor function R2-373
 WtGet function R2-376
 WtInit function R2-380, R2-432, PG-480
 WtInput function R2-382
 WtList function R2-385
 WtLoop function R2-387, PG-494
 WtMainLoop function R2-389
 WtPointer function R2-390
 WtSet function R2-392, PG-480
 WtTable function R2-395
 WtTimer function R2-402, PG-489
 WtWorkProc unction R2-404
 wvsetup (WVSETUP.COM) file
 methods of executing UG-12
 WwButtonBox function R2-406, PG-434
 WwCommand function R2-410, PG-459
 WwControlsBox function R2-414,
 PG-437
 WwDialog function R2-419, PG-453
 WwDrawing function R2-422, PG-439
 WwFileSelection function R2-426,
 PG-456
 WwGetValue function PG-466
 description R2-430
 WwInit function PG-413—PG-414
 WwLayout function R2-434, PG-420
 WwList function R2-439, PG-446
 WwLoop function R2-443, PG-414
 WwMainWindow function R2-444,
 PG-416
 WwMenuBar function R2-447, PG-426
 WwMenuItem function R2-451
 WwMessage function R2-453, PG-449
 WwOptionsMenu function R2-457,
 PG-428

WwPopupMenu function R2-461,
 PG-427
 WwRadioBox function R2-464, PG-436
 WwSetValue function R2-468, PG-466
 WwTable function R2-472
 WwText function R2-481, PG-443
 WwToolBox function R2-485, PG-435

X

X server
 closing connection UG-A-72
 connecting to PG-414
 !X system variable, fields of R2-554—
 R2-560
 X Window System
 \$DISPLAY environment variable
 UG-A-69
 clients UG-A-69
 color translation UG-A-84
 damage repair UG-A-7
 DECW\$DISPLAY logical UG-A-69
 DEVICE procedure UG-A-71
 graphics function codes UG-A-75
 IDs UG-A-97
 keywords UG-A-71
 overview UG-A-69
 pixmaps UG-A-85—UG-A-86
 private colormaps UG-A-82
 providing a GUI for application
 UG-A-70
 resources PG-464
 servers UG-A-69
 shared colormaps UG-A-81
 using with PV-WAVE UG-53
 visual classes UG-A-77
 X11 socket R2-61
 .Xdefaults file PG-465
 XDR
 data
 allowed I/O routines PG-205
 byte variables PG-206
 differences from normal I/O
 PG-205

- string variables PG-206
- transferring PG-206
- files
 - conventions for reading and writing PG-210
 - creating with C programs PG-208
 - description of PG-205
 - opening PG-205
 - reading, byte data PG-206, PG-207
 - reading, READU procedure PG-209
- routines
 - for transferring complex data PG-210
 - table of PG-210
- Xlib PG-406
- XOR operator
 - description of PG-50
 - example UG-A-95
 - operator precedence PG-33
 - truth table PG-43
- Xt Intrinsic PG-406, PG-414, PG-481
- Xt TimeOut function PG-489
- XY plots
 - examples UG-62
 - scaling axes UG-63—UG-64
- XYOUTS procedure R2-490, UG-69, UG-121

- ZOOM procedure R2-493, UG-142
- zooming
 - data in strip chart R2-337
 - images R2-493, UG-140
 - in 3D window R1-86
 - of reference cube R2-309, R2-331
- ZROOTS procedure R2-495

Y

- Y axis, scaling with YNozero UG-64
- !Y system variable, fields of R2-561

Z

- !Z system variable, fields of R2-561
- Z-buffer
 - output
 - description UG-A-99
 - DEVICE procedure UG-A-99
 - keywords UG-A-99
 - using to create special effects R1-543, UG-A-99

