



What's New in IDL 5.4



IDL Version 5.4
September, 2000 Edition
Copyright © Research Systems, Inc.
All Rights Reserved

Restricted Rights Notice

The IDL[®] software program and the accompanying procedures, functions, and documentation described herein are sold under license agreement. Their use, duplication, and disclosure are subject to the restrictions stated in the license agreement. Research Systems, Inc., reserves the right to make changes to this document at any time and without notice.

Limitation of Warranty

Research Systems, Inc. makes no warranties, either express or implied, as to any matter not expressly set forth in the license agreement, including without limitation the condition of the software, merchantability, or fitness for any particular purpose.

Research Systems, Inc. shall not be liable for any direct, consequential, or other damages suffered by the Licensee or any others resulting from use of the IDL software package or its documentation.

Permission to Reproduce this Manual

If you are a licensed user of this product, Research Systems, Inc. grants you a limited, non-transferable license to reproduce this particular document provided such copies are for your use only and are not sold or distributed to third parties. All such copies must contain the title page and this notice page in their entirety.

Acknowledgments

IDL[®] is a registered trademark of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein. Software \equiv Vision[™] is a trademark of Research Systems, Inc.

Numerical Recipes[™] is a trademark of Numerical Recipes Software. Numerical Recipes routines are used by permission.

GRG2[™] is a trademark of Windward Technologies, Inc. The GRG2 software for nonlinear optimization is used by permission.

NCSA Hierarchical Data Format (HDF) Software Library and Utilities
Copyright © 1988-1998 The Board of Trustees of the University of Illinois
All rights reserved.

CDF Library
Copyright © 1999
National Space Science Data Center
NASA/Goddard Space Flight Center

NetCDF Library
Copyright © 1993-1996 University Corporation for Atmospheric Research/Unidata

HDF EOS Library
Copyright © 1996 Hughes and Applied Research Corporation

This software is based in part on the work of the Independent JPEG Group.

This product contains StoneTable[™], by StoneTablet Publishing. All rights to StoneTable[™] and its documentation are retained by StoneTablet Publishing, PO Box 12665, Portland OR 97212-0665. Copyright © 1992-1997 StoneTablet Publishing

WASTE text engine © 1993-1996 Marco Piovaneli

Portions of this software are copyrighted by INTERSOLV, Inc., 1991-1998.

Other trademarks and registered trademarks are the property of the respective trademark holders.



Contents

Chapter 1:	
Overview of New Features in IDL 5.4	7
Visualization Enhancements in IDL	8
Analysis Enhancements in IDL 5.4	17
IDL Language Enhancements	31
LZW/GIF No Longer Supported in IDL	41
File I/O Enhancements	43
Development Environment Enhancements	46
Installation and Licensing Enhancements	54
Application Development Enhancements	57
IDL Wavelet Toolkit Enhancements	59
New and Enhanced IDL Utilities	62
New and Enhanced IDL Objects	66
New and Enhanced IDL Routines	88
New and Updated System Variables	126

Features Obsoleted in IDL 5.4	128
Platforms Supported in this Release	134

Chapter 2:

Date/Time Plotting in IDL 135

Overview	136
How to Generate Date/Time Data	138
Displaying Date/Time Data on an Axis in Direct Graphics	140
Displaying Date/Time Data on an Axis in Object Graphics	148

Chapter 3:

New IDL Routines 157

ARRAY_EQUAL	158
BESELK	159
BREAK	161
COLORMAP_APPLICABLE	162
CONTINUE	163
FILE_CHMOD	164
FILE_DELETE	168
FILE_EXPAND_PATH	170
FILE_MKDIR	172
FILE_TEST	173
FILE_WHICH	177
HOUGH	179
LAGUERRE	187
LEGENDRE	189
MAKE_DLL	192
MAP_2POINTS	201
MATRIX_MULTIPLY	205
MEMORY	207
RADON	210
SAVGOL	219
SOCKET	223
SPHER_HARM	227
SWITCH	230
TIMEGEN	232
WV_CWT	237

WV_DENOISE	239
WV_FN_GAUSSIAN	243
WV_FN_MORLET	246
WV_FN_PAUL	249
XDXF	252
XPCOLOR	256
XPLOT3D	257
XROI	264
XVOLUME	273
Chapter 4:	
New Objects	279
IDLffShape	280
Index	313



Chapter 1: Overview of New Features in IDL 5.4

This chapter contains the following topics:

Visualization Enhancements in IDL	8	IDL Wavelet Toolkit Enhancements	59
Analysis Enhancements in IDL 5.4	17	New and Enhanced IDL Utilities	62
IDL Language Enhancements	31	New and Enhanced IDL Objects	66
LZW/GIF No Longer Supported in IDL . . .	41	New and Enhanced IDL Routines	88
File I/O Enhancements	43	New and Updated System Variables	126
Development Environment Enhancements .	46	Features Obsoleted in IDL 5.4	128
Installation and Licensing Enhancements .	54	Platforms Supported in this Release	134
Application Development Enhancements . .	57		

Visualization Enhancements in IDL

The following enhancements have been made in the area of Visualization in the IDL 5.4 release:

- [New Visualization Utilities](#)
- [Double-Precision Support for Visualization](#)
- [Enhanced Date/Time Support for Plotting in IDL](#)
- [Elimination of Limits on the Number of Contour Levels](#)
- [Improved Preview Functionality for PostScript Files](#)
- [New Printer Support for UNIX Platforms](#)
- [Windows Metafile Format \(WMF\) Support for Direct Graphics](#)
- [New Reverse Axis Plotting Example for Object Graphics](#)
- [Ability to Specify Values in Points for the IDLgrPattern Object](#)

New Visualization Utilities

IDL 5.4 now contains new visualization utilities which can be used as stand-alone applications tools which help you create applications. These utilities can also be embedded within IDL applications that you develop.

For more information, see [“New and Enhanced IDL Utilities”](#) on page 62.

Double-Precision Support for Visualization

IDL routines and objects that can be used for visualization now accept double-precision data without converting it to single-precision. This allows for greater precision and flexibility when visualizing data. For routines that can return an array of data, a keyword has been added to allow you to choose between the default single-precision and an optional double-precision.

The following is a simple example of how double-precision data can now be displayed in IDL plotting:

```
PRO dp_plot_example

DEVICE, RETAIN=2, DECOMPOSED=0
!P.BACKGROUND=255
!P.COLOR=0
```



```

secPerYear = 365.24d*86400
time = [1d-43, 1d-35, 1d-12, 1d-6, 0.01, 1, 15, 180, 210, $
1d6*secPerYear, 1d10*secPerYear]
temperature = [1d32, 1d27, 1d15, 1d13, 1d11, 1d10, 3d9, 1d9, 1d8, $
4000, 2.725]

WINDOW, 0, xsize=400, ysize=300
PLOT, time, temperature, /NOCLIP, $
    PSYM = -2, XSTYLE = 1, YSTYLE = 1, /XLOG, /YLOG, $
    YRANGE = [1,1d33], XTICKS = 6,$
    XTICKV = [1d-43,1d-30,1d-20,1d-10,1,1d10,1d20], $
    XTITLE = 'Time since Big Bang (sec)', $
    YTITLE = 'Temperature (K)', $
    TITLE = 'Temperature of the Universe'
END

```

This results in the following plot:

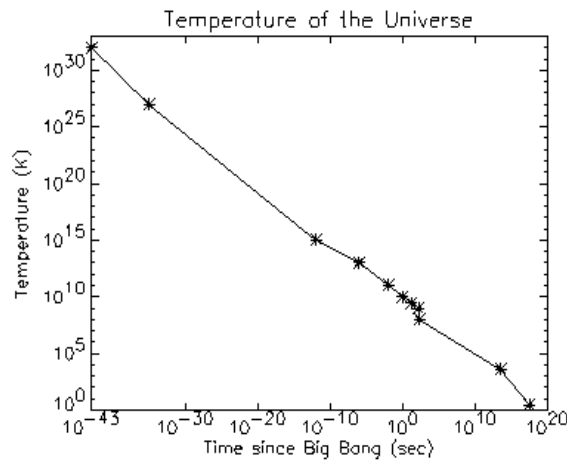


Figure 1-1: Double-Precision Plotting in IDL

IDL System Variables Now Supporting Double Precision

The following IDL system variable fields now support double precision:

- !P.T
- ![XYZ].CRANGE
- ![XYZ].RANGE

- `![XYZ].S`
- `![XYZ].TICKFORMAT`
- `![XYZ].TICKV`

For more information on specific changes to these IDL System Variables, see [“New and Updated System Variables”](#) on page 126.

IDL Routines Now Supporting Double Precision

The following IDL routines now support double precision:

- | | | |
|------------------------------|-----------------------------|-------------------------|
| • <code>AXIS</code> | • <code>LIVE_CONTOUR</code> | • <code>T3D</code> |
| • <code>CONVERT_COORD</code> | • <code>LIVE_PLOT</code> | • <code>TV</code> |
| • <code>CONTOUR</code> | • <code>LIVE_SURFACE</code> | • <code>TVCRS</code> |
| • <code>COORD2TO3</code> | • <code>PLOT</code> | • <code>TVSCL</code> |
| • <code>CREATE_VIEW</code> | • <code>PLOTS</code> | • <code>VERT_T3D</code> |
| • <code>CURSOR</code> | • <code>POLYFILL</code> | • <code>XOBJVIEW</code> |
| • <code>DRAW_ROI</code> | • <code>POLYSHADE</code> | • <code>XYOUTS</code> |
| • <code>ISOCONTOUR</code> | • <code>SURFACE</code> | |
| • <code>OPLOT</code> | • <code>SHADE_SURF</code> | |

For more information on specific changes to these IDL routines, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

IDL Objects Now Supporting Double Precision

The following IDL objects now support double precision:

- | | | |
|------------------------------|------------------------------|-----------------------------|
| • <code>IDLanROI</code> | • <code>IDLgrLight</code> | • <code>IDLgrSurface</code> |
| • <code>IDLanROIGroup</code> | • <code>IDLgrModel</code> | • <code>IDLgrText</code> |
| • <code>IDLgrAxis</code> | • <code>IDLgrPlot</code> | • <code>IDLgrView</code> |
| • <code>IDLgrBuffer</code> | • <code>IDLgrPolygon</code> | • <code>IDLgrVolume</code> |
| • <code>IDLgrColorbar</code> | • <code>IDLgrPolyline</code> | • <code>IDLgrVRML</code> |
| • <code>IDLgrContour</code> | • <code>IDLgrPrinter</code> | • <code>IDLgrWindow</code> |

- IDLgrImage
- IDLgrLegend
- IDLgrROI
- IDLgrROIGroup

For more information on specific changes to IDL Objects, see [“New and Updated Keywords/Arguments to IDL Object Methods”](#) on page 67.

IDL Utilities Now Supporting Double Precision

The following IDL system utility now supports double precision:

- XOBJVIEW

For more information on specific changes to XOBJVIEW, see [“New Keywords/Arguments to Existing IDL Utilities”](#) on page 64.

Enhanced Date/Time Support for Plotting in IDL

IDL routines and objects used for plotting have been enhanced to make it easier to display date/time data along axis. The following figure shows an example of the capabilities of the new date/time support:

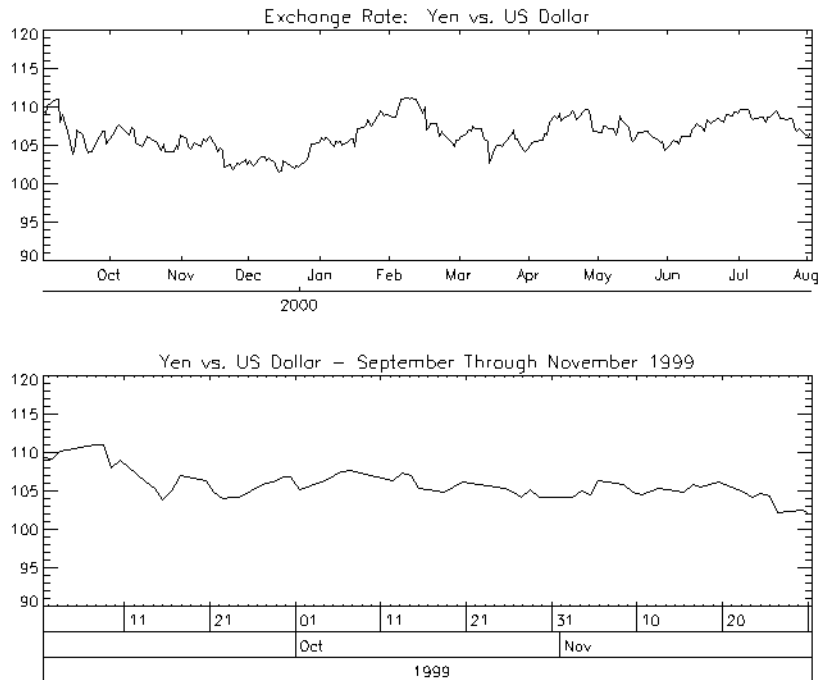


Figure 1-2: New Date/Time Display Along Axis

The enhancements for date/time support for plotting in IDL are:

- A new TIMEGEN array creation routine — The new TIMEGEN function returns an array (with the specified dimensions) of double-precision floating-point values that represent times by Julian dates.
- New date/time keywords for IDL Direct Graphics — Four keywords have been modified or added to Direct Graphics routines to provide improved capabilities for date/time axis labeling. These keywords are: TICKLAYOUT, TICKUNITS, TICKINTERVAL, and TICKFORMAT.

- New date/time keywords for IDL Object Graphics — Four keywords have been modified or added to the IDLgrAxis class to provide improved capabilities for date/time axis labeling. The keywords are: TICKLAYOUT, TICKUNITS, TICKINTERVAL, and TICKFORMAT.
- New Fields for the !X, !Y, and !Z system variables — These new fields provide easier date/time axis labeling capabilities. They are: TICKLAYOUT, TICKUNITS, TICKINTERVAL.
- Improvements to LABEL_DATE — The LABEL_DATE routine now accepts format strings that include codes for sub-seconds. Also, LABEL_DATE will accept an array of DATE_FORMATS and a *Level* argument so that it may be used for multi-level date/time axes.

For more information on date/time support, see [Chapter 2, “Date/Time Plotting in IDL”](#).

Elimination of Limits on the Number of Contour Levels

Previous to this release, the CONTOUR routine has been limited to a fixed number of levels (most recently 60) that can be rendered. This limitation has now been removed. CONTOUR will now accept vectors that contain more than 60 elements for each of the C_* keywords (C_ANNOTATION, C_COLORS, C_LABELS, C_LINestyle, C_ORIENTATION, C_SPACING, and C_THICK) as well as for the LEVELS keyword. The NLEVELS keyword may now be set to a value greater than 60.

Improved Preview Functionality for PostScript Files

In IDL 5.4, you now have the ability to specify the resolution of the preview when creating a PostScript or an Encapsulated PostScript file. You can now specify the width, height, and depth of the preview with the new PRE_XSIZE, PRE_YSIZE, and PRE_DEPTH keywords to the DEVICE routine.

For more information on specific changes to IDL Objects, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

New Printer Support for UNIX Platforms

In IDL 5.4, the Xprinter (UNIX printer support) has been upgraded to version 3.3. This provides the following added functionality:

- Support for the following printer models with the addition of the associated PPD files:

HP LaserJet 4Si MX PS 600 dpi	HP LaserJet 8000
HP LaserJet 5/5M PostScript	Tektronix 560
HP LaserJet 5P/5MP	Lexmark Optra S 2455
HP LaserJet 5Si	Lexmark Optra Color 1200
HP LaserJet 4000	

- Support for advanced features such as 1200 DPI, duplex printing, and multiple paper tray features on printers that provide these capabilities.
- Support for PostScript Level II compression.

Note

For more information on Bristol's XPrinter 3.3 visit the Bristol web site at:
<http://www.bristol.com>

Windows Metafile Format (WMF) Support for Direct Graphics

IDL now supports writing to the Windows Metafile Format (WMF). This format is used by Windows to store vector graphics in order to exchange graphics information between applications. This format is only available on Windows platforms.

To write to this format, you use the SET_PLOT procedure and specify 'METAFILE' as the device. You can then use the DEVICE procedure to modify the attributes of the file. The following DEVICE keywords are supported for Metafile:

CLOSE_FILE	INDEX_COLOR
FILENAME	SET_CHARACTER_SIZE
GET_CURRENT_FONT	SET_FONT
GET_FONTNAMES	TRUE_COLOR

GET_FONTNUM	TT_FONT
GLYPH_CACHE	XSIZE
INCHES	YSIZE

For example, the following will create a WMF file for a simple plot:

```
;Create X and Y Axis data
x=findgen(10)
y=findgen(10)

;Save current device name
mydevice=!D.NAME

;Set the device to Metafile
SET_PLOT, 'METAFILE'

;Name the file to be created
DEVICE, FILE='test.emf'

;Create the plot
PLOT, x, y

;Close the device which creates the Metafile
DEVICE, /CLOSE

;Set the device back to the original
SET_PLOT, mydevice
```

New Reverse Axis Plotting Example for Object Graphics

A new example has been included with IDL to show how to reverse the order of axis tick values using Object Graphics. You can run this example by entering `EX_REVERSE_PLOT.PRO` at the IDL command line. You can view the source for this example, `EX_REVERSE_PLOT.PRO`, in the `examples/visual` directory.

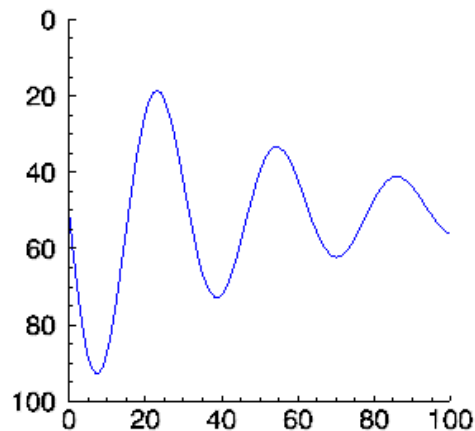


Figure 1-3: Reverse Axis Plotting Example

Ability to Specify Values in Points for the IDLgrPattern Object

Previously, IDLgrPattern has used a pattern description that is defined in terms of pixels units. To facilitate “what you see is what you get” (WYSIWYG) behavior, the units of measure for the PATTERN, SPACING and THICK keywords will now be points (rather than pixels). With this new functionality, it is easy to re-use the same pattern on more than one destination device (even if the destinations have varying resolutions).

Analysis Enhancements in IDL 5.4

The following enhancements have been made in the area of Analysis in the IDL 5.4 release:

- Improved FFT Performance
- New Hough and Radon Transform Functions
- New Legendre Polynomial Functions
- New Laguerre Polynomial Function
- New Savitzky-Golay Smoothing Filter
- New MAP_2POINTS Function
- Enhanced IBETA and IGAMMA Functions
- Enhanced ROBERTS and SOBEL Functions
- Enhancement to Bessel Functions
- Ability to Retrieve the Number of Vertices in IDLanROI
- Enhanced MIN_CURVE_SURF Function
- Enhanced Probability Functions
- Enhanced TRIGRID Function
- Enhanced Integration Functions
- Enhanced FACTORIAL Function
- Enhanced HISTOGRAM Function
- Enhanced Curve-Fitting Functions

Improved FFT Performance

The FFT function now uses an improved algorithm that is more efficient at handling data sets with a length containing powers of 2, 3, or 5. The FFT function in previous versions of IDL only took advantage of data lengths that are powers of 2. The new FFT algorithm extends this advantage to powers of 3 and 5. The new FFT performance is up to three times faster for data sets rich in powers of 2, 3, or 5, depending on the data set size and platform. In addition, the new FFT is more

accurate for primes and powers of 2, and more memory efficient, requiring half the memory for some data sets.

Note

You may notice a negligible difference in results from the FFT function in previous versions of IDL.

New Hough and Radon Transform Functions

The new **HOUGH** and **RADON** functions have been added to IDL. The Hough (P. V. C. Hough, 1962) and Radon (J. Radon, 1917) transforms are used to detect lines within two-dimensional images. The Hough transform maps each image pixel into a sinusoid within the Hough domain, while the Radon transform maps lines within an image into a single pixel within the Radon domain. Both transforms are widely used for image processing, remote sensing, computer vision, and seismic analysis. The Hough transform, in particular, can be used for automatic extraction and classification of features in satellite images. The Radon transform is used in computed tomography (CT) to reconstruct two-dimensional tissue slices from a series of X-ray projections.

The following figure shows an example of the use of the new **HOUGH** function. The top image shows three lines drawn within a random array of pixels that represent noise. The center image shows the Hough transform, displaying sinusoids for points that lie on the same line in the original image. The bottom image shows the Hough backprojection, after setting the threshold to retain only those lines that contain more than 20 points. The Hough inverse transform, or backprojection, transforms each point in the Hough domain into a straight line in the image.

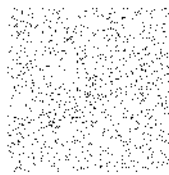
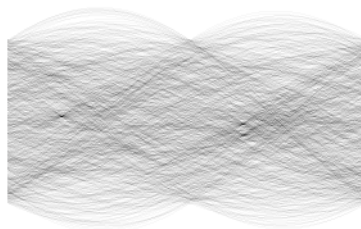
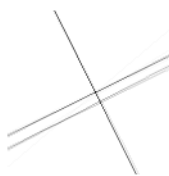
Noise and Lines**Hough Transform****Hough Backprojection**

Figure 1: HOUGH example showing random pixels (top), Hough transform (center) and Hough backprojection (bottom)

The next figure shows an example of the use of the new RADON function. The top image is an image of a ring and random pixels, or noise. The center image is the Radon transform, and displays the line integrals through the image. The bottom image is the Radon backprojection, after filtering all noise except for the two strong horizontal stripes in the middle image.

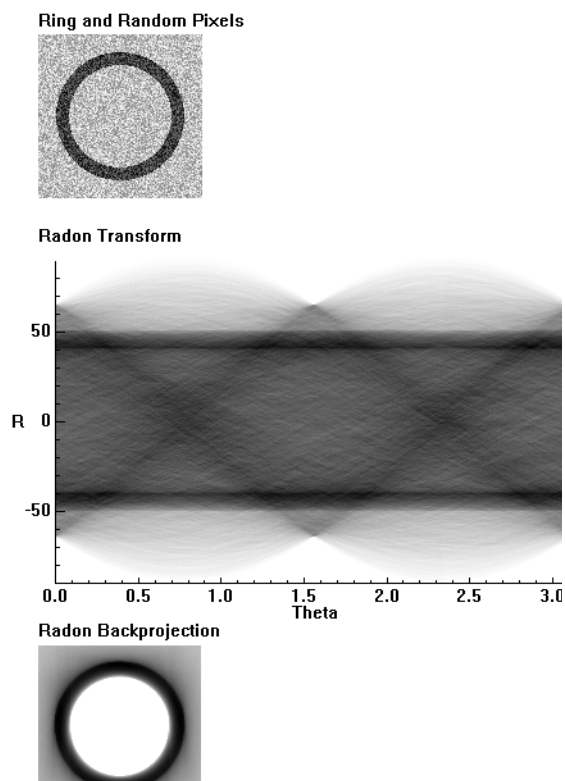


Figure 2: Radon Example - Original image (top), Radon transform (center), and backprojection of the filtered RADON transform (bottom)

New Legendre Polynomial Functions

Two new functions have been added to IDL for evaluation of Legendre polynomials:

Function	Description
LEGENDRE	The LEGENDRE function returns the value of the associated Legendre polynomial.
SPHER_HARM	The SPHER_HARM function returns the value of the spherical harmonic, which is a function of two coordinates on a spherical surface.

Table 1-1: New Legendre Polynomial Functions

New Laguerre Polynomial Function

The [LAGUERRE](#) function has been added to IDL for the evaluation of Laguerre polynomials. Laguerre polynomials are used in quantum mechanics, for example, where the wave function for the hydrogen atom is given by the Laguerre differential equation.

New Savitzky-Golay Smoothing Filter

The new [SAVGOL](#) function returns the coefficients of a Savitzky-Golay smoothing filter, which can then be applied using the CONVOL function. The Savitzky-Golay smoothing filter, also known as least squares or DISPO (digital smoothing polynomial), can be used to smooth a noisy signal.

The following figure illustrates the new SAVGOL function. In this example, we have created a noisy 400-point vector with 4 Gaussian peaks of decreasing width. Then, we plot the vector smoothed with a 33-point boxcar smoother (the SMOOTH function), the vector smoothed with 33-point wide Savitzky-Golay filter of degree 4, and finally the first derivative of the noisy signal and the first derivative using the Savitzky-Golay filter of degree 4. Notice how the Savitzky-Golay filter preserves the high peaks but does not do as much smoothing on the flatter regions, and how the filter is able to construct a good approximation of the first derivative.

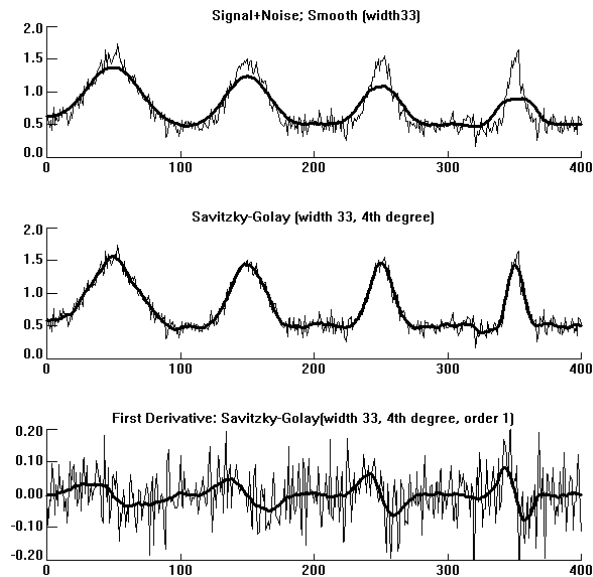


Figure 1-4: New Savitzky-Golay Smoothing Filter

New MAP_2POINTS Function

The new [MAP_2POINTS](#) function has been added to IDL to return parameters such as distance, azimuth, and path relating to the great circle or rhumb line connecting two points on a sphere.

Enhanced IBETA and IGAMMA Functions

The following enhancements have been made to the IBETA and IGAMMA functions:

- The functions now work on arrays as well as scalars (provided the arrays have the same size).
- The functions now fully support double-precision.
- The new EPS and ITMAX keywords allow the user to specify the desired accuracy and number of iterations, respectively.

For more information on specific changes to IBETA and IGAMMA, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

Enhanced ROBERTS and SOBEL Functions

The ROBERTS and SOBEL functions have been enhanced such that the resulting image array has the same dimensions and data type as the original array. In previous versions of IDL, the resulting image array was returned as type INT, regardless of the data type of the original array.

Enhancement to Bessel Functions

The Bessel functions, BESELI, BESELJ, and BESELY, have been modified to accept non-integer orders. Also, the new BESELK function has been added.

Ability to Retrieve the Number of Vertices in IDLanROI

You can now retrieve the number of vertices being used by a region in an IDLanROI object. The N_VERTS keyword, used to represent the number of vertices, has been added to the IDLanROI::GetProperty method to specify a named variable that will contain the number of vertices currently being used by the region.

For more information on specific changes to IDL Objects, see [“New and Updated Keywords/Arguments to IDL Object Methods”](#) on page 67.

Enhanced MIN_CURVE_SURF Function

The SPHERE and CONST keywords have been added to the MIN_CURVE_SURF function to support minimum curvature surface interpolation over a sphere.

For more information on specific changes to MIN_CURVE_SURF, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

Enhanced Probability Functions

The CHISQR_PDF, F_PDF, GAUS_PDF, and T_PDF functions have been enhanced to accept array values for all arguments.

Enhanced TRIGRID Function

The TRIGRID function has been enhanced to allow specification of irregularly spaced rectangular output grids. The new XOUT and YOUT keywords can be set to vectors specifying the output grid X and Y values.

Enhanced Integration Functions

The QROMB, QROMO, and QSIMP functions are now fully re-entrant, and can be called from within the user-supplied integration functions. This allows you to

perform double (or multiple) integration by calling QROMB with a user-supplied IDL function that calls QROMB within itself.

Enhanced FACTORIAL Function

The FACTORIAL function now accepts input as either a scalar or an array. Additionally, the new UL64 keyword has been added so that the results can be returned as unsigned 64-bit integers.

Enhanced HISTOGRAM Function

The NBINS keyword has been added to the HISTOGRAM function to allow the user to explicitly specify the number of bins to use.

Enhanced Curve-Fitting Functions

Several curve-fitting routines have been modified to make them consistent with one another, to correct the value returned by the SIGMA keyword, and to add functionality. The following changes have been made:

LINFIT

- For consistency with other curve-fitting routines, the COVAR and YFIT output keywords have been added.
- For consistency with other curve-fitting routines, the SDEV keyword has been replaced by MEASURE_ERRORS, which has the same definition and meaning as SDEV. For backwards compatibility, the SDEV keyword is still accepted, but new code should use the MEASURE_ERRORS keyword.

LMFIT

- The definition of the SIGMA keyword has changed. If you do not specify error estimates (via the MEASURE_ERRORS keyword), then you are assuming that your user-supplied model (or the default quadratic), is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in SIGMA are multiplied by the correction factor $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X , and M is the number of coefficients. In versions of IDL prior to 5.4, this correction factor was not being applied. For example, the following code yields different results in IDL 5.3 and IDL 5.4:

```
; Define an 11-element vector of independent variable data:
X = DINDGEN(11)
```



```

; Define an 11-element vector of dependent variable data:
Y = 3 - 4*x + 5*x^2 + 0.5*randomn(1,11)

; Compute fit:
A = [0, 0, 0]
result = LMFIT(X, Y, A, SIGMA=sigma)
PRINT, 'Coefficients:      ', A
PRINT, 'Standard errors: ', sigma

```

IDL 5.3 prints incorrect results:

```

Coefficients:      2.8768212      -3.9525263      5.0026831
Standard errors:    0.76185273      0.35445878      0.034139437

```

IDL 5.4 prints the correct results:

```

Coefficients:      2.8768212      -3.9525263      5.0026831
Standard errors:    0.28439646      0.13231799      0.012744110

```

For more information, see section 15.2 of *Numerical Recipes in C* (Second Edition).

- The WEIGHTS keyword is obsolete and has been replaced by the MEASURE_ERRORS keyword. Code that uses the WEIGHTS keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. Note that the definition of the MEASURE_ERRORS keyword is not the same as the WEIGHTS keyword. Using the WEIGHTS keyword, $\text{SQRT}(1/\text{WEIGHTS}[i])$ represents the measurement error for each point $Y[i]$. Using the MEASURE_ERRORS keyword, the measurement error for each point is represented as simply $\text{MEASURE_ERRORS}[i]$. The following code demonstrates the difference between the use of the old WEIGHTS keyword and the new MEASURE_ERRORS keyword:

Assume we have the following data and measurement errors:

```

; Define an 11-element vector of independent variable data:
X = FINDGEN(11)

; Define an 11-element vector of dependent variable data:
Y = 3 - 4*x + 5*x^2

; Assume Gaussian measurement errors for each point:
measure_errors = REPLICATE(0.5, 11)

```

Using the obsolete WEIGHTS keyword, we would compute the fit as follows:

```

A = [0, 0, 0]
result = LMFIT(X, Y, A, WEIGHTS=1/measure_errors^2, SIGMA=sigma)

```

```
PRINT, 'Coefficients:      ', A
PRINT, 'Standard errors: ', sigma
```

Using the new `MEASURE_ERRORS` keyword, we now compute the fit as follows. Note that the same measurement errors are used:

```
A = [0, 0, 0]
result = LMFIT(X, Y, A, MEASURE_ERRORS=measure_errors, $
              SIGMA=sigma)
PRINT, 'Coefficients:      ', A
PRINT, 'Standard errors: ', sigma
```

In both cases, IDL prints:

```
Coefficients:      2.99998      -3.99999      5.00000
Standard errors:    0.380926      0.177229      0.0170697
```

POLY_FIT

- A new `MEASURE_ERRORS` keyword has been added to `POLY_FIT`, replacing the `POLYFITW` function. Note, however, that the definition of the `MEASURE_ERRORS` keyword to `POLY_FIT` is different from the definition of the *Weights* argument to `POLYFITW`. In `POLYFITW`, `SQRT(1/Weights[i])` represented the measurement error for each point $Y[i]$. Now, for consistency with other curve-fitting routines, `POLY_FIT` defines the measurement error for each point as `MEASURE_ERRORS[i]`. Code using `POLYFITW` will continue to work as before, but new code should use `POLY_FIT`. If you wish to convert existing code using `POLYFITW` to use the new `MEASURE_ERRORS` keyword to `POLY_FIT`, you must change the values you supply. For example, assume we have the following data and measurement errors:

```
; Define an 11-element vector of independent variable data:
X = FINDGEN(11)

; Define an 11-element vector of dependent variable data:
Y = 3 - 4*x + 5*x^2

; Assume Gaussian measurement errors of 0.5 for each point:
measure_errors = REPLICATE(0.5, 11)
```

To compute the weighted second degree polynomial fit using `POLYFITW`:

```
weight = 1/measure_errors^2
result = POLYFITW(X, Y, weight, 2)
PRINT, 'Coefficients:      ', result[*]
```

Using the improved `POLY_FIT` function, we can compute the fit as follows:

```

; Compute using the improved POLY_FIT routine:
result = POLY_FIT(X, Y, 2, MEASURE_ERRORS = measure_errors)
PRINT, 'Coefficients:      ', result[*]

```

In both cases, IDL prints:

```
Coefficients:  3.00032      -4.00015      5.00002
```

- The *Yfit* argument to POLY_FIT is now a keyword YFIT. For backwards compatibility, the argument will still be accepted.
- The *Yband* argument to POLY_FIT is now a keyword YBAND. For backwards compatibility, the argument will still be accepted.
- The *Sigma* argument to POLY_FIT is now a keyword YERROR. For backwards compatibility, the argument will still be accepted. Note that the description of the argument *Sigma* incorrectly stated that *Sigma* was the “standard deviation of the returned coefficients.” Actually, *Sigma* (now keyword YERROR) is the standard error between YFIT and Y.
- The *Corrm* argument to POLY_FIT is now a keyword COVAR. For backwards compatibility, the argument will still be accepted. Note that the description of the argument *Corrm* stated that *Corrm* was the “correlation matrix.” Actually, *Corrm* (now keyword COVAR) is the covariance matrix.
- POLY_FIT now returns the fit parameters CHISQ and SIGMA, which give the chi-square goodness-of-fit and the standard deviation of the returned coefficients, respectively.
- A new STATUS keyword has been added to POLY_FIT to allow the function to return a status value that can be used to programmatically determine whether the operation was successful.

POLYFITW

The POLYFITW function is obsolete, and has been replaced by the MEASURE_ERRORS keyword to POLY_FIT.

REGRESS

- For consistency with the other curve-fitting routines, the arguments *Weights*, *Yfit*, *Const*, *Sigma*, *Ftest*, *R*, *Rmul*, *Chisq*, and *Status* are now keywords MEASURE_ERRORS, YFIT, CONST, SIGMA, FTEST, CORRELATION, MCORRELATION, CHISQ, and STATUS, respectively. The arguments are still supported for backward compatibility, but new code should use the keywords instead.

- The definition of the MEASURE_ERRORS keyword is different from the *Weights* argument that it has replaced. Using the *Weights* argument, $\text{SQRT}(1/\text{Weights}[i])$ represents the measurement error for each point $Y[i]$. Now, for consistency with other curve-fitting routines, the measurement error for each point is represented as simply $\text{MEASURE_ERRORS}[i]$. Also, the *RELATIVE_WEIGHT* keyword is no longer necessary. Now, if the *MEASURE_ERRORS* keyword is not provided, then *REGRESS* assumes you want no weighting.

Assume we have the following data:

```
; Create two vectors of independent variable data:
X1 = [1.0, 2.0, 4.0, 8.0, 16.0, 32.0]
X2 = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
; Combine into a 2x6 array
X = [TRANPOSE(X1), TRANPOSE(X2)]

; Create a vector of dependent variable data:
Y = 5 + 3*X1 - 4*X2
```

The following examples illustrate the difference between the old method and the new method, with and without weighting:

No weighting:

```
; Old method:
Weights = REPLICATE(1.0, N_ELEMENTS(Y))
; Compute the fit using multiple linear regression:
result = REGRESS(X, Y, Weights, yfit, const, Sigma, $
    /RELATIVE_WEIGHT)
PRINT, 'Coefficients:      ', result[*]
PRINT, 'Standard errors: ', sigma

; New method. Note that the Weights arguments and
; RELATIVE_WEIGHT keyword are not needed:
result = REGRESS(X, Y, SIGMA=sigma)
PRINT, 'COEFFICIENTS:      ', result[*]
PRINT, 'Standard errors: ', sigma
```

In both cases, IDL prints:

```
Coefficients:      3.00000      -3.99999
Standard errors:    1.38052e-006      8.75298e-006
```

Weighting:

```
; Assume Gaussian measurement errors for each point:
measure_errors = REPLICATE(0.5, N_ELEMENTS(Y))
```

```

; Old method:

Weights = 1/measure_errors^2
; Compute the fit using multiple linear regression:
result = REGRESS(X, Y, Weights, Yfit, Const, Sigma)
PRINT, 'Coefficients:      ', result[*]
PRINT, 'Standard errors: ', Sigma

; New method. Note the change in definition of Weights:

weights = 1/measure_errors
result = REGRESS(X, Y, SIGMA=sigma, $
    MEASURE_ERRORS=measure_errors)
PRINT, 'Coefficients:      ', result[*]
PRINT, 'Standard errors: ', sigma

```

In both cases, IDL prints:

```

Coefficients:      3.00000      -3.99999
Standard errors:    0.0444831    0.282038

```

SVDFIT

- The definition of the SIGMA keyword has changed. If you do not specify error estimates (via the MEASURE_ERRORS keyword), then you are assuming that the polynomial (or your user-supplied model) is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in SIGMA are multiplied by the correction factor $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X , and M is the number of coefficients. In versions of IDL prior to 5.4, this correction factor was not being applied. For example, the following code yields different results in IDL 5.3 and IDL 5.4:

```

; Define an 11-element vector of independent variable data:
X = FINDGEN(11)

; Define an 11-element vector of dependent variable data:
Y = 3 - 4*x + 5*x^2 + 0.5*randomn(1,11)

; Compute the quadratic fit:
result = SVDFIT(X, Y, 3, SIGMA=sigma)
PRINT, 'Coefficients:      ', result
PRINT, 'Standard errors: ', sigma

```

IDL 5.3 prints incorrect results:

```

Coefficients:      2.87686      -3.95254      5.00268
Standard errors:    0.761852    0.354459    0.0341395

```

IDL 5.4 prints the correct results:

```
Coefficients:      2.87680      -3.95253      5.00268
Standard errors:   0.284396      0.132318      0.0127441
```

For more information, see section 15.2 of *Numerical Recipes in C* (Second Edition).

- The WEIGHTS keyword is obsolete and has been replaced by the MEASURE_ERRORS keyword. Code that uses the WEIGHTS keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. Note that the definition of the MEASURE_ERRORS keyword is not the same as the WEIGHTS keyword. Using the WEIGHTS keyword, $1/\text{WEIGHTS}[i]$ represents the measurement error for each point $Y[i]$. Using the MEASURE_ERRORS keyword, the measurement error is represented as simply $\text{MEASURE_ERRORS}[i]$. The following code demonstrates the difference between the use of the old WEIGHTS keyword and the new MEASURE_ERRORS keyword:

Assume we have the following data and measurement errors:

```
; Define an 11-element vector of independent variable data:
X = FINDGEN(11)

; Define an 11-element vector of dependent variable data:
Y = 3 - 4*x + 5*x^2

; Assume Gaussian measurement errors for each point:
measure_errors = REPLICATE(0.5, 11)
```

Using the obsolete WEIGHTS keyword, we would compute the fit as follows:

```
A = [0, 0, 0]
result = SVDFIT(X, Y, A=A, WEIGHTS=1/measure_errors, $
    SIGMA=sigma)
PRINT, 'Coefficients: ', A
PRINT, 'Standard errors: ', sigma
```

Using the new MEASURE_ERRORS keyword, we now compute the fit as follows. Note that the same measurement errors are used:

```
A = [0, 0, 0]
result = SVDFIT(X, Y, A=A, MEASURE_ERRORS=measure_errors, $
    SIGMA=sigma)
PRINT, 'Coefficients: ', A
PRINT, 'Standard errors: ', sigma
```

IDL Language Enhancements

The following enhancements have been made to the IDL language in the IDL 5.4 release:

- Large File Support for Windows Platforms
- New 64-Bit Memory Support
- New Support for Reading Compressed Files with Associated Variables
- New File Handling Routines
- New Date Attributes for Retrieving File Status
- New Support for Converting System Times
- Improvements for Formatted Input Using `READ` and `READF`
- New Function for Testing Equality of Arrays
- New Function for Multiplying Transposed Arrays
- New Program Control Statements
- Enhanced `RESOLVE_ROUTINE` Function
- `CALL_EXTERNAL` Enhancement to Automatically Write and Compile Intermediate Glue Code on the Fly
- Enhanced Ability for Spawning Processes
- New Support for TCP/IP Client Side Sockets
- New AltiVec Support for Macintosh
- Relaxed Rules for Combining Structures
- New C `printf`-Style Quoted String Format Code
- Enhanced `WHERE` Function

Large File Support for Windows Platforms

IDL 5.4 now supports accessing files larger than 2.1 GB on the Windows platform. You now can use the 64-bit integer data type to read and write data from files on the following platforms that support the use of a large file capable file system:

- Windows 95, 98, NT 4.0, 2000 (with NTFS)

- SUN Solaris (Intel and SPARC systems)
- HP-UX
- SGI Irix
- Compaq Tru64 UNIX

When reading and writing to files smaller than 2.1 GB, IDL uses longword integers for file position arguments (e.g. `POINT_LUN`, `FSTAT`) and keywords. When accessing files larger than 2.1 GB, IDL will automatically use signed 64-bit integers in order to be able to properly represent the offset.

New 64-Bit Memory Support

IDL 5.4 now provides 64-bit memory support on some platforms which allows you to create individual variables that exceed 2.1 GB in size. You can freely exchange `.sav` files between 64-bit and 32-bit versions of IDL with the exception that the 32-bit version of IDL cannot restore more than 2.1 GB of data from a `.sav` file due to the 32-bit limitation.

Platforms that Support 64-Bit IDL

The platforms that support 64-bit are:

- Tru64 UNIX on Compaq Alpha hardware
- Sun Solaris 7 and 8 on SPARC (64-bit Ultra hardware)
- Linux on Compaq Alpha hardware

Note the following on 64-bit version of IDL for Sun Solaris:

- The 32-bit version of IDL for Sun Solaris will continue to be supported but as a separate 32-bit build. In IDL 5.4, there are now two versions of IDL for Sun SPARC platforms (a 32-bit version and a 64-bit version). During installation, you have a choice of which versions to install. You can select the 32-bit, 64-bit, or both if needed.
- To run 64-bit IDL on Sun Solaris, you will need an UltraSparc platform (sun4u) running the 64-bit Solaris 7 (or later) operating system kernel. A 32-bit Solaris kernel is also available for the UltraSparc, and is commonly installed on systems with CPUs slower than 200Mz unless you take special action when installing Solaris to cause the 64-bit kernel to run. 64-bit IDL will not run on an UltraSparc using the 32-bit OS kernel. To see which kernel you are running, execute the `isainfo` command at the UNIX prompt:

```
% /bin/isainfo -b
```


If a value of 32 is returned, you are running a 32-bit kernel. If a value of 64 is returned, you are running a 64-bit kernel. If `isainfo` is not present on your machine, you are probably running Solaris 2.6, which is 32-bit.

- The 32-bit version of IDL will run correctly on a 64-bit version of Solaris. You do not have to install the 64-bit version of IDL unless you require the ability to access more than 2.1 GB of memory.
- If you have both the 32-bit and 64-bit versions of IDL installed on Solaris, the `idl` and `idlde` commands will start the 64-bit version. If you wish to start the 32-bit version, you can use the `-32` option to the `idl` or `idlde` commands. If you have only the 32-bit version of IDL installed, the `idl` and `idlde` commands will start the 32-bit version.
- The DXF and Dataminer extensions to IDL are not available with the 64-bit Sun Solaris version of IDL. If you need access to this functionality, you should install and use the 32-bit version of IDL (along with the 64-bit version) to access the DXF and Dataminer extension.

For more information on the Solaris operating system, see Sun Microsystem's web page at docs.sun.com as well as the installation instructions included with your Sun Solaris media.

New Support for Reading Compressed Files with Associated Variables

You can now read compressed files that have been associated to a variable using the `ASSOC` function. In previous releases of IDL, you could not associate variables to a file that was opened using the `OPEN` procedure with the `COMPRESS` keyword.

Note

Associated file variables cannot be used for output with files opened using the `COMPRESS` keyword to `OPEN`.

New File Handling Routines

There are six new routines in IDL 5.4 that enhance IDL's ability to perform file handling operations. These functions are:

New Routine	Description
FILE_CHMOD	Changes the access permissions for a file.
FILE_DELETE	Deletes files and empty directories.
FILE_EXPAND_PATH	Returns the full path to a file.
FILE_MKDIR	Creates a directory.
FILE_TEST	Returns whether or not a file exists and attributes about that file.
FILE_WHICH	Searches for a file in a directory path you specify.

Table 1-2: New File Handling Routines in IDL 5.4

New Date Attributes for Retrieving File Status

The FSTAT function in IDL 5.4 now has the ability to return the following information about file status:

- Creation date of the file
- Last date the file was accessed
- Last date the file was modified

The values returned are in seconds since 1 January 1970 UTC.

For descriptions of the enhancements to FSTAT, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

New Support for Converting System Times

The SYSTIME function has been enhanced to:

- Format an input argument giving the number of seconds past January 1, 1970 as a string that represents the date in the format:

DOW MON DD HH:MM:SS YEAR

where DOW is the day of the week, MON is the month, DD is the day of the month, HH is the hour, MM is the minute, SS is the second, and YEAR is the year.

- Output the date string in Universal Time Coordinated (UTC) rather than being adjusted for the current time zone.

For descriptions of the new argument and keyword to SYSTIME, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

Improvements for Formatted Input Using READ and READF

The READ and READF routines now understand all three possible stream file line termination conventions on all platforms:

- Macintosh — CR
- UNIX — LF
- Windows/DOS — CR/LF

IDL running on any operating system can transparently read from files using any of these conventions.

New Function for Testing Equality of Arrays

The ARRAY_EQUAL function is a fast way to compare data for equality in situations where the index of the elements that differ are not of interest. This operation is much faster than using TOTAL(A NE B), because it stops the comparison as soon as the first inequality is found, an intermediate array is not created, and only one pass is made through the data. For best speed, ensure that the types of the operands are the same.

New Function for Multiplying Transposed Arrays

The new MATRIX_MULTIPLY function has been added to IDL to provide a more efficient means of multiplying transposed arrays. The MATRIX_MULTIPLY function calculates the IDL matrix-multiply operator (#) of two (possibly transposed) arrays. The transpose operation (if desired) is done simultaneously with the multiplication, thus conserving memory and increasing the speed of the operation.

New Program Control Statements

Three new program control statements have been added in IDL 5.4:

Statement	Description
BREAK	The BREAK statement provides a convenient way to immediately exit from a loop (FOR, WHILE, REPEAT), CASE, or SWITCH statement without resorting to GOTO statements.
CONTINUE	The CONTINUE statement provides a convenient way to immediately start the next iteration of the enclosing FOR, WHILE, or REPEAT loop.
SWITCH	The SWITCH statement is used to select one statement for execution from multiple choices, depending upon the value of the expression following the word SWITCH. This statement is similar to the CASE statement. Whereas CASE executes at most one statement within the CASE block, SWITCH executes the first matching statement and any following statements in the SWITCH block.

Table 1-3: New Program Control Statements

For more information on IDL's program control statements, see [Chapter 11](#), “Program Control” in *Building IDL Applications*.

Enhanced RESOLVE_ROUTINE Function

The new COMPILE_FULL_FILE keyword has been added to the RESOLVE_ROUTINE function. When compiling a file to find the routine specified using the *Name* argument, IDL normally stops compiling when the desired routine is found. Set COMPILE_FULL_FILE to cause the entire file to be compiled regardless of *Name* being encountered before the end of the file.

CALL_EXTERNAL Enhancement to Automatically Write and Compile Intermediate Glue Code on the Fly

The new CALL_EXTERNAL AUTO_GLUE keyword causes CALL_EXTERNAL to write the intermediate glue code (previously written by IDL users) that converts the IDL calling convention to the argument list actually needed by the target function.

It then uses the new `MAKE_DLL` procedure to compile and link the glue code. You simply need to make the `CALL_EXTERNAL` call and all of the intermediate glue code is created and used quietly behind the scenes on your behalf. The result is the ability to call arbitrary functions from a sharable library from IDL without writing a line of C or worrying about the details of compiling and linking sharable libraries on Windows, UNIX, and VMS platforms.

Enhanced Ability for Spawning Processes

The `SPAWN` procedure for UNIX platforms has been enhanced in the following ways:

- You can now capture the exit status from a child process started by IDL.
- You can now capture `stderr` from a child process started by IDL.

The `SPAWN` procedure for Windows platforms has been enhanced in the following ways:

- You can now capture the exit status from a child process started by IDL.
- You can now capture errors from a child process started by IDL.
- You can hide (minimize) the command interpreter window.
- You can run Win32 executables without using the command interpreter window.
- You can capture the output from `SPAWN` and display it in the IDL Development Environment Log window instead of having it go to the command interpreter window.
- You can execute processes and specify not to wait.
- You can spawn a subprocess and have the IDL process continue executing in parallel with the spawned subprocess.

For descriptions of the new arguments and keywords to `SPAWN`, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

New Support for TCP/IP Client Side Sockets

The new `SOCKET` procedure, supported on UNIX and Windows platforms, opens a client side TCP/IP Internet socket as an IDL file unit. Such files can be used in the standard manner with any of IDL’s input/output routines.

New AltiVec Support for Macintosh

With the introduction of the G4 series of PowerPC processors, a new vector unit has been introduced to the architecture. Called “AltiVec” by Motorola and “The Velocity Engine” by Apple, this unit provides a new set of vector operations which greatly enhance the performance of processing certain kinds of operations.

IDL 5.4 has added AltiVec support for array operations of the following types:

- Byte
- Integer
- Complex
- Unsigned Integer
- Long
- Unsigned Long
- Floating-Point

Other operations that have been accelerated:

- Addition
- Subtraction
- Multiplication
- Division

These accelerated operators apply to *array* <operator> *array* calculations as well as *array* <operator> *scalar* and *scalar* <operator> *array* calculations.

Relaxed Rules for Combining Structures

IDL 5.4 now allows concatenation and assignment of structures with different but compatible definitions. For example, consider the following structures:

```
s1 = { moose1, a:fltarr(10, 10), b:23 }
s2 = { moose2, x:fltarr(100), z:intarr(1) }
```

These statements are different, but they are compatible in terms of actual memory layout. In IDL 5.4, the following statements are now allowed which would have produced errors in previous releases:

```
s3 = [s1, s2]
s3[1] = s2
```

This eliminates a problem commonly encountered with anonymous structures. For example:

```
s1 = REPLICATE({ a:fltarr(10,10), b:23 }, 10)
s1[4] = { a:FINDGEN(10, 10), b:79 }
```

In previous versions, the two anonymous structures in these statements are considered to yield different types and therefore would not have been allowed. In IDL 5.4, these statements are recognized to be of different types, but still compatible and the statements are allowed.

New C printf-Style Quoted String Format Code

IDL's explicitly formatted specifications, which are based on those found in the FORTRAN language, are extremely powerful and capable of specifying almost any desired output. However, they require fairly verbose specifications, even in simple cases. In contrast, the C language (and the many languages influenced by C) have a different style of format specification used by functions such as `printf()` and `sprintf()`. Most programmers are very familiar with such formats. In this style, text and format codes (prefixed by a % character) are intermixed in a single string. User-supplied arguments are substituted into the format in place of the format specifiers. Although less powerful, this style of format is easier to read and write in common simple cases.

IDL now supports the use of `printf`-style formats within format specifications, using a special variant of the Quoted String Format Code in which the opening quote starts with a % character (e.g. `%"` or `%'` rather than `"` or `'`). The presence of this % before the opening quote (with no white space between them) tells IDL that this is a `printf`-style quoted string and not a standard quoted string.

As a simple example, consider the following IDL statement that uses normal quoted string format codes:

```
PRINT, FORMAT='("I have ", I0, " monkeys, ", A, ".")', $
      23, 'Scott'
```

Executing this statement yields the output:

```
I have 23 monkeys, Scott.
```

Using a `printf`-style quoted string format code instead, this statement could be written:

```
PRINT, FORMAT='(%"I have %d monkeys, %s.")', 23, 'Scott'
```

These above statements are completely equivalent in their action.

For more information on `printf`-style formats, see [“C printf-Style Quoted String Format Code”](#) in Chapter 8 of *Building IDL Applications*.

Enhanced WHERE Function

The `COMPLEMENT` and `NCOMPLEMENT` keywords have been added to the `WHERE` function to return the subscripts and number of zero elements in the input array. For descriptions of the new keywords to `WHERE`, see [“WHERE”](#) on page 120.

LZW/GIF No Longer Supported in IDL

Research Systems is no longer able to support reading and writing GIF (Graphics Interchange Format) images or LZW (Lempel-Zif-Welch) compression for TIFF images. The LZW technology has been patented by the Unisys Corporation. Note that any users of GIF/LZW technology are required to enter into a license agreement with Unisys Corporation.

The following are the related changes to IDL:

- **READ_GIF, WRITE_GIF, and QUERY_GIF** — These routines no longer are able to read, write, and query GIF files. If you use these routines in any IDL application, when executing, IDL will produce an error message and execution will halt.

As an alternative to GIF, you can use the Portable Network Graphics (PNG) format. This allows you to easily search and replace many of your calls to `READ_GIF`, `WRITE_GIF`, and `QUERY_GIF` with `READ_PNG`, `WRITE_PNG`, and `QUERY_PNG`. If you are currently using any GIF files in your IDL applications, you will need to convert them to PNG.

The PNG format is a new standard intended to replace GIF as a dominant network format. PNG handles 8-bit and 24-bit images and uses a lossless compression scheme to compress images. For more information, see [READ_PNG](#), [WRITE_PNG](#), and [QUERY_PNG](#) in the *IDL Reference Guide*.

`READ_PNG`, already an IDL function, has been enhanced in this release so that you can now call it as a procedure allowing it to be easily used as a replacement for `READ_GIF`. Note that the `READ_PNG` function is still a supported IDL routine.

- **READ_IMAGE, WRITE_IMAGE, and QUERY_IMAGE** — These routines no longer support the ability to access GIF files.
- **DIALOG_READ_IMAGE and DIALOG_WRITE_IMAGE** — These routines no longer support the ability to access GIF files.
- **ANNOTATE** — This routine no longer supports the ability to save an annotated image as a GIF file.
- **READ_TIFF and WRITE_TIFF** — These routines no longer support the ability to read and write TIFF files with LZW compression. If you have created TIFF files in previous releases of IDL that use LZW compression, you will no longer be able to access those files using `READ_TIFF`. If you set the

COMPRESSION keyword to WRITE_TIFF to a value of 1 (previously this created a TIFF file using LZW compression), the resulting TIFF file will be created using the PackBits compression.

- **LIVE_EXPORT** — This routine no longer supports the ability to export a TIFF file with LZW compression. If you set the COMPRESSION keyword to a value of 1 (previously this created a TIFF file using LZW compression), the resulting TIFF file will be created using the PackBits compression.
- **IDL Insight** — The Insight application has been removed from IDL. Insight uses LZW compression for saving compressed project files and therefore can no longer be included with IDL.

Research Systems apologizes for any inconvenience this may cause. For more information on this topic and information on existing GIF conversion utilities, visit www.ResearchSystems.com/IDL.

File I/O Enhancements

The following file I/O enhancements have been made in the IDL 5.4 release:

- [New Support for ESRI Shapefiles](#)
- [Improved Performance with the READ_ASCII Function](#)
- [Library Updates](#)
- [Enhanced READ_PNG and WRITE_PNG Functions](#)
- [Enhancements to the Quality of MPEG Movies](#)
- [Windows Input/Output Behavior Improved](#)

New Support for ESRI Shapefiles

IDL now provides support for Shapefiles through the use of a Shapefile Object, IDLffShape. This object encapsulates all functionality that is required to access a Shapefile.

For more information on the new IDLffShape class, see [Chapter 4, “New Objects”](#).

Improved Performance with the READ_ASCII Function

The performance of READ_ASCII has been significantly improved when reading large ASCII files.

Library Updates

The following libraries have been updated in the IDL 5.4 release:

Library	New Version	Previous Version
DXF	2.003	1.010
PNG	1.0.5	.89c
ZLIB	1.1.3	1.04

Table 1-4: Updated Libraries in IDL 5.4

Enhanced READ_PNG and WRITE_PNG Functions

READ_PNG and WRITE_PNG have been changed to read and write PNG files in top-to-bottom order. PNG files should now have the correct orientation when transferred from other applications to IDL. An ORDER keyword has been added to provide compatibility with PNG files written using earlier versions of IDL.

Enhancements to the Quality of MPEG Movies

In IDL 5.4, new keywords have been added to help control the level of compression and motion prediction used when creating MPEG movies. Now you can weigh the final quality/file size versus the amount of time needed to create an MPEG movie.

Note

MPEG support in IDL requires a special license. For more information, contact your Research Systems sales representative or technical support.

For more information, see MPEG_OPEN and XINTERANIMATE in [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91 and IDLgrMPEG in [“New and Updated Keywords/Arguments to IDL Object Methods”](#) on page 67.

Windows Input/Output Behavior Improved

In order to make read/write operations under Windows work correctly the way they do under Unix, read/write operations under Windows are handled differently in IDL 5.4 than in earlier versions of IDL. In some cases, this may require changes to existing IDL code. Before we look at an example in which code would need to be updated, the following is a brief background of this issue.

Under Microsoft Windows, a file is read or written as an uninterrupted stream of bytes—there is no record structure at the operating system level. Lines in a Windows text file are terminated by the character sequence CR LF (carriage return, line feed).

The Microsoft C runtime library considers a file to be in either binary or text mode, and its behavior differs depending on the current mode of the file. The programmer confusion caused by this distinction is a cause of many C/C++ program bugs. IDL is not affected by this quirk of the C runtime library, and no special action is required to work around it. Read/write operations are handled the same in Windows as in Unix: when IDL performs a formatted I/O operation, it reads/writes the CR/LF line termination. When it performs a binary operation, it simply reads/writes raw data.

Versions of IDL prior to IDL 5.4 (5.3 and earlier), however, were affected by the text/binary distinction made by the C library. The BINARY and NOAUTOMODE

keywords to the OPEN procedures were provided to allow the user to change IDL's default behavior during read/write operations. In IDL 5.4 and later versions, these keywords are no longer necessary. They continue to be accepted in order to allow older code to compile and run without modification, but they are completely ignored and can be safely removed from code that does not need to run on those older versions of IDL.

Some rare Windows-specific code that contains special workarounds using `BINARY` or `NOAUTOMODE` may require modification or removal in order for the program to work correctly under IDL 5.4. Once modified, such code will be simpler, and portable to other operating systems. For example, assume you used the following code in the Windows version of IDL 5.3 to write a Unix-format text file:

```
OPENW, 1, 'unix.txt', /BINARY, /NOAUTOMODE
PRINTF, 1, 'A line of text'
```

The above code would create a text file that looks like this:

```
A line of textLF
```

where LF is a linefeed character. If you were to run the above code in IDL 5.4, the resulting text file would look like this:

```
A line of textCRLF
```

where CR is the carriage return character and LF is the linefeed character. Using IDL 5.4, this code should be updated as follows:

```
OPENW, 1, 'unix.txt'
WRITEU, 1, 'A line of text' + STRING(10B)
```

This code creates the following text file in both IDL 5.3 and IDL 5.4:

```
A line of textLF
```

Development Environment Enhancements

The following enhancements have been made to the IDL Development Environment in the IDL 5.4 release:

- [Improved IDL Projects](#)
- [Importing IDL Preferences & Macros from Previous Releases](#)
- [New Preferences for Windows Always on Top for the IDLDE](#)
- [New Error Window for Macintosh](#)
- [New Editor Window on Macintosh](#)
- [Running With Breakpoints in the Macintosh Editor Window](#)
- [Improved General Preferences Dialog Box](#)

Improved IDL Projects

IDL Projects, introduced in IDL 5.3, allow you to easily develop applications in IDL. You can manage, compile, run, and create distributions of all the files you will need to develop your IDL application. All of your application files can be organized so that they are easier to access and easier to export to other developers, colleagues, or users. IDL Projects are a great benefit to development teams working on a large project as well as individual developers managing multiple projects.

Many improvements have been made to IDL Projects in IDL 5.4 to allow for better project management as well as for cross-platform compatibility. The following improvements have been made to IDL Projects in this release:

- You can now create and name your own groups for storing files as well as create your own filters for specifying which types of files are to be stored in a group. To change your groups, select **Project** → **Groups...** The Project Groups dialog is displayed:

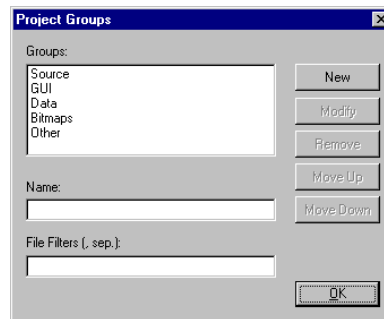


Figure 1-5: New Project Groups Dialog

- You can now select multiple files for editing, deleting, compiling, and setting of attributes by pressing the Ctrl and Shift keys when selecting files.
- Projects will now store all breakpoint information for .pro files. When you have created breakpoints for a file and then save the project, the next time you open that file through your project, the breakpoint information will be restored.
- Only files that can be compiled are shown in the Build dialog.
- A new tool bar for the IDL Project window has been added on Windows and UNIX platforms that was previously available on the Macintosh platform. This toolbar allows quick access for frequently used tasks.



Displays the Project Options dialog for setting or modifying the current project's options.



Displays the Add/Remove Files dialog for adding or removing files from the current project.



Compiles all the files in the current project.



Builds the current project.



Runs the current project.



Displays the File Properties dialog for setting or modifying the current file's properties.

Importing IDL Preferences & Macros from Previous Releases

IDL for Windows now has new support for importing preferences and user-defined macros from a previous release of IDL.

The first time you start IDL for Windows after installing, you will be prompted for whether or not you want to import preferences or user-defined macros from a previous release of IDL.

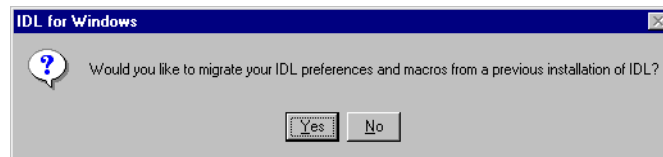


Figure 1-6: Importing IDL Preferences from Previous Releases

Note

It is not necessary to explicitly import macros from previous releases of IDL on UNIX, VMS, or Macintosh platforms. IDL preferences and macros are automatically imported on these platforms.

If you do not want to import preferences and user-defined macros, select **No** and IDL will start. If you want to import from a previous release, select **Yes**. The new **Import IDL Preferences** dialog displays.

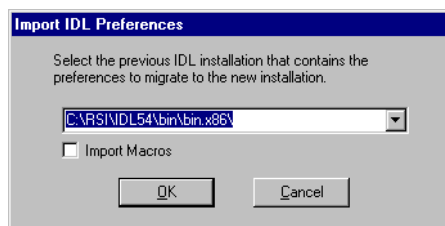


Figure 1-7: Import IDL Preferences Dialog

This dialog displays the paths to the previous IDL installations on your machine in the drop-down list box. Select the path to the previous release of IDL from which to

import preferences and user-defined macros. If you want to import any user-defined macros from this installation as well, select the **Import Macros** check box. Click **OK**. The preferences and user-defined macros are imported and then IDL will start.

Note

If you chose not to import user-defined macros or if you want to import macros from several previous installations, you can select **Macros**→**Import**, and select the previous release from which to import the macros.

New Preferences for Windows Always on Top for the IDLDE

A new preference has been added on the Windows platform so that you can specify whether graphic windows you create will remain on top of the IDL Development Environment window or if they can be hidden behind the IDL Development Environment window when it is in focus. IDL graphics windows that are affected by this preference are those created through IDL Direct Graphics (i.e. WINDOW, PLOT, SURFACE, CONTOUR procedures) or in IDL Object Graphics through the creation of an IDLgrWindow object.

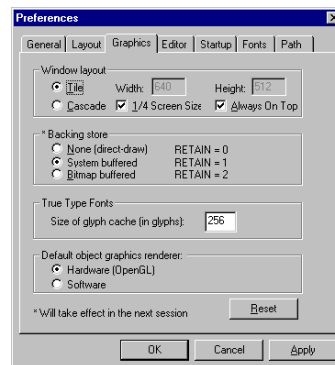


Figure 1-8: New “Always On Top” Preference

New Error Window for Macintosh

A new window has been added on the Macintosh platform for displaying compilation errors. During compilation, the Error Window will display all the errors encountered.

Clicking on the error displays that line of the program that contains the error in the IDL Editor window.

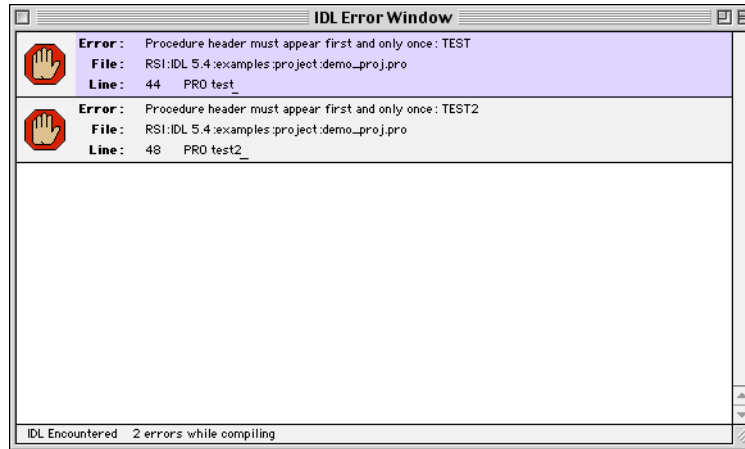


Figure 1-9: New Error Window for Macintosh

New Editor Window on Macintosh

The Editor window has been improved and now has a button bar with a path box and an icon which indicates whether the file is writable or read-only. The buttons on the button bar from left to right are **Save**, **Print**, **Compile**, and **Run** and are provided for ease of access during editing. The path box is simply an informational box and is not editable. The icon at the right side of the button bar shows a pencil to indicate the file is writable, or a lock to indicate the file is read-only.

Multiple Panes in the Editor Window

You can create multiple editing panes within the Editor window enabling you to edit multiple sections of the program without having to scroll back and forth. To open a second pane, click on the button at the top of the vertical scroll bar on the right and drag it until a second horizontal base is seen. When you release the mouse button a second pane with the same program appears. More than two panes are possible in an Editor window, as long as each pane exceeds the minimum size necessary.

The Breakpoint Column

On the left side of each pane in the Editor window is a border used to display break points, flag compiler errors, and the current executing line of code. Rows with tick

marks indicate program lines with executable IDL statements. You can set and unset breakpoints on these lines by clicking on the tick mark or breakpoint. Click on the tick mark to set the breakpoint, and click on the disabled breakpoint to display the tick mark again.

The Line Box

The line number button box at the bottom left of an IDL Editor window displays the line number of the insertion point in the active pane. To relocate the cursor on another line in the same pane, click in the box and specify the line number in the **Go To Line** field of the new dialog box. Clicking the line number box is a shortcut for the **Go To Line** option from the **Search** menu for the active pane in the Editor window.

Function Drop Down List

The button with parentheses and a down arrow to the right of the Line box brings up a drop down list containing the functions and procedures defined in the current `.pro` file. Choosing a function or procedure from the list moves the cursor to that function or procedure definition in the active pane of the Editor window.

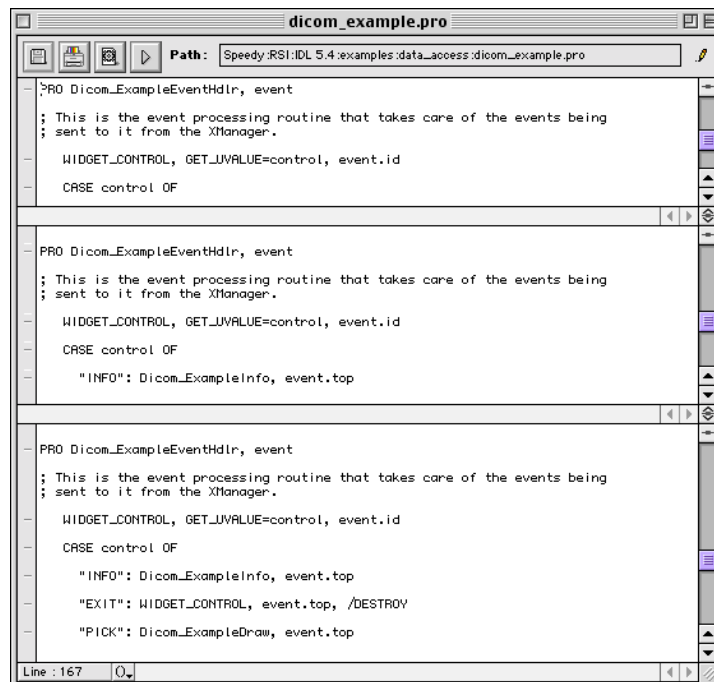


Figure 1-10: The IDL Editor Window

Running With Breakpoints in the Macintosh Editor Window

When you set breakpoints in a .pro file and compile and run the program, the Editor window buttons change to allow you to step through the program using the breakpoints. The four buttons at the top of the window become step buttons (see the following figure) which call the various executive commands for stepping through a program: at the left is **Step Out** which calls .OUT, next is **Step Over** which calls .STEPOVER, then **Step In** which calls .STEP, and the fourth button is **Continue** which calls .CONTINUE.

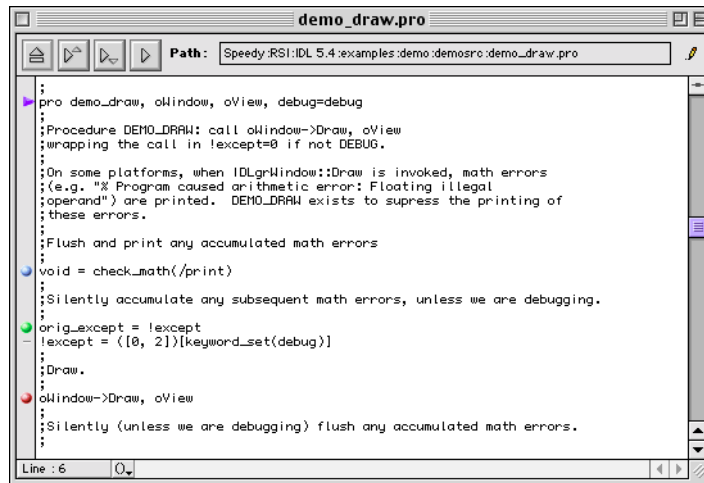


Figure 1-11: Running with Breakpoints in the Editor Window

Improved General Preferences Dialog Box

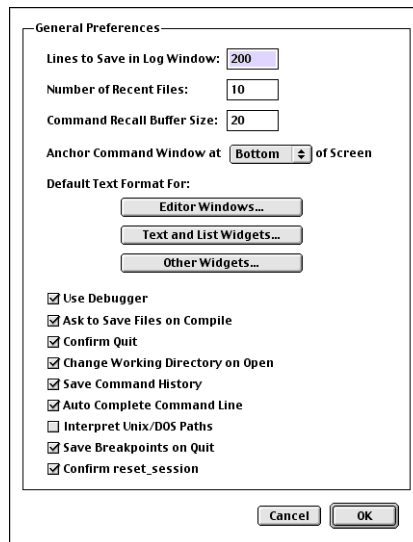


Figure 1-12: The General Preferences Dialog

These preferences control the general appearance and behavior of IDL. The following two selections have been added to the **General Preferences** dialog box.

Auto Complete Command Line

If checked this option enables IDL to compare commands as they are being typed at the command line prompt against the commands in the recall buffer. IDL auto completes the command when a unique match to a previous command is found. This is particularly useful for commands used quite often, such as **Print**.

Save Breakpoints on Quit

If selected, all IDL program breakpoints are saved from session to session until this check box is deselected. When this option is deselected, all saved breakpoints are cleared upon exiting IDL.

Installation and Licensing Enhancements

The following enhancements have been made to IDL Installation and Licensing in the IDL 5.4 release:

- [New Licensing Wizard](#)
- [Improved Floating License Management Utilities](#)
- [New QUEUE Startup Command Line Option](#)

New Licensing Wizard

The new licensing wizard for Windows and UNIX platforms has been designed with user convenience in mind. The wizard allows you to easily create and send a license request. The new request process automatically recognizes all installed Research Systems software products so that you need only a single license file to run any Research Systems product. When your license file arrives, the wizard automatically recognizes where you have installed IDL and guides you to save the license file in the appropriate location.

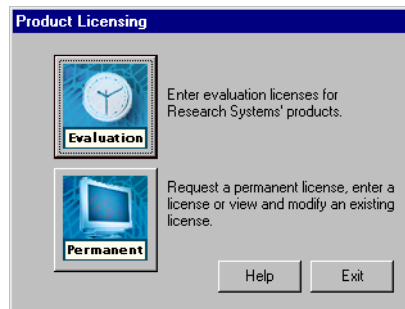


Figure 1-13: New Licensing Wizard

For more information, see your Installation Guide.

Improved Floating License Management Utilities

This version of IDL offers the FLEXlm License Manager Control Panel, a simple graphical interface that easily allows you to configure your license manager service for floating licenses. Under the Setup tab, you can browse to select the appropriate files and with a click of the mouse, arrange to have the license manager automatically

started any time your server is booted. Other features of this utility allow you to start or stop the license manager and to complete simple diagnostics with a click of the mouse.

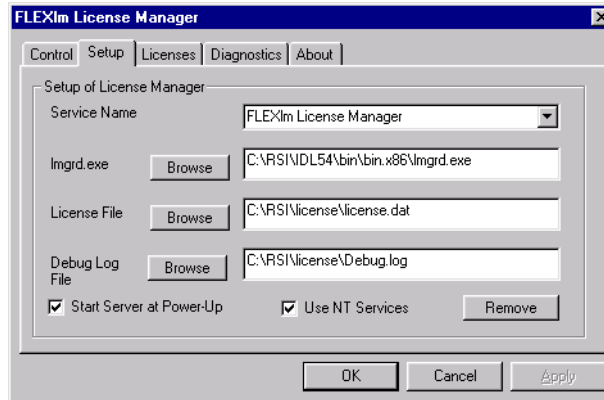


Figure 1-14: New FLEXlm License Manager Control Panel

New QUEUE Startup Command Line Option

Improved IDL functionality now allows users of counted floating licenses to choose to start IDL with the new **QUEUE** argument, assuring that a license will be issued for IDL before beginning an IDL task such as a batch processing job. Using this method of starting IDL when a counted license is unavailable, IDL will not issue a prompt for licensing, but will continue to wait until one becomes available. Using the new **QUEUE** startup switch is especially useful during batch processing since previously, the IDL command log window message, asking if you wanted to wait for an available license, was unavailable for viewing. To start IDL with the new **QUEUE** option, use one of the following methods:

Platform	Method
UNIX	Enter the following at the UNIX prompt: <code>idl -queue</code>

Table 1-5: **QUEUE** Startup Command Line Option

Platform	Method
VMS	Enter the following at the VMS prompt: IDL /QUEUE
Windows	Change the shortcut properties of the desktop IDL 5.4 icon so the target line reads: C:\RSI-Directory\bin\bin.x86\idlde.exe -queue where <i>RSI-Directory</i> is the directory where you have installed IDL.

Table 1-5: QUEUE Startup Command Line Option

Application Development Enhancements

The following enhancements have been made in the IDL 5.4 release.

Modifications to the `DIALOG_PICKFILE` Function

The `FILTER` keyword to `DIALOG_PICKFILE` has been enhanced to allow you to specify an IDL variable containing a string value or an array of string values for filtering the files in the file list. This keyword is used to reduce the number of files to choose from. If the value contains a vector of strings, multiple filters are used to filter the files.

For example, you may want to include a filter so that only files of type `.jpg`, `.tif`, or `.png` show in the file selection window. To accomplish this, you would use:

```
file = DIALOG_PICKFILE(/READ, $  
    FILTER = ['*.jpg', '*.tif', '*.png'])
```

This would result in the following dialog:

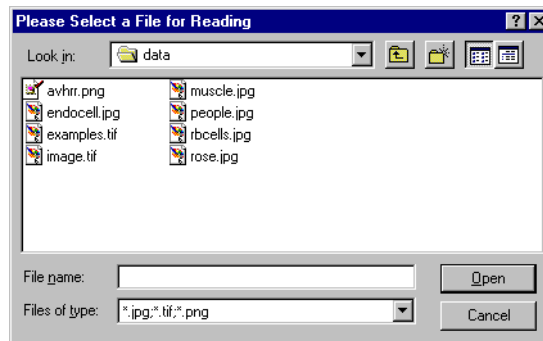


Figure 1-15: Example of `DIALOG_PICKFILE` Filter

Note

On UNIX, the `FILTER` keyword does not support specifying more than one filter. If you specify more than one filter, all files in the current directory will be displayed.

Additional Support for Calling Online Help from Your Application

Two keywords have been added to the `ONLINE_HELP` procedure to allow you to:

- Access a Windows HTML Help file with the new `HTML_HELP` keyword.
- Display the Contents window of the Help system with the new `TOPICS` keyword.

For more information on specific changes to `ONLINE_HELP`, see [“New and Updated Keywords/Arguments to IDL Routines”](#) on page 91.

IDL Wavelet Toolkit Enhancements

The IDL Wavelet Toolkit version 1.1 offers enhanced functionality and new features.

- Enhanced data input for ASCII data files, especially for “time-series” data
- Implementation of the continuous wavelet transform
- New Morlet and Paul wavelet functions for use with the continuous wavelet transform
- Improved visualization for the 3D wavelet power spectrum

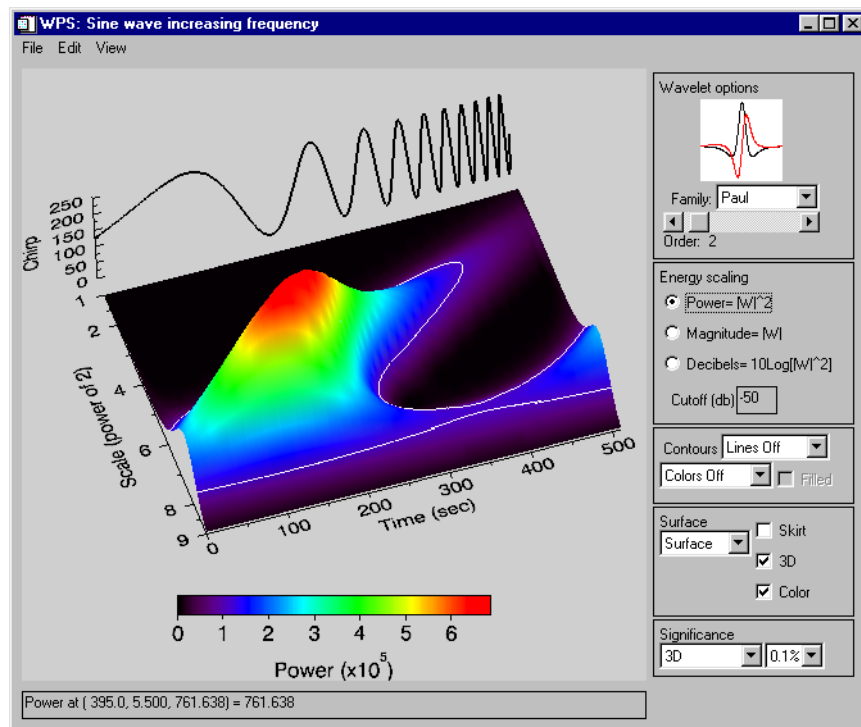


Figure 1-16: The 3D wavelet power spectrum with the Paul continuous wavelet.

New Functions

The following table describes the new functions for the IDL Wavelet Toolkit. These functions are accessible either from the Wavelet Toolkit applet, or directly from the IDL command line.

Command	Description
WV_CWT	Returns the one-dimensional continuous wavelet transform of the input array. The transform is done using a user-inputted wavelet function.
WV_DENOISE	Uses the wavelet transform to filter (or de-noise) a multi-dimensional array.
WV_FN_GAUSSIAN	Constructs wavelet coefficients for the Gaussian wavelet function.
WV_FN_MORLET	Constructs wavelet coefficients for the Morlet wavelet function.
WV_FN_PAUL	Constructs wavelet coefficients for the Paul wavelet function.

Table 1-6: New Wavelet Toolkit functions

New and Updated Keywords/Arguments

The following is a list of new and updated keywords and arguments to existing routines in the IDL Wavelet Toolkit.

[WV_APPLET](#)

Keyword/Argument	Description
Input	This argument can now either be a string representing a save file to open, or an array of data.

WV_CW_WAVELET

Keyword/Argument	Description
DISCRETE	Set this keyword to include only discrete wavelets in the list of wavelet functions. Set this keyword to zero to include only continuous wavelets. The default is to include all available wavelets.
NO_COLOR	If this keyword is set, the wavelet functions will be drawn in black and white.
NO_DRAW_WINDOW	If this keyword is set, the draw window will not be included within the widget.
VALUE	Set this keyword to an anonymous structure of the form {FAMILY: ' ', ORDER: 0d} representing the initial value for the widget.

WV_PLOT3D_WPS

Keyword/Argument	Description
Input	This argument can now either be a string representing the file to open, or an array of data.
SURFACE_STYLE	Set this keyword to an integer specifying the initial style to use for the three-dimensional surface. Valid values are: <ul style="list-style-type: none"> • 0 = Off • 1 = Points • 2 = Mesh • 3 = Shaded • 4 = XZ lines • 5 = YZ lines • 6 = Lego • 7 = Lego fill

New and Enhanced IDL Utilities

IDL 5.4 now contains utilities that can be used in several ways:

- As stand-alone applications
- As tools for helping you create applications
- Embedded within IDL applications that you develop

All of these utilities are located in the `lib/utilities` directory and have been added to your path at install time. Some of these utilities existed in previous versions of IDL but have been improved.

These utilities may be updated in subsequent IDL releases to take advantage of new features and technologies.

New and Existing IDL Utilities

The following table lists the IDL utilities. Note that utilities that existed in previous versions have been listed here since they have moved within the directory structure.

Utility	Description
XBM_EDIT	The XBM_EDIT utility allows you to create and edit icons for use with IDL widgets as bitmap labels for widget buttons. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XDISPLAYFILE	The XDISPLAYFILE utility displays an ASCII text file using a widget interface. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XDXF	The XDXF procedure is a utility for displaying and interactively manipulating DXF objects.
XFONT	The XFONT utility creates a modal widget for selecting and viewing an X Windows font. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.

Table 1-7: New and Existing IDL Utilities

Utility	Description
XINTERANIMATE	The XINTERANIMATE procedure displays an animated sequence of images using off-screen pixmaps or memory buffers. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XLOADCT	The XLOADCT utility displays the current color table and allows you to select a different color table to be loaded from a predefined color table list. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XMTOOL	The XMTOOL utility displays a list of widgets currently being managed by the XMANAGER. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XOBJVIEW	The XOBJVIEW utility can be used to quickly and easily view and manipulate IDL Object Graphics on screen. It displays given objects in an IDL widget with toolbar buttons and menus providing functionality for manipulating, printing, and exporting the resulting graphic. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XPALETTE	The XPALETTE utility allows you to create and modify color tables using the RGB, CMY, HSV, or HLS color systems. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XPCOLOR	The XPCOLOR procedure allows you to adjust the value of the current foreground plotting color, <code>!P.COLOR</code> . The new plotting foreground color is saved in the <code>COLORS</code> common block and loaded to the display.
XPLOT3D	The XPLOT3D utility is used to create and interactively manipulate 3D plots.

Table 1-7: New and Existing IDL Utilities

Utility	Description
XROI	The new XROI procedure is a utility for interactively defining and obtaining information about regions of interest. Freehand and polygon ROIs can be drawn, and information such as minimum, maximum, and mean pixel values and histogram plots can be displayed.
XSURFACE	The XSURFACE utility can be used to quickly and easily view surface plots. Different controls are provided to change the viewing angle and other plot parameters. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XVAREEDIT	The XVAREEDIT utility allows you to edit any IDL variable. This utility was in the <code>lib</code> directory in previous releases. It is now located in the <code>lib/utilities</code> directory.
XVOLUME	The new XVOLUME procedure is a utility for viewing and interactively manipulating volumes and isosurfaces.

Table 1-7: New and Existing IDL Utilities

New Keywords/Arguments to Existing IDL Utilities

The following is a list of the new keywords to existing IDL utilities:

[XOBJVIEW](#)

Keyword/Argument	Description
BACKGROUND	Set this keyword to a three-element [r, g, b] color vector specifying the background color of the XOBJVIEW window.
DOUBLE_VIEW	Set this keyword to force XOBJVIEW to set the DOUBLE property on the IDLgrView that it uses to display graphical data.

Keyword/Argument	Description
REFRESH	Set this keyword to the widget ID of the XOBJVIEW instance to be refreshed. To retrieve the widget ID of an instance of XOBJVIEW, first call XOBJVIEW with the TLB keyword. To refresh that instance of XOBJVIEW, call XOBJVIEW again and set REFRESH to the value retrieved by the TLB keyword in the earlier call to XOBJVIEW.
TLB	Set this keyword to a named variable that upon return will contain the widget ID of the top level base.

New and Enhanced IDL Objects

This section describes the following:

- [New Object Classes](#)
- [New Object Methods](#)
- [New and Updated Keywords/Arguments to IDL Object Methods](#)

New Object Classes

The following table describes the new object classes in IDL 5.4:

Object Class	Description
IDLffShape	An object that contains geometry, connectivity and attributes for graphics primitives.

New Object Methods

New and existing IDL Object Graphics classes have been updated to include the following new methods:

New Methods	Description
IDLffShape::AddAttribute	This method adds an attribute to a Shapefile.
IDLffShape::Cleanup	This method performs all cleanup on the Shapefile object.
IDLffShape::Close	This method closes a Shapefile.
IDLffShape::DestroyEntity	This method destroys the specified entities of a Shapefile.
IDLffShape::GetAttribute	This method retrieves the attributes for the entities you specify.
IDLffShape::GetEntity	This method returns an array of Shapefile entity structures.
IDLffShape::GetProperty	This method returns the values of properties associated with the Shapefile object.

New Methods	Description
IDLffShape::Init	This method initializes or constructs a Shapefile object.
IDLffShape::Open	This method opens a specified shapefile.
IDLffShape::PutEntity	This method inserts an entity into the Shapefile object.
IDLffShape::SetAttributes	This method modifies the attributes for a specified entity in a Shapefile object.

New and Updated Keywords/Arguments to IDL Object Methods

The following table describes the new and updated keywords/arguments to IDL objects.

IDLanROI::AppendData

Keyword/Argument	Description
X, Y, Z	The values specified with these arguments are now maintained as double-precision if in the Init or SetProperty method the input data was of type DOUBLE or if the DOUBLE keyword was set. Otherwise, the values are maintained as single-precision.
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLanROI::ComputeGeometry

Keyword/Argument	Description
AREA	The value returned in the variable you specify with this keyword are now double-precision.
CENTROID	The value returned in the variable you specify with this keyword are now double-precision.
PERIMETER	The value returned in the variable you specify with this keyword are now double-precision.

Keyword/Argument	Description
SPATIAL_SCALE	The value for this keyword may now be double-precision and will no longer be converted to single-precision.

IDLanROI::ComputeMask

Keyword/Argument	Description
LOCATION	The value for this keyword may now be double-precision and will no longer be converted to single-precision.

IDLanROI::GetProperty

Keyword/Argument	Description
N_VERTS	Set this keyword to a named variable that will contain the number of vertices currently being used by the region.
ROI_XRANGE, ROI_YRANGE, ROI_ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLanROI::Init

Keyword/Argument	Description
X, Y, Z	The values specified with these arguments are now maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, the values are stored as single-precision.
DATA	The value for this property is now stored as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, it is stored as single-precision.
DOUBLE	Set this keyword to indicate that data provided by any of the X, Y, or Z arguments or the DATA keyword will be stored in this object as double-precision floating-point.

IDLanROI::RemoveData

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLanROI::ReplaceData

Keyword/Argument	Description
X, Y, Z	The values specified with these arguments are now maintained as double-precision if in the Init or SetProperty method, the inputted data was of type DOUBLE or if the DOUBLE keyword was set. Otherwise, the values are maintained as single-precision.
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLanROI::Rotate

Keyword/Argument	Description
CENTER	The values for this keyword may now be double-precision and will no longer be converted to single-precision.

IDLanROI::Scale

Keyword/Argument	Description
Sx, Sy, Sz	The values for these arguments may now be double-precision and will no longer be converted to single-precision.

IDLanROI::Translate

Keyword/Argument	Description
Tx, Ty, Tz	The values for this argument may now be double-precision and will no longer be converted to single-precision.

IDLanROIGroup::ComputeMesh

Keyword/Argument	Description
VERTICES	The values returned in this argument are double-precision if the DOUBLE keyword was set in the IDLanROI::Init method for any ROI in the group. Otherwise, the values returned are single-precision.
SURFACE_AREA	The value returned in the variable you specify with this keyword are now double-precision.

IDLanROIGroup::GetProperty

Keyword/Argument	Description
ROIGROUP_XRANGE, ROIGROUP_YRANGE, ROIGROUP_ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLanROIGroup::Rotate

Keyword/Argument	Description
CENTER	The values for this keyword may now be double-precision and will no longer be converted to single-precision.

IDLanROIGroup::Scale

Keyword/Argument	Description
Sx, Sy, Sz	The values for these arguments may now be double-precision and will no longer be converted to single-precision.

IDLanROIGroup::Translate

Keyword/Argument	Description
Tx, Ty, Tz	The values for this argument may now be double-precision and will no longer be converted to single-precision.

IDLgrAxis::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrAxis::GetProperty

Keyword/Argument	Description
CRANGE	The values returned in the variable specified with this keyword are double-precision.
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with this keyword are double-precision.

IDLgrAxis::Init

Keyword/Argument	Description
AM_PM	Set this keyword to an array of 2 strings to be used for the names of the AM and PM strings when processing explicitly formatted dates (CAPA, CapA, and CapA format codes) with the TICKFORMAT keyword.
DAYS_OF_WEEK	Set this keyword to an array of 7 strings to be used for the names of the days of the week when processing explicitly formatted dates (CDWA, CDwA, and CdwA format codes) with the TICKFORMAT keyword.
LOCATION	The value for this property is now stored as double-precision.

Keyword/Argument	Description
MONTHS	Set this keyword to an array of 12 strings to be used for the names of the months of the year when processing explicitly formatted dates (CMOA, CMoA, and CmoA format codes) with the TICKFORMAT keyword.
RANGE	The value for this property is now stored as double-precision.
TICKFORMAT	<ol style="list-style-type: none"> 1. This keyword may now be set to either a single string or an array of strings. Each string corresponds to a level of the axis. 2. If any of the strings is the name of a callback function, the third argument to that function (that is the argument indicating the value of the tickmark) will be double-precision (rather than single-precision). 3. If any of the strings is the name of a callback function and if the TICKUNITS keyword is set to one or more non-empty strings, the callback function will be called with four parameters: <i>Axis</i>, <i>Index</i>, <i>Value</i> and <i>Level</i>, where: <ul style="list-style-type: none"> • <i>Axis</i>, <i>Index</i>, and <i>Value</i> are the same as before. • <i>Level</i> is the <i>Index</i> of the axis level for the current tick value to be labelled (<i>Level</i> indices start at 0).
TICKINTERVAL	Set this keyword to a scalar indicating the interval between major tick marks for the first axis level. The default value is computed according to the axis RANGE and the number of major tick marks (MAJOR). This keyword takes precedence over MAJOR.

Keyword/Argument	Description
TICKLAYOUT	<p>Set this keyword to a scalar that indicates the style to draw each level of the axis. Valid values are:</p> <ul style="list-style-type: none"> • 0 — The axis line, major tick marks, and tick labels are drawn. • 1 — Only the labels for the major tick marks are drawn. • 2 — Each major tick interval is outlined by a box.
TICKUNITS	<p>Set this keyword to a string (or vector of strings) to indicate the units to be used for axis tick labeling. Valid values are:</p> <ul style="list-style-type: none"> • “Numeric” (the default) • “Year” • “Month” • “Day” • “Hour” • “Minute” • “Second” • “Time” — Use this value to indicate that the units are generic time units. IDL will compute appropriate default intervals and tick formats based on the range of values covered by the axis. <p>You can specify more than one type of unit. The axis levels will be drawn in the order in which you specify the strings, with the first unit being drawn nearest to the primary axis line.</p>
TICKLEN	The value for this property is now stored as double-precision.
TICKVALUES	The value for this property is now stored as double-precision.
XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The value for these properties is now stored as double-precision.

IDLgrBuffer::GetTextDimensions

Keyword/Argument	Description
DESCENT	The values returned in the variable you specify with this keyword are now double-precision. Note that the return value of this method is now double-precision.

IDLgrBuffer::PickData

Keyword/Argument	Description
XYZLOCATION	The values returned in this variable are now double-precision.

IDLgrClipboard::GetTextDimensions

Keyword/Argument	Description
DESCENT	The values returned in the variable you specify with this keyword are now double-precision. Note that the return value of this method is now double-precision.

IDLgrColorbar::ComputeDimensions

Keyword/Argument	Description
n/a	This method now returns the dimensions of a colorbar object as double-precision values.

IDLgrColorbar::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLgrColorbar::Init

Keyword/Argument	Description
TICKLEN	The value for this property is now stored as double-precision.

Keyword/Argument	Description
TICKVALUES	The value for this property is now stored as double-precision.
XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The value for these properties is now stored as double-precision.

IDLgrContour::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrContour::GetProperty

Keyword/Argument	Description
GEOM	The values returned in the variable you specify with this keyword are now either single or double-precision, depending upon the precision used to store the geometry.
XRANGE, YRANGE, Z RANGE	The values returned in the variable you specify with this keyword are now double-precision.

IDLgrContour::Init

Keyword/Argument	Description
Values	The values specified with this argument are now maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE_DATA keyword is set. Otherwise, the data is maintained as single-precision.
ANISOTROPY	The values for this property are now stored as double-precision.
C_VALUE	The values for this property are now stored as double-precision.

Keyword/Argument	Description
DATA_VALUES	The values stored in this property are maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE_DATA keyword is set. Otherwise, the values are maintained as single-precision.
DOUBLE_DATA	Set this keyword to indicate that the object is to store data provided by either the <i>Values</i> argument or the DATA_VALUES keyword parameter in double-precision floating point. Otherwise, the data is stored in single-precision floating point. IDL converts any value data already stored in the object to the requested precision, if necessary.
DOUBLE_GEOM	Set this keyword to indicate that the object is to store data provided by any of the GEOMX, GEOMY, or GEOMZ keyword parameters in double-precision floating-point. Otherwise, the data is stored in single-precision floating point. IDL converts any geometry data already stored in the object to the requested precision, if necessary.
GEOMX, GEOMY, GEOMZ	The values stored in these properties are maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE_DATA keyword is set. Otherwise, the values are maintained as single-precision.
MAX_VALUE, MIN_VALUE, TICKINTERVAL, TICKLEN, XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The values for these properties are now stored as double-precision.

IDLgrImage::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrImage::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLgrImage::Init

Keyword/Argument	Description
DIMENSIONS, LOCATION, XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The values for these properties are now stored as double-precision.

IDLgrLegend::ComputeDimensions

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrLegend::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The value for these properties is now stored as double-precision.

IDLgrLegend::Init

Keyword/Argument	Description
ITEM_THICK	This keyword now accepts floating-point values.
XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The value for these properties is now stored as double-precision.

IDLgrLight::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrLight::Init

Keyword/Argument	Description
LOCATION, XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The values for these properties are now stored as double-precision.

IDLgrModel::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrModel::Init

Keyword/Argument	Description
TRANSFORM	The value for this property is now stored as double-precision.

IDLgrModel::Rotate

Keyword/Argument	Description
Axis, Angle	This method now accepts these arguments as double-precision values without converting them to single-precision. The resulting transformation matrix is stored in the object in double-precision.

IDLgrModel::Scale

Keyword/Argument	Description
Axis, Angle	This method now accepts these arguments as double-precision values without converting them to single-precision. The resulting transformation matrix is stored in the object in double-precision.

IDLgrModel::Translate

Keyword/Argument	Description
Axis, Angle	This method now accepts these arguments as double-precision values without converting them to single-precision. The resulting transformation matrix is stored in the object in double-precision.

IDLgrMPEG::Init

Keyword/Argument	Description
BITRATE	<p>Set this keyword to a double-precision value to specify the MPEG movie bit rate. Higher bit rates will create higher quality MPEGs but will increase file size. The following table describes the valid values:</p> <ul style="list-style-type: none"> • MPEG 1 — 0.1 to 104857200.0 • MPEG 2 — 0.1 to 429496729200.0 <p>If you do not set this keyword, IDL computes the BITRATE value based upon the value you have specified for the QUALITY keyword. The value of BITRATE returned by IDLgrMPEG::GetProperty is either the value computed by IDL from the QUALITY value or the last non-zero valid value stored in this property.</p> <p>Note - Only use the BITRATE keyword if changing the QUALITY keyword value does not produce the desired results. It is highly recommended to set the BITRATE to at least several times the frame rate to avoid unusable MPEG files or file generation errors.</p>

Keyword/Argument	Description
IFRAME_GAP	<p>Set this keyword to a positive integer value that specifies the number of frames between I frames to be created in the MPEG file. I frames are full-quality image frames that may have a number of predicted or interpolated frames between them.</p> <p>If you do not specify this keyword, IDL computes the IFRAME_GAP value based upon the value you have specified for the QUALITY keyword. The value of IFRAME_GAP returned by IDLgrMPEG::GetProperty is either the value computed by IDL from the QUALITY value or the last non-zero valid value stored in this property.</p> <p>Note - Only use the IFRAME_GAP keyword if changing the QUALITY keyword value does not produce the desired results.</p>
MOTION_VEC_LENGTH	<p>Set this keyword to an integer value specifying the length of the motion vectors to be used to generate predictive frames. The following table describes the valid values:</p> <ul style="list-style-type: none"> • 1 — Small motion vectors. • 2 — Medium motion vectors. • 3 — Large motion vectors. <p>If you do not set this keyword, IDL computes the MOTION_VEC_LENGTH value based upon the value you have specified for the QUALITY keyword. The value of MOTION_VEC_LENGTH returned by IDLgrMPEG::GetProperty is either the value computed by IDL from the QUALITY value or the last non-zero valid value stored in this property.</p> <p>Note - Only use the MOTION_VEC_LENGTH keyword if changing the QUALITY value does not produce the desired results.</p>

Keyword/Argument	Description
QUALITY	<p>Set this keyword to an integer value between 0 (low quality) and 100 (high quality) inclusive to specify the quality at which the MPEG stream is to be stored. Higher quality values result in lower rates of time compression and less motion prediction which provide higher quality MPEGs but with substantially larger file size. Lower quality factors may result in longer MPEG generation times. The default is 50.</p> <p>Note - Since MPEG uses JPEG (lossy) compression, the original picture quality can't be reproduced even when setting QUALITY to its' highest setting.</p>

IDLgrPattern::Init

Keyword/Argument	Description
PATTERN, SPACING	These values are now specified in points rather than pixels.

IDLgrPlot::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrPlot::GetProperty

Keyword/Argument	Description
ZRANGE	The value for this property is now stored as double-precision.

IDLgrPlot::Init

Keyword/Argument	Description
X, Y	The values specified with these arguments are now maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, they are stored as single-precision.

Keyword/Argument	Description
DATA _X , DATA _Y	The value for these properties is now stored as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, it is stored as single-precision.
DOUBLE	Set this keyword to indicate that data provided by any of the X or Y arguments or DATA _X or DATA _Y keywords will be stored in this object as double-precision floating-point.
MAX_VALUE, MIN_VALUE, XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV, ZVALUE	The value for these properties is now stored as double-precision.

IDLgrPolygon::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrPolygon::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The value for these properties is now stored as double-precision.

IDLgrPolygon::Init

Keyword/Argument	Description
X, Y, Z	The value for these arguments is now stored as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, they are stored as single-precision.

Keyword/Argument	Description
DATA	The value for this property is now stored as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, it is stored as single-precision.
DOUBLE	Set this keyword to indicate that data provided by any of the X, Y, or Z arguments or DATA keyword will be stored in this object as double-precision floating-point.
XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The value for these properties is now stored as double-precision.

IDLgrPolyline::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrPolyline::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The value for these properties is now stored as double-precision.

IDLgrPolyline::Init

Keyword/Argument	Description
X, Y, Z	The values specified with these arguments are now maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, it is stored as single-precision.
DATA	The value for this property is now stored as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, it is stored as single-precision.

Keyword/Argument	Description
DOUBLE	Set this keyword to indicate that data provided by any of the X, Y, or Z arguments or DATA keyword will be stored in this object as double-precision floating-point.
XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The value for these properties is now stored as double-precision.

IDLgrPrinter::GetTextDimensions

Keyword/Argument	Description
DESCENT	The values returned in the variable you specify with this keyword are now double-precision. Note that the return value for this method is now double-precision.

IDLgrROI::Init

Keyword/Argument	Description
X, Y, Z	The values specified with these arguments are now maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, they are stored as single-precision.
DOUBLE	Set this keyword to indicate that data provided by any of the X, Y, or Z arguments or DATA keyword (inherited) will be stored in this object as double-precision floating-point.
XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The value for these properties is now stored as double-precision.

IDLgrSurface::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrSurface::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE Z RANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLgrSurface::Init

Keyword/Argument	Description
X, Y, Z	The values specified with these arguments are now maintained as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, it is stored as single-precision.
DATA X, DATA Y, DATA Z	The value for these properties is now stored as double-precision if the input data is of type DOUBLE or if the DOUBLE keyword is set. Otherwise, it is stored as single-precision.
DOUBLE	Set this keyword to indicate that data provided by any of the X, Y, or Z arguments or DATA X, DATA Y, or DATA Z keywords will be stored in this object as double-precision floating-point.
MAX_VALUE, MIN_VALUE, SKIRT, XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The value for these properties is now stored as double-precision.

IDLgrSymbol::Init

Keyword/Argument	Description
SIZE	The values for this property are nw stored as double-precision.

IDLgrText::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision data.

IDLgrText::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLgrText::Init

Keyword/Argument	Description
CHAR_DIMENSIONS, LOCATIONS, XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The values for these properties are now stored as double-precision.

IDLgrView::Init

Keyword/Argument	Description
DOUBLE	If set, IDL calculates the transformations used for the modeling and view transforms using double-precision floating point arithmetic.
EYE, VIEWPLANE, ZCLIP	The values for these properties are now stored as double-precision.

IDLgrVolume::GetCTM

Keyword/Argument	Description
n/a	This method now returns double-precision values.

IDLgrVolume::GetProperty

Keyword/Argument	Description
XRANGE, YRANGE, ZRANGE	The values returned in the variable specified with these keywords are now double-precision.

IDLgrVolume::Init

Keyword/Argument	Description
XCOORD_CONV, YCOORD_CONV, ZCOORD_CONV	The values returned in the variable specified with these keywords are now double-precision.

IDLgrVRML::GetTextDimensions

Keyword/Argument	Description
DESCENT	The values returned in the variable you specify with this keyword are now double-precision.

IDLgrWindow::GetTextDimensions

Keyword/Argument	Description
DESCENT	The values returned in the variable you specify with this keyword are now double-precision.

IDLgrWindow::Pickdata

Keyword/Argument	Description
XLOCATION, YLOCATION, ZLOCATION	The values returned in the variable specified with these keywords are now double-precision.

New and Enhanced IDL Routines

This section describes the following:

- [New IDL Routines](#)
- [New and Updated Keywords/Arguments to IDL Routines](#)
- [Updated Common Graphics Keywords](#)

New IDL Routines

The following is a list of new functions, procedures, statements, and executive commands added to IDL.

Routine	Description
ARRAY_EQUAL	This function provides a fast way to compare data for equality in situations where the index of the elements that differ are not of interest. For best speed, ensure that the operands are the same data type.
BESELK	This function returns the K Bessel function of order N for the argument X .
BREAK	This statement immediately exits from a loop (FOR, WHILE, REPEAT), CASE, or SWITCH statement.
COLORMAP_APPLICABLE	This function determines whether the current visual class supports the use of a color map, and if so, whether color map changes affect pre-displayed Direct Graphics or if the graphics must be redrawn to pick up color map changes. Note that this routine was included in previous releases of IDL but was undocumented.
CONTINUE	This statement immediately starts the next iteration of the enclosing FOR, WHILE, or REPEAT loop.

Routine	Description
FILE_CHMOD	This procedure allows the user to change the current access permissions (modes) associated with a file or directory.
FILE_DELETE	This procedure deletes a file or empty directory, if the process has the necessary permissions to remove the file as defined by the current operating system. FILE_CHMOD can be used to change file protection settings.
FILE_EXPAND_PATH	This function expands a given file or partial directory name to its fully qualified name.
FILE_MKDIR	This procedure creates a new directory, or directories, with the default access permissions for the current process. If a specified directory has non-existent parent directories, FILE_MKDIR automatically creates all the intermediate directories as well.
FILE_TEST	This function checks files for existence and other attributes without first having to open the file.
FILE_WHICH	This function separates a specified file path into its component directories, and searches each directory in turn for a specific file. This command is modeled after the UNIX <code>which(1)</code> command.
HOUGH	This function returns the Hough transform of a two-dimensional image.
LAGUERRE	This function returns the value of the associated Laguerre polynomial.
LEGENDRE	This function returns the value of the associated Legendre polynomial.
MAKE_DLL	This procedure builds a sharable library from C language code which is suitable for use by the dynamic linking features in IDL (CALL_EXTERNAL , LINKIMAGE , and the dynamically linkable modules (DLMs)).

Routine	Description
MAP_2POINTS	This function returns parameters such as distance, azimuth, and path relating to the great circle or rhumb line connecting two points on a sphere.
MATRIX_MULTIPLY	This function calculates the IDL matrix-multiply operator (#) of two (possibly transposed) arrays. This is more efficient than # in some situations.
MEMORY	This function returns information on the amount of dynamic memory currently in use by the IDL session.
RADON	This function returns the Radon transform of a two-dimensional image.
SAVGOL	This function returns the coefficients of a Savitzky-Golay smoothing filter.
SOCKET	This procedure, supported on UNIX or Microsoft Windows platforms, opens a client side TCP/IP Internet socket as an IDL file unit. Such files can be used in the standard manner with any Input/Output routines in IDL.
SPHER_HARM	This function returns the value of the spherical harmonic function.
SWITCH	This statement selects one statement for execution from multiple choices, depending upon the value of an expression. This statement is similar to the CASE statement. Whereas CASE executes at most one statement within the CASE block, SWITCH executes the first matching statement and any following statements in the SWITCH block.
TIMEGEN	This function returns an array (with the specified dimensions) of double-precision floating-point values that represent Julian date/times.

Routine	Description
READ_PNG	This procedure has been added to ease the conversion for the removal of the READ_GIF procedure from IDL. This new procedure has the same functionality as the READ_PNG function. When converting from the READ_GIF procedure to the READ_PNG procedure, note that READ_PNG accepts the same arguments as READ_GIF but does not accept the CLOSE and MULTIPLE keywords.

New and Updated Keywords/Arguments to IDL Routines

The following is a list of new and updated keywords and arguments to existing IDL routines.

ASSOC

Keyword/Argument	Description
n/a	You can now use ASSOC to read data from compressed files.

AXIS

Keyword/Argument	Description
X, Y, Z	AXIS now accepts the X, Y, and/or Z arguments as double-precision floating point values without converting them to single-precision.

BESELI, BESELJ, BESELK, BESELY

Keyword/Argument	Description
N	This argument can now be either an integer or a real number.

BINOMIAL

Keyword/Argument	Description
P	This argument can now be either a scalar or an array.
DOUBLE	Set this keyword to force the computation to be done in double-precision arithmetic.
GAUSSIAN	Set this keyword to use the Gaussian approximation, by using the normalized variable $Z = (V - NP)/\text{SQRT}(NP(1 - P))$.

BLK_CON

Keyword/Argument	Description
DOUBLE	Set this keyword to force the computation to be done in double precision.

CALL_EXTERNAL

Keyword/Argument	Description
AUTO_GLUE	Set this keyword to enable the CALL_EXTERNAL Auto Glue feature. Use of AUTO_GLUE implies the PORTABLE keyword.
CC	If present, a template string to be used in generating the C compiler command(s) to compile the automatically generated glue function. For a more complete description of this keyword, see MAKE_DLL .
COMPILE_DIRECTORY	Specifies the directory to use for creating the necessary intermediate files and the final glue function sharable library. For a more complete description of this keyword, see MAKE_DLL .
EXTRA_CFLAGS	If present, a string supplying extra options to the command used to execute the C compiler. For a more complete description of this keyword, see MAKE_DLL .

Keyword/Argument	Description
EXTRA_LFLAGS	If present, a string supplying extra options to the command used to execute the linker. For a more complete description of this keyword, see MAKE_DLL .
IGNORE_EXISTING_GLUE	Normally, if Auto Glue finds a pre-existing glue function, it will use it without attempting to build it again. Set IGNORE_EXISTING_GLUE to override this caching behavior and force CALL_EXTERNAL to rebuild the glue function sharable library.
LD	If present, a template string to be used in generating the linker command to build the glue function sharable library. For a more complete description of this keyword, see MAKE_DLL .
NOCLEANUP	If set, CALL_EXTERNAL will not remove intermediate files generated in order to build the glue function sharable library after the library has been built. This keyword can be used to preserve information for debugging in case of error, or for additional information on how Auto Glue works. For a more complete description of this keyword, see MAKE_DLL .
SHOW_ALL_OUTPUT	Auto Glue normally produces no output unless an error prevents successful building of the glue function sharable library. Set SHOW_ALL_OUTPUT to see all output produced by the process of building the library. For a more complete description of this keyword, see MAKE_DLL .
VERBOSE	If set, VERBOSE causes CALL_EXTERNAL to issue informational messages as it carries out the task of locating, building, and executing the glue function. For a more complete description of this keyword, see MAKE_DLL .

CEIL

Keyword/Argument	Description
L64	Set this keyword so that the result type is 64-bit integer regardless of the input type.

CLOSE

Keyword/Argument	Description
EXIT_STATUS	Set this keyword to a named variable that will contain the exit status reported by a UNIX child process started via the UNIT keyword to SPAWN.
FORCE	Set this keyword to force the file to be closed regardless of any errors that occur in the process.

CONTOUR

Keyword/Argument	Description
X, Y, Z	The X, Y and/or Z arguments are now accepted as double-precision floating point vectors/arrays without converting them to single-precision.
CLOSED	Set CLOSED=0 along with PATH_INFO and/or PATH_XY to return path information for contours that are not closed.
LEVELS	Now accepts a vector of double-precision floating point values without converting them to single-precision.
NLEVELS	Should be a positive integer.
PATH_DOUBLE	The new PATH_DOUBLE keyword has been added to allow a choice as to whether PATH_* information should be returned in single-precision or double-precision.

Keyword/Argument	Description
PATH_FILENAME	For this keyword, a secondary structure (CONTOUR_DBL_HEADER) is introduced for the case that the new PATH_DOUBLE keyword is set. This structure is the same as the CONTOUR_HEADER structure except that the VALUE field is a double-precision (rather than single-precision) floating point value.
PATH_INFO	For this keyword, a secondary structure (CONTOUR_DBL_PATH_STRUCTURE) is introduced for the case that the new PATH_DOUBLE keyword is set. This structure is the same as the CONTOUR_PATH_STRUCTURE except that the VALUE field is a double-precision (rather than a single-precision) floating point value. To return path information for contours that are not closed, set CLOSED=0.
PATH_XY	This keyword will now return an array of double-precision (rather than single-precision) coordinate values in the case that the new /PATH_DOUBLE keyword is set. To return path information for contours that are not closed, set CLOSED=0.

CONVERT_COORD

Keyword/Argument	Description
X, Y, Z	Accepts the X, Y and/or Z arguments as double-precision floating point vectors without converting them to single-precision.
DOUBLE	Set this keyword to specify the results should be returned in double-precision.

COORD2TO3

Keyword/Argument	Description
	This routine now returns a three-element vector of double-precision (rather than single-precision) values.

CREATE_VIEW

Keyword/Argument	Description
AX, AY, AZ, PERSP, XMAX, XMIN, YMAX, YMIN, ZFAC, ZMAX, ZMIN, ZOOM	These keywords now accept double-precision values and will no longer be converted to single-precision values.

CURSOR

Keyword/Argument	Description
NORMAL, DATA	If set, the X and Y arguments will contain double-precision values (rather than single-precision values).

CURVEFIT

Keyword/Argument	Description
DOUBLE	Set this keyword to force the computation to be performed in double-precision arithmetic.

CV_COORD

Keyword/Argument	Description
DOUBLE	Set this keyword to force the computation to be done in double-precision arithmetic.

CW_CLR_INDEX

Keyword/Argument	Description
VALUE	Set this keyword to the index of the color that is to be initially selected. The default is the START_COLOR.

CW_FILESEL

Keyword/Argument	Description
WARN_EXIST	Set this keyword to produce a question dialog if the user selects a file that already exists. This keyword is useful when creating a “write” dialog. The default is to allow any filename to be quietly accepted, whether it exists or not.

DEVICE

Keyword/Argument	Description
PRE_DEPTH (PS)	Set this keyword to a value indicating the bit depth to be used for the preview in the PostScript file. Valid values are 1 (for black and white preview) and 8 (for 8-bit grayscale preview). This keyword applies only if the PREVIEW keyword is nonzero. The default depth is 8.
PRE_XSIZE (PS)	Set this keyword to the width to be used for the preview in the PostScript file. PRE_XSIZE is specified in centimeters, unless the INCHES keyword is set. This keyword applies only if the PREVIEW keyword value is nonzero. The default is 1.77778 inches (128 pixels at 72dpi).
PRE_YSIZE (PS)	Set this keyword to the height to be used for the preview in the PostScript file. PRE_YSIZE is specified in centimeters, unless the INCHES keyword is set. This keyword applies only if the PREVIEW keyword value is nonzero. The default is 1.77778 inches (128 pixels at 72dpi).

DIALOG_PICKFILE

Keyword/Argument	Description
FILTER	Set this keyword to a string value or an array of strings specifying the file types to be displayed in the file list. This keyword is used to reduce the number of files to choose from. Note that in UNIX, passing an array using the FILTER keyword will result in the inclusion of all files in the current directory.

DIALOG_READ_IMAGE

Keyword/Argument	Description
GET_PATH	Set this keyword to a named variable in which the path of the selection is returned.

DIALOG_WRITE_IMAGE

Keyword/Argument	Description
WARN_EXIST	Set this keyword to produce a question dialog if the user selects a file that already exists. The default is to quietly overwrite the file.

DOUBLE

Keyword/Argument	Description
Expression	This argument is the expression to be converted to double-precision, floating-point.
Offset	This argument is the offset from beginning of the <i>Expression</i> data area.
D_i	When extracting fields of data, the D_i arguments specify the dimensions of the result. The dimension parameters can be any scalar expression. Up to eight dimensions can be specified. If no dimension arguments are given, the result is taken to be scalar.

DRAW_ROI

Keyword/Argument	Description
oROI	Since the IDLanROI object now supports single or double-precision vertices, this routine now supports single or double-precision.

FACTORIAL

Keyword/Argument	Description
N	This argument can now be either a scalar or an array.
UL64	Set this keyword to return the results as unsigned 64-bit integers.

FLOOR

Keyword/Argument	Description
L64	Set this keyword so that the result type is 64-bit integer regardless of the input type.

FREE_LUN

Keyword/Argument	Description
EXIT_STATUS	Set this keyword to a named variable that will contain the exit status reported by a UNIX child process started via the UNIT keyword to SPAWN.
FORCE	Set this keyword to force the file to be closed regardless of any errors that occur in the process.

FSTAT

Keyword/Argument	Description
n/a	<p>The following new fields are returned in the FSTAT structure:</p> <ul style="list-style-type: none"> • ATIME — Date of last access • CTIME — Creation date • MTIME — Date of last modification <p>All are reported in seconds since 1 January 1970 UTC.</p>

GETENV

Keyword/Argument	Description
ENVIRONMENT	<p>Set this keyword to return a string array containing all environment variables set in the current process, one variable per entry, in the format (Variable = <i>value</i>). This keyword is for UNIX only.</p>

HANNING

Keyword/Argument	Description
DOUBLE	<p>Set this keyword to force the computation to be done in double precision.</p>

HISTOGRAM

Keyword/Argument	Description
BINSIZE	<p>When the new NBINS keyword is specified and BINSIZE is not specified, the default is $BINSIZE = (MAX - MIN) / (NBINS - 1)$.</p>
L64	<p>By default, the return value of HISTOGRAM is 32-bit integer when possible, and 64-bit integer if the number of elements being processed requires it. Set L64 to force 64-bit integers to be returned in all cases. L64 controls the type of <i>Result</i> as well as the output from the REVERSE_INDICES keyword.</p>

Keyword/Argument	Description
MAX	If the new NBINS keyword is specified, the value for MAX will be adjusted to $\text{NBINS} \times \text{BINSIZE} + \text{MIN}$. This ensures that the last bin has the same width as the other bins.
NBINS	Set this keyword to the number of bins to use.
REVERSE_INDICES	Set this keyword to a named variable in which the list of reverse indices is returned.

IBETA, IGAMMA

Keyword/Argument	Description
DOUBLE	Set this keyword to force the computation to be done in double precision.
EPS	Set this keyword to the relative accuracy, or tolerance.
ITER	Set this keyword to a named variable that will contain the actual number of iterations performed.
ITMAX	Set this keyword to specify the maximum number of iterations. The default value is 100.

ISOCONTOUR

Keyword/Argument	Description
Outverts	Vertices are now returned in double-precision floating point if the new DOUBLE keyword is set.
C_VALUE	Now accepts a vector of double-precision floating point values, independent of the setting of the new DOUBLE keyword.
DOUBLE	This new keyword allows you to specify that computations are to be carried out in double-precision and to return resulting vertices as double-precision values.
LEVEL_VALUES	Now returns a vector of double-precision floating point values, independent of the setting of the new DOUBLE keyword.

JULDAY

Keyword/Argument	Description
Month, Day, Year, Hour, Minute, Second	These arguments now accept array values.

LABEL_DATE

Keyword/Argument	Description
AM_PM,	Set this keyword to a string of 2 names to be used for the names of the AM and PM strings.
DATE_FORMATS	This keyword now accepts format strings that include codes for sub-seconds. DATE_FORMATS can now also accept a string array for use with a multi-level axis.
DAYS_OF_WEEK	Set this keyword to a string array of 7 names to be used for the days of the week.
OFFSET	Set this keyword to a value representing the offset to be added to each tick value before conversion to a label. This keyword is usually used when the tick values are measured relative to a certain starting time.
ROUND_UP	Set this keyword to force times to be rounded up to the smallest time unit that is present in the DATE_FORMAT string.

LEEFILT

Keyword/Argument	Description
DOUBLE	Set this keyword to force the computation to be performed in double precision.

LINFIT

Keyword/Argument	Description
COVAR	Set this keyword to a named variable that will contain the Covariance matrix of the coefficients.
MEASURE_ERRORS	Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y . Note - This keyword has replaced the SDEV keyword. MEASURE_ERRORS has the same definition and meaning as SDEV. For backwards compatibility, the SDEV keyword is still accepted, but new code should use the MEASURE_ERRORS keyword.
YFIT	Set this keyword equal to a named variable that will contain the vector of calculated Y values.

LIVE_CONTOUR

Keyword/Argument	Description
DOUBLE	Set this keyword to force LIVE_CONTOUR to use double-precision to draw the result. This has the same effect as specifying data in the Zn argument using IDL variables of type DOUBLE.

LIVE_PLOT

Keyword/Argument	Description
DOUBLE	Set this keyword to force LIVE_PLOT to use double-precision to draw the result. This has the same effect as specifying data in the YVector argument using IDL variables of type DOUBLE.

LIVE_SURFACE

Keyword/Argument	Description
DOUBLE	Set this keyword to force LIVE_SURFACE to used double-precision to draw the result. This has the same effect as specifying data in the Data argument using IDL variables of type DOUBLE.

LMFIT

Keyword/Argument	Description
MEASURE_ERRORS	<p>Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y.</p> <p>Note - This keyword has replaced the WEIGHTS keyword. Code that uses the WEIGHTS keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. Note that the definition of the MEASURE_ERRORS keyword is not the same as the WEIGHTS keyword. Using the WEIGHTS keyword, $\text{SQRT}(1/\text{WEIGHTS}[i])$ represents the measurement error for each point $Y[i]$. Using the MEASURE_ERRORS keyword, the measurement error for each point is represented as simply $\text{MEASURE_ERRORS}[i]$. For an example, see “LMFIT” on page 24.</p>
SIGMA	<p>The definition of the SIGMA keyword has changed. If you do not specify error estimates (via the MEASURE_ERRORS keyword), then you are assuming that your user-supplied model (or the default quadratic), is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in SIGMA are multiplied by the correction factor $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X, and M is the number of coefficients. In versions of IDL prior to 5.4, this correction factor was not being applied. For an example, see “LMFIT” on page 24.</p>

MIN_CURVE_SURF

Keyword/Argument	Description
CONST	Set this keyword to fit data on the sphere with a constant baseline, otherwise, data on the sphere is fit with a baseline that contains a constant term plus linear X, Y, and Z terms.
SPHERE	Set this keyword to perform interpolation on the surface of a sphere.

MPEG_OPEN

Keyword/Argument	Description
BITRATE	<p>Set this keyword to a double-precision value to specify the MPEG movie bit rate. Higher bit rates will create higher quality MPEGs but will increase file size. The following table describes the valid values:</p> <ul style="list-style-type: none"> • MPEG 1 — 0.1 to 104857200.0 • MPEG 2 — 0.1 to 429496729200.0 <p>If you do not set this keyword, IDL computes the BITRATE value based upon the value you have specified for the QUALITY keyword.</p> <p>Note - Only use the BITRATE keyword if changing the QUALITY keyword value does not produce the desired results. It is highly recommended to set the BITRATE to at least several times the frame rate to avoid unusable MPEG files or file generation errors.</p>

Keyword/Argument	Description
IFRAME_GAP	<p>Set this keyword to a positive integer value that specifies the number of frames between I frames to be created in the MPEG file. I frames are full-quality image frames that may have a number of predicted or interpolated frames between them.</p> <p>If you do not specify this keyword, IDL computes the IFRAME_GAP value based upon the value you have specified for the QUALITY keyword.</p> <p>Note - Only use the IFRAME_GAP keyword if changing the QUALITY keyword value does not produce the desired results.</p>
MOTION_VEC_LENGTH	<p>Set this keyword to an integer value specifying the length of the motion vectors to be used to generate predictive frames. Valid values include:</p> <ul style="list-style-type: none"> • 1 — Small motion vectors. • 2 — Medium motion vectors. • 3 — Large motion vectors. <p>If you do not set this keyword, IDL computes the MOTION_VEC_LENGTH value based upon the value you have specified for the QUALITY keyword.</p> <p>Note - Only use the MOTION_VEC_LENGTH keyword if changing the QUALITY value does not produce the desired results.</p>
QUALITY	<p>Set this keyword to an integer value between 0 (low quality) and 100 (high quality) inclusive to specify the quality at which the MPEG stream is to be stored. Higher quality values result in lower rates of time compression and less motion prediction which provide higher quality MPEGs but with substantially larger file size. Lower quality factors may result in longer MPEG generation times. The default is 50.</p> <p>Note - Since MPEG uses JPEG (lossy) compression, the original picture quality can't be reproduced even when setting QUALITY to its highest setting.</p>

ONLINE_HELP

Keyword/Argument	Description
TOPICS	If set, the Topics dialog of the help system will be displayed for the specified help file.
HTML_HELP	If set, the Windows HTML Help system is used. All other keywords to ONLINE_HELP behave as specified, but the HTML help system is utilized. Note that a default file extension of .chm is used, not .hlp.

OPEN

Keyword/Argument	Description
RAWIO	The pre-existing keyword NOSTDIO has been renamed RAWIO to reflect the fact that stdio may or may not actually be used. IDL will continue to accept NOSTDIO as a synonym for RAWIO.
STDIO	Set this keyword to force the file to be opened via the standard C I/O library (stdio) rather than any other native OS API that might usually be used. This is not generally necessary and is intended for use with dynamically linked 3rd party code where the details of how I/O is performed is relevant.

OPLOT

Keyword/Argument	Description
X, Y	OPLOT now accepts the X and/or Y arguments as double-precision floating point vectors without converting them to single-precision.

PLOT

Keyword/Argument	Description
X, Y	PLOT now accepts X and/or Y arguments as double-precision floating point vectors without converting them to single-precision.

PLOTS

Keyword/Argument	Description
X, Y, Z	PLOTS now accepts the X, Y, and/or Z arguments as double-precision floating point vectors without converting them to single-precision.

POLY_FIT

Keyword/Argument	Description
CHISQ	Set this keyword to a named variable that will contain the value of the chi-square goodness-of-fit.
COVAR	This keyword has replaced the <i>Corrm</i> argument. For backwards compatibility, the argument will still be accepted.
MEASURE_ERRORS	<p>Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y.</p> <p>Note - This keyword has replaced the POLYFITW function. Note, however, that the definition of the MEASURE_ERRORS keyword to POLY_FIT is different from the definition of the <i>Weights</i> argument to POLYFITW. In POLYFITW, $\text{SQRT}(1/\text{Weights}[i])$ represented the measurement error for each point $Y[i]$. Now, for consistency with other curve-fitting routines, POLY_FIT defines the measurement error for each point as $\text{MEASURE_ERRORS}[i]$. Code using POLYFITW will continue to work as before, but new code should use POLY_FIT. If you wish to convert existing code using POLYFITW to use the new MEASURE_ERRORS keyword to POLY_FIT, you must change the values you supply. For an example, see “POLY_FIT” on page 26.</p>
SIGMA	Set this keyword to a named variable that will contain the 1-sigma uncertainty estimates for the returned parameters.

Keyword/Argument	Description
YERROR	This keyword has replaced the <i>Sigma</i> argument. For backwards compatibility, the argument will still be accepted.
YFIT	This keyword has replaced the <i>Yfit</i> argument. For backwards compatibility, the argument will still be accepted.
STATUS	Set this keyword to a named variable to receive the status of the operation. Possible status values are: <ul style="list-style-type: none"> • 0 = Successful completion. • 1 = Singular array (which indicates that the inversion is invalid). <i>Result</i> is NaN. • 2 = Warning that a small pivot element was used and that significant accuracy was probably lost. • 3 = Undefined (NaN) error estimate was encountered.

POLYFILL

Keyword/Argument	Description
X, Y, Z	Now accepts X, Y, and/or Z as double-precision values without converting them to single-precision.

POLYSHADE

Keyword/Argument	Description
X, Y, Z	Now accepts X, Y, and/or Z as double-precision values without converting them to single-precision.

RANDOMN, RANDOMU

Keyword/Argument	Description
DOUBLE	Set this keyword to force the computation to be done using double-precision arithmetic.
LONG	Set this keyword to return integer uniform random deviates in the range $[1...2^{31} - 2]$. If LONG is set, all other keywords are ignored.

READ_JPEG

Keyword/Argument	Description
UNIT	When opening a file intended for use with the UNIT keyword, if the filename does not end in <code>.jpg</code> , or <code>.jpeg</code> , you must specify the STDIO keyword to OPEN in order for the file to be compatible with READ_JPEG.

READ_PNG

Keyword/Argument	Description
ORDER	Set this keyword to indicate that the rows of the image should be drawn from the bottom to top. By default the rows are drawn from top to bottom.

READ_TIFF

Keyword/Argument	Description
CHANNELS	Set this keyword to a scalar or vector giving the channel numbers to be returned for a multi-channel image, starting with zero. The default is to return all of the channels.

Keyword/Argument	Description
INTERLEAVE	<p>For multi-channel images, set this keyword to one of the following values to force the <i>Result</i> to have a specific interleaving, regardless of the type of interleaving in the file being read:</p> <ul style="list-style-type: none"> • 0 = Pixel interleaved: <i>Result</i> will have dimensions [<i>Channels</i>, <i>Columns</i>, <i>Rows</i>]. • 1 = Scanline (row) interleaved: <i>Result</i> will have dimensions [<i>Columns</i>, <i>Channels</i>, <i>Rows</i>]. • 2 = Planar interleaved: <i>Result</i> will have dimensions [<i>Columns</i>, <i>Rows</i>, <i>Channels</i>]. <p>If this keyword is not specified, the <i>Result</i> will always be pixel interleaved, regardless of the type of interleaving in the file being read. For files stored in planar-interleave format, this keyword is ignored if the <i>R</i>, <i>G</i>, and <i>B</i> arguments are specified.</p>

READU

Keyword/Argument	Description
TRANSFER_COUNT	This keyword is now accepted on all platforms.

REGRESS

Keyword/Argument	Description
CHISQ	<p>Set this keyword equal to a named variable that will contain the value of the chi-square goodness-of-fit.</p> <p>Note - This keyword replaces the <i>Chisq</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>
CONST	<p>Set this keyword to a named variable that will contain the constant term of the fit.</p> <p>Note - This keyword replaces the <i>Const</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>

Keyword/Argument	Description
CORRELATION	<p>Set this keyword to a named variable that will contain the vector of linear correlation coefficients.</p> <p>Note - This keyword replaces the <i>R</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>
DOUBLE	<p>Set this keyword to force computations to be done in double-precision arithmetic.</p>
FTEST	<p>Set this keyword to a named variable that will contain the F-value for the goodness-of-fit test.</p> <p>Note - This keyword replaces the <i>Ftest</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>
MCORRELATION	<p>Set this keyword to a named variable that will contain the multiple linear correlation coefficient.</p> <p>Note - This keyword replaces the <i>Rmul</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>
SIGMA	<p>Set this keyword to a named variable that will contain the 1-sigma uncertainty estimates for the returned parameters.</p> <p>Note - This keyword replaces the <i>Sigma</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>
STATUS	<p>Set this keyword to a named variable that will contain the status of the operation. Possible status values are:</p> <ul style="list-style-type: none"> • 0 = successful completion • 1 = singular array (which indicates that the inversion is invalid) • 2 = warning that a small pivot element was used and that significant accuracy was probably lost. <p>Note - This keyword replaces the <i>Status</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>

Keyword/Argument	Description
MEASURE_ERRORS	<p>Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y.</p> <p>Note - This keyword has replaced the <i>Weights</i> argument. The definition of MEASURE_ERRORS is different from the <i>Weights</i> argument that it has replaced. Using the <i>Weights</i> argument, $\text{SQRT}(1/\text{Weights}[i])$ represents the measurement error for each point $Y[i]$. Now, for consistency with other curve-fitting routines, the measurement error for each point is represented as simply MEASURE_ERRORS[i]. Also, the RELATIVE_WEIGHT keyword is no longer necessary. Now, if the MEASURE_ERRORS keyword is not provided, then REGRESS assumes you want no weighting. For an example of how to use the new MEASURE_ERRORS keyword, see “REGRESS” on page 27.</p>
YFIT	<p>Set this keyword to a named variable that will contain the vector of calculated Y values.</p> <p>Note - This keyword replaces the <i>Yfit</i> argument. The argument is still accepted for backward compatibility, but the keyword should be used in all new code.</p>

RESOLVE_ROUTINE

Keyword/Argument	Description
COMPILE_FULL_FILE	Set COMPILE_FULL_FILE to compile the entire file regardless of encountering the specified routine in <i>Name</i> .

REVERSE

Keyword/Argument	Description
OVERWRITE	Set this keyword to conserve memory by doing the transformation “in-place.” THE result overwrites the previous contents of the array.

ROUND

Keyword/Argument	Description
L64	Set this keyword so that the result type is 64-bit integer regardless of the input type.

SET_PLOT

Keyword/Argument	Description
Device	This argument now accepts the “METAFILE” device. For more information, see “Windows Metafile Format (WMF) Support for Direct Graphics” on page 14.

SHADE_SURF

Keyword/Argument	Description
X, Y, Z	These arguments now accept double-precision values without converting them to single-precision.

SIZE

Keyword/Argument	Description
DIMENSIONS	Set this keyword to return the dimensions of <i>Expression</i> . The result is a 32-bit integer when possible, and 64-bit integer if the number of elements in <i>Expression</i> requires it. Set L64 to force 64-bit integers to be returned in all cases.
L64	Set this keyword to force 64-bit integers to be returned in all cases. In addition to affecting the default result, L64 also affects the output from the DIMENSIONS, N_ELEMENTS, and STRUCTURE keywords.
N_ELEMENTS	Set this keyword to return the number of data elements in <i>Expression</i> . Setting this keyword is equivalent to using the N_ELEMENTS function. The result is a 32-bit integer when possible, and 64-bit integer if the number of elements requires it.

Keyword/Argument	Description
STRUCTURE	Set this keyword to return all available information about <i>Expression</i> in a structure. The result is an IDL_SIZE (32-bit) structure when possible, and an IDL_SIZE64 structure otherwise.

SORT

Keyword/Argument	Description
L64	Set this keyword so that the result type is 64-bit integer regardless of the input type.

SPAWN

Keyword/Argument	Description
ErrResult	A named variable in which to place the error output (stderr) from the child process. (UNIX and Windows only.)
EXIT_STATUS	Set this keyword to Return the exit status for the child process. The meaning of this value is operating system dependent
FORCE	Set this keyword to force the file to be closed regardless if errors occur in the process.
HIDE	Set this keyword so that the command interpreter shell window is minimized to prevent the user from seeing it. (Windows only)
LOG_OUTPUT	Set this keyword so that the command interpreter window is minimized (as with HIDE) and all output is diverted to the IDLDE log window. (Windows only)
NOSHELL	This keyword is now supported on Windows platforms.
NOWAIT	Set this keyword so that the IDL process continues executing in parallel with the subprocess. (Windows, Macintosh, and VMS only)

Keyword/Argument	Description
NULL_STDIN	Set this keyword so that the null device <code>/dev/null</code> (UNIX) or NUL (Windows) is connected to the standard input of the child process. (UNIX and Windows only)
STDERR	Set this keyword so that the child's error output (stderr) is combined with the standard output and returned in <i>Result</i> . STDERR and the <i>ErrResult</i> argument are mutually exclusive. (UNIX and Windows only)

SURFACE

Keyword/Argument	Description
X, Y, Z	Now accepts double-precision values without converting them to single-precision.
SKIRT	Now accepts double-precision values without converting them to single-precision.

SVDC

Keyword/Argument	Description
ITMAX	Set this keyword to specify the maximum number of iterations. The default value is 30.

SVDFIT

Keyword/Argument	Description
MEASURE_ERRORS	<p>Set this keyword to a vector containing standard measurement errors for each point $Y[i]$. This vector must be the same length as X and Y.</p> <p>Note - The WEIGHTS keyword is obsolete and has been replaced by the MEASURE_ERRORS keyword. Code that uses the WEIGHTS keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. Note that the definition of the MEASURE_ERRORS keyword is not the same as the WEIGHTS keyword. Using the WEIGHTS keyword, $1/\text{WEIGHTS}[i]$ represents the measurement error for each point $Y[i]$. Using the MEASURE_ERRORS keyword, the measurement error is represented as simply $\text{MEASURE_ERRORS}[i]$. For an example, see “SVDFIT” on page 29.</p>
SIGMA	<p>The definition of the SIGMA keyword has changed. If you do not specify error estimates (via the MEASURE_ERRORS keyword), then you are assuming that the polynomial (or your user-supplied model) is the correct model for your data, and therefore, no independent goodness-of-fit test is possible. In this case, the values returned in SIGMA are multiplied by the correction factor $\text{SQRT}(\text{CHISQ}/(N-M))$, where N is the number of points in X, and M is the number of coefficients. In versions of IDL prior to 5.4, this correction factor was not being applied. For an example, see “SVDFIT” on page 29.</p>

SYSTIME

Keyword/Argument	Description
ElapsedSeconds	If the <i>SecondsFlag</i> argument (previously called <i>Arg</i>) is zero, the <i>ElapsedSeconds</i> argument may be set to the number of seconds past 1 January 1970 UTC. In this case, SYSTIME returns the corresponding date/time string (rather than the string for the current time). The returned date/time string is adjusted for the local time zone, unless the UTC keyword is set.
UTC	Set this keyword to specify that the value returned by SYSTIME is to be returned in Universal Time Coordinated (UTC) rather than being adjusted for the current time zone. UTC time is defined as Greenwich Mean Time updated with leap seconds. UTC can be used with the JULIAN keyword.

T3D

Keyword/Argument	Description
OBLIQUE PERSPECTIVE ROTATE SCALE TRANSLATE	These keywords now accept double-precision values (without conversion to single-precision).

TRIGRID

Keyword/Argument	Description
XOUT, YOUT	Set these keywords to a vector specifying the output grid <i>X</i> and <i>Y</i> values. If these keywords are supplied, the <i>GS</i> and <i>Limits</i> arguments are ignored. Use these keywords to specify irregularly spaced rectangular output grids.

TV

Keyword/Argument	Description
X, Y	Now accepts X and Y as double-precision values without converting them to single-precision.

TVCRS

Keyword/Argument	Description
X, Y	Now accepts X and Y as double-precision values without converting them to single-precision.

TVSCL

Keyword/Argument	Description
X, Y	Now accepts X and Y as double-precision values without converting them to single-precision.

VALUE_LOCATE

Keyword/Argument	Description
L64	Set this keyword so that the result type is 64-bit integer regardless of the input type.

VERT_T3D

Keyword/Argument	Description
DOUBLE	Set this keyword so that the results are returned in double-precision.

WHERE

Keyword/Argument	Description
COMPLEMENT	Set this keyword to a named variable that receives the subscripts of the zero elements of <i>Array_Expression</i> . These are the subscripts that are not returned in <i>Result</i> . Together, <i>Result</i> and COMPLEMENT specify every subscript in <i>Array_Expression</i> . If there are no zero elements in <i>Array_Expression</i> , COMPLEMENT returns a scalar integer with the value -1.
NCOMPLEMENT	Set this keyword to a named variable that receives the number of zero elements found in <i>Array_Expression</i> . This value is the number of subscripts that will be returned via the COMPLEMENT keyword if it is specified.
L64	Set this keyword so that the result type is 64-bit integer regardless of the input type.

WRITE_JPEG

Keyword/Argument	Description
UNIT	When opening a file intended for use with the UNIT keyword, if the filename does not end in <code>.jpg</code> , or <code>.jpeg</code> , you must specify the STDIO keyword to OPEN in order for the file to be compatible with WRITE_JPEG.

WRITE_PNG

Keyword/Argument	Description
ORDER	Set this keyword to indicate that the rows of the image should be written from the bottom to top. By default, the rows are written from top to bottom.

WRITEU

Keyword/Argument	Description
TRANSFER_COUNT	Now accepted on all platforms.

XINTERANIMATE

Keyword/Argument	Description
MPEG_BITRATE	<p>Set this keyword to a double-precision value to specify the MPEG movie bit rate. Higher bit rates will create higher quality MPEGs but will increase file size. The following table describes the valid values:</p> <ul style="list-style-type: none"> • MPEG 1 — 0.1 to 104857200.0 • MPEG 2 — 0.1 to 429496729200.0 <p>If you do not set this keyword, IDL computes the MPEG_BITRATE value based upon the value you have specified for the MPEG_QUALITY keyword.</p> <p>Note - Only use the MPEG_BITRATE keyword if changing the MPEG_QUALITY keyword value does not produce the desired results. It is highly recommended to set the MPEG_BITRATE to at least several times the frame rate to avoid unusable MPEG files or file generation errors.</p>
MPEG_IFRAME_GAP	<p>Set this keyword to a positive integer value that specifies the number of frames between I frames to be created in the MPEG file. I frames are full-quality image frames that may have a number of predicted or interpolated frames between them.</p> <p>If you do not specify this keyword, IDL computes the MPEG_IFRAME_GAP value based upon the value you have specified for the MPEG_QUALITY keyword.</p> <p>Note - Only use the MPEG_IFRAME_GAP keyword if changing the MPEG_QUALITY keyword value does not produce the desired results.</p>

Keyword/Argument	Description
MPEG_MOTION_VEC_LENGTH	<p>Set this keyword to an integer value specifying the length of the motion vectors to be used to generate predictive frames. Valid values include:</p> <ul style="list-style-type: none"> • 1 — Small motion vectors. • 2 — Medium motion vectors. • 3 — Large motion vectors. <p>If you do not set this keyword, IDL computes the MPEG_MOTION_VEC_LENGTH value based upon the value you have specified for the MPEG_QUALITY keyword.</p> <p>Note - Only use the MPEG_MOTION_VEC_LENGTH keyword if changing the MPEG_QUALITY value does not produce the desired results.</p>
MPEG_QUALITY	<p>Set this keyword to an integer value between 0 (low quality) and 100 (high quality) inclusive to specify the quality at which the MPEG stream is to be stored. Higher quality values result in lower rates of time compression and less motion prediction which provide higher quality MPEGs but with substantially larger file size. Lower quality factors may result in longer MPEG generation times. The default is 50.</p> <p>Note - Since MPEG uses JPEG (lossy) compression, the original picture quality can't be reproduced even when setting QUALITY to its highest setting.</p>

XYOUTS

Keyword/Argument	Description
X, Y	Now accepts X and Y as double-precision values without converting them to single-precision.

Updated Common Graphics Keywords

The following is a list of updated common graphics keywords to existing IDL routines.

CLIP

Affected Routines	Description
CONTOUR, DRAW_ROI, OPLOT, PLOT, PLOTS, POLYFILL, SURFACE, XYOUTS	This keyword now accepts a vector of double-precision values without converting them to single-precision.

[XYZ]RANGE

Affected Routines	Description
AXIS, CONTOUR, PLOT, SHADE_SURF, SURFACE	This keyword now accepts a 2-element vector of double-precision values without converting them to single-precision.

[XYZ]TICKFORMAT

Affected Routines	Description
AXIS, CONTOUR, PLOT, SHADE_SURF, SURFACE	<ol style="list-style-type: none"> 1. This keyword may now be set to either a single string or an array of strings. Each string corresponds to a level of the axis. 2. If any of the strings is the name of a callback function, the third argument to that function (that is the argument indicating the value of the tickmark) will be double-precision. 3. If any of the strings is the name of a callback function and if the [XYZ]TICKUNITS keyword is set to one or more non-empty strings, the callback function will be called with four parameters: Axis, Index, Value and Level, where Axis, Index, and Value are the same as before, and Level is the Index of the axis level for the current tick value to be labelled (Level indices start at 0).

[XYZ]TICK_GET

Affected Routines	Description
AXIS, CONTOUR, PLOT, SHADE_SURF, SURFACE	The value returned is now a vector of double-precision floating point values.

[XYZ]TICKINTERVAL

Affected Routines	Description
AXIS, CONTOUR, PLOT, SHADE_SURF, SURFACE	Set this keyword to a scalar to indicate the interval between major tick marks for the first axis level.

[XYZ]TICKLAYOUT

Affected Routines	Description
AXIS, CONTOUR, PLOT, SHADE_SURF, SURFACE	<p>Set this keyword to a scalar that indicates the style to draw each level of the axis. Valid values are:</p> <ul style="list-style-type: none"> • 0 — The axis line, major tick marks, and tick labels are drawn. • 1 — Only the labels for the major tick marks are drawn. • 2 — Each major tick interval is outlined by a box.

[XYZ]TICKUNITS

Affected Routines	Description
AXIS, CONTOUR, PLOT, SHADE_SURF, SURFACE	<p>Set this keyword to a string (or vector of strings) to indicate the units to be used for axis tick labeling. Valid values are:</p> <ul style="list-style-type: none"> • “Numeric” (the default) • “Year” • “Month” • “Day” • “Hour” • “Minute” • “Second” • “Time” — Use this value to indicate that the units are generic time units. IDL will compute appropriate default intervals and tick formats based on the range of values covered by the axis. <p>You can specify more than one type of unit. The axis levels will be drawn in the order in which you specify the strings with the first unit being drawn nearest to the primary axis line.</p>

[XYZ]TICKV

Affected Routines	Description
AXIS, CONTOUR, PLOT, SHADE_SURF, SURFACE	<p>[XYZ]TICKV will now accept a vector of double-precision values without converting them to single-precision.</p>

New and Updated System Variables

The following system variables have been added or updated in IDL 5.4:

System Variable	Description
!MAKE_DLL	A new system variable used to configure how IDL uses the standard system C compiler and linker to generate sharable libraries for the current platform. The <code>AUTO_GLUE</code> keyword to the <code>CALL_EXTERNAL</code> function and <code>MAKE_DLL</code> procedure uses the standard system C compiler and linker to generate sharable libraries that can be used by IDL in various contexts (<code>CALL_EXTERNAL</code> , <code>DLMs</code> , <code>LINKIMAGE</code>). For more information, see “ !MAKE_DLL System Variable ” on page 196.
!P.T	<code>!P.T</code> has changed from a single-precision 4-by-4 array of floating-point values to a double-precision floating-point array of values.
!VERSION	The <code>!VERSION</code> system variable has two new fields called <code>MEMORY_BITS</code> and <code>FILE_OFFSET_BITS</code> that tell you how many bits are used by the current IDL to access memory and files, respectively.
![XYZ].CRANGE	<code>![XYZ].CRANGE</code> , formerly a 2-element vector of single-precision floating point values, is now a 2-element vector of double-precision floating point values.
![XYZ].RANGE	<code>![XYZ].RANGE</code> , formerly a 2-element vector of single-precision floating point values, is now a 2-element vector of double-precision floating point values.
![XYZ].S	<code>![XYZ].S</code> , formerly a 2-element vector of single-precision floating point values, is now a 2-element vector of double-precision floating point values.

System Variable	Description
![XYZ]. TICKFORMAT	The third argument (that is, the argument indicating the value at the tick mark) for any callback functions set via the ![XYZ].TICKFORMAT field will now become a double-precision floating point value (rather than a single-precision floating point value).
![XYZ]. TICKLAYOUT	![XYZ].TICKLAYOUT is a scalar that indicates the style to be used to draw each level of the axis.
![XYZ]. TICKINTERVAL	![XYZ].TICKINTERVAL is a scalar indicating the interval between major tick marks for the first axis level. This setting takes precedence over ![XYZ].TICKS.
![XYZ]. TICKUNITS	![XYZ].TICKUNITS is a string (or a vector of strings) indicating the units to be used for axis tick labeling.
![XYZ]. TICKV	![XYZ].TICKV, formerly a vector of single-precision floating point values, is now a vector of double-precision floating point values.

Features Obsoleted in IDL 5.4

Obsoleted Routines

The following routines were present in IDL Version 5.3 but became obsolete in IDL Version 5.4. These routines have been replaced with new routines or new keywords to existing routines that offer enhanced functionality. These obsoleted routines should not be used in new IDL code.

Routine	Replaced by
POLYFITW	POLY_FIT, MEASURE_ERRORS keyword
RIEMANN	RADON

Table 1-8: Routines Obsoleted in IDL 5.4

Obsoleted Keywords and Arguments

The following keywords and arguments became obsolete in IDL Version 5.4. These keywords and arguments have been replaced with new routines or new keywords to existing routines that offer enhanced functionality. These obsoleted keywords and arguments should not be used in new IDL code.

Routine	Keyword/Argument	Description
LINFIT	SDEV	This keyword has been replaced by the MEASURE_ERRORS keyword. The definition of the MEASURE_ERRORS keyword is identical to that of the SDEV keyword. The SDEV keyword is still accepted for backwards compatibility.

Table 1-9: Keywords/Arguments Obsoleted in IDL 5.4

Routine	Keyword/Argument	Description
LMFIT	WEIGHTS	This keyword has been replaced by the MEASURE_ERRORS keyword. Code that uses the WEIGHTS keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. Note that the definition of the MEASURE_ERRORS keyword is not the same as the WEIGHTS keyword. Using the WEIGHTS keyword, $\text{SQRT}(1/\text{WEIGHTS}[i])$ represents the measurement error for each point $Y[i]$. Using the MEASURE_ERRORS keyword, the measurement error for each point is represented as simply $\text{MEASURE_ERRORS}[i]$.
	BINARY	This keyword is no longer necessary on Windows for input/output. Still accepted, but quietly ignored, for backward compatibility.
	NOAUTOMODE	This keyword is no longer necessary on Windows for input/output. Still accepted, but quietly ignored, for backward compatibility.
OPEN	NOSTDIO	This keyword has been renamed RAWIO to reflect the fact that stdio may or may not actually be used. Still accepted as a synonym for RAWIO.

Table 1-9: Keywords/Arguments Obsolete in IDL 5.4

Routine	Keyword/Argument	Description
POLY_FIT	Yfit	The <i>Yfit</i> argument has been replaced by the YFIT keyword. Code using this argument will continue to work as before, but new code should use the keyword instead.
	Yband	The <i>Yband</i> argument has been replaced by the YBAND keyword. Code using this argument will continue to work as before, but new code should use the keyword instead.
	Sigma	The <i>Sigma</i> argument has been replaced by the YERROR keyword. Code using this argument will continue to work as before, but new code should use the keyword instead.
	Corrm	The <i>Corrm</i> argument has been replaced by the COVAR keyword. Code using this argument will continue to work as before, but new code should use the keyword instead.

Table 1-9: Keywords/Arguments Obsoleted in IDL 5.4

Routine	Keyword/Argument	Description
REGRESS	Weights	The <i>Weights</i> argument has been replaced by the MEASURE_ERRORS keyword. Code that uses the <i>Weights</i> argument will continue to work as before, but new code should use the MEASURE_ERRORS keyword instead. Note that the definition of the MEASURE_ERRORS keyword is different from that of the <i>Weights</i> argument. Using the <i>Weights</i> argument, $\text{SQRT}(1/\text{Weights}[i])$ represents the measurement error for each point $Y[i]$. Using MEASURE_ERRORS, the measurement error for each point is represented as simply MEASURE_ERRORS[<i>i</i>]. Also note that the RELATIVE_WEIGHTS keyword is not necessary when using the MEASURE_ERRORS keyword.
	Yfit	The <i>Yfit</i> argument has been replaced by the YFIT keyword.
	Const	The <i>Const</i> argument has been replaced by the CONST keyword.
	Sigma	The <i>Sigma</i> argument has been replaced by the SIGMA keyword.
	Ftest	The <i>Ftest</i> argument has been replaced by the FTEST keyword.

Table 1-9: Keywords/Arguments Obsolete in IDL 5.4

Routine	Keyword/Argument	Description
REGRESS (continued)	R	The <i>R</i> argument has been replaced by the CORRELATION keyword.
	Rmul	The <i>Rmul</i> argument has been replaced by the MCOLLATION keyword.
	Chisq	The <i>Chisq</i> argument has been replaced by the CHISQR keyword.
	Status	The <i>Status</i> argument has been replaced by the STATUS keyword.
	RELATIVE_WEIGHT	This keyword is no longer necessary. Code using the <i>Weights</i> argument and RELATIVE_WEIGHT keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword, for which case the RELATIVE_WEIGHT keyword is not necessary. Using the <i>Weights</i> argument, it was necessary to specify the RELATIVE_WEIGHT keyword if no weighting was desired. This is not the case with the MEASURE_ERRORS keyword—when MEASURE_ERRORS is omitted, REGRESS assumes you want no weighting.

Table 1-9: Keywords/Arguments Obsolete in IDL 5.4

Routine	Keyword/Argument	Description
SVDFIT	WEIGHTS	This keyword has been replaced by the MEASURE_ERRORS keyword. Code that uses the WEIGHTS keyword will continue to work as before, but new code should use the MEASURE_ERRORS keyword. Note that the definition of the MEASURE_ERRORS keyword is not the same as the WEIGHTS keyword. Using the WEIGHTS keyword, $1/\text{WEIGHTS}[i]$ represents the measurement error for each point $Y[i]$. Using the MEASURE_ERRORS keyword, the measurement error is represented as simply $\text{MEASURE_ERRORS}[i]$.

Table 1-9: Keywords/Arguments Obsolete in IDL 5.4

Platforms Supported in this Release

IDL 5.4 supports the following platforms and operating systems:

Platform	Vendor	Hardware	Operating System	Supported Versions
VMS	Compaq	Alpha	VMS	7.1
UNIX†	Compaq	Alpha	Tru64 UNIX	4.0
	Compaq	Alpha	Linux	Red Hat 6.2††
	HP	PA-RISC	HP-UX	10.20, 11.0
	IBM	RS/6000	AIX	4.3
	Intel	Intel x86	Linux	Red Hat 6.0, 6.2††
	SGI	Mips	IRIX	6.4, 6.5
	SUN	SPARC	Solaris	2.6, 7, 8
	SUN	SPARC (64-bit Ultra)	Solaris	7, 8
	SUN	Intel x86	Solaris	2.6, 7
Windows	Microsoft	Intel x86	Windows	95b, 98, NT 4.0, 2000
Macintosh	Apple	PowerMAC†††	MacOS	8.x, 9.x

Table 1-10: Platforms Supported in IDL 5.4

† For UNIX, the supported versions indicate that IDL was either built (the lowest version listed) or tested on that version. You can install and run IDL on other versions that are binary compatible with those listed.

†† IDL 5.4 was built on the Linux 2.2 kernel with glibc 2.1 using Red Hat Linux. If your version of Linux is compatible with these, it is possible that you can install and run IDL on your version.

††† Includes G3, G4 and iMac



Chapter 2: Date/Time Plotting in IDL

This chapter contains the following topics:

Overview	136	Displaying Date/Time Data on an Axis in Direct Graphics	140
How to Generate Date/Time Data	138	Displaying Date/Time Data on an Axis in Object Graphics	148

Overview

Dates and times are among the many types of information that numerical data can represent. IDL provides a number of routines that offer specialized support for generating, analyzing, and displaying date- and time- based data (herein referred to as date/time data).

Julian Dates and Times

Within IDL, dates and times are typically stored as Julian dates. A Julian date is defined to be the number of days elapsed since noon on January 1, 4713 BCE. Following the astronomical convention, a Julian day is defined to start at 12pm (noon). The following table shows a few examples of calendar dates and their corresponding Julian dates.

Calendar Date	Julian Date
January 1, 4713 B.C.E., at 12pm	0
January 2, 4713 B.C.E., at 12pm	1
January 1, 2000 at 12pm	2451545

Table 2-1: Example Julian Dates

Julian dates can also include fractional portions of a day, thereby incorporating hours, minutes, and seconds. If the day fraction is included in a Julian date, it is represented as a double-precision floating point value. The day fraction is computed as follows:

$$dayFraction = \frac{hour}{24.d} + \frac{minute}{1440.d} + \frac{seconds}{86400.d}$$

One advantage of using Julian dates to represent dates and times is that a given date/time can be stored within a single variable (rather than storing the year, month, day, hour, minute, and second information in six different variables). Because each Julian date is simply a number, IDL's numerical routines can be applied to Julian dates just as for any other type of number.

Precision of Date/Time Data

The precision of any numerical value is defined as the smallest possible number that can be added to that value that produces a new value different from the first. Precision is typically limited by the data type of the variable used to store the number and the magnitude of the number itself. Within IDL, the following guide should be used when choosing a data format for date/time data:

- Time values that require a high precision, and that span a range of a few days or less, should be stored as double-precision values in units of time elapsed since the starting time, rather than in Julian date format. An example would be the seconds elapsed since the beginning of an experiment. In this case, the data can be treated within IDL as standard numeric data without the need to utilize IDL's specialized date/time features.
- Date values that do not include the time of day may be stored as long-integer Julian dates. The Julian date format has the advantage of being compact (one value per date) and being evenly spaced in days. As an example, January 1st for the years 2000, 2001, and 2002 can be stored as Julian days 2451545, 2451911, and 2452276. The precision of this format is 1 day.
- Date values where it is necessary to include the time of day can be stored as double-precision Julian dates, with the time included as a day fraction. Because of the large magnitude of the Julian date (such as Julian day 2451545 for 1 January 2000), the precision of most Julian dates is limited to 1 millisecond (0.001 seconds).

To determine the precision of a Julian date/time value, you can use the IDL **MACHAR** function:

```
; Set date to January 1, 2000, at 12:15pm
julian = JULDAY(1,1,2000,12,15,0)

; get machine characteristics
machine = MACHAR(/DOUBLE)

; multiply by floating-point precision
precision = julian*machine.eps

; convert to seconds
PRINT, precision*86400d0
```

How to Generate Date/Time Data

The **TIMEGEN** function returns an array of double precision floating point values that represent date/time in terms of Julian dates. The first value of the returned array corresponds to a start date/time, and each subsequent value corresponds to the start date/time plus that array element's one-dimensional subscript multiplied by a step size for a given date/time unit. Unlike the other array generation routines in IDL, **TIMEGEN** includes a **START** keyword, which is necessary if the starting date/time is originally provided in calendar (month, day, year) form.

The following example begins with a start date of March 1, 2000 and increments every month for a full year:

```
date_time = TIMEGEN(12, UNIT = 'Months', $
    START = JULDAY(3, 1, 2000))
```

where the **UNIT** keyword is set to 'Months' to increment by month and the **START** keyword is set to the Julian date form of March 1, 2000.

The results of the above call to **TIMEGEN** can be output using either of the following methods:

1. Using the **CALDAT** routine to convert the Julian dates to calendar dates:

```
CALDAT, date_time, month, day, year
FOR i = 0, (N_ELEMENTS(date_time) - 1) DO PRINT, $
    month[i], day[i], year[i], $
    FORMAT = '(i2.2, "/", i2.2, "/", i4)'
```

2. Using the calendar format codes:

```
PRINT, date_time, format = '(C(CMOI2.2, "/", CDI2.2, "/", CYI))'
```

The resulting calendar dates are printed out as follows:

```
03/01/2000
04/01/2000
05/01/2000
06/01/2000
07/01/2000
08/01/2000
09/01/2000
10/01/2000
11/01/2000
12/01/2000
01/01/2001
02/01/2001
```

The TIMEGEN routine contains several keywords to provide specific date/time data generation. For more information, see [TIMEGEN](#) in the *IDL Reference Guide*.

Displaying Date/Time Data on an Axis in Direct Graphics

You can display date/time data on plots, contours, and surfaces through the tick settings of the date/time axis. Date/time data can be displayed on any axis (x, y or z). The date/time data is stored as Julian dates, but the `LABEL_DATE` function and `AXIS` keywords allow you to display this data as calendar dates. The following examples show how to display one-dimensional and two-dimensional date/time data.

Displaying Date/Time Data on a Plot Display

Date/time data usually comes from measuring data values at specific times. For example, the displacement (in inches) of an object might be recorded at every second for 37 seconds after the initial recording of 59 minutes and 30 seconds after 2 o'clock pm (14 hundred hours) on the 30th day of March in the year 2000 as follows

```
number_samples = 37
date_time = TIMEGEN(number_samples, UNITS = 'Seconds', $
    START = JULDAY(3, 30, 2000, 14, 59, 30))
displacement = SIN(10.*!DTOR*FINDGEN(number_samples))
```

Normally, this type of data would be imported into IDL from a data file. However, this section is designed specifically to show how to display date/time data, not how to import data from a file; therefore, the data for this example is created with the above IDL commands.

Before displaying this one-dimensional data with the `PLOT` routine, the format of the date/time values is specified through the `LABEL_DATE` routine as follows

```
date_label = LABEL_DATE(DATE_FORMAT = ['%I:%S'])
```

where `%I` represents minutes and `%S` represents seconds.

The resulting format is specified in the call to the `PLOT` routine with the `XTICKFORMAT` keyword:

```
PLOT, date_time, displacement, /XSTYLE, $
    ; displaying titles.
    TITLE = 'Measured Signal', $
    XTITLE = 'Time (seconds)', $
    YTITLE = 'Displacement (inches)', $
    ; applying date/time formats to X-axis labels.
    XTICKFORMAT = 'LABEL_DATE', $
    XTICKUNITS = 'Time', $
    XTICKINTERVAL = 5
```

The XTICKUNITS keyword is set to note the tick labels contain date/time data. The XTICKINTERVAL keyword is set to place the major tick marks at every five second interval. These keyword settings produce the following results:

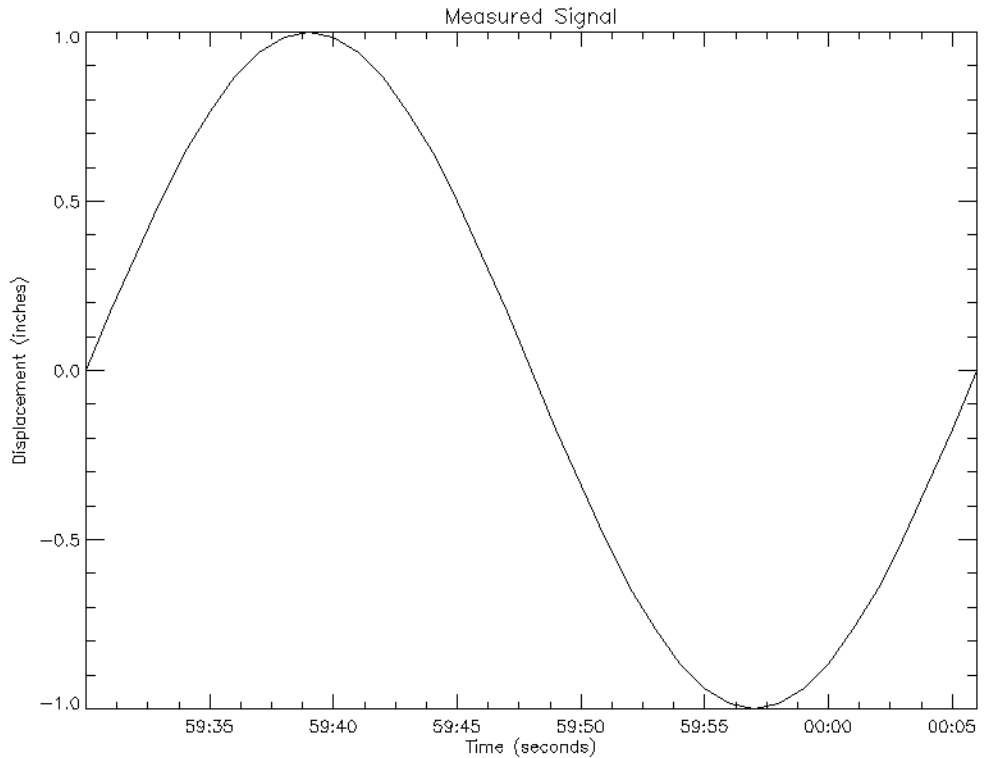


Figure 2-1: Displaying Date/Time data with PLOT

The above display shows the progression of the date/time variable, but it does not include all of the date/time data we generated with the TIMEGEN routine. This data also includes hour, month, day, and year information. IDL can display this information with additional levels to the date/time axis. You can control the number of levels to draw and the units used at each level with the XTICKUNITS keyword. You can specify the formatting for these levels by changing the DATE_FORMAT keyword setting to the LABEL_DATE routine:

```
date_label = LABEL_DATE(DATE_FORMAT = $
    ['%I:%S', '%H', '%D %M, %Y'])
```

where %H represents hours, %D represents days, %M represents months, and %Y represents years. Notice DATE_FORMAT is specified with a three element vector. Date/time data can be displayed on an axis with three levels. The format of these levels are specified through this vector.

In this example, the first level (closest to the axis) will contain minute and second values separated by a colon (%I:%S). The second level (just below the first level) will contain the hour values (%H). The third level (the final level farthest from the axis) will contain the day and month values separated by a space and year value separated from the day and month values by a comma (%D %M, %Y). For more information, see [LABEL_DATE](#) in the *IDL Reference Guide*.

Besides the above change to the LABEL_DATE routine, you must also change the settings of the keywords to the PLOT routine to specify a multiple level axis:

```
PLOT, date_time, displacement, /XSTYLE, $
    ; displaying titles.
    TITLE = 'Measured Signal', $
    XTITLE = 'Time (seconds)', $
    YTITLE = 'Displacement (inches)', $
    ; applying date/time formats to X-axis labels.
    POSITION = [0.2, 0.25, 0.9, 0.9], $
    XTICKFORMAT = ['LABEL_DATE', 'LABEL_DATE', 'LABEL_DATE'], $
    XTICKUNITS = ['Time', 'Hour', 'Day'], $
    XTICKINTERVAL = 5
```

The POSITION keyword is set to allow the resulting display to contain all three levels and the title of the date/time axis. The XTICKFORMAT is now set to a string array containing an element for each level of the axis. The XTICKUNITS keyword is set to note the unit of each level. These keyword settings produce the following results:

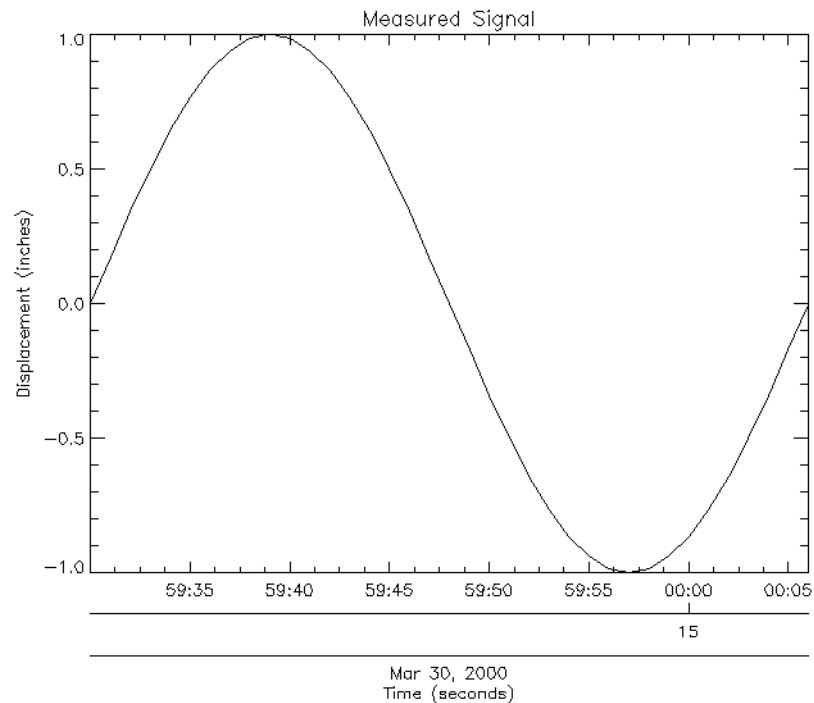


Figure 2-2: Displaying Three Levels of Date/Time data with PLOT

Notice the three levels of the x -axis. These levels are arranged as specified by the previous call to the LABEL_DATE routine.

Displaying Date/Time Data on a Contour Display

Another possible example may be the surface temperature (in degrees Celsius) of each degree of a single circle on a sphere recorded at every second for 37 seconds after the initial recording of 59 minutes and 30 seconds after 2 o'clock pm (14 hundred hours) on the 30th day of March in the year 2000:

```
number_samples = 37
date_time = TIMEGEN(number_samples, UNITS = 'Seconds', $
    START = JULDAY(3, 30, 2000, 14, 59, 30))
angle = 10.*FINDGEN(number_samples)
temperature = BYTSCL(SIN(10.*!DTOR* $
    FINDGEN(number_samples)) # COS(!DTOR*angle))
```

Since the final contour display will be filled, we should define a color table:

```
DEVICE, DECOMPOSED = 0
LOADCT, 5
```

The call to the `DEVICE` command with the `DECOMPOSED` keyword set to zero allows color tables to be used on TrueColor displays, which may be the default setting on some systems. The call to the `LOADCT` routine loads the Standard Gamma-II (number 5) color table, which is a part of IDL's libraries.

As with the one-dimensional case, the format of the date/time values is specified through the `LABEL_DATE` routine as follows

```
date_label = LABEL_DATE( DATE_FORMAT = $
    ['%I:%S', '%H', '%D %M, %Y'] )
```

where `%I` represents minutes, `%S` represents seconds, `%H` represents hours, `%D` represents days, `%M` represents months, and `%Y` represents years.

The first level (closest to the axis) will contain minute and second values separated by a colon (`%I:%S`). The second level (just below the first level) will contain the hour values (`%H`). The third level (the final level farthest from the axis) will contain the day and month values separated by a space and year value separated from the day and month values by a comma (`%D %M, %Y`).

The resulting format is specified by using the `CONTOUR` routine with the `XTICKFORMAT` keyword:

```
CONTOUR, temperature, angle, date_time, $
    ; specifying contour levels and fill colors.
    LEVELS = BYTSCL(INDGEN(8)), /XSTYLE, /YSTYLE, $
    C_COLORS = BYTSCL(INDGEN(8)), /FILL, $
    ; displaying titles.
    TITLE = 'Measured Temperature (degrees Celsius)', $
    XTITLE = 'Angle (degrees)', $
    YTITLE = 'Time (seconds)', $
    ; applying date/time formats to X-axis labels.
    POSITION = [0.2, 0.25, 0.9, 0.9], $
    YTICKFORMAT = ['LABEL_DATE', 'LABEL_DATE', 'LABEL_DATE'], $
    YTICKUNITS = ['Time', 'Hour', 'Day'], $
    YTICKINTERVAL = 5, $
    YTICKLAYOUT = 2
    ; Applying contour lines over the original contour display.
    CONTOUR, temperature, angle, date_time, /OVERPLOT, $
    LEVELS = BYTSCL(INDGEN(8))
```

As in the plot example, the `POSITION` keyword is set to allow the resulting display to contain all three levels and the title of the date/time axis. The `YTICKUNITS`

keyword is set to note the unit of each level. And the YTICKINTERVAL keyword is set to place the major tick marks at every five second interval.

This example also contains the YTICKLAYOUT keyword. By default, this keyword is set to 0, which provides the date/time layout shown in the plot example. In this example, YTICKLAYOUT is set to 2, which rotates and boxes the tick labels to provide the following results:

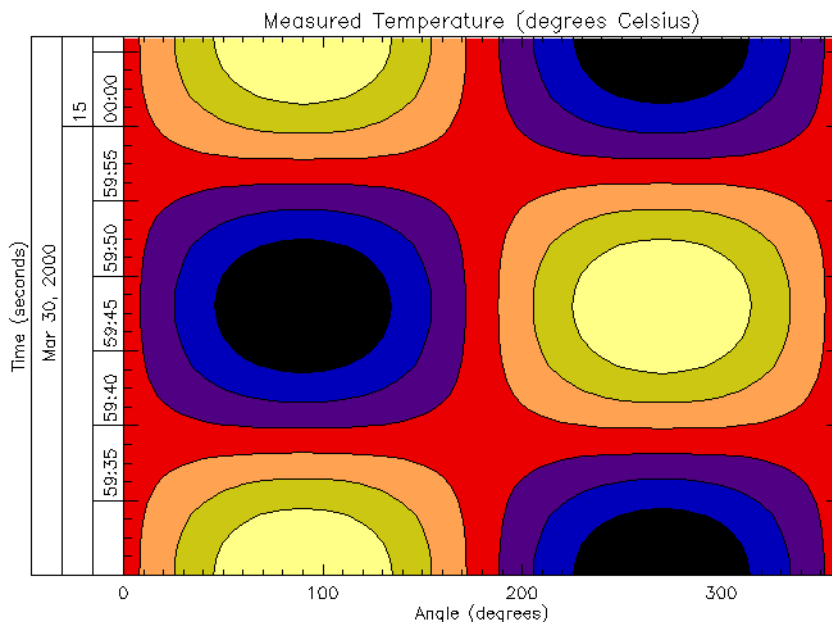


Figure 2-3: Displaying Date/Time Data with CONTOUR

Using System Variables to Display Date/Time Data

The settings we used to display our date/time data could have been specified through system variables instead of keywords. The following table shows the relationship between these keywords and their system variables:

Keywords	System Variables
[XYZ]TICKUNITS	![XYZ].TICKUNITS

Table 2-2: Relationship Between Keywords and System Variables

Keywords	System Variables
[XYZ]TICKINTERVAL	![XYZ].TICKINTERVAL
[XYZ]TICKLAYOUT	![XYZ].TICKLAYOUT

Table 2-2: Relationship Between Keywords and System Variables

Usually, keywords are used more frequently than system variables, but system variables are better when trying to establish a consistent display style. For example, we could have established a date/time axis style with these system variables before producing our previous displays:

```

; Establishing an axis style.
!X.TICKFORMAT = ['LABEL_DATE', 'LABEL_DATE', 'LABEL_DATE']
!X.TICKUNITS = ['Time', 'Hour', 'Day']
!X.TICKINTERVAL = 5
!X.TICKLAYOUT = 2
; Displaying data.
PLOT, date_time, displacement, /XSTYLE, $
    TITLE = 'Measured Signal', $
    XTITLE = 'Time (seconds)', $
    YTITLE = 'Displacement (inches)', $
    POSITION = [0.2, 0.7, 0.9, 0.9]
CONTOUR, temperature, date_time, angle, /FILL, $
    LEVELS = BYTSCL(INDGEN(8)), /XSTYLE, /YSTYLE, $
    C_COLORS = BYTSCL(INDGEN(8)), /NOERASE, $
    TITLE = 'Measured Temperature (degrees Celsius)', $
    XTITLE = 'Angle (degrees)', $
    YTITLE = 'Time (seconds)', $
    POSITION = [0.2, 0.25, 0.9, 0.45]
CONTOUR, temperature, date_time, angle, /OVERPLOT, $
    LEVELS = BYTSCL(INDGEN(8))
!X.TICKLAYOUT = 0
!X.TICKINTERVAL = 0
!X.TICKUNITS = ''
!X.TICKFORMAT = ''

```

Notice these system variables are set to their default values after the two displays are shown. When using system variables, instead of keywords, remember to reset them back to their default values. The above example produces the following results:

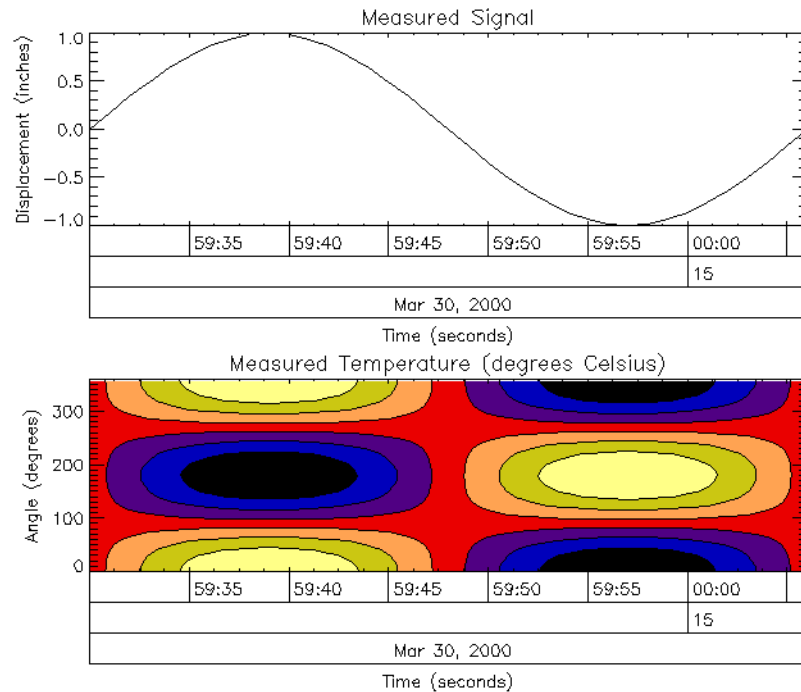


Figure 2-4: Date/Time Axis Style Established With System Variables

Displaying Date/Time Data on an Axis in Object Graphics

You can display date/time data on plots, contours, and surfaces through the tick settings of the date/time axis. Date/time data can be displayed on any axis (x , y or z). The date/time data is stored as Julian dates, but the `LABEL_DATE` routine and axis keywords allow you to display this data as calendar dates. The following examples show how to display one-dimensional and two-dimensional date/time data.

Displaying Date/Time Data on a Plot Display

Date/time data usually comes from measuring data values at specific times. For example, the displacement (in inches) of an object might be recorded at every second for 37 seconds after the initial recording of 59 minutes and 30 seconds after 2 o'clock pm (14 hundred hours) on the 30th day of March in the year 2000 as follows

```
number_samples = 37
date_time = TIMEGEN(number_samples, UNITS = 'Seconds', $
    START = JULDAY(3, 30, 2000, 14, 59, 30))
displacement = SIN(10.*!DTOR*FINDGEN(number_samples))
```

Normally, this type of data would be imported into IDL from a data file. However, this section is designed specifically to show how to display date/time data, not how to import data from a file; therefore, the data for this example is created with the above IDL commands.

Before displaying this one-dimensional data with the `IDLgrPlot` object, the format of the date/time values is specified through the `LABEL_DATE` routine:

```
date_label = LABEL_DATE(DATE_FORMAT = ['%I:%S'])
```

where `%I` represents minutes and `%S` represents seconds.

Before applying the results from `LABEL_DATE`, we must first create (initialize) our display objects:

```
oPlotWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = [800, 600])
oPlotView = OBJ_NEW('IDLgrView', /DOUBLE)
oPlotModel = OBJ_NEW('IDLgrModel')
oPlot = OBJ_NEW('IDLgrPlot', date_time, displacement, $
    /DOUBLE)
```

The `oPlotModel` object will contain the `IDLgrPlot` and `IDLgrAxis` objects. The `oPlotView` object contains the `oPlotModel` object with the `DOUBLE` keyword. The

DOUBLE keyword is set for the oPlotView and oPlot objects because the date/time data is made up of double-precision floating-point values.

Although the date/time part of the data will actually be contained and displayed through the IDLgrAxis object, the oPlot object is created first to provide a display region for the axes:

```
oPlot -> GetProperty, XRANGE = xr, YRANGE = yr
xs = NORM_COORD(xr)
xs[0] = xs[0] - 0.5
ys = NORM_COORD(yr)
ys[0] = ys[0] - 0.5
oPlot -> SetProperty, XCOORD_CONV = xs, YCOORD_CONV = ys
```

The NORM_COORD routine is used to create a normalized (0 to 1) display coordinate system. This coordinate system will also apply to the IDLgrAxis objects:

```
; X-axis title.
oTextXAxis = OBJ_NEW('IDLgrText', 'Time (seconds)')
; X-axis (date/time axis).
oPlotXAxis = OBJ_NEW('IDLgrAxis', 0, /EXACT, RANGE = xr, $
    XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextXAxis, $
    LOCATION = [xr[0], yr[0]], TICKDIR = 0, $
    TICKLEN = (0.02*(yr[1] - yr[0])), $
    TICKFORMAT = ['LABEL_DATE'], TICKINTERVAL = 5, $
    TICKUNITS = ['Time'])
; Y-axis title.
oTextYAxis = OBJ_NEW('IDLgrText', 'Displacement (inches)')
; Y-axis.
oPlotYAxis = OBJ_NEW('IDLgrAxis', 1, /EXACT, RANGE = yr, $
    XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextYAxis, $
    LOCATION = [xr[0], yr[0]], TICKDIR = 0, $
    TICKLEN = (0.02*(xr[1] - xr[0])))
; plot title.
oPlotText = OBJ_NEW('IDLgrText', 'Measured Signal', $
    LOCATIONS = [(xr[0] + xr[1])/2., $
    (yr[1] + (0.02*(yr[0] + yr[1])))], $
    XCOORD_CONV = xs, YCOORD_CONV = ys, $
    ALIGNMENT = 0.5)
```

The TICKFORMAT, TICKINTERVAL, and TICKUNITS keywords specify the X-axis as a date/time axis.

These objects are now added to the oPlotModel object and this model is added to the oPlotView object:

```
oPlotModel -> Add, oPlot
oPlotModel -> Add, oPlotXAxis
oPlotModel -> Add, oPlotYAxis
```

```
oPlotModel -> Add, oPlotText
oPlotView -> Add, oPlotModel
```

Now the `oPlotView` object, which contains all of these objects, can be viewed in the `oPlotWindow` object:

```
oPlotWindow -> Draw, oPlotView
```

The `Draw` method to the `oPlotWindow` object produces the following results:

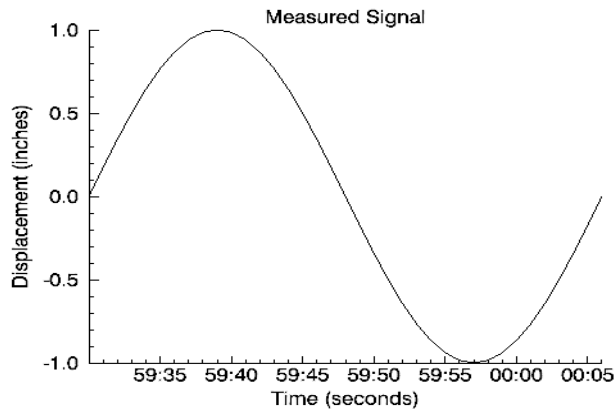


Figure 2-5: Displaying Date/Time data with IDLgrPlot

The above display shows the progression of the date/time variable, but it does not include all of the date/time data we generated with the `TIMEGEN` routine. This data also includes hour, month, day, and year information. IDL can display this information with additional levels to the date/time axis. You can control the number of levels to draw and the units used at each level with the `TICKUNITS` keyword. You can specify the formatting for these levels by changing the `DATE_FORMAT` keyword setting to the `LABEL_DATE` routine:

```
date_label = LABEL_DATE(DATE_FORMAT = $
    ['%I:%S', '%H', '%D %M, %Y'])
```

where `%H` represents hours, `%D` represents days, `%M` represents months, and `%Y` represents years. Notice `DATE_FORMAT` is specified with a three-element vector. Date/time data can be displayed on an axis with three levels. The format of these levels are specified through this vector.

In this example, the first level (closest to the axis) will contain minute and second values separated by a colon (`%I:%S`). The second level (just below the first level) will

contain the hour values (%H). The third level (the final level farthest from the axis) will contain the day and month values separated by a space and year value separated from the day and month values by a comma (%D %M, %Y). For more information, see [LABEL_DATE](#) in the *IDL Reference Guide*.

Besides the above change to the LABEL_DATE routine, we must also change the settings of the IDLgrAxis properties to specify a multiple level axis:

```
oPlotXAxis -> SetProperty, $
    TICKFORMAT = ['LABEL_DATE', 'LABEL_DATE', 'LABEL_DATE'], $
    TICKUNITS = ['Time', 'Hour', 'Day']
```

The TICKFORMAT is now set to a string array containing an element for each level of the axis. The TICKUNITS keyword is set to note the unit of each level. These property settings produce the following results:

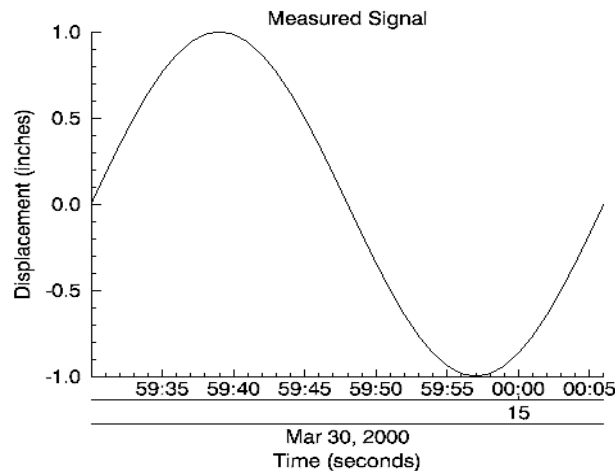


Figure 2-6: Displaying Three Levels of Date/Time data with IDLgrPlot

Notice the three levels of the X-axis. These levels are arranged as specified by the previous call to the LABEL_DATE routine.

To maintain IDL's memory, the object references for oPlotView, oTextXAxis, and oTextYAxis should be destroyed. Therefore, after the display is drawn, the OBJ_DESTROY routine should be called:

```
OBJ_DESTROY, [oPlotView, oTextXAxis, oTextYAxis]
```

The display will remain until closed, but the object references are now freed from IDL's memory.

Displaying Date/Time Data on a Contour Display

Another possible example may be the surface temperature (in degrees Celsius) of each degree of a single circle on a sphere recorded at every second for 37 seconds after the initial recording of 59 minutes and 30 seconds after 2 o'clock pm (14 hundred hours) on the 30th day of March in the year 2000

```
number_samples = 37
date_time = TIMEGEN(number_samples, UNITS = 'Seconds', $
    START = JULDAY(3, 30, 2000, 14, 59, 30))
angle = 10.*FINDGEN(number_samples)
temperature = BYTSCL(SIN(10.*!DTOR* $
    FINDGEN(number_samples))) # COS(!DTOR*angle))
```

As with the one-dimensional case, the format of the date/time values is specified through the LABEL_DATE routine as follows

```
date_label = LABEL_DATE(DATE_FORMAT = $
    ['%I:%S', '%H', '%D %M, %Y'])
```

where %I represents minutes, %S represents seconds, %H represents hours, %D represents days, %M represents months, and %Y represents years.

The first level (closest to the axis) will contain minute and second values separated by a colon (%I:%S). The second level (just below the first level) will contain the hour values(%H). The third level (the final level farthest from the axis) will contain the day and month values separated by a space and year value separated from the day and month values by a comma (%D %M, %Y).

Since the final contour display will be filled, we should define a color palette:

```
oContourPalette = OBJ_NEW('IDLgrPalette')
oContourPalette -> LoadCT, 5
```

As in the one-dimensional example, the display must be initialized:


```

oContourWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = [800, 600])
oContourView = OBJ_NEW('IDLgrView', /DOUBLE)
oContourModel = OBJ_NEW('IDLgrModel')
oContour = OBJ_NEW('IDLgrContour', temperature, $
    GEOMX = angle, GEOMY = date_time, GEOMZ = 0., $
    /PLANAR, /FILL, PALETTE = oContourPalette, $
    /DOUBLE_GEOM, C_VALUE = BYTSCL(INDGEN(8)), $
    C_COLOR = BYTSCL(INDGEN(8)))
; Applying contour lines over the original contour display.
oContourLines = OBJ_NEW('IDLgrContour', temperature, $
    GEOMX = angle, GEOMY = date_time, GEOMZ = 0.001, $
    /PLANAR, /DOUBLE_GEOM, C_VALUE = BYTSCL(INDGEN(8)))

```

The `oContourModel` object will contain the `IDLgrContour` and `IDLgrAxis` objects. The `oContourView` object contains the `oContourModel` with the `DOUBLE` keyword. The `DOUBLE` and `DOUBLE_GEOM` keywords are set for the `oContourView` and `oContour` objects because date/time data is made up of double-precision floating-point values.

Although the date/time part of the data will actually be contained and displayed through the `IDLgrAxis` object, the `oContour` object is created first to provide a display region for the axes:

```

oContour -> GetProperty, XRange = xr, YRange = yr, ZRange = zr
xs = NORM_COORD(xr)
xs[0] = xs[0] - 0.5
ys = NORM_COORD(yr)
ys[0] = ys[0] - 0.5
oContour -> SetProperty, XCOORD_CONV = xs, YCOORD_CONV = ys
oContourLines -> SetProperty, XCOORD_CONV = xs, YCOORD_CONV = ys

```

The `oContourLines` object is created to display contour lines over the filled contours. Note these lines have a `GEOMZ` difference of 0.001 from the filled contours. This difference is provided to display the lines over the filled contours and not in the same view plane. The `NORM_COORD` routine is used to create a normalized (0 to 1) display coordinate system. This coordinate system will also apply to the `IDLgrAxis` objects:

```

; X-axis title.
oTextXAxis = OBJ_NEW('IDLgrText', 'Angle (degrees)')
; X-axis.
oContourXAxis = OBJ_NEW('IDLgrAxis', 0, /EXACT, RANGE = xr, $
    XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextXAxis, $
    LOCATION = [xr[0], yr[0], zr[0] + 0.001], TICKDIR = 0, $
    TICKLEN = (0.02*(yr[1] - yr[0])))
; Y-axis title.
oTextYAxis = OBJ_NEW('IDLgrText', 'Time (seconds)')

```

```

; Y-axis (date/time axis).
oContourYAxis = OBJ_NEW('IDLgrAxis', 1, /EXACT, RANGE = yr, $
    XCOORD_CONV = xs, YCOORD_CONV = ys, TITLE = oTextYAxis, $
    LOCATION = [xr[0], yr[0], zr[0] + 0.001], TICKDIR = 0, $
    TICKLEN = (0.02*(xr[1] - xr[0])), $
    TICKFORMAT = ['LABEL_DATE', 'LABEL_DATE', 'LABEL_DATE'], $
    TICKUNITS = ['Time', 'Hour', 'Day'], $
    TICKLAYOUT = 2)
oContourText = OBJ_NEW('IDLgrText', $
    'Measured Temperature (degrees Celsius)', $
    LOCATIONS = [(xr[0] + xr[1])/2., $
        (yr[1] + (0.02*(yr[0] + yr[1])))], $
    XCOORD_CONV = xs, YCOORD_CONV = ys, $
    ALIGNMENT = 0.5)

```

The **TICKFORMAT**, **TICKINTERVAL**, and **TICKUNITS** keywords specify the Y-axis as a date/time axis, which contains three levels related to the formats presented in the call to the **LABEL_DATE** routine. This example also contains the **TICKLAYOUT** keyword. By default, this keyword is set to 0, which provides the date/time layout shown in the plot example. In this example, **TICKLAYOUT** is set to 2, which rotates and boxes the tick labels.

These objects are now added to the **oContourModel** object and this model is added to the **oContourView** object:

```

oContourModel -> Add, oContour
oContourModel -> Add, oContourLines
oContourModel -> Add, oContourXAxis
oContourModel -> Add, oContourYAxis
oContourModel -> Add, oContourText
oContourView -> Add, oContourModel

```

Now the **oContourView** object, which contains all of these objects, can be viewed in the **oContourWindow** object:

```

oContourWindow -> Draw, oContourView

```

The **Draw** method to **oContourWindow** produces the following results:

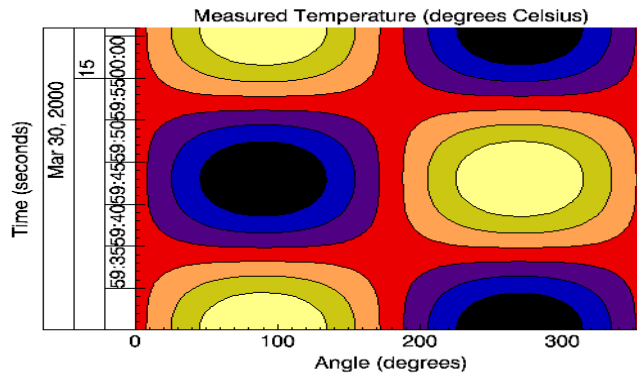


Figure 2-7: Displaying Date/Time data with IDLgrContour

Notice the three levels of the y-axis. These levels are arranged as specified by the previous call to the LABEL_DATE routine.

To maintain IDL's memory, the object references for oContourView, oContourPalette, oTextXAxis, and oTextYAxis should be destroyed. Therefore, after the display is drawn, the OBJ_DESTROY routine should be called:

```
OBJ_DESTROY, [oContourView, oContourPalette, $
              oTextXAxis, oTextYAxis]
```

The display will remain until closed, but the object references are now freed from IDL's memory.



Chapter 3: New IDL Routines

This chapter describes IDL Routines introduced in IDL version 5.4.

- [ARRAY_EQUAL](#)
- [BESELK](#)
- [BREAK](#)
- [COLORMAP_APPLICABLE](#)
- [CONTINUE](#)
- [FILE_CHMOD](#)
- [FILE_DELETE](#)
- [FILE_EXPAND_PATH](#)
- [FILE_MKDIR](#)
- [FILE_TEST](#)
- [FILE_WHICH](#)
- [HOUGH](#)
- [LAGUERRE](#)
- [LEGENDRE](#)
- [MAKE_DLL](#)
- [MAP_2POINTS](#)
- [MATRIX_MULTIPLY](#)
- [MEMORY](#)
- [RADON](#)
- [SAVGOL](#)
- [SOCKET](#)
- [SPHER_HARM](#)
- [SWITCH](#)
- [TIMEGEN](#)
- [WV_CWT](#)
- [WV_DENOISE](#)
- [WV_FN_GAUSSIAN](#)
- [WV_FN_MORLET](#)
- [WV_FN_PAUL](#)
- [XDXF](#)
- [XPCOLOR](#)
- [XPLOT3D](#)
- [XROI](#)
- [XVOLUME](#)

ARRAY_EQUAL

The `ARRAY_EQUAL` function is a fast way to compare data for equality in situations where the index of the elements that differ are not of interest. This operation is much faster than using `TOTAL(A NE B)`, because it stops the comparison as soon as the first inequality is found, an intermediate array is not created, and only one pass is made through the data. For best speed, ensure that the operands are of the same data type.

Arrays may be compared to scalars, in which case each element is compared to the scalar. For two arrays to be equal, they must have the same number of elements. If the types of the operands differ, the type of the least precise is converted to that of the most precise, unless the `NO_TYPECONV` keyword is specified to prevent it. This function works on all numeric types and strings.

Syntax

Result = `ARRAY_EQUAL(Op1 , Op2 [, /NO_TYPECONV])`

Return Value

Returns 1 (true) if, and only if, all elements of *Op1* are equal to *Op2*; returns 0 (false) at the first instance of inequality.

Arguments

Op1, Op2

The variables to be compared.

Keywords

NO_TYPECONV

By default, `ARRAY_EQUAL` converts operands of different types to a common type before performing the equality comparison. Set `NO_TYPECONV` to disallow this implicit type conversion. If `NO_TYPECONV` is specified, operands of different types are never considered to be equal, even if their numeric values are the same.

Example

```
; Return True (1) if all elements of a are equal to a 0 byte:
IF ARRAY_EQUAL(a, 0b) THEN ...
; Return True (1) if all elements of a are equal all elements of b:
IF ARRAY_EQUAL(a, b) THEN ...
```

BESELK

The BESELK function returns the K Bessel function of order N for the argument X . The BESELK function is adapted from “SPECFUN - A Portable FORTRAN Package of Special Functions and Test Drivers”, W. J. Cody, Algorithm 715, *ACM Transactions on Mathematical Software*, Vol 19, No. 1, March 1993.

Syntax

Result = BESELK(X , N)

Return Value

If X is double-precision, the result is double precision, otherwise the argument is converted to floating-point and the result is floating-point.

Arguments

X

The expression for which the K Bessel function is required. The result will have the same dimensions as X .

N

The order of the K Bessel function to calculate. N should be greater than or equal to 0 and less than 20, and can be either an integer or a real number.

Keywords

None

Example

The following example plots the I and K Bessel functions for orders 0, 1 and 2:

```
X = FINDGEN(40)/10

;Plot I and K Bessel Functions:
PLOT, X, BESELI(X, 0), MAX_VALUE=4, $
  TITLE = 'I and K Bessel Functions'
OPLOT, X, BESELI(X, 1)
OPLOT, X, BESELI(X, 2)
OPLOT, X, BESELK(X, 0), LINESSTYLE=2
OPLOT, X, BESELK(X, 1), LINESSTYLE=2
```

```

OPLOT, X, BESELK(X, 2), LINESTYLE=2

;Annotate plot:
xcoords = [.18, .45, .95, 1.4, 1.8, 2.4]
ycoords = [2.1, 2.1, 2.1, 1.8, 1.6, 1.4]
labels = ['!8K!X!D0', '!8K!X!D1', '!8K!X!D2', '!8I!X!D0',
          '!8I!X!D1', '!8I!X!D2']
XYOUTS, xcoords, ycoords, labels, /DATA

```

This results in the following plot:

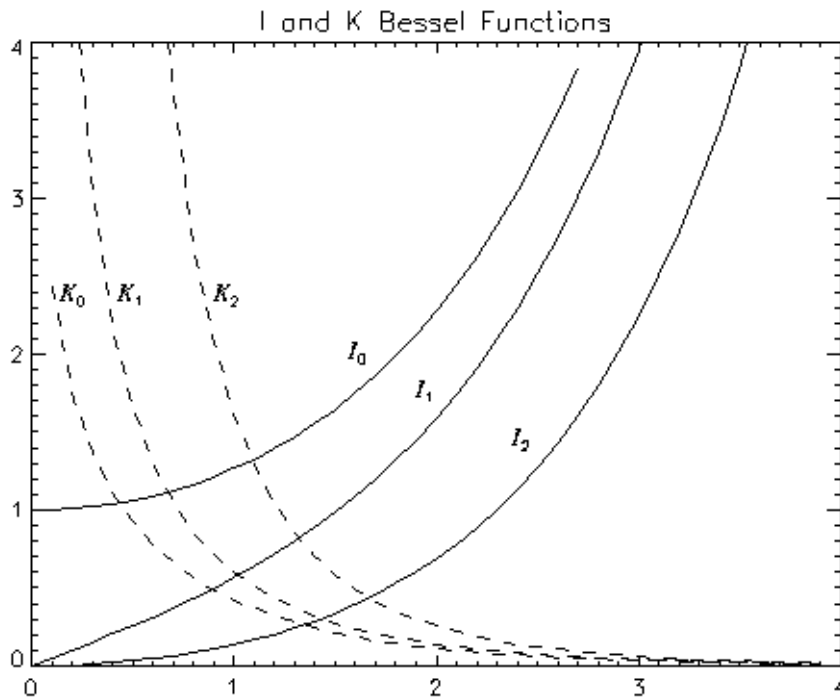


Figure 3-1: *I and K Bessel Functions.*

See Also

[BESELI](#), [BESELJ](#), [BESELY](#)

BREAK

The BREAK statement provides a convenient way to immediately exit from a loop (FOR, WHILE, REPEAT), CASE, or SWITCH statement without resorting to GOTO statements.

Note

BREAK is an IDL statement. For information on using statements, see [Chapter 11](#), “Program Control” in *Building IDL Applications*.

Syntax

BREAK

Example

This example exits the enclosing WHILE loop when the value of i hits 5.

```
I = 0
WHILE (1) DO BEGIN
    i = i + 1
    IF (i eq 5) THEN BREAK
ENDWHILE
```

COLORMAP_APPLICABLE

The COLORMAP_APPLICABLE function determines whether the current visual class supports the use of a colormap, and if so, whether colormap changes affect pre-displayed Direct Graphics or if the graphics must be redrawn to pick up colormap changes.

This routine is written in the IDL language. Its source code can be found in the file `colormap_applicable.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = COLORMAP_APPLICABLE(*redrawRequired*)

Return Value

The function returns a long value of 1 if the current visual class allows modification of the color table, and 0 otherwise.

Arguments

redrawRequired

A named variable to retrieve a value indicating whether the visual class supports automatic updating of graphics. The value is 0 if the graphics are updated automatically, or 1 if the graphics must be redrawn to pick up changes to the colormap.

Keywords

None.

Example

To determine whether to redisplay an image after a colormap change:

```
result = COLORMAP_APPLICABLE(redrawRequired)
IF ((result GT 0) AND (redrawRequired GT 0)) THEN BEGIN
    my_redraw
ENDIF
```

CONTINUE

The CONTINUE statement provides a convenient way to immediately start the next iteration of the enclosing FOR, WHILE, or REPEAT loop.

Note

Do not confuse the CONTINUE statement described here with the .CONTINUE executive command. The two constructs are not related, and serve completely different purposes.

Note

CONTINUE is not allowed within CASE or SWITCH statements. This is in contrast with the C language, which does allow this.

For more information on using CONTINUE and other IDL program control statements, see [Chapter 11, “Program Control”](#) in *Building IDL Applications*.

Syntax

CONTINUE

Example

This example presents one way (not necessarily the best) to print the even numbers between 1 and 10.

```
FOR I = 1,10 DO BEGIN
    ; If odd, start next iteration:
    IF (I AND 1) THEN CONTINUE
    PRINT, I
ENDFOR
```

FILE_CHMOD

The `FILE_CHMOD` procedure allows you to change the current access permissions (sometimes known as modes on UNIX platforms) associated with a file or directory. File modes are specified using the standard Posix convention of three protection classes (user, group, other), each containing three attributes (read, write, execute). These permissions can be specified as an octal bitmask in which desired permissions have their associated bit set and unwanted ones have their bits cleared. This is the same format familiar to users of the UNIX `chmod(1)` command).

Keywords are available to specify permissions without the requirement to specify a bitmask, providing a simpler way to handle many situations. All of the keywords share a similar behavior: Setting them to a non-zero value adds the specified permission to the *Mode* argument. Setting the keyword to 0 removes that permission.

To find the current protection settings for a given file, you can use the `GET_MODE` keyword to the `FILE_TEST` function.

Syntax

```
FILE_CHMOD, File [, Mode] [, /A_EXECUTE | /A_READ | /A_WRITE]
[, /G_EXECUTE | /G_READ | /G_WRITE]
[, /O_EXECUTE | /O_READ | /O_WRITE]
[, /U_EXECUTE | /U_READ | /U_WRITE]
```

UNIX-Only Keywords: [, /SETGID] [, /SETUID] [, /STICKY_BIT]

Arguments

File

A scalar or array of file or directory names for which protection modes will be changed.

Mode

An optional bit mask specifying the absolute protection settings to be applied to the files. If *Mode* is not supplied, `FILE_CHMOD` looks up the current modes for the file and uses it instead. Any additional modes specified via keywords are applied relative to the value in *Mode*. Setting a keyword adds the necessary mode bits to *Mode*, and clearing it by explicitly setting a keyword to 0 removes those bits from *Mode*.

The values of the bits in these masks correspond to those used by the UNIX `chmod(2)` system call and `chmod(1)` user command, and are given in the following

table. Since these bits are usually manipulated in groups of three, octal notation is commonly used when referring to them. When constructing a mode, the following platform specific considerations should be kept in mind:

- The `setuid`, `setgid`, and sticky bits are specific to the UNIX operating system, and have no meaning elsewhere. `FILE_CHMOD` ignores them on non-UNIX systems. The UNIX kernel may quietly refuse to set the sticky bit if you are not the root user. Consult the `chmod(2)` man page for details.
- The VMS operating system has four permission classes, unlike the 3 supported by UNIX. Furthermore, each class has an additional bit (DELETE) not supported by UNIX. IDL uses the C runtime library `chmod()` function supplied by the operating system to translate between the UNIX convention used by IDL and the native VMS permission masks. It maps the VMS SYSTEM and OWNER classes to the user class, GROUP to group, and WORLD to other. The DELETE bit is combined with the WRITE bit.
- The Microsoft Windows and Macintosh operating systems do not have 3 permission classes like UNIX does. Therefore, setting for all three classes are combined into a single request.
- The Microsoft Windows and Macintosh operating systems always allow read access to any files visible to a program. `FILE_CHMOD` therefore ignores any requests to remove read access.
- The Microsoft Windows and Macintosh operating systems do not maintain an execute bit for their files. Windows uses the file suffix to decide if a file is executable, and Macintosh IDL only considers files of type APPL to be executable. Therefore, `FILE_CHMOD` cannot change the execution status of a file on these platforms. Such requests are quietly ignored.

Bit	Octal Mask	Meaning
12	'4000'o	Setuid: Set user ID on execution.
11	'2000'o	Setgid: Set group ID on execution.
10	'1000'o	Turn on sticky bit. See the UNIX documentation on <code>chmod(2)</code> for details.
9	'0400'o	Allow read by owner.
8	'0200'o	Allow write by owner.

Table 3-1: UNIX `chmod(2)` mode bits

Bit	Octal Mask	Meaning
7	'0100' o	Allow execute by owner.
6	'0040' o	Allow read by group.
5	'0020' o	Allow write by group.
4	'0010' o	Allow execute by group.
3	'0004' o	Allow read by others.
2	'0002' o	Allow write by others.
1	'0001' o	Allow execute by others.

Table 3-1: UNIX chmod(2) mode bits

Keywords

A_EXECUTE

Execute access for all three (user, group, other) categories.

A_READ

Read access for all three (user, group, other) categories.

A_WRITE

Write access for all three (user, group, other) categories.

G_EXECUTE

Execute access for the group category.

G_READ

Read access for the group category.

G_WRITE

Write access for the group category.

O_EXECUTE

Execute access for the other category.

O_READ

Read access for the other category.

O_WRITE

Write access for the other category.

U_EXECUTE

Execute access for the user category.

U_READ

Read access for the user category.

U_WRITE

Write access for the user category.

UNIX-Only Keywords**SETGID**

The Set Group ID bit.

SETUID

The Set User ID bit.

STICKY_BIT

Sets the sticky bit.

Example

In the first example, we make the file `moose.dat` read only to everyone except the owner of the file, but not change any other settings:

```
FILE_CHMOD, 'moose.dat', /U_WRITE, G_WRITE=0, O_WRITE=0
```

In the next example, we make the file readable and writable to the owner and group, but read-only to anyone else, and remove any other modes:

```
FILE_CHMOD, 'moose.dat', '664'&
```

FILE_DELETE

The FILE_DELETE procedure deletes a file or empty directory, if the process has the necessary permissions to remove the file as defined by the current operating system. FILE_CHMOD can be used to change file protection settings.

Syntax

```
FILE_DELETE, File1 [... FileN] [, /QUIET]
```

Arguments

FileN

A scalar or array of file or directory names to be deleted, one name per string element. Directories must be specified in the native syntax for the current operating system. See “Operating System Syntax” below for additional details.

Keywords

QUIET

FILE_DELETE will normally issue an error if it is unable to remove a requested file or directory. If QUIET is set, no error is issued and FILE_DELETE simply moves on to the next requested item.

Operating System Syntax

The syntax used to specify directories for removal depends on the operating system in use, and is in general the same as you would use when issuing commands to the operating system command interpreter.

Microsoft Windows users must be careful to not specify a trailing backslash at the end of a specification. For example:

```
FILE_DELETE, 'c:\mydir\myfile'
```

and not:

```
FILE_DELETE, 'c:\mydir\myfile\'
```

For VMS users, the syntax for creating a subdirectory (as with the CREATE/DIRECTORY DCL command) is not symmetric with that used to delete it (using the DELETE,/DIRECTORY). FILE_DELETE follows the same rules. For

instance, to create a subdirectory of the current working directory named `bullwinkle` and then remove it:

```
FILE_MKDIR, ['.bullwinkle']  
FILE_DELETE, 'bullwinkle.dir'
```

Example

In this example, we remove an empty directory named `moose`. On the Macintosh, UNIX, or Windows operating systems:

```
FILE_DELETE, 'moose'
```

To do the same thing under VMS:

```
FILE_DELETE, 'moose.dir'
```

FILE_EXPAND_PATH

The FILE_EXPAND_PATH function expands a given file or partial directory name to its fully qualified name regardless of the current working directory.

Note

This routine should be used only to make sure that file paths are fully qualified, but not to expand wildcard characters (e.g. *). The behavior of FILE_EXPAND_PATH when it encounters a wildcard is platform dependent, and should not be depended on. These differences are due to the underlying operating system, and are beyond IDL's control. To expand wildcards and obtain fully qualified paths, combine the FINDFILE function with FILE_EXPAND_PATH:

```
A = FILE_EXPAND_PATH(FINDFILE( '*.pro' ))
```

Syntax

Result = FILE_EXPAND_PATH (*Path*)

Return Value

FILE_EXPAND_PATH returns a fully qualified file path that completely specifies the location of *Path* without the need to consider the user's current working directory.

Arguments

Path

A scalar or array of file or directory names to be fully qualified.

Keywords

None.

Example

In this example, we change directories to the IDL lib directory and expand the file path for the DIST function:

```
cd, FILEPATH(' ', SUBDIRECTORY=['lib'])  
print, FILE_EXPAND_PATH('dist.pro')
```

This results in the following if run on a UNIX system:

```
/usr/local/rsi/idl_5.4/lib/dist.pro
```

See Also

[FINDFILE](#)

FILE_MKDIR

The FILE_MKDIR procedure creates a new directory, or directories, with the default access permissions for the current process.

Note

Use the FILE_CHMOD procedure to alter access permissions.

If a specified directory has non-existent parent directories, FILE_MKDIR automatically creates all the intermediate directories as well.

Syntax

FILE_MKDIR, *File1* [... *FileN*]

Arguments

FileN

A scalar or array of directory names to be created, one name per string element. Directories must be specified in the native syntax for the current operating system.

Keywords

None.

Example

To create a subdirectory named `moose` in the current working directory on the Macintosh, UNIX, or Windows operating systems:

```
FILE_MKDIR, 'moose'
```

To do the same thing under VMS:

```
FILE_MKDIR, '[.moose]'
```

FILE_TEST

The FILE_TEST function checks files for existence and other attributes without having to first open the file.

Syntax

Result = FILE_TEST(*File* [, /DIRECTORY | , /EXECUTABLE | , /READ | , /REGULAR | , /WRITE | , /ZERO_LENGTH] [, GET_MODE=*variable*])

UNIX-Only Keywords: [, /BLOCK_SPECIAL | , /CHARACTER_SPECIAL | , /DANGLING_SYMLINK | , /NAMED_PIPE | , /SETGID | , /SETUID | , /SOCKET | , /STICKY_BIT | , /SYMLINK]

UNIX and VMS-Only Keywords: [, /GROUP | , /USER]

Return Value

FILE_TEST returns 1 (true), if the specified file exists and all of the attributes specified by the keywords are also true. If no keywords are present, a simple test for existence is performed. If the file does not exist or one of the specified attributes is not true, then FILE_TEST returns 0 (false).

Arguments

File

A scalar or array of file names to be tested. The result is of type integer with the same number of elements as *File*.

Keywords

DIRECTORY

Set this keyword to return 1 (true) if *File* exists and is a directory.

EXECUTABLE

Set this keyword to return 1 (true) if *File* exists and is executable. The source of this information differs between operating systems:

- UNIX and VMS: IDL checks the per-file information (the execute bit) maintained by the operating system.

- Microsoft Windows: The determination is made on the basis of the file name extension (e.g. .exe).
- Macintosh: Files of type ‘APPL’ (proper applications) are reported as executable. This corresponds to “Double Clickable” applications.

GET_MODE

Set this keyword to a named variable to receive the UNIX style mode (permission) mask for the specified file. The bits in these masks correspond to those used by the UNIX `chmod(2)` system call, and are explained in detail in the description of the *Mode* argument to the [FILE_CHMOD](#) procedure. When interpreting the value returned by this keyword, the following platform specific details should be kept in mind:

- The `setuid`, `setgid`, and sticky bits are specific to the UNIX operating system, and will never be returned on any other platform. Consult the `chmod(2)` man page and/or other UNIX programming documentation for more details.
- The VMS operating system has four permission classes, unlike the three supported by UNIX. Furthermore, each class has an additional bit (DELETE) not supported by UNIX. IDL uses the C runtime library `stat()` function supplied by the operating system to translate between the UNIX convention used by IDL and the native VMS permission masks. It maps the VMS OWNER to the user class, GROUP to group, and WORLD to other. The DELETE bit is combined with the WRITE bit.
- The Microsoft Windows and Macintosh operating systems do not have 3 permission classes like UNIX does. Therefore, IDL returns the same settings for all three classes.
- The Microsoft Windows and Macintosh operating systems do not maintain an execute bit for their files. Windows uses the file suffix to decide if a file is executable, and Macintosh IDL only considers files of type ‘APPL’ to be executable.

READ

Set this keyword to return 1 (true) if *File* exists and is readable by the user.

REGULAR

Set this keyword to return 1 (true) if *File* exists and is a regular disk file and not a directory, pipe, socket, or other special file type.

WRITE

Set this keyword to return 1 (true) if *File* exists and is writable by the user.

ZERO_LENGTH

Set this keyword to return 1 (true) if *File* exists and has zero length.

Note

The length of a directory is highly system dependent and does not necessarily correspond to the number of files it contains. In particular, it is possible for an empty directory to report a non-zero length. RSI does not recommend using the ZERO_LENGTH keyword on directories, as the information returned cannot be used in a meaningful way.

UNIX-Only Keywords**BLOCK_SPECIAL**

Set this keyword to return 1 (true) if *File* exists and is a block special device.

CHARACTER_SPECIAL

Set this keyword to return 1 (true) if *File* exists and is a character special device.

DANGLING_SYMLINK

Set this keyword to return 1 (true) if *File* is a symbolic link that points at a non-existent file.

NAMED_PIPE

Set this keyword to return 1 (true) if *File* exists and is a named pipe (fifo) device.

SETGID

Set this keyword to return 1 (true) if *File* exists and has its Set-Group-ID bit set.

SETUID

Set this keyword to return 1 (true) if *File* exists and has its Set-User-ID bit set.

SOCKET

Set this keyword to return 1 (true) if *File* exists and is a UNIX domain socket.

STICKY_BIT

Set this keyword to return 1 (true) if *File* exists and has its sticky bit set.

SYMLINK

Set this keyword to return 1 (true) if *File* exists and is a symbolic link that points at an existing file.

UNIX and VMS-Only Keywords**GROUP**

Set this keyword to return 1 (true) if *File* exists and belongs to the same effective group ID (GID) as the IDL process.

USER

Set this keyword to return 1 (true) if *File* exists and belongs to the same effective user ID (UID) as the IDL process.

Example

Does my IDL distribution support the IRIX operating system?

```
result = FILE_TEST(!DIR + '/bin/bin.sgi', /DIRECTORY)
PRINT, 'IRIX IDL Installed: ', result ? 'yes' : 'no'
```


FILE_WHICH

The FILE_WHICH function separates a specified file path into its component directories, and searches each directory in turn for a specific file. This command is modeled after the UNIX `which(1)` command.

This routine is written in the IDL language. Its source code can be found in the file `file_which.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = FILE_WHICH( [Path, ] File [, /INCLUDE_CURRENT_DIR] )
```

Return Value

Returns the path for the first file for the given name found by searching the specified path. If FILE_WHICH does not find the desired file, a NULL string is returned.

Arguments

Path

A search path to be searched. If *Path* is not present, the value of the IDL `!PATH` system variable is used.

File

The file to look for in the directories given by *Path*.

Keywords

INCLUDE_CURRENT_DIR

If set, FILE_WHICH looks in the current directory before starting to search *Path* for *File*. When IDL searches for a routine to compile, it looks in the current working directory before searching `!PATH`. The INCLUDE_CURRENT_DIR keyword allows FILE_WHICH to mimic this behavior.

Example

To find the location of this routine:

```
Result = FILE_WHICH('file_which.pro')
```

To find the location of the UNIX `ls` command:

```
Result = FILE_WHICH(getenv('PATH'), 'ls')
```

HOUGH

The HOUGH function implements the Hough transform, used to detect straight lines within a two-dimensional image. This function can be used to return either the Hough transform, which transforms each nonzero point in an image to a sinusoid in the Hough domain, or the Hough backprojection, where each point in the Hough domain is transformed to a straight line in the image.

Syntax

Hough Transform:

```
Result = HOUGH( Array [, /DOUBLE] [, DRHO=scalar] [, DX=scalar]
[, DY=scalar] [, /GRAY] [, NRHO=scalar] [, NTHETA=scalar] [, RHO=variable]
[, RMIN=scalar] [, THETA=variable] [, XMIN=scalar] [, YMIN=scalar] )
```

Hough Backprojection:

```
Result = HOUGH( Array, /BACKPROJECT, RHO=variable, THETA=variable
[, /DOUBLE] [, DX=scalar] [, DY=scalar] [, NX=scalar] [, NY=scalar]
[, XMIN=scalar] [, YMIN=scalar] )
```

Return Value

The result of this function is a two-dimensional floating-point array, or a complex array if the input image is complex. If *Array* is double-precision, or if the DOUBLE keyword is set, the result is double-precision, otherwise, the result is single-precision.

Hough Transform Theory

The Hough transform is defined for a function $A(x, y)$ as:

$$H(\theta, \rho) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} A(x, y) \delta(\rho - x \cos \theta - y \sin \theta) dx dy$$

where δ is the Dirac delta-function. With $A(x, y)$, each point (x, y) in the original image, A , is transformed into a sinusoid $\rho = x\cos\theta - y\sin\theta$, where ρ is the perpendicular distance from the origin of a line at an angle θ :

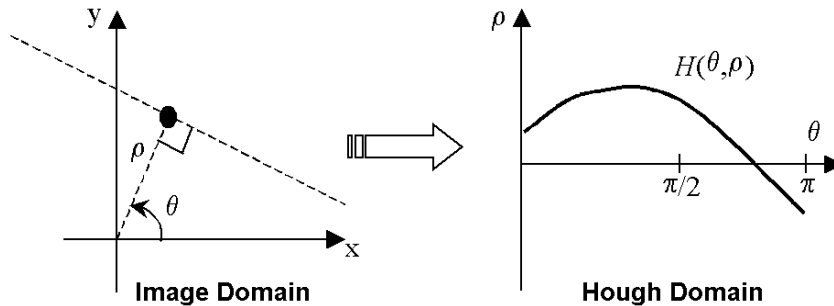


Figure 3-2: Hough Transform

Points that lie on the same line in the image will produce sinusoids that all cross at a single point in the Hough transform. For the inverse transform, or backprojection, each point in the Hough domain is transformed into a straight line in the image.

Usually, the Hough function is used with binary images, in which case $H(\theta, \rho)$ gives the total number of sinusoids that cross at point (θ, ρ) , and hence, the total number of points making up the line in the original image. By choosing a threshold T for $H(\theta, \rho)$, and using the inverse Hough function, you can filter the original image to keep only lines that contain at least T points.

How IDL Implements the Hough Transform

Consider an image A_{mn} of dimensions M by N , with array indices $m = 0, \dots, M-1$ and $n = 0, \dots, N-1$.

The discrete formula for the HOUGH function for A_{mn} is:

$$H(\theta, \rho) = \sum_m \sum_n A_{mn} \delta(\rho, [\rho'])$$

where the brackets $[]$ indicate rounding to the nearest integer, and

$$\rho' = (m\Delta x + x_{\min})\cos\theta + (n\Delta y + y_{\min})\sin\theta$$

The pixels are assumed to have spacing Δx and Δy in the x and y directions. The delta-function is defined as:

$$\delta(\rho, [\rho']) = \begin{cases} 1 & \rho = [\rho'] \\ 0 & \text{otherwise} \end{cases}$$

How IDL Implements the Hough Backprojection

The backprojection, B_{mn} , contains all of the straight lines given by the (θ, ρ) points given in $H(\theta, \rho)$. The discrete formula is

$$B_{mn} = \begin{cases} \sum_{\theta} \sum_{\rho} H(\theta, \rho) \delta(n, [am + b]) & |\sin \theta| > \frac{\sqrt{2}}{2} \\ \sum_{\theta} \sum_{\rho} H(\theta, \rho) \delta(m, [a'n + b']) & |\sin \theta| \leq \frac{\sqrt{2}}{2} \end{cases}$$

where the slopes and offsets are given by:

$$a = -\frac{\Delta x}{\Delta y} \frac{\cos \theta}{\sin \theta} \qquad b = \frac{\rho - x_{\min} \cos \theta - y_{\min} \sin \theta}{\Delta y \sin \theta}$$

$$a' = \frac{1}{a} \qquad b' = \frac{\rho - x_{\min} \cos \theta - y_{\min} \sin \theta}{\Delta x \cos \theta}$$

Arguments

Array

The two-dimensional array of size M by N which will be transformed. If the keyword GRAY is not set, then, for the forward transform, *Array* is treated as a binary image with all nonzero pixels considered as 1.

Keywords

BACKPROJECT

If set, the backprojection is computed, otherwise, the forward transform is computed. When BACKPROJECT is set, *Result* will be an array of dimension NX by NY .

Note

The Hough transform is not one-to-one: each point (x, y) is not mapped to a single (θ, ρ). Therefore, instead of the original image, the backprojection, or inverse transform, returns an image containing the set of all lines given by the (θ, ρ) points.

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

DRHO

Set this keyword equal to a scalar specifying the spacing $\Delta\rho$ between ρ coordinates, expressed in the same units as *Array*. The default is $1/\text{SQRT}(2)$ times the diagonal distance between pixels, $[(DX^2 + DY^2)/2]^{1/2}$. A larger value produces a coarser resolution by mapping multiple pixels onto a single ρ ; this is useful for images that do not contain perfectly straight lines. A smaller value may produce undersampling by trying to map fractional pixels onto ρ , and is not recommended. If BACKPROJECT is specified, this keyword is ignored.

DX

Set this keyword equal to a scalar specifying the spacing between the horizontal (X) coordinates. The default is 1.0.

DY

Set this keyword equal to a scalar specifying the spacing between the vertical (Y) coordinates. The default is 1.0.

GRAY

Set this keyword to perform a weighted Hough transform, with the weighting given by the pixel values. If GRAY is not set, the image is treated as a binary image with all nonzero pixels considered as 1. If BACKPROJECT is specified, this keyword is ignored.

NRHO

Set this keyword equal to a scalar specifying the number of ρ coordinates to use. The default is $2 \text{ CEIL}[(\text{MAX}(X^2 + Y^2))^{1/2} / \text{DRHO}] + 1$. If BACKPROJECT is specified, this keyword is ignored.

NTHETA

Set this keyword equal to a scalar specifying the number of θ coordinates to use over the interval $[0, \pi]$. The default is $\text{CEIL}(\pi [\text{MAX}(X^2 + Y^2)]^{1/2} / \text{DRHO})$. A larger value will produce smoother results, and is useful for filtering before backprojection. A smaller value will result in broken lines in the transform, and is not recommended. If BACKPROJECT is specified, this keyword is ignored.

NX

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of horizontal coordinates in the output array. The default is $\text{FLOOR}(2 \text{MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform this keyword is ignored.

NY

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of vertical coordinates in the output array. The default is $\text{FLOOR}(2 \text{MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform, this keyword is ignored.

RHO

For the forward transform, set this keyword to a named variable that, on exit, will contain the radial (ρ) coordinates. If BACKPROJECT is specified, this keyword must contain the ρ coordinates of the input *Array*.

RMIN

Set this keyword equal to a scalar specifying the minimum ρ coordinate to use for the forward transform. The default is $-0.5(\text{NRHO} - 1) \text{DRHO}$. If BACKPROJECT is specified, this keyword is ignored.

THETA

For the forward transform, set this keyword to a named variable containing a vector of angular (θ) coordinates to use for the transform. If NTHETA is specified instead, and THETA is set to a named variable, then on exit THETA will contain the θ coordinates. If BACKPROJECT is specified, this keyword must contain the θ coordinates of the input *Array*.

XMIN

Set this keyword equal to a scalar specifying the X coordinate of the lower-left corner of the input *Array*. The default is $-(M-1)/2$, where *Array* is an M by N array. If BACKPROJECT is specified, set this keyword equal to a scalar specifying the X

coordinate of the lower-left corner of the *Result*. In this case the default is $-DX (NX-1)/2$.

YMIN

Set this keyword equal to a scalar specifying the Y coordinate of the lower-left corner of the input *Array*. The default is $-(N-1)/2$, where *Array* is an *M* by *N* array. If *BACKPROJECT* is specified, set this keyword equal to a scalar specifying the Y coordinate of the lower-left corner of the *Result*. In this case the default is $-DY (NY-1)/2$.

Example

This example computes the Hough transform of a random set of pixels:

```
PRO hough_example

;Create an image with a random set of pixels
seed = 12345 ; remove this line to get different random images
array = RANDOMU(seed,128,128) GT 0.95

;Draw three lines in the image
x = FINDGEN(32)*4
array[x,0.5*x+20] = 1b
array[x,0.5*x+30] = 1b
array[-0.5*x+100,x] = 1b

;Create display window, set graphics properties
WINDOW, XSIZE=330,YSIZE=630, TITLE='Hough Example'
!P.BACKGROUND = 255 ; white
!P.COLOR = 0 ; black
!P.FONT=2
ERASE

XYOUTS, .1, .94, 'Noise and Lines', /NORMAL
;Display the image. 255b changes black values to white:
TVSCL, 255b - array, .1, .72, /NORMAL

;Calculate and display the Hough transform
result = HOUGH(array, RHO=rho, THETA=theta)
XYOUTS, .1, .66, 'Hough Transform', /NORMAL
TVSCL, 255b - result, .1, .36, /NORMAL

;Keep only lines that contain more than 20 points:
result = (result - 20) > 0

;Find the Hough backprojection and display the output
backproject = HOUGH(result, /BACKPROJECT, RHO=rho, THETA=theta)
```



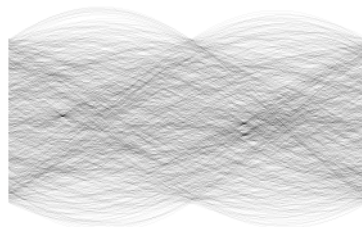
```
XYOUTS, .1, .30, 'Hough Backprojection', /NORMAL  
TVSCL, 255b - backproject, .1, .08, /NORMAL  
  
END
```

The following figure displays the output of this example. The top image shows three lines drawn within a random array of pixels that represent noise. The center image shows the Hough transform, displaying sinusoids for points that lie on the same line in the original image. The bottom image shows the Hough backprojection, after setting the threshold to retain only those lines that contain more than 20 points. The Hough inverse transform, or backprojection, transforms each point in the Hough domain into a straight line in the image.

Noise and Lines



Hough Transform



Hough Backprojection



Figure 3-3: HOUGH example showing random pixels (top), Hough transform (center) and Hough backprojection (bottom)

See Also

[RADON](#)

References

1. Gonzalez, R.C., and R.E. Woods. *Digital Image Processing*. Reading, MA: Addison Wesley, 1992.
2. Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
3. Toft, Peter. *The Radon Transform: Theory and Implementation*. Denmark: Technical University; 1996. Ph.D. Thesis.
4. Weeks, Arthur. R. *Fundamentals of Electronic Image Processing*. New York: SPIE Optical Engineering Press, 1996.

LAGUERRE

The LAGUERRE function returns the value of the associated Laguerre polynomial $L_n^k(x)$. The associated Laguerre polynomials are solutions to the differential equation:

$$xy'' + (k + 1 - x)y' + ny = 0$$

with orthogonality constraint:

$$\int_0^\infty e^{-x} x^{k+1} L_m^k(x) L_n^k(x) dx = \frac{(n+k)!}{n!} \delta_{mn}$$

Laguerre polynomials are used in quantum mechanics, for example, where the wave function for the hydrogen atom is given by the Laguerre differential equation.

This routine is written in the IDL language. Its source code can be found in the file `laguerre.pro` in the `lib` subdirectory of the IDL distribution.

This routine is written in the IDL language. Its source code can be found in the file `laguerre.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = LAGUERRE(*X*, *N* [, *K*] [, COEFFICIENTS=*variable*] [, /DOUBLE])

Return Value

This function returns a scalar or array with the same dimensions as *X*. If *X* is double-precision or if the DOUBLE keyword is set, the result is double-precision complex, otherwise the result is single-precision complex.

Arguments

X

The value(s) at which $L_n^k(x)$ is evaluated. *X* can be either a scalar or an array.

N

A scalar integer, $N \geq 0$, specifying the order *n* of $L_n^k(x)$. If *N* is of type float, it will be truncated.

K

A scalar, $K \geq 0$, specifying the order k of $L_n^k(x)$. If K is not specified, the default $K = 0$ is used and the Laguerre polynomial, $L_n(x)$, is returned.

Keywords**COEFFICIENTS**

Set this keyword to a named variable that will contain the polynomial coefficients in the expansion $C[0] + C[1]x + C[2]x^2 + \dots$.

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Example

To compute the value of the Laguerre polynomial at the following X values:

```
;Define the parametric X values:
X = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]

;Compute the Laguerre polynomial of order N=2, K=1:
result = LAGUERRE(X, 2, 1)

;Print the result:
PRINT, result
```

IDL prints:

```
3.000000    2.420000    1.880000    1.380000    0.920000    0.500000
```

This is the exact solution vector to six-decimal accuracy.

See Also

[LEGENDRE](#), [SPHER_HARM](#)

LEGENDRE

The LEGENDRE function returns the value of the associated Legendre polynomial $P_l^m(x)$. The associated Legendre functions are solutions to the differential equation:

$$(1 - x^2)y'' - 2xy' + \left[l(l+1) - \frac{m^2}{(1 - x^2)} \right] y = 0$$

with orthogonality constraints:

$$\int_{-1}^{+1} P_l^m(x) P_k^n(x) dx = \frac{2}{2l+1} \frac{(l+m)!}{(l-m)!} \delta_{lk} \delta_{mn}$$

The Legendre polynomials are the solutions to the Legendre equation with $m = 0$. For positive m , the associated Legendre functions can be written in terms of the Legendre polynomials as:

$$P_l^m(x) = (-1)^m (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_l(x)$$

Associated polynomials for negative m are related to positive m by:

$$P_l^{-m}(x) = (-1)^m \frac{(l-m)!}{(l+m)!} P_l^m(x)$$

LEGENDRE is based on the routine *plgndr* described in section 6.8 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

Result = LEGENDRE(*X*, *L* [, *M*] [, /DOUBLE])

Return Value

If all arguments are scalar, the function returns a scalar. If all arguments are arrays, the function matches up the corresponding elements of *X*, *L*, and *M*, returning an array with the same dimensions as the smallest array. If one argument is a scalar and the other arguments are arrays, the function uses the scalar value with each element

of the arrays, and returns an array with the same dimensions as the smallest input array.

If any of the arguments are double-precision or if the **DOUBLE** keyword is set, the result is double-precision, otherwise the result is single-precision.

Arguments

X

The expression for which $P_l^m(x)$ is evaluated. Values for X must be in the range $-1 \leq X \leq 1$.

L

An integer scalar or array, $L \geq 0$, specifying the order l of $P_l^m(x)$. If L is of type float, it will be truncated.

M

An integer scalar or array, $-L \leq M \leq L$, specifying the order m of $P_l^m(x)$. If M is not specified, then the default $M = 0$ is used and the Legendre polynomial, $P_l(x)$, is returned. If M is of type float, it will be truncated.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Examples

Example 1

Compute the value of the Legendre polynomial at the following X values:

```
; Define the parametric X values:
X = [-0.75, -0.5, -0.25, 0.25, 0.5, 0.75]

; Compute the Legendre polynomial of order L=2:
result = LEGENDRE(X, 2)

; Print the result:
PRINT, result
```

The result of this is:

```
0.343750 -0.125000 -0.406250 -0.406250 -0.125000 0.343750
```

Example 2

Compute the value of the associated Legendre polynomial at the same X values:

```
; Compute the associated Legendre polynomial of order L=2, M=1:  
result = LEGENDRE(X, 2, 1)  
; Print the result:  
PRINT, result
```

IDL prints:

```
1.48824  1.29904  0.726184 -0.726184 -1.29904 -1.48824
```

This is the exact solution vector to six-decimal accuracy.

See Also

[SPHER_HARM](#), [LAGUERRE](#)

MAKE_DLL

The MAKE_DLL procedure builds a sharable library from C language code which is suitable for use by IDL's dynamic linking features such as CALL_EXTERNAL, LINKIMAGE, and dynamically loadable modules (DLMs). MAKE_DLL reduces the complexity of building sharable libraries by providing a stable cross-platform method for the user to describe the desired library, and issuing the necessary operating system commands to build the library.

Note

MAKE_DLL is supported under UNIX, VMS, and Microsoft Windows, but is not available for the Macintosh.

Although MAKE_DLL is very convenient, it is not intended for use as a general purpose compiler. Instead, MAKE_DLL is specifically targeted to solving the most common IDL dynamic linking problem: building a sharable library from C language source files that are usable by IDL. Because of this, the following requirements apply:

- You must have a C compiler installed on your system. It is easiest to use the compiler used to build IDL, because MAKE_DLL already knows how to use that compiler without any additional configuring. To determine which compiler was used, query the !MAKE_DLL system variable with a print statement such as the following:

```
PRINT, !MAKE_DLL.COMPILER_NAME
```

- MAKE_DLL only compiles programs written in the C language; it does not understand Fortran, C++, or any other languages.
- MAKE_DLL provides only the functionality necessary to build C code intended to be linked with IDL. Not every possible option supported by the C compiler or system linker is addressed, only those commonly needed by IDL-related C code.

MAKE_DLL solves the most common IDL-centric problem of linking C code with IDL. To do more than this or to use a different language requires a system-specific building process (e.g. make files, projects, etc...).

Syntax

```
MAKE_DLL, InputFiles [, OutputFile], ExportedRoutineNames [, CC=string]  
[, COMPILE_DIRECTORY=path] [, DLL_PATH=variable]  
[, EXPORTED_DATA=string] [, EXTRA_CFLAGS=string]  
[, EXTRA_LFLAGS=string] [, INPUT_DIRECTORY=path] [, LD=string]  
[, /NOCLEANUP] [, OUTPUT_DIRECTORY=path] [, /SHOW_ALL_OUTPUT]  
[, /VERBOSE]
```

VMS-Only Keywords: [/VAX_FLOAT]

Arguments

InputFiles

A string (scalar or array) giving the names of the input C program files to be compiled by MAKE_DLL. These names should not include any directory path information or the .c suffix, they are simply the base file names.

The input directory is specified using the INPUT_DIRECTORY keyword, and the .c file suffix is assumed.

OutputFile

The base name of the resulting sharable library. This name should not include any directory path information or the sharable library suffix, which differs between platforms (for example: .so, .a, .sl, .exe, .dll).

The output directory can be specified using the OUTPUT_DIRECTORY keyword.

If the *OutputFile* argument is omitted, the first name given by *InputFile* is used as the base name of output file.

ExportedRoutineNames

A string (scalar or array) specifying the names of the routines to be exported (i.e., are visible for linking) from the resulting sharable library.

Keywords

CC

If present, a template string to use in generating the C compiler commands to compile *InputFiles*. If CC is not specified, the value given by the !MAKE_DLL.CC system variable is used by default. See the discussion of !MAKE_DLL for a description of how to write the format string for CC.

COMPILE_DIRECTORY

To build a sharable library, MAKE_DLL requires a place to create the necessary intermediate files and possibly the final library itself. If COMPILE_DIRECTORY is specified, the directory specified is used. If COMPILE_DIRECTORY is not specified, the directory given by the !MAKE_DLL.COMPILE_DIRECTORY system variable is used.

DLL_PATH

If present, the name of a variable to receive the complete file path for the newly created sharable library. The location of the resulting sharable library depends on the setting of the OUTPUT_DIRECTORY or COMPILE_DIRECTORY keywords as well as the !MAKE_DLL.COMPILE_DIRECTORY system variable, and different platforms use different file suffixes to indicate sharable libraries. Use of the DLL_PATH keyword makes it possible to determine the resulting file path in a simple and portable manner.

EXPORTED_DATA

A string (scalar or array) containing the names of variables to be exported (i.e., are visible for linking) from the resulting sharable library.

EXTRA_CFLAGS

If present, a string supplying extra options for the command used to execute the C compiler to compile the files given by *InputFiles*. This keyword is frequently used to specify header file include directories. This text is inserted in place of the %X format code in the compile string. See the discussion of the CC keyword and !MAKE_DLL.CC system variable for more information.

EXTRA_LFLAGS

If present, a string supplying extra options for the command used to execute the linker when combining the object files to produce the sharable library. This keyword is frequently used to specify libraries to be included in the link, and is inserted in place of the %X format code in the linker string. See the discussion of the LD keyword and !MAKE_DLL.LD system variable for more information.

INPUT_DIRECTORY

If present, the path to the directory containing the source C files listed in *InputFiles*. If INPUT_DIRECTORY is not specified, the directory given by COMPILE_DIRECTORY is assumed to contain the files.

LD

If present, a template string to use when generating the linker command to generate the resulting sharable library. If LD is not specified, the value given by the `!MAKE_DLL.LD` system variable is used by default. See the discussion of `!MAKE_DLL` for a description of how to write the format string for LD.

NOCLEANUP

To produce a sharable library, `MAKE_DLL` produces several intermediate files:

1. A shell script (UNIX), command file (VMS), or batch file (Windows) that is then executed via SPAWN to build the library.
2. A linker options file. This file is used to control the linker. `MAKE_DLL` uses it to cause the routines given by the *ExportedRoutineNames* argument (and `EXPORTED_DATA` keyword) to be exported from the resulting sharable library. The general platform terminology is shown below.

Platform	Linker Options File Terminology
UNIX	export file, or linker map file
VMS	linker options file (.OPT)
Windows	a .DEF file

Table 3-2: Platform Terminology for Linker Options File

3. Object files, resulting from compiling the source C files given by the *InputFiles* argument.
4. A log file that captures the output from executing the script, and which can be used for debugging in case of error.

By default, `MAKE_DLL` deletes all of these intermediate files once the sharable library has been successfully built. Setting the `NOCLEANUP` keyword prevents `MAKE_DLL` from removing them.

Note

Set the `NOCLEANUP` keyword (possibly in conjunction with `VERBOSE`) for trouble shooting, or to read the files for additional information on how `MAKE_DLL` works.

OUTPUT_DIRECTORY

By default, MAKE_DLL creates the resulting sharable library in the compile directory specified by the COMPILE_DIRECTORY keyword or the !MAKE_DLL.COMPILE_DIRECTORY system variable. The OUTPUT_DIRECTORY keyword can be used to override this and explicitly specify where the library file should go.

SHOW_ALL_OUTPUT

MAKE_DLL normally produces no output unless an error prevents successful building of the sharable library. Set SHOW_ALL_OUTPUT to see all output produced by the spawned process building the library.

VERBOSE

If set, VERBOSE causes MAKE_DLL to issue informational messages as it carries out the task of building the sharable library. These messages include information on the intermediate files created to build the library and how they are used.

VMS-Only Keywords

This keyword is for VMS platforms only, and is ignored on all other platforms.

VAX_FLOAT

If set, specifies the sharable library to be compiled for VAX F (single) or D (double) floating point formats. The default is to use the IEEE format used by IDL.

!MAKE_DLL System Variable

The MAKE_DLL procedure and CALL_EXTERNAL function's AUTO_GLUE keyword use the standard system C compiler and linker to generate sharable libraries that can be used by IDL in various contexts (CALL_EXTERNAL, DLMs, LINKIMAGE). There is a great deal of variation possible in the use of these tools between different platforms, operating system versions, and compiler releases. The !MAKE_DLL system variable is used to configure how IDL uses them for the current platform.

The !MAKE_DLL structure is defined as follows:

```
{ !MAKE_DLL, COMPILE_DIRECTORY:'', COMPILER_NAME:'', CC:'', LD:'' }
```

The meaning of the fields of !MAKE_DLL are given in Table D-2. When expanding !MAKE_DLL.CC and !MAKE_DLL.LD, IDL substitutes text in place of the

PRINTF style codes described in the following table. These codes are case-insensitive, and can be either upper or lower case.

Note

It is possible to use C compilers other than the one assumed by RSI in !MAKE_DLL to build sharable libraries. To do so, you can alter the contents of !MAKE_DLL or use the CC and/or LD keyword to MAKE_DLL and CALL_EXTERNAL. Please understand that RSI cannot and does not maintain a list of all possible compilers and the necessary compiler options. This information is available in your compiler and system documentation. It is the programmers responsibility to understand the rules for their chosen compiler.

Field	Meaning
COMPILE_DIRECTORY	<p>IDL requires a place to create the intermediate files necessary to build a sharable library, and possibly the final library itself. Unless told to use an explicit directory, it uses the directory given by the COMPILE_DIRECTORY field of !MAKE_DLL. If the IDL_MAKE_DLL_COMPILE_DIRECTORY environment variable is set, IDL uses its value to initialize the COMPILE_DIRECTORY field. Otherwise, IDL supplies a standard location.</p> <p>Note - Note that if the directory given by !MAKE_DLL.COMPILE_DIRECTORY does not exist when IDL needs it, IDL automatically creates it for you.</p>
COMPILER_NAME	<p>A string containing the name of the C compiler used by RSI to build the currently running IDL. This field is not used by IDL, and exists solely for informational purposes and to help the end user decide which C compiler to install on their system.</p>

Table 3-3: Meaning of !MAKE_DLL fields

Field	Meaning
CC	A string used by IDL as a template to construct the command for using the C compiler. This template uses PRINTF style substitution codes, as described in the following table.
LD	A string used by IDL as a template to construct the command for using the linker. This template uses PRINTF style substitution codes, as described in the following table.

Table 3-3: Meaning of !MAKE_DLL fields (Continued)

The following table describes the substitution codes for the CC and LD fields:

Code	Meaning
%B %b	The base name of a C file to compile. For example, if the C file is <code>moose.c</code> , then %B substitutes <code>moose</code> .
%C %c	The name of the C file.
%E %e	The name of the linker options file. This file, which is automatically generated by IDL as needed, is used to control the linker. Under UNIX, the system documentation refers to this as an export file, or a linker map file. VMS calls it a linker options file (<code>.OPT</code>). Microsoft Windows calls it a <code>.DEF</code> file.
%F %f	A floating point switch to C compiler. This is only meaningful under VMS, and corresponds to the <code>VAX_FLOAT</code> keyword to <code>MAKE_DLL</code> and <code>CALL_EXTERNAL</code> .
%L %l	The name of the resulting sharable library. IDL constructs this name by using the base name (%B) and adding the appropriate suffix for the current platform (<code>.dll</code> , <code>.so</code> , <code>.sl</code> , <code>.exe</code> , ...).
%O %o	An object file name. IDL constructs this name by using the base name (%B) and adding the appropriate suffix for the current platform (<code>.o</code> , <code>.obj</code>).

Table 3-4: Description of CC and LD Field Codes

Code	Meaning
%X %x	When expanding !MAKE_DLL.CC, any text supplied via the EXTRA_CFLAGS keyword to MAKE_DLL or CALL_EXTERNAL is inserted in place of %X. IDL does not interpret this text. It is the users responsibility to ensure that it is meaningful in the command. When expanding !MAKE_DLL.LD, the text from the EXTRA_LFLAGS keyword is substituted. The primary use for this code is to include necessary header include directories and link libraries.
%%	Replaced with a single % character.

Table 3-4: Description of CC and LD Field Codes (Continued)

Example 1

Testmodule DLM

The IDL distribution contains an example of a simple DLM (dynamically loadable module) in the `external/dlm` subdirectory. This example consists of a single C source file, and the desired sharable library exports a single function called `IDL_Load`. The following `MAKE_DLL` statement builds this sharable library, leaving the resulting file in the directory given by `!MAKE_DLL.COMPILE_DIRECTORY`:

```
; Locate the source file:
INDIR = FILEPATH('', SUBDIRECTORY=['external', 'dlm'])
; Build the sharable library:
MAKE_DLL, 'testmodule', 'IDL_Load', INPUT_DIRECTORY=INDIR
```

Example 2

Using GCC

IDL is built with the standard vendor-supported C compiler in order to get maximum integration with the target system. `MAKE_DLL` assumes that you have the same compiler installed on your system and its defaults are targeted to use it. To use other compilers, you tell `MAKE_DLL` how to use them.

For example, many IDL users have the `gcc` compiler installed on their systems. This example (tested under 32-bit Solaris 7 using `gcc 2.95.2`) shows how to use `gcc` to build the testmodule sharable library from the previous example:

```
; We need the include directory for the IDL export.h header
```

```
; file. One way to get this is to extract it from the
; !MAKE_DLL system variable using the STREGEX function
INCLUDE=STREGEX(!MAKE_DLL.CC, '-I[^ ]+', /EXTRACT)
; Locate the source file
INDIR = FILEPATH('', SUBDIRECTORY=['external', 'dlm'])
; Build the sharable library, using the CC keyword to specify gcc:
MAKE_DLL, 'testmodule', 'IDL_Load', INPUT_DIRECTORY=INDIR, $
    CC='gcc -c -fPIC '+ INCLUDE + '%C -o %O'
```


MAP_2POINTS

The MAP_2POINTS function returns parameters such as distance, azimuth, and path relating to the great circle or rhumb line connecting two points on a sphere.

This routine is written in the IDL language. Its source code can be found in the file `map_2points.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

```
Result = MAP_2POINTS( lon0, lat0, lon1, lat1 [, DPATH=value | , /METERS |  
    , /MILES | , NPATH=integer{ 2 or greater } | , /PARAMETERS | , RADIUS=value ]  
    [, /RADIANS] [, /RHUMB] )
```

Return Value

This function returns a two-element vector containing the distance and azimuth of the great circle or rhumb line connecting the two points, P0 to P1, in the specified angular units, unless one or more of the keywords NPATH, DPATH, METERS, MILES, PARAMETERS, or RADIUS is specified. See the keyword descriptions for the return value associated with each of these keywords.

If MILES, METERS, or RADIUS is not set, distances are angular distance, from 0 to 180 degrees (or 0 to !PI if the RADIANS keyword is set). Azimuth is measured in degrees or radians, east of north.

Arguments

Lon0, Lat0

Longitude and latitude of the first point, P0.

Lon1, Lat1

Longitude and latitude of the second point, P1.

Keywords

DPATH

Set this keyword to a value specifying the maximum angular distance between the points on the path in the prevalent units, degrees or radians.

METERS

Set this keyword to return the distance between the two points in meters, calculated using the Clarke 1866 equatorial radius of the earth.

MILES

Set this keyword to return the distance between the two points in miles, calculated using the Clarke 1866 equatorial radius of the earth.

NPATH

Set this keyword to a value specifying the number of points to return. If this keyword is set, the function returns a (2, NPATH) array containing the longitude/latitude of the points on the great circle or rhumb line connecting P0 and P1. For a great circle, the points will be evenly spaced in distance, while for a rhumb line, the points will be evenly spaced in longitude.

Note

This keyword must be set to an integer of 2 or greater.

PARAMETERS

Set this keyword to return the parameters determining the great circle connecting the two points, $[\sin(c), \cos(c), \sin(az), \cos(az)]$, where c is the great circle angular distance, and az is the azimuth of the great circle at P0, in degrees east of north.

RADIANS

Set this keyword if inputs and angular outputs are to be specified in radians. The default is degrees.

RADIUS

Set this keyword to a value specifying the radius of the sphere to be used to calculate the distance between the two points. If this keyword is specified, the function returns the distance between the two points calculated using the given radius.

RHUMB

Set this keyword to return the distance and azimuth of the rhumb line connecting the two points, P0 to P1. The default is to return the distance and azimuth of the great circle connecting the two points. A rhumb line is the line of constant direction connecting two points.

Examples

The following examples use the geocoordinates of two points, Boulder and London:

```
B = [ -105.19, 40.02] ;Longitude, latitude in degrees.
L = [ -0.07, 51.30]
```

Example 1

Print the angular distance and azimuth, from B, of the great circle connecting the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1])
```

IDL prints 67.854333 40.667833

Example 2

Print the angular distance and course (azimuth), connecting the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1],/RHUMB)
```

IDL prints 73.966283 81.228057

Example 3

Print the distance in miles between the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1],/MILES)
```

IDL prints 4693.5845

Example 4

Print the distance in miles along the rhumb line connecting the two points:

```
PRINT, MAP_2POINTS(B[0], B[1], L[0], L[1], /MILES, /RHUMB)
```

IDL prints 5116.3571

Example 5

Display a map containing the two points, and annotate the map with both the great circle and the rhumb line path between the points, drawn at one degree increments:

```
MAP_SET, /MOLLWEIDE, 40,-50, /GRID, SCALE=75e6,/CONTINENTS
PLOTS, MAP_2POINTS(B[0], B[1], L[0], L[1],/RHUMB, DPATH=1)
PLOTS, MAP_2POINTS(B[0], B[1], L[0], L[1],DPATH=1)
```

This displays the following map:

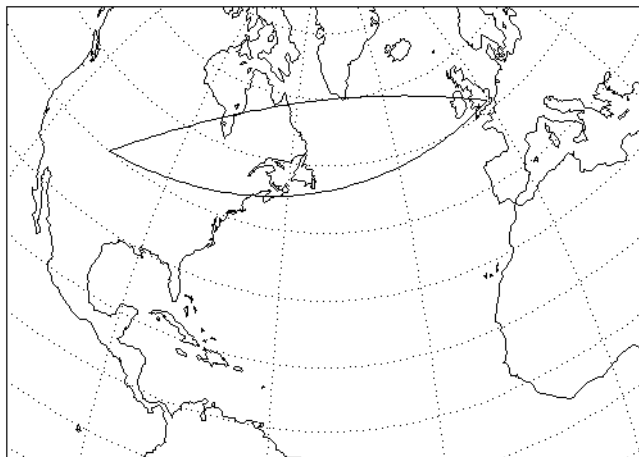


Figure 3-4: Map annotated with great circle and rhumb line path between Boulder and London, drawn at one degree increments.

See Also

[MAP_SET](#)

MATRIX_MULTIPLY

The MATRIX_MULTIPLY function calculates the IDL # operator of two (possibly transposed) arrays. The transpose operation (if desired) is done simultaneously with the multiplication, thus conserving memory and increasing the speed of the operation. If the arrays are not transposed, then MATRIX_MULTIPLY is equivalent to using the # operator.

Syntax

Result = MATRIX_MULTIPLY(*A*, *B* [, /ATRANSPOSE] [, /BTRANSPOSE])

Return Value

The type for the result depends upon the input type. For byte or integer arrays, the result has the type of the next-larger integer type that could contain the result (for example, byte, integer, or long input returns type long integer). For floating-point, the result has the same type as the input.

For the case of no transpose, the resulting array has the same number of columns as the first array and the same number of rows as the second array. The second array must have the same number of columns as the first array has rows.

Note

If *A* and *B* arguments are vectors, then $C = \text{MATRIX_MULTIPLY}(A, B)$ is a matrix with $C_{ij} = A_i B_j$. Mathematically, this is equivalent to the outer product, usually denoted by $A \otimes B$.

Arguments

A

The left operand for the matrix multiplication. Dimensions higher than two are ignored.

B

The right operand for the matrix multiplication. Dimensions higher than two are ignored.

Keywords

ATRANSPOSE

Set this keyword to multiply using the transpose of A .

BTRANSPOSE

Set this keyword to multiply using the transpose of B .

The # Operator vs. MATRIX_MULTIPLY

The following table illustrates how various operations are performed using the # operator versus the MATRIX_MULTIPLY function:

# Operator	Function
$A \# B$	MATRIX_MULTIPLY(A, B)
transpose(A) # B	MATRIX_MULTIPLY($A, B, /ATRANSPOSE$)
$A \#$ transpose(B)	MATRIX_MULTIPLY($A, B, /BTRANSPOSE$)
transpose(A) # transpose(B)	MATRIX_MULTIPLY($A, B, /ATRANSPOSE, /BTRANSPOSE$)

Table 3-5: The # Operator vs. MATRIX_MULTIPLY

Note

MATRIX_MULTIPLY can also be used in place of the ## operator. For example, $A \## B$ is equivalent to MATRIX_MULTIPLY(B, A), and $A \## \text{TRANPOSE}(B)$ is equivalent to MATRIX_MULTIPLY($B, A, /ATRANSPOSE$).

See Also

“[Multiplying Arrays](#)” in Chapter 16 of *Using IDL*

MEMORY

The MEMORY function returns information on the amount of dynamic memory currently in use by the IDL session if no keywords are set. If a keyword is set, MEMORY returns the specified quantity.

Syntax

Result = MEMORY([, /CURRENT | , /HIGHWATER | , /NUM_ALLOC | , /NUM_FREE | , /STRUCTURE] [, /L64])

Return Value

The return value is a vector that is always of integer type. The following table describes the information returned if no keywords are set:

Element	Contents
<i>Result</i> [0]	Amount of dynamic memory (in bytes) currently in use by the IDL session.
<i>Result</i> [1]	The number of times IDL has made a memory allocation request from the underlying system.
<i>Result</i> [2]	The number of times IDL has made a request to free memory from the underlying system.
<i>Result</i> [3]	High water mark: The maximum amount of dynamic memory used since the last time the MEMORY function or HELP, /MEMORY procedure was called.

Table 3-6: MEMORY Function Return Values

Arguments

None.

Keywords

The following keywords determine the return value of the MEMORY function. Except for L64, all of the keywords are mutually exclusive — specify at most one of the following.

CURRENT

Set this keyword to return the amount of dynamic memory (in bytes) currently in use by the IDL session.

HIGHWATER

Set this keyword to return the maximum amount of dynamic memory used since the last time the MEMORY function or HELP/MEMORY procedure was called. This can be used to determine maximum memory use of a code sequence as shown in the example below.

L64

By default, the result of MEMORY is 32-bit integer when possible, and 64-bit integer if the size of the returned values requires it. Set L64 to force 64-bit integers to be returned in all cases.

Note

Only 64-bit versions of IDL are capable of using enough memory to require 64-bit MEMORY output. Check the value of !VERSION.MEMORY_BITS to see if your IDL is 64-bit or not.

NUM_ALLOC

Returns the number of times IDL has requested dynamic memory from the underlying system.

NUM_FREE

Returns the number of times IDL has returned dynamic memory to the underlying system.

STRUCTURE

Set this keyword to return all available information about Expression in a structure. The result will be an IDL_MEMORY (32-bit) structure when possible, and an IDL_MEMORY64 structure otherwise. Set L64 to force an IDL_MEMORY64 structure to be returned in all cases.

The following are descriptions of the fields in the returned structure:

Field	Description
CURRENT	Current dynamic memory in use.
NUM_ALLOC	Number of calls to allocate memory.
NUM_FREE	Number of calls to free memory.
HIGHWATER	Maximum dynamic memory used since last call for this information.

Table 3-7: STRUCTURE Field Descriptions

Example

To determine how much dynamic memory is required to execute a sequence of IDL code:

```
; Get current allocation and reset the high water mark:
start_mem = MEMORY(/CURRENT)

; Arbitrary code goes here.

PRINT, 'Memory required: ', MEMORY(/HIGHWATER) - start_mem
```

The MEMORY function can also be used in conjunction with DIALOG_MESSAGE as follows:

```
; Get current dynamic memory in use:
mem = MEMORY(/CURRENT)
; Prepare dialog message:
message = 'Current amount of dynamic memory used is '
sentence = message + STRTRIM(mem,2)+' bytes.'
; Display the dialog message containing memory usage statement:
status = DIALOG_MESSAGE (sentence, /INFORMATION)
```

See Also

[HELP](#)

RADON

The RADON function implements the Radon transform, used to detect features within a two-dimensional image. This function can be used to return either the Radon transform, which transforms lines through an image to points in the Radon domain, or the Radon backprojection, where each point in the Radon domain is transformed to a straight line in the image.

Syntax

Radon Transform:

```
Result = RADON( Array [, /DOUBLE] [, DRHO=scalar] [, DX=scalar]
[, DY=scalar] [, /GRAY] [, /LINEAR] [, NRHO=scalar] [, NTHETA=scalar]
[, RHO=variable] [, RMIN=scalar] [, THETA=variable] [, XMIN=scalar]
[, YMIN=scalar] )
```

Radon Backprojection:

```
Result = RADON( Array, /BACKPROJECT, RHO=variable, THETA=variable
[, /DOUBLE] [, DX=scalar] [, DY=scalar] [, /LINEAR] [, NX=scalar]
[, NY=scalar] [, XMIN=scalar] [, YMIN=scalar] )
```

Return Value

The result of this function is a two-dimensional floating-point array, or a complex array if the input image is complex. If *Array* is double-precision, or if the DOUBLE keyword is set, the result is double-precision, otherwise, the result is single-precision.

Radon Transform Theory

The Radon transform is used to detect features within an image. Given a function $A(x, y)$, the Radon transform is defined as:

$$R(\theta, \rho) = \int_{-\infty}^{\infty} A(\rho \cos \theta - s \sin \theta, \rho \sin \theta + s \cos \theta) ds$$

This equation describes the integral along a line s through the image, where ρ is the distance of the line from the origin and θ is the angle from the horizontal.

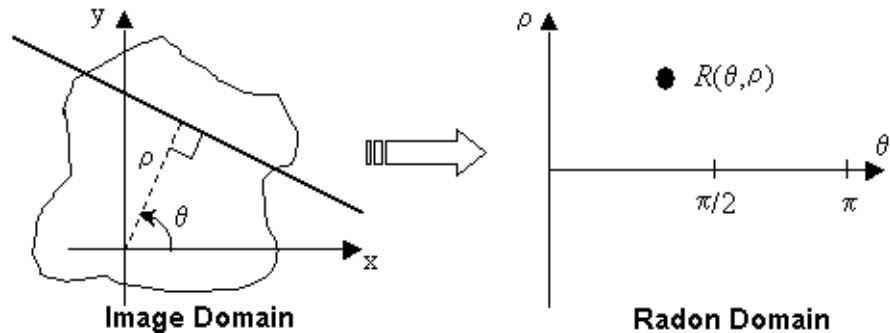


Figure 3-5: The Radon Transform

In medical imaging, each point $R(\theta, \rho)$ is called a ray-sum, while the resulting image is called a shadowgram. An image can be reconstructed from its ray-sums using the backprojection operator:

$$B(x, y) = \int_0^\pi R(\theta, x \cos \theta + y \sin \theta) d\theta$$

where the output, $B(x, y)$, is an image of $A(x, y)$ blurred by the Radon transform.

How IDL Implements the Radon Transform

To avoid the use of a two-dimensional interpolation and decrease the interpolation errors, the Radon transform equation is rotated by θ , and the interpolation is then done along the line s . The transform is divided into two regions, one for nearly-horizontal lines ($45^\circ < \theta < 135^\circ$), and the other for steeper lines ($\theta \leq 45^\circ$; $135^\circ \leq \theta \leq 180^\circ$), where θ is assumed to lie on the interval $[0^\circ, 180^\circ]$.

For nearest-neighbor interpolation (the default), the discrete transform formula for an image $A(m, n)$ [$m = 0, \dots, M-1, n = 0, \dots, N-1$] is:

$$R(\theta, \rho) = \begin{cases} \frac{\Delta x}{|\sin \theta|} \sum_m A(m, [am + b]) & |\sin \theta| > \frac{\sqrt{2}}{2} \\ \frac{\Delta y}{|\cos \theta|} \sum_n A([a'n + b'], n) & |\sin \theta| \leq \frac{\sqrt{2}}{2} \end{cases}$$

where brackets $[\cdot]$ indicate rounding to the nearest integer, and the slope and offsets are given by:

$$a = -\frac{\Delta x \cos \theta}{\Delta y \sin \theta} \quad b = \frac{\rho - x_{\min} \cos \theta - y_{\min} \sin \theta}{\Delta y \sin \theta}$$

$$a' = \frac{1}{a} \quad b' = \frac{\rho - x_{\min} \cos \theta - y_{\min} \sin \theta}{\Delta x \cos \theta}$$

For linear interpolation, the transform is:

$$R(\theta, \rho) = \begin{cases} \frac{\Delta x}{|\sin \theta|} \sum_m (1-w)A(m, \lfloor am + b \rfloor) + wA(m, \lfloor am + b \rfloor + 1) & |\sin \theta| > \frac{\sqrt{2}}{2} \\ \frac{\Delta y}{|\cos \theta|} \sum_n (1-w)A(\lfloor a'n + b' \rfloor, n) + wA(\lfloor a'n + b' \rfloor + 1, n) & |\sin \theta| \leq \frac{\sqrt{2}}{2} \end{cases}$$

where the slope and offsets are the same as above, and $\lfloor \cdot \rfloor$ indicates flooring to the nearest lower integer. The weighting w is given by the difference between $am + b$ and its floored value, or between $a'n + b'$ and its floored value.

How IDL Implements the Radon Backprojection

For the backprojection transform, the discrete formula for nearest-neighbor interpolation is:

$$B(m, n) = \Delta \theta \sum_t R(\theta_t, [\rho])$$

with the nearest-neighbor for ρ given by:

$$\rho = \{(m\Delta x + x_{\min})\cos\theta_t + (n\Delta y + y_{\min})\sin\theta_t - \rho_{\min}\}\Delta\rho^{-1}$$

For backprojection with linear interpolation:

$$B(m,n) = \Delta\theta \sum_t (1-w)R(\theta_t, \lfloor \rho \rfloor) + wR(\theta_t, \lfloor \rho \rfloor + 1)$$

$$w = \rho - \lfloor \rho \rfloor$$

and ρ is the same as in the nearest-neighbor.

Arguments

Array

The two-dimensional array of size M by N to be transformed.

Keywords

BACKPROJECT

If set, the backprojection is computed, otherwise, the forward transform is computed.

Note

The Radon backprojection does not return the original image. Instead, it returns an image blurred by the Radon transform. Because the Radon transform is not one-to-one, multiple (x, y) points are mapped to a single (θ, ρ) .

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

DRHO

Set this keyword equal to a scalar specifying the spacing between ρ coordinates, expressed in the same units as *Array*. The default is one-half of the diagonal distance between pixels, $0.5[(DX^2 + DY^2)]^{1/2}$. Smaller values produce finer resolution, and are useful for zooming in on interesting features. Larger values may result in

undersampling, and are not recommended. If BACKPROJECT is specified, this keyword is ignored.

DX

Set this keyword equal to a scalar specifying the spacing between the horizontal (x) coordinates. The default is 1.0.

DY

Set this keyword equal to a scalar specifying the spacing between the vertical (y) coordinates. The default is 1.0.

GRAY

Set or omit this keyword to perform a weighted Radon transform, with the weighting given by the pixel values. If GRAY is explicitly set to zero, the image is treated as a binary image with all nonzero pixels considered as 1.

LINEAR

Set this keyword to use linear interpolation rather than the default nearest-neighbor sampling. Results are more accurate but slower when linear interpolation is used.

NRHO

Set this keyword equal to a scalar specifying the number of ρ coordinates to use. The default is $2 \text{ CEIL}([\text{MAX}(x^2 + y^2)]^{1/2} / \text{DRHO}) + 1$. If BACKPROJECT is specified, this keyword is ignored.

NTHETA

Set this keyword equal to a scalar specifying the number of θ coordinates to use over the interval $[0, \pi]$. The default is $\text{CEIL}(\pi [(M^2 + N^2)/2]^{1/2})$. Larger values produce smoother results, and are useful for filtering before backprojection. Smaller values result in broken lines in the transform, and are not recommended. If BACKPROJECT is specified, this keyword is ignored.

NX

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of horizontal coordinates in the output *Result*. The default is $\text{FLOOR}(2 \text{ MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform this keyword is ignored.

NY

If BACKPROJECT is specified, set this keyword equal to a scalar specifying the number of vertical coordinates in the output *Result*. The default is $\text{FLOOR}(2 \text{ MAX}(|\text{RHO}|)(\text{DX}^2 + \text{DY}^2)^{-1/2} + 1)$. For the forward transform, this keyword is ignored.

RHO

For the forward transform, set this keyword to a named variable that will contain the radial (ρ) coordinates. If BACKPROJECT is specified, this keyword must contain the ρ coordinates of the input *Array*. The ρ coordinates should be evenly spaced and in increasing order.

RMIN

Set this keyword equal to a scalar specifying the minimum ρ coordinate to use for the forward transform. The default is $-0.5(\text{NRHO} - 1) \text{ DRHO}$. If BACKPROJECT is specified, this keyword is ignored.

THETA

For the forward transform, set this keyword to a named variable containing a vector of angular (θ) coordinates to use for the transform. If NTHETA is specified instead, and THETA is set to a named variable, on exit THETA will contain the θ coordinates. If BACKPROJECT is specified, this keyword must contain the θ coordinates of the input *Array*.

XMIN

Set this keyword equal to a scalar specifying the x -coordinate of the lower-left corner of the input *Array*. The default is $-(M-1)/2$, where *Array* is an M by N array. If BACKPROJECT is specified, set this keyword equal to a scalar specifying the x -coordinate of the lower-left corner of the *Result*. In this case the default is $-\text{DX} (\text{NX}-1)/2$.

YMIN

Set this keyword equal to a scalar specifying the y -coordinate of the lower-left corner of the input *Array*. The default is $-(N-1)/2$, where *Array* is an M by N array. If BACKPROJECT is specified, set this keyword equal to a scalar specifying the y -coordinate of the lower-left corner of the *Result*. In this case, the default is $-\text{DY} (\text{NY}-1)/2$.

Example

This example displays the Radon transform and the Radon backprojection:

```

PRO radon_example

    DEVICE, DECOMPOSED=0

    ;Create an image with a ring plus random noise:
    x = (LINDGEN(128,128) MOD 128) - 63.5
    y = (LINDGEN(128,128)/128) - 63.5
    radius = SQRT(x^2 + y^2)
    array = (radius GT 40) AND (radius LT 50)
    array = array + RANDOMU(seed,128,128)

    ;Create display window, set graphics properties:
    WINDOW, XSIZE=440,YSIZE=700, TITLE='Radon Example'
    !P.BACKGROUND = 255 ; white
    !P.COLOR = 0 ; black
    !P.FONT=2
    ERASE

    XYOUTS, .05, .94, 'Ring and Random Pixels', /NORMAL
    ;Display the image. 255b changes black values to white:
    TVSCL, 255b - array, .05, .75, /NORMAL

    ;Calculate and display the Radon transform:
    XYOUTS, .05, .70, 'Radon Transform', /NORMAL
    result = RADON(array, RHO=rho, THETA=theta)
    TVSCL, 255b - result, .08, .32, /NORMAL
    PLOT, theta, rho, /NODATA, /NOERASE, $
        POSITION=[0.08,0.32, 1, 0.68], $
        XSTYLE=9,YSTYLE=9,XTITLE='Theta', YTITLE='R'

    ;For simplicity in this example, remove everything except
    ;the two stripes. A better (and more complicated) method would
    ;be to choose a threshold for the result at each value of theta,
    ;perhaps based on the average of the result over the theta
    ;dimension.
    result[*,0:55] = 0
    result[*,73:181] = 0
    result[*,199:*] = 0

    ;Find the Radon backprojection and display the output:
    XYOUTS, .05, .26, 'Radon Backprojection', /NORMAL
    backproject = RADON(result, /BACKPROJECT, RHO=rho, THETA=theta)
    TVSCL, 255b - backproject, .05, .07, /NORMAL

END

```


The following figure displays the program output. The top image is an image of a ring and random pixels, or noise. The center image is the Radon transform, and displays the line integrals through the image. The bottom image is the Radon backprojection, after filtering all noise except for the two strong horizontal stripes in the middle image.

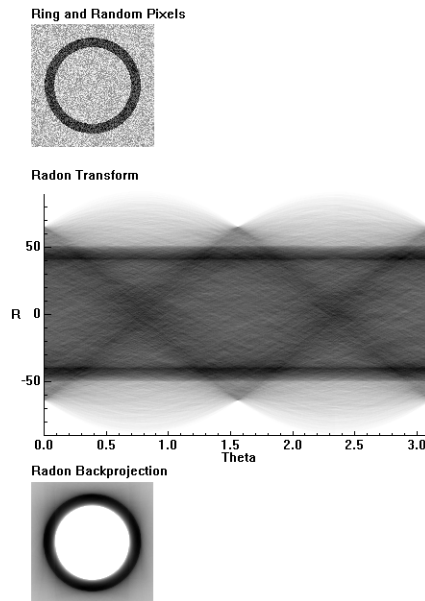


Figure 3-6: Radon Example - Original image (top), Radon transform (center), and backprojection of the altered Radon transform (bottom).

See Also

[HOUGH](#), [VOXEL_PROJ](#)

References

1. Herman, Gabor T. *Image Reconstruction from Projections*. New York: Academic Press, 1980.
2. Hiriyanaiyah, H. P. X-ray computed tomography for medical imaging. *IEEE Signal Processing Magazine*, March 1997: 42-58.

3. Jain, Anil K. *Fundamentals of Digital Image Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
4. Toft, Peter. *The Radon Transform: Theory and Implementation*. Denmark: Technical University; 1996. Ph.D. Thesis.

SAVGOL

The SAVGOL function returns the coefficients of a Savitzky-Golay smoothing filter, which can then be applied using the CONVOL function. The Savitzky-Golay smoothing filter, also known as least squares or DISPO (digital smoothing polynomial), can be used to smooth a noisy signal.

The filter is defined as a weighted moving average with weighting given as a polynomial of a certain degree. The returned coefficients, when applied to a signal, perform a polynomial least-squares fit within the filter window. This polynomial is designed to preserve higher moments within the data and reduce the bias introduced by the filter. The filter can use any number of points for this weighted average.

This filter works especially well when the typical peaks of the signal are narrow. The heights and widths of the curves are generally preserved.

Tip

You can use this function in conjunction with the CONVOL function for smoothing and optionally for numeric differentiation.

This routine is written in the IDL language. Its source code can be found in the file `savgol.pro` in the `lib` subdirectory of the IDL distribution.

SAVGOL is based on the Savitzky-Golay Smoothing Filters described in section 14.8 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

Result = SAVGOL(*Nleft*, *Nright*, *Order*, *Degree* [, /DOUBLE])

Return Value

This function returns an array of floating-point numbers that are the coefficients of the smoothing filter.

Arguments

Nleft

An integer specifying the number of data points to the left of each point to include in the filter.

Nright

An integer specifying the number of data points to the right of each point to include in the filter.

Note

Larger values of *Nleft* and *Nright* produce a smoother result at the expense of flattening sharp peaks.

Order

An integer specifying the order of the derivative desired. For smoothing, use order 0. To find the smoothed first derivative of the signal, use order 1, for the second derivative, use order 2, etc.

Note

Order must be less than or equal to the value specified for *Degree*.

Degree

An integer specifying the degree of smoothing polynomial. Typical values are 2 to 4. Lower values for *Degree* will produce smoother results but may introduce bias, higher values for *Degree* will reduce the filter bias, but may “over fit” the data and give a noisier result.

Note

Degree must be less than the filter width ($Nleft + Nright + 1$).

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Tip

The DOUBLE keyword is recommended for *Degree* greater than 9.

Example

The following example creates a noisy 400-point vector with 4 Gaussian peaks of decreasing width. It then plots the original vector, the vector smoothed with a 33-point Boxcar smoother (the SMOOTH function), and the vector smoothed with 33-point wide Savitzky-Golay filter of degree 4. The bottom plot shows the first derivative of the noisy signal and the first derivative using the Savitzky-Golay filter of degree 4:

```
n = 401 ; number of points
np = 4 ; number of peaks
; Form the baseline:
y = REPLICATE(0.5, n)
; Index the array:
x = FINDGEN(n)
; Add each Gaussian peak:
FOR i=0, np-1 DO BEGIN
    c = (i + 0.5) * FLOAT(n)/np ; Center of peak
    peak = -(3 * (x-c) / (75. / 1.5 ^ i))^2
    ; Add Gaussian. Cutoff of -50 avoids underflow errors for
    ; tiny exponentials:
    y = y + EXP(peak>(-50))
ENDFOR
; Add noise:
y1 = y + 0.10 * RANDOMN(-121147, n)

!P.MULTI=[0,1,3]

; Boxcar smoothing width 33:
PLOT, x, y1, TITLE='Signal+Noise; Smooth (width33)'
OPLOT, SMOOTH(y1, 33, /EDGE_TRUNCATE), THICK=3

; Savitzky-Golay with 33, 4th degree polynomial:
savgolFilter = SAVGOL(16, 16, 0, 4)
PLOT, x, y1, TITLE='Savitzky-Golay (width 33, 4th degree)'
OPLOT, x, CONVOL(y1, savgolFilter, /EDGE_TRUNCATE), THICK=3

; Savitzky-Golay width 33, 4th degree, 1st derivative:
savgolFilter = SAVGOL(16, 16, 1, 4)
PLOT, x, DERIV(y1), YRANGE=[-0.2, 0.2], TITLE=$
    'First Derivative: Savitzky-Golay(width 33, 4th degree, order 1)'
OPLOT, x, CONVOL(y1, savgolFilter, /EDGE_TRUNCATE), THICK=3
```

The following is the resulting plot. Notice how the Savitzky-Golay filter preserves the high peaks but does not do as much smoothing on the flatter regions. Note also

that the Savitzky-Golay filter is able to construct a good approximation of the first derivative.

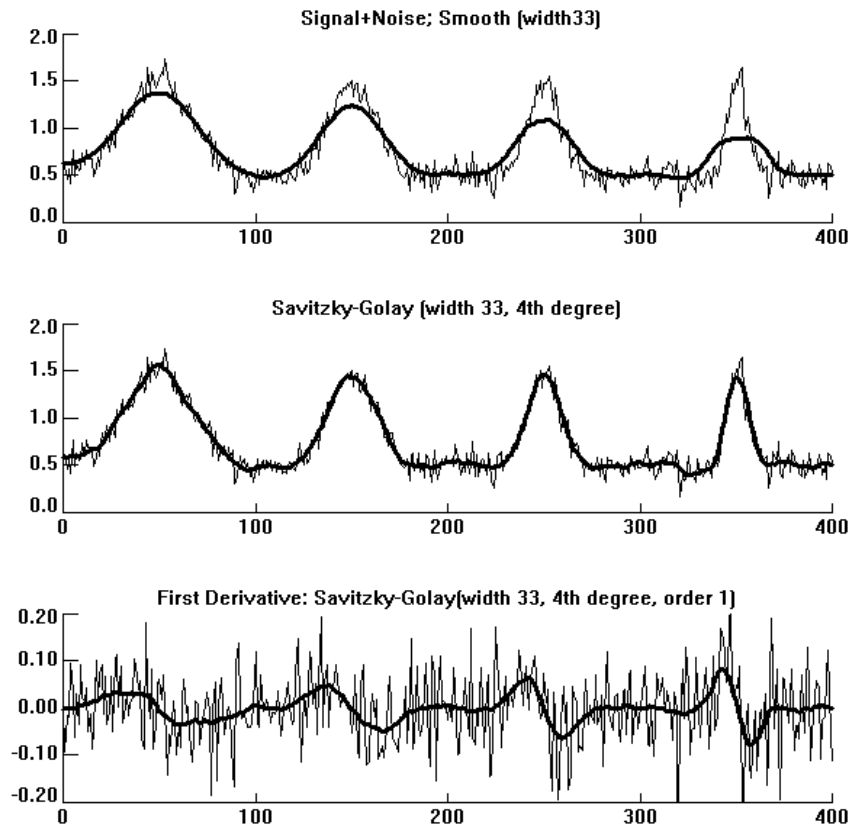


Figure 3-7: SAVGOL Example

See Also

[CONVOL](#), [DIGITAL_FILTER](#), [SMOOTH](#)

SOCKET

The SOCKET procedure, supported on UNIX and Microsoft Windows platforms, opens a client-side TCP/IP Internet socket as an IDL file unit. Such files can be used in the standard manner with any of IDL's Input/Output routines.

Tip

RSI recommends that you don't use the EOF procedure as a way to check to see if a socket is empty. It is recommended that you structure your communication across the socket so that using EOF is not necessary to know when the communication is complete.

Syntax

```
SOCKET, Unit, Host, Port [, CONNECT_TIMEOUT=value] [, ERROR=variable]  
[, /GET_LUN] [, /RAWIO] [, READ_TIMEOUT=value] [, /SWAP_ENDIAN]  
[, /SWAP_IF_BIG_ENDIAN] [, /SWAP_IF_LITTLE_ENDIAN] [, WIDTH=value]  
[, WRITE_TIMEOUT=value]
```

UNIX-Only Keywords: [, /STDIO]

Arguments

Unit

The unit number to associate with the opened socket.

Host

The name of the host to which the socket is connected. This can be either a standard Internet host name (e.g. `ftp.ResearchSystems.com`) or a dot-separated numeric address (e.g. `192.5.156.21`).

Port

The port to which the socket is connected on the remote machine. If this is a well-known port (as contained in the `/etc/services` file on a UNIX host), then you can specify its name (e.g. `daytime`); otherwise, specify a number.

Keywords

CONNECT_TIMEOUT

Set this keyword to the number of seconds to wait before giving up and issuing an error to shorten the connect timeout from the system-supplied default. Most experts recommend that you not specify an explicit timeout, and instead use your operating system defaults.

Note

Although you can use `CONNECT_TIMEOUT` to shorten the timeout, you cannot increase it past the system-supplied default.

ERROR

A named variable in which to place the error status. If an error occurs in the attempt to open File, IDL normally takes the error handling action defined by the `ON_ERROR` and/or `ON_IOERROR` procedures. `SOCKET` always returns 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`.

GET_LUN

Set this keyword to use the `GET_LUN` procedure to set the value of *Unit* before the file is opened. Instead of using the two statements:

```
GET_LUN, Unit
OPENR, Unit, 'data.dat'
```

you can use the single statement:

```
OPENR, Unit, 'data.dat', /GET LUN
```

RAWIO

Set this keyword to disable all use of the standard operating system I/O for the file, in favor of direct calls to the operating system. This allows direct access to devices, such as tape drives, that are difficult or impossible to use effectively through the standard I/O. Using this keyword has the following implications:

- No formatted or associated (ASSOC) I/O is allowed on the file. Only `READU` and `WRITEU` are allowed.
- Normally, attempting to read more data than is available from a file causes the unfilled space to be set to zero and an error to be issued. This does not happen

with files opened with RAWIO. When using RAWIO, the programmer must check the transfer count, either via the TRANSFER_COUNT keywords to READU and WRITEU, or the FSTAT function.

- The EOF and POINT_LUN functions cannot be used with a file opened with RAWIO.
- Each call to READU or WRITEU maps directly to UNIX read(2) and write(2) system calls. The programmer must read the UNIX system documentation for these calls and documentation on the target device to determine if there are any special rules for I/O to that device. For example, the size of data that can be transferred to many cartridge tape drives is often forced to be a multiple of 512 bytes.

READ_TIMEOUT

Set this keyword to the number of seconds to wait for data to arrive before giving up and issuing an error. By default, IDL blocks indefinitely until the data arrives. Typically, this option is unnecessary on a local network, but it is useful with networks that are slow or unreliable.

SWAP_ENDIAN

Set this keyword to swap byte ordering for multi-byte data when performing binary I/O on the specified file. This is useful when accessing files also used by another system with byte ordering different than that of the current host.

SWAP_IF_BIG_ENDIAN

Setting this keyword is equivalent to setting SWAP_ENDIAN; it only takes effect if the current system has big endian byte ordering. This keyword does not refer to the byte ordering of the input data, but to the computer hardware.

SWAP_IF_LITTLE_ENDIAN

Setting this keyword is equivalent to setting SWAP_ENDIAN; it only takes effect if the current system has little endian byte ordering. This keyword does not refer to the byte ordering of the input data, but to the computer hardware.

WIDTH

The desired output width. When using the defaults for formatted output, IDL 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.

- Otherwise, a default of 80 columns is used.

The WIDTH keyword allows the user to override this default.

WRITE_TIMEOUT

Set this keyword to the number of seconds to wait to send data before giving up and issuing an error. By default, IDL blocks indefinitely until it is possible to send the data. Typically, this option is unnecessary on a local network, but it is useful with networks that are slow or unreliable.

UNIX-Only Keywords

STDIO

Under UNIX, forces the file to be opened via the standard C I/O library (stdio) rather than any other more native OS API that might usually be used. This is primarily of interest to those who intend to access the file from external code, and is not necessary for most uses.

Note

Under Windows, the STDIO feature is not possible. Requesting it causes IDL to throw an error.

Example

Most UNIX systems maintain a daytime server on the daytime port (port 13). There servers send a 1 line response when connected to, containing the current time of day.

```
; To obtain the current time from the host bullwinkle:
SOCKET, 1, 'bullwinkle', 'daytime'
date= ''
READF, 1, date
CLOSE, 1
PRINT, date
```

IDL prints:

```
Wed Sep 15 17:20:27 1999
```

SPHER_HARM

The SPHER_HARM function returns the value of the spherical harmonic $Y_{lm}(\theta, \phi)$, $-l \leq m \leq l$, $l \geq 0$, which is a function of two coordinates on a spherical surface.

The spherical harmonics are related to the associated Legendre polynomial by:

$$Y_{lm}(\theta, \phi) = \sqrt{\frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}} P_l^m(\cos \theta) e^{im\phi}$$

For negative m the following relation is used:

$$Y_{l,-m}(\theta, \phi) = (-1)^m Y_{lm}^*(\theta, \phi)$$

where $*$ represents the complex conjugate.

This routine is written in the IDL language. Its source code can be found in the file `spher_harm.pro` in the `lib` subdirectory of the IDL distribution.

Syntax

Result = SPHER_HARM(*Theta*, *Phi*, *L*, *M*, [, /DOUBLE])

Return Value

SPHER_HARM returns a complex scalar or array containing the value of the spherical harmonic function. The return value has the same dimensions as the input arguments *Theta* and *Phi*. If one argument (*Theta* or *Phi*) is a scalar and the other argument is an array, the function uses the scalar value with each element of the array, and returns an array with the same dimensions as the input array.

If either *Theta* or *Phi* are double-precision or if the DOUBLE keyword is set, the result is double-precision complex, otherwise the result is single-precision complex.

Arguments

Theta

The value of the polar (colatitudinal) coordinate θ at which $Y_{lm}(\theta, \phi)$ is evaluated. *Theta* can be either a scalar or an array.

Phi

The value of the azimuthal (longitudinal) coordinate ϕ at which $Y_{lm}(\theta, \phi)$ is evaluated. *Phi* can be either a scalar or an array.

L

A scalar integer, $L \geq 0$, specifying the order l of $Y_{lm}(\theta, \phi)$. If L is of type float, it will be truncated.

M

A scalar integer, $-L \leq M \leq L$, specifying the azimuthal order m of $Y_{lm}(\theta, \phi)$. If M is of type float, it will be truncated.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

Example

This example visualizes the electron probability density for the hydrogen atom in state 3d0. (Feynman, Leighton, and Sands, 1965: The Feynman Lectures on Physics, Calif. Inst. Tech, Ch. 19):

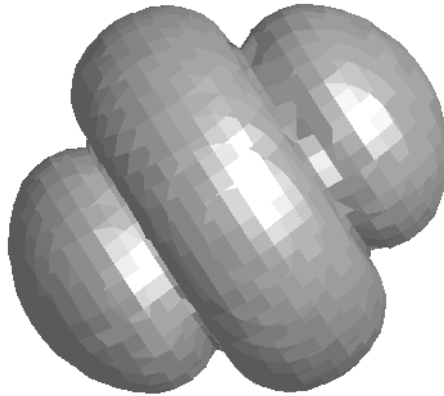
```
; Define a data cube (N x N x N)
n = 41L
a = 60*FINDGEN(n)/(n-1) - 29.999 ; [-1,+1]
x = REBIN(a, n, n, n)           ; X-coordinates of cube
y = REBIN(REFORM(a,1,n), n, n, n) ; Y-coordinates
z = REBIN(REFORM(a,1,1,n), n, n, n); Z-coordinates

; Convert from rectangular (x,y,z) to spherical (phi, theta, r)
spherCoord = CV_COORD(FROM_RECT= $
  TRANSPOSE([x[*]], [y[*]], [z[*]]]), /TO_SPHERE)
phi = REFORM(spherCoord[0,*], n, n, n)
theta = REFORM(!PI/2 - spherCoord[1,*], n, n, n)
r = REFORM(spherCoord[2,*], n, n, n)

; Find electron probability density for hydrogen atom in state 3d0
; Angular component
L = 2 ; state "d" is electron spin L=2
M = 0 ; Z-component of spin is zero
angularState = SPHER_HARM(theta, phi, L, M)
; Radial component for state n=3, L=2
radialFunction = EXP(-r/2)*(r^2)
```

```
waveFunction = angularState*radialFunction  
probabilityDensity = ABS(waveFunction)^2  
  
SHADE_VOLUME, probabilityDensity, $  
    0.1*MEAN(probabilityDensity), vertex, poly  
oPolygon = OBJ_NEW('IDLgrPolygon', vertex, $  
    POLYGON=poly, COLOR=[180,180,180])  
XOBJVIEW, oPolygon
```

The results are shown in the following figure (rotated in XOBJVIEW for clarity):



*Figure 3-8: SPHER_HARM Example of Hydrogen Atom
(object rotated in XOBJVIEW for clarity)*

See Also

[LEGENDRE](#), [LAGUERRE](#)

SWITCH

The SWITCH statement is used to select one statement for execution from multiple choices, depending upon the value of the expression following the word SWITCH.

Each statement that is part of a SWITCH statement is preceded by an expression that is compared to the value of the SWITCH expression. SWITCH executes by comparing the SWITCH expression with each selector expression in the order written. If a match is found, program execution jumps to that statement and execution continues from that point. Whereas CASE executes at most one statement within the CASE block, SWITCH executes the first matching statement and any following statements in the SWITCH block. Once a match is found in the SWITCH block, execution falls through to any remaining statements. For this reason, the BREAK statement is commonly used within SWITCH statements to force an immediate exit from the SWITCH block.

The ELSE clause of the SWITCH statement is optional. If included, it matches any selector expression, causing its code to be executed. For this reason, it is usually written as the last clause in the switch statement. The ELSE statement is executed only if none of the preceding statement expressions match. If an ELSE clause is not included and none of the values match the selector, program execution continues immediately below the SWITCH without executing any of the SWITCH statements.

SWITCH is similar to the CASE statement. For more information on using SWITCH and other IDL program control statements, as well as the differences between SWITCH and CASE, see [Chapter 11, “Program Control”](#) in *Building IDL Applications*.

Syntax

```
SWITCH expression OF
    expression: statement
    ...
    expression: statement
ELSE: statement
ENDSWITCH
```

Example

This example illustrates how, unlike CASE, SWITCH executes the first matching statement and any following statements in the SWITCH block:

```
x=2

SWITCH x OF
  1: PRINT, 'one'
  2: PRINT, 'two'
  3: PRINT, 'three'
  4: PRINT, 'four'
ENDSWITCH
```

IDL Prints:

```
two
three
four
```

See Also

[CASE](#)

TIMEGEN

The TIMEGEN function returns an array, with specified dimensions, of double-precision floating-point values that represent times in terms of Julian dates.

The Julian date is the number of days elapsed since Jan. 1, 4713 B.C.E., plus the time expressed as a day fraction. Following the astronomical convention, the day is defined to start at 12 PM (noon). Julian date 0.0d is therefore Jan. 1, 4713 B.C.E. at 12:00:00.

The first value of the returned array corresponds to a Julian date start time, and each subsequent value corresponds to the next Julian date in the sequence. The sequence is determined by specifying the time unit (such as months or seconds) and the step size, or spacing, between the units. You can also construct more complicated arrays by including smaller time units within each major time interval.

A small offset is added to each Julian date to eliminate roundoff errors when calculating the day fraction from the hour, minute, second. This offset is given by the larger of EPS and EPS*Julian, where Julian is the integer portion of the Julian date and EPS is the double-precision floating-point precision parameter from [MACHAR](#). For typical Julian dates the offset is approximately 6×10^{-10} (which corresponds to 5×10^{-5} seconds). This offset ensures that when the Julian date is converted back to the hour, minute, and second, the hour, minute, and second will have the same integer values.

Tip

Because of the large magnitude of the Julian date (1 Jan 2000 is Julian day 2451545), the precision of most Julian dates is limited to 1 millisecond (0.001 seconds). If you are not interested in the date itself, you can improve the precision by subtracting a large offset or setting the START keyword to zero.

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000, respectively.

Syntax

```
Result = TIMEGEN( [D1,...,D8 | , FINAL=value] [, DAYS=vector]
[, HOURS=vector] [, MINUTES=vector] [, MONTHS=vector] [, SECONDS=vector]
[, START=value] [, STEP_SIZE=value] [, UNITS=string] [, YEAR=value] )
```


Arguments

D_i

The dimensions of the result. The dimension parameters may be any scalar expression. Up to eight dimensions may be specified. If the dimension arguments are not integer values, IDL will truncate them to integer values before creating the new array. The dimension arguments are required unless keyword **FINAL** is set, in which case they are ignored.

Keywords

DAYS

Set this keyword to a scalar or a vector giving the day values that should be included within each month. This keyword is ignored if the **UNITS** keyword is set to “Days”, “Hours”, “Minutes”, or “Seconds”.

Note

Day values that are beyond the end of the month will be set equal to the last day for that month. For example, setting `DAY=[31]` will automatically return the last day in each month.

FINAL

Set this keyword to a double-precision value representing the Julian date/time to use as the last value in the returned array. In this case, the dimension arguments are ignored and *Result* is a one-dimensional array, with the number of elements depending upon the step size. The **FINAL** time may be less than the **START** time, in which case `STEP_SIZE` should be negative.

Note

If the step size is not an integer then the last element may not be equal to the **FINAL** time. In this case, **TIMEGEN** will return enough elements such that the last element is less than or equal to **FINAL**.

HOURS

Set this keyword to a scalar or a vector giving the hour values that should be included within each day. This keyword is ignored if **UNITS** is set to “Hours”, “Minutes”, or “Seconds”.

MINUTES

Set this keyword to a scalar or a vector giving the minute values that should be included within each hour. This keyword is ignored if UNITS is set to “Minutes” or “Seconds”.

MONTHS

Set this keyword to a scalar or a vector giving the month values that should be included within each year. This keyword is ignored if UNITS is set to “Months”, “Days”, “Hours”, “Minutes”, or “Seconds”.

SECONDS

Set this keyword to a scalar or a vector giving the second values that should be included within each minute. This keyword is ignored if UNITS is set to “Seconds”.

START

Set this keyword to a double-precision value representing the Julian date/time to use as the first value in the returned array. The default is 0.0d [corresponding to January 1, 4713 B.C.E. at 12 pm (noon)].

Note

If subintervals are provided by MONTHS, DAYS, HOURS, MINUTES, or SECONDS, then the first element may not be equal to the START time. In this case the first element in the returned array will be greater than or equal to START.

Tip

Other array generation routines in IDL (such as FINDGEN) do not allow you to specify a starting value because the resulting array can be added to a scalar representing the start value. For TIMEGEN it is correct to add a scalar to the array if the units are days, hours, minutes, seconds, or sub-seconds. For example:

```
MyTimes = TIMEGEN(365, UNITS="Days") + SYSTIME(/JULIAN)
```

However, if the units are months or years, the start value is necessary because the number of days in a month or year can vary depending upon the year in which they fall (for instance, consider leap years). For example:

```
MyTimes = TIMEGEN(12, UNITS="Months", START=JULDAY(1,1,2000))
```

STEP_SIZE

Set this keyword to a scalar value representing the step size between the major intervals of the returned array. The step size may be negative. The default step size is 1. When the UNITS keyword is set to “Years” or “Months”, the STEP_SIZE value is rounded to the nearest integer.

UNITS

Set this keyword to a scalar string indicating the time units to be used for the major intervals for the generated array. Valid values include:

- “Years” or “Y”
- “Months” or “M”
- “Days” or “D”
- “Hours” or “H”
- “Minutes” or “I”
- “Seconds” or “S”

The case (upper or lower) is ignored. If this keyword is not specified, then the default for UNITS is the time unit that is larger than the largest keyword present:

Largest Keyword Present	Default UNITS
SECONDS= <i>vector</i>	“Minutes”
MINUTES= <i>vector</i>	“Hours”
HOURS= <i>vector</i>	“Days”
DAYS= <i>vector</i>	“Months”
MONTHS= <i>vector</i>	“Years”
YEAR= <i>value</i>	“Years”

Table 3-8: Defaults for the UNITS keyword

If none of the above keywords are present, the default is UNITS=“Days”.

YEAR

Set this keyword to a scalar giving the starting year. If YEAR is specified then the starting year from START is ignored.

Examples

- Generate an array of 366 time values that are one day apart starting with January 1, 2000:

```
MyDates = TIMEGEN(366, START=JULDAY(1,1,2000))
```

- Generate an array of 20 time values that are 12 hours apart starting with the current time:

```
MyTimes = TIMEGEN(20, UNITS='Hours', STEP_SIZE=12, $
    START=SYSTIME(/JULIAN))
```

- Generate an array of time values that are 1 hour apart from 1 January 2000 until the current time:

```
MyTimes = TIMEGEN(START=JULDAY(1,1,2000), $
    FINAL=SYSTIME(/JULIAN), UNITS='Hours')
```

- Generate an array of time values composed of seconds, minutes, and hours that start from the current hour:

```
MyTimes = TIMEGEN(60, 60, 24, $
    START=FLOOR(SYSTIME(/JULIAN)*24)/24d, UNITS='S')
```

- Generate an array of 24 time values with monthly intervals, but with subintervals at 5 PM on the first and fifteenth of each month:

```
MyTimes = TIMEGEN(24, START=FLOOR(SYSTIME(/JULIAN)), $
    DAYS=[1,15], HOURS=17)
```

See Also

“Format Codes” in Chapter 8 of Building IDL Applications, [CALDAT](#), [JULDAY](#), [LABEL_DATE](#), [SYSTIME](#)

WV_CWT

The `WV_CWT` function returns the one-dimensional continuous wavelet transform of the input *Array*. The transform is done using a user-inputted wavelet function.

Syntax

```
Result = WV_CWT(Array, Family, Order [, /DOUBLE]  
               [, DSCALE=scalar] [, NSCALE=scalar] [, /PAD]  
               [, SCALE=variable] [, START_SCALE=scalar])
```

Return Value

The result is a two-dimensional array of type complex or double complex, containing the continuous wavelet transform of the input *Array*.

Arguments

Array

A one-dimensional array of length *N*, of floating-point or complex type.

Family

A scalar string giving the name of the wavelet function to use for the transform.

Order

The order number, or parameter, for the wavelet function given by *Family*.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

DSCALE

Set this keyword to a scalar value giving the spacing between scale values, in logarithmic units. The default is 0.25, which gives four subscales within each major scale.

NSCALE

Set this keyword to a scalar value giving the total number of scale values to use for the wavelet transform. The default is $\lceil \log_2(N/START_SCALE) \rceil / DSCALE + 1$.

PAD

Set this keyword to force *Array* to be padded with zeroes before computing the transform. Enough zeroes are added to make the total length of *Array* equal to the next-higher power-of-two greater than $2N$. Padding with zeroes prevents wraparound of the *Array* and speeds up the fast Fourier transform.

Note

Padding with zeroes reduces, but does not eliminate, edge effects caused by the discontinuities at the start and end of the data.

SCALE

Set this keyword to a named variable in which to return the scale values used for the continuous wavelet transform. The *SCALE* values range from *START_SCALE* up to $\text{START_SCALE} \cdot 2^{[(\text{NSCALE}-1)\text{DSCALE}]}$.

START_SCALE

Set this keyword to a scalar value giving the starting scale, in non-dimensional units. The default is 2, which gives a starting scale that is twice the spacing between *Array* elements.

Reference

Torrence and Compo, 1998: A Practical Guide to Wavelet Analysis. *Bull. Amer. Meteor. Soc.*, **79**, 61–78.

See Also

[WV_FN_MORLET](#), [WV_FN_PAUL](#)

WV_DENOISE

The WV_DENOISE function uses the wavelet transform to filter (or de-noise) a multi-dimensional array.

WV_DENOISE computes the discrete wavelet transform of *Array*, and then discards wavelet coefficients smaller than a certain threshold. WV_DENOISE then computes the inverse wavelet transform on the filtered coefficients and returns the result.

Syntax

```
Result = WV_DENOISE(Array [, Family, Order]  
    [, COEFFICIENTS=value] [, CUTOFF=variable]  
    [, DENOISE_STATE=variable] [, /DOUBLE]  
    [, DWT_FILTERED=variable] [, PERCENT=value]  
    [, THRESHOLD=value] [, WPS_FILTERED=variable])
```

Return Value

The result is an array of the same dimensions as the input *Array*. If *Array* is double precision or /DOUBLE is set then the result is double precision, otherwise the result is single precision.

Arguments

Array

A one-dimensional array of length N, of floating-point or complex type.

Family

A scalar string giving the name of the wavelet function to use for the transform.

Order

The order number, or parameter, for the wavelet function given by *Family*. If not specified the default for the wavelet function will be used.

Note

If you pass in a DENOISE_STATE structure, then *Family* and *Order* may be omitted. In this case the values from DENOISE_STATE are used.

Keywords

COEFFICIENTS

Set this keyword to a scalar specifying the number of wavelet coefficients to retain in the filtered wavelet transform. This keyword is ignored if keyword PERCENT is present.

CUTOFF

Set this keyword to a named variable that, upon return, will contain the cutoff value of wavelet power that was used for the threshold.

DENOISE_STATE

This is both an input and an output keyword. If this keyword is set to a named variable, then on exit, DENOISE_STATE will contain the following structure:

Tag	Type	Definition
FAMILY	STRING	Name of the wavelet function used.
ORDER	DOUBLE	Order for the wavelet function.
DWT	FLT/DBLARR	Discrete wavelet transform of <i>Array</i>
WPS	FLT/DBLARR	Wavelet power spectrum, equal to $ DWT ^2$
SORTED	FLT/DBLARR	Percent-normalized WPS, sorted
CUMULATIVE	FLT/DBLARR	Cumulative sum of SORTED
COEFFICIENTS	LONG	Number of coefficients retained
PERCENT	DOUBLE	Percent of coefficients retained

Table 3-9: The structure tags for DENOISE_STATE.

Note

If the DOUBLE keyword is set then the arrays will be of type double.

Upon input, if DENOISE_STATE is set to a structure with the above form, then DWT, WPS, SORTED, and CUMULATIVE will not be recomputed by WV_DENOISE. This is useful if you want to make multiple calls to WV_DENOISE using the same *Array*.

Warning

No error checking is made on the input values. The values should not be modified between calls to `DENOISE_STATE`.

DOUBLE

Set this keyword to force the computation to be done using double-precision arithmetic.

DWT_FILTERED

Set this keyword to a named variable in which the filtered discrete wavelet transform will be returned.

PERCENT

Set this keyword to a scalar specifying the percentage of cumulative power to retain.

Note

If neither `COEFFICIENTS` nor `PERCENT` is present then all of the coefficients are retained (i.e. no filtering is done).

THRESHOLD

Set this keyword to a scalar specifying the type of threshold. The actual threshold, T , is set using `COEFFICIENTS` or `PERCENT`. Possible values are:

Value	Description
0	Hard threshold (this is the default). The hard threshold sets all wavelet coefficients with magnitude less than or equal to T to zero.
1	Soft threshold. The soft threshold sets all $DWT[i]$ with magnitude less than T to zero, and also linearly reduces the magnitude of the each retained wavelet coefficient by T : Positive coefficients are set equal to $DWT[i] - T$, while negative coefficients are set equal to $DWT[i] + T$.

Table 3-10: THRESHOLD Values

WPS_FILTERED

Set this keyword to a named variable in which the filtered wavelet power spectrum will be returned.

Example

Remove the noise from a 256 x 256 image:

```
image = dist(256) + 10*randomn(1, 256, 256)
; Keep only 100 out of 65536 coefficients:
denoise = WV_DENOISE(image, 'Daubechies', COEFF=100, $
    DENOISE_STATE=denoise_state)

window, xsize=512, ysize=300
tvsc1, image, 0
tvsc1, denoise, 1
xyouts, [128, 384], [10, 10], ['Image', 'Filtered'], $
    /device, align=0.5, charsize=2
print, 'Percent of power retained: ', denoise_state.percent
```

IDL prints:

```
Percent of power retained:          99.973450
```

Change to a “soft” threshold (use DENOISE_STATE to avoid re-computing):

```
denoise2 = WV_DENOISE(image, COEFF=100, $
    DENOISE_STATE=denoise_state, THRESHOLD=1)
```

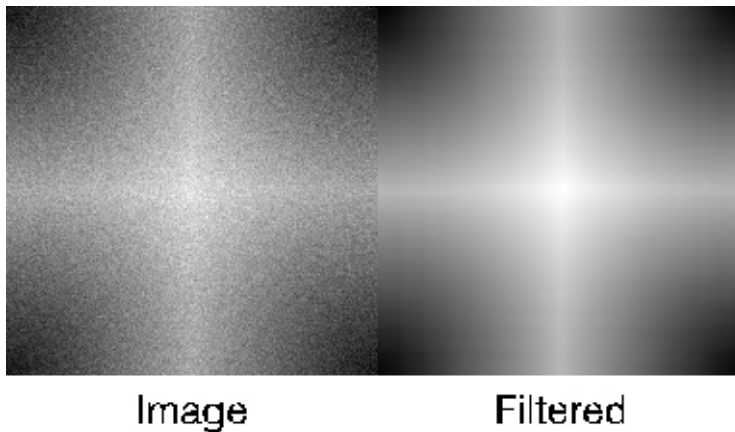


Figure 3-9: Example of de-noising an image.

WV_FN_GAUSSIAN

The WV_FN_GAUSSIAN function constructs wavelet coefficients for the Gaussian wavelet function. In real space, the Gaussian wavelet function is proportional to the m -th order derivative of a Gaussian, $\exp(-x^2/2)$. The Gaussian second derivative, $(x^2 - 1) \exp(-x^2/2)$, is often referred to as the Marr wavelet.

Syntax

Result = WV_FN_GAUSSIAN([*Order*] [, *Scale*, *N*]
[, /DOUBLE] [, FREQUENCY=*variable*] [, /SPATIAL] [, WAVELET=*variable*])

Return Value

The returned value of this function is an anonymous structure of information about the particular wavelet.

Tag	Type	Definition
FAMILY	STRING	'Gaussian'
ORDER_NAME	STRING	'Derivative'
ORDER_RANGE	DBLARR(3)	Valid orders [first, last, default]
ORDER	DOUBLE	The chosen <i>Order</i>
DISCRETE	INT	0 [0=continuous, 1=discrete]
ORTHOGONAL	INT	0 [0=nonorthogonal, 1=orthogonal]
SYMMETRIC	INT	1 [0=asymmetric, 1=symm.]
SUPPORT	DOUBLE	Infinity [Compact support width]
MOMENTS	INT	1 [Number of vanishing moments]
REGULARITY	DOUBLE	Infinity [Number of continuous derivatives]
E_FOLDING	DOUBLE	SQRT(2) [Autocorrelation e -fold distance]
FOURIER_PERIOD	DOUBLE	Ratio of Fourier wavelength to scale

Table 3-11: The structure tags for *Result*.

Arguments

Order

A scalar that specifies the non-dimensional order parameter for the wavelet. The default is 2.

Scale

A scalar that specifies the scale at which to construct the wavelet function.

N

An integer that specifies the number of points in the wavelet function. For Fourier space (SPATIAL=0), the frequencies are constructed following the FFT convention:

For N even: 0, 1/N, 2/N, ..., (N-2)/(2N), 1/2, -(N-2)/(2N), ..., -1/N.

For N odd: 0, 1/N, 2/N, ..., (N-1)/(2N), -(N-1)/(2N), ..., -1/N.

For real space (/SPATIAL), the spatial coordinates are $-(N-1)/2 \dots (N-1)/2$.

Note

If none of the above arguments are present then the function will simply return the *Result* structure using the default *Order*.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

FREQUENCY

Set this keyword to a named variable in which to return the frequency array used to construct the wavelet. This variable will be undefined if SPATIAL is set.

SPATIAL

Set this keyword to return the wavelet function in real space. The default is to return the wavelet function in Fourier space.

WAVELET

Set this keyword to a named variable in which to return the wavelet function.

Reference

Torrence and Compo, 1998: A Practical Guide to Wavelet Analysis. *Bull. Amer. Meteor. Soc.*, **79**, 61–78.

Example

Plot the Gaussian wavelet function at scale=20:

```
n = 1000 ; pick a nice number of points
info = WV_FN_GAUSSIAN( 2, 20, n, /SPATIAL, $
    WAVELET=wavelet)
plot, wavelet
```

Now plot the same wavelet in Fourier space:

```
info = WV_FN_GAUSSIAN( 2, 20, n, $
    FREQUENCY=frequency, WAVELET=wave_fourier)
plot, frequency, wave_fourier, $
    xrange=[-0.2,0.2], thick=2
```

WV_FN_MORLET

The WV_FN_MORLET function constructs wavelet coefficients for the Morlet wavelet function. In real space, the Morlet wavelet function consists of a complex exponential modulated by a Gaussian envelope: $\pi^{-1/4} s^{-1/2} \exp[i k x / s] \exp[-(x / s)^2 / 2]$, where s is the wavelet scale, k is a non-dimensional parameter, and x is the position.

Syntax

```
Result = WV_FN_MORLET( [Order] [, Scale, N]
[, /DOUBLE] [, FREQUENCY=variable] [, /SPATIAL] [, WAVELET=variable])
```

Return Value

The returned value of this function is an anonymous structure of information about the particular wavelet.

Tag	Type	Definition
FAMILY	STRING	'Morlet'
ORDER_NAME	STRING	'Parameter'
ORDER_RANGE	DBLARR(3)	[3, 24, 6] Valid orders [first, last, default]
ORDER	DOUBLE	The chosen <i>Order</i>
DISCRETE	INT	0 [0=continuous, 1=discrete]
ORTHOGONAL	INT	0 [0=nonorthogonal, 1=orthogonal]
SYMMETRIC	INT	1 [0=asymmetric, 1=symm.]
SUPPORT	DOUBLE	Infinity [Compact support width]
MOMENTS	INT	1 [Number of vanishing moments]
REGULARITY	DOUBLE	Infinity [Number of continuous derivatives]
E_FOLDING	DOUBLE	SQRT(2) [Autocorrelation e -fold distance]
FOURIER_PERIOD	DOUBLE	Ratio of Fourier wavelength to scale

Table 3-12: The structure tags for Result.

Arguments

Order

A scalar that specifies the non-dimensional order parameter for the wavelet. The default is 6.

Scale

A scalar that specifies the scale at which to construct the wavelet function.

N

An integer that specifies the number of points in the wavelet function. For Fourier space (SPATIAL=0), the frequencies are constructed following the FFT convention:

For N even: 0, 1/N, 2/N, ..., (N-2)/(2N), 1/2, -(N-2)/(2N), ..., -1/N.

For N odd: 0, 1/N, 2/N, ..., (N-1)/(2N), -(N-1)/(2N), ..., -1/N.

For real space (/SPATIAL), the spatial coordinates are $-(N-1)/2 \dots (N-1)/2$.

Note

If none of the above arguments are present then the function will simply return the *Result* structure using the default *Order*.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

FREQUENCY

Set this keyword to a named variable in which to return the frequency array used to construct the wavelet. This variable will be undefined if SPATIAL is set.

SPATIAL

Set this keyword to return the wavelet function in real space. The default is to return the wavelet function in Fourier space.

WAVELET

Set this keyword to a named variable in which to return the wavelet function.

Reference

Torrence and Compo, 1998: A Practical Guide to Wavelet Analysis. *Bull. Amer. Meteor. Soc.*, **79**, 61–78.

Example

Plot the Morlet wavelet function at scale=100:

```
n = 1000 ; pick a nice number of points
info = WV_FN_MORLET( 6, 100, n, /SPATIAL, $
    WAVELET=wavelet)
plot, float(wavelet), THICK=2
oplot, imaginary(wavelet)
```

Now plot the same wavelet in Fourier space:

```
info = WV_FN_MORLET( 6, 100, n, $
    FREQUENCY=frequency, WAVELET=wave_fourier)
plot, frequency, wave_fourier, $
    xrange=[-0.2,0.2], thick=2
```


WV_FN_PAUL

The WV_FN_PAUL function constructs wavelet coefficients for the Paul wavelet function. In real space, the Paul wavelet function is proportional to the complex polynomial $(1 - i x / s)^{(-m-1)}$, where s is the wavelet scale, m is a non-dimensional parameter, and x is the position.

Syntax

```
Result = WV_FN_PAUL( [Order] [, Scale, N]
[, /DOUBLE] [, FREQUENCY=variable] [, /SPATIAL] [, WAVELET=variable])
```

Return Value

The returned value of this function is an anonymous structure of information about the particular wavelet.

Tag	Type	Definition
FAMILY	STRING	'Paul'
ORDER_NAME	STRING	'Parameter'
ORDER_RANGE	DBLARR(3)	[1, 20, 4] Valid orders [first, last, default]
ORDER	DOUBLE	The chosen <i>Order</i>
DISCRETE	INT	0 [0=continuous, 1=discrete]
ORTHOGONAL	INT	0 [0=nonorthogonal, 1=orthogonal]
SYMMETRIC	INT	1 [0=asymmetric, 1=symm.]
SUPPORT	DOUBLE	Infinity [Compact support width]
MOMENTS	INT	1 [Number of vanishing moments]
REGULARITY	DOUBLE	Infinity [Number of continuous derivatives]
E_FOLDING	DOUBLE	1/sqrt(2) [Autocorrelation e -fold distance]
FOURIER_PERIOD	DOUBLE	Ratio of Fourier wavelength to scale

Table 3-13: The structure tags for Result.

Arguments

Order

A scalar that specifies the non-dimensional order for the wavelet. The default is 4.

Scale

A scalar that specifies the scale at which to construct the wavelet function.

N

An integer that specifies the number of points in the wavelet function. For Fourier space (SPATIAL=0), the frequencies are constructed following the FFT convention:

For N even: 0, 1/N, 2/N, ..., (N-2)/(2N), 1/2, -(N-2)/(2N), ..., -1/N.

For N odd: 0, 1/N, 2/N, ..., (N-1)/(2N), -(N-1)/(2N), ..., -1/N.

For real space (/SPATIAL), the spatial coordinates are $-(N-1)/2 \dots (N-1)/2$.

Note

If none of the above arguments are present then the function will simply return the *Result* structure using the default *Order*.

Keywords

DOUBLE

Set this keyword to force the computation to be done in double-precision arithmetic.

FREQUENCY

Set this keyword to a named variable in which to return the frequency array used to construct the wavelet. This variable will be undefined if SPATIAL is set.

SPATIAL

Set this keyword to return the wavelet function in real space. The default is to return the wavelet function in Fourier space.

WAVELET

Set this keyword to a named variable in which to return the wavelet function.

Reference

Torrence and Compo, 1998: A Practical Guide to Wavelet Analysis. *Bull. Amer. Meteor. Soc.*, **79**, 61–78.

Example

Plot the Paul wavelet function at scale=100:

```
n = 1000 ; pick a nice number of points
info = WV_FN_PAUL( 6, 100, n, /SPATIAL, $
    WAVELET=wavelet)
plot, float(wavelet), THICK=2
oplot, imaginary(wavelet)
```

Now plot the same wavelet in Fourier space:

```
info = WV_FN_PAUL( 6, 100, n, $
    FREQUENCY=frequency, WAVELET=wave_fourier)
plot, frequency, wave_fourier, $
    xrange=[-0.2,0.2], thick=2
```

XDXF

The XDXF procedure is a utility for displaying and interactively manipulating DXF objects.

Syntax

```
XDXF [, Filename] [, /BLOCK] [, GROUP=widget_id] [, /MODAL]  
[, SCALE=value] [, /TEST] [keywords to XOBJVIEW]
```

Arguments

Filename

A string specifying the name of the DXF file to display. If this argument is not specified, a file selection dialog is opened.

Keywords

XDXF accepts the keywords to [XOBJVIEW](#). In addition, XDXF supports the following keywords:

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XDXF block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

The widget ID of the widget that calls XDXF. When this ID is specified, the death of the caller results in the death of XDXF.

MODAL

Set this keyword to block processing of events from other widgets until the user quits XDXF. A group leader must be specified (via the GROUP keyword) for the MODAL keyword to have any effect. By default, XDXF does not block event processing.

SCALE

Set this keyword to the zoom factor for the initial view. The default is $1/\text{SQRT}(3)$. This default value provides the largest possible view of the object, while ensuring that no portion of the object will be clipped by the XDXF window, regardless of the object's orientation.

TEST

If this keyword is set, the file `heart.dxf` in the IDL distribution is automatically opened in XDXF.

Using XDXF

XDXF displays a resizable top-level base with a menu and draw widget used to display and manipulate the orientation of a DXF object.

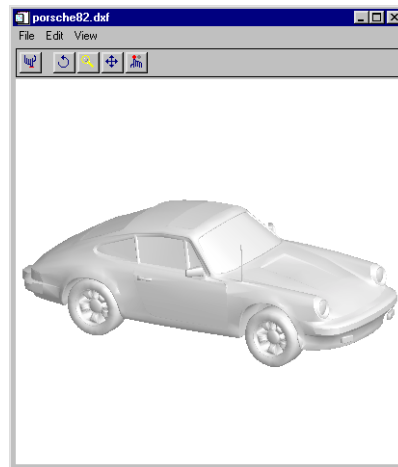


Figure 3-10: The XDXF Utility

XDXF also displays a dialog that contains block and layer information and allows the user to turn on and off the display of individual layers.

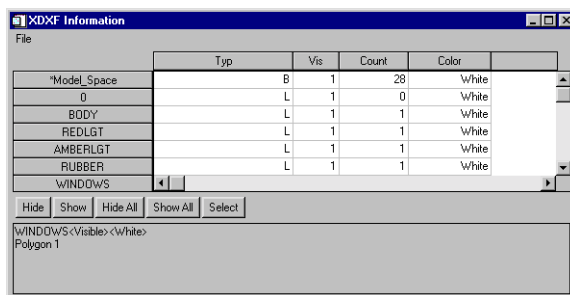


Figure 3-11: The XDXF Information Dialog

The XDXF Toolbar

The XDXF toolbar contains the following buttons:



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the object and drag to rotate.



Pan: Click the left mouse button on the object and drag to pan.



Zoom: Click the left mouse button on the object and drag to zoom in or out.



Select: Click on the object. The name of the selected object is displayed, if the object has a name, otherwise its class is displayed.

The XDXF Information Dialog

The XDXF Information dialog displays information about the blocks and layers contained in the currently displayed object, and allows you to turn on and off the display of each layer.

To show or hide layers in the DXF object, select the layer from the list of layers on the left of the dialog, and click the **Show** or **Hide** button. Alternatively, you can click in the “Vis” field for the desired layer. To show or hide all layers, click the **Show All** or **Hide All** buttons.

Example

Display the file `heart.dxf`, contained in the IDL distribution:

```
XDXF, FILEPATH('heart.dxf', $  
    SUBDIR=['examples', 'data'])
```

See Also

[IDLffDXF](#)

XPCOLOR

The XPCOLOR procedure is a utility that allows you to adjust the value of the current plotting color (foreground) using sliders, and store the desired color in the global system variable, !P.COLOR.

When XPCOLOR is called from the IDL input command line, the **Set Plot Color** dialog box appears. The dialog has two buttons (**Done** and **Help**) a single color swatch window, three sliders, and a pulldown menu with the four color systems: red, green, blue (RGB); cyan, magenta, yellow (CMY); hue, saturation, value (HSV); and hue, lightness, and saturation (HLS).

When you have chosen the color system and adjusted the sliders to your liking, click **Done** to store the color selected in the !P.COLOR system variable. Any plots generated in IDL afterwards use the color selected as the plotting (foreground) color until !P.COLOR is changed again.

Note

For a more flexible color editor, use the XPALETTE User Library routine.

This routine is written in the IDL language. Its source code can be found in the file `xpcolor.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XPCOLOR [, GROUP=widget_id ]
```

Arguments

None.

Keywords

GROUP

Set this keyword to the group leader widget ID as passed to XMANAGER.

XPLOT3D

The XPLOT3D procedure is a utility for creating and interactively manipulating 3D plots.

This routine is written in the IDL language. Its source code can be found in the file `xplot3d.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XPLOT3D, X, Y, Z [, /BLOCK] [, COLOR=[r,g,b]] [, /DOUBLE_VIEW]
[, GROUP=widget_id] [, LINESYLE={0 | 1 | 2 | 3 | 4 | 5 | 6}] [, /MODAL]
[, NAME=string] [, /OVERPLOT] [, SYMBOL=objref(s)] [, /TEST]
[, THICK=points{ 1.0 to 10.0}] [, TITLE=string] [, XRANGE=[min, max]]
[, YRANGE=[min, max]] [, ZRANGE=[min, max]] [, XTITLE=string]
[, YTITLE=string] [, ZTITLE=string]
```

Arguments

X

A vector of X data values.

Y

A vector of Y data values.

Z

A vector of Z data values.

Keywords

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XPLOT3D block, any earlier calls to XMANAGER must have been called with the

NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

COLOR

Set this keyword to an $[r, g, b]$ triplet specifying the color of the curve.

DOUBLE_VIEW

Set this keyword to cause XPLOT3D to set the DOUBLE property on the IDLgrView that it uses to display the plot.

GROUP

Set this keyword to the widget ID of the widget that calls XPLOT3D. When this keyword is specified, the death of the caller results in the death of XPLOT3D.

LINestyle

Set this keyword to a value indicating the line style that should be used to draw the curve. The value can be either an integer value specifying a pre-defined line style, or a 2-element vector specifying a stippling pattern.

To use a pre-defined line style, set the LINestyle keyword to one of the following integer values:

- 0 = Solid line (the default)
- 1 = dotted
- 2 = dashed
- 3 = dash dot
- 4 = dash dot dot dot
- 5 = long dash
- 6 = no line drawn

To define your own stippling pattern, specify a two-element vector $[repeat, bitmask]$, where repeat indicates the number of times consecutive runs of 1s or 0s in the bitmask should be repeated. (That is, if three consecutive 0s appear in the bitmask and the value of repeat is 2, then the line that is drawn will have six consecutive bits turned off.) The value of repeat must be in the range $1 \leq repeat \leq 255$.

The bitmask indicates which pixels are drawn and which are not along the length of the line. The bitmask is most conveniently specified as a 16-bit hexadecimal value.

For example, `LINESTYLE = [2, 'F0F0'X]` describes a dashed line (8 bits on, 8 bits off, 8 bits on, 8 bits off).

MODAL

Set this keyword to block processing of events from other widgets until the user quits `XPLOT3D`. The `MODAL` keyword does not require a group leader to be specified. If no group leader is specified, and the `MODAL` keyword is set, `XPLOT3D` fabricates an invisible group leader for you.

Note

To be modal, `XPLOT3D` does not require that its caller specify a group leader. This is unlike other IDL widget procedures such as `XLOADCT`, which, to be modal, do require that their caller specify a group leader. These other procedures were implemented this way to encourage the caller to create a modal widget that will be well-behaved with respect to layering and iconizing. (See [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) on page 1536 for more information.)

To provide a simple means of invoking `XPLOT3D` as a modal widget in applications that contain no other widgets, `XPLOT3D` can be invoked as `MODAL` without specifying a group leader, in which case `XPLOT3D` fabricates an invisible group leader for you. For applications that contain multiple widgets, however, it is good programming practice to supply an appropriate group leader when invoking `XPLOT3D`, `/MODAL`. As with other IDL widget procedures with names prefixed with “X”, specify the group leader via the `GROUP` keyword.

NAME

Set this keyword to a string specifying the name for the data curve being plotted. The name is displayed on the `XPLOT3D` toolbar when the curve is selected with the mouse. (To select the curve with the mouse, `XPLOT3D` must be in select mode. You can put `XPLOT3D` in select mode by clicking on the rightmost button on the `XPLOT3D` toolbar.)

OVERPLOT

Set this keyword to draw the curve in the most recently created view. The `TITLE`, `[XYZ]TITLE`, `[XYZ]RANGE`, and `MODAL` keywords are ignored if this keyword is set.

SYMBOL

Set this keyword to a vector containing one or more instances of the `IDLgrSymbol` object class to indicate the plotting symbols to be used at each vertex of the polyline.

If there are more vertices than elements in **SYMBOL**, the elements of the **SYMBOL** vector are cyclically repeated. By default, no symbols are drawn. To remove symbols from a polyline, set **SYMBOL** to a scalar.

TEST

If set, the *X*, *Y*, and *Z* arguments are not required (and are ignored if provided). A sinusoidal curve is displayed instead. This allows you to test code that uses **XPLOT3D** without having to specify plot data.

THICK

Set this keyword to a value between 1.0 and 10.0, specifying the line thickness to be used to draw the polyline, in points. The default is 1.0 points.

TITLE

Set this keyword to a string to appear in the **XPLOT3D** title bar.

XRANGE

Set this keyword to a 2-element array of the form [*min*, *max*] specifying the X-axis range.

YRANGE

Set this keyword to a 2-element array of the form [*min*, *max*] specifying the Y-axis range.

ZRANGE

Set this keyword to a 2-element array of the form [*min*, *max*] specifying the Z-axis range.

XTITLE

Set this keyword to a string specifying the title for the X axis of the plot.

YTITLE

Set this keyword to a string specifying the title for the Y axis of the plot.

ZTITLE

Set this keyword to a string specifying the title for the Z axis of the plot.

Using XPLOT3D

XPLOT3D displays a resizable top-level base with a menu, toolbar and draw widget, as shown in the following figure:

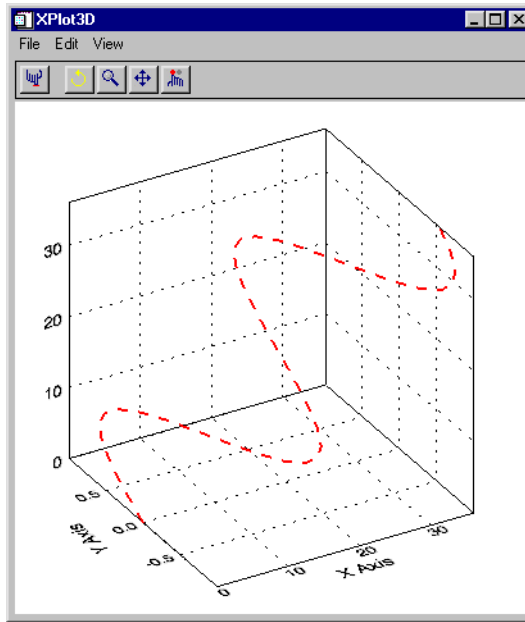


Figure 3-12: The XPLOT3D Utility

The XPLOT3D Toolbar

The XPLOT3D toolbar contains the following buttons:



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the plot and drag to rotate.



Zoom: Click the left mouse button on the plot and drag to zoom in or out.



Pan: Click the left mouse button on the plot and drag to pan.



Select: Click on a curve to display the curve name (if defined with the **NAME** keyword) on the XPLOT3D toolbar. If no name was defined for the curve, “IDLGRPOLYLINE” is displayed.

Projecting Data onto Plot “Walls”

To turn on or off the projection of data onto the walls of the box enclosing the 3D plot, select **All On**, **All Off**, **XY**, **YZ**, or **XZ** from the **View** → **2D Projection** menu.

Changing the Axis Type

The **View** → **Axes** menu allows you to select one of the following types of axes:

- Simple Axes — displays the X, Y, and Z axes as lines.
- Box Axes — displays the X, Y, and Z axes as planes.
- No Axes — turns off the display of axes.

Example

The following example displays two curves in XPLOT3D, using a custom plotting symbol for one of the curves:

```
;Define plot data:
X = INDGEN(20)
Y1 = SIN(X/3.)
Y2 = COS(X/3.)
Z = X

;Display curve 1 in XPLOT3D:
XPLOT3D, X, Y1, Z, NAME='Curve1', THICK=2

;Define custom plotting symbols:
oOrb = OBJ_NEW('orb', COLOR=[0, 0, 255])
oOrb->Scale, .75, .1, .5
oSymbol = OBJ_NEW('IDLgrSymbol', oOrb)

;Overplot curve 2 in XPLOT3D:
XPLOT3D, X, Y2, Z, COLOR=[0,255,0], NAME='Curve2', $
    SYMBOL=oSymbol, THICK=2, /OVERPLOT
```

This code results in the following:

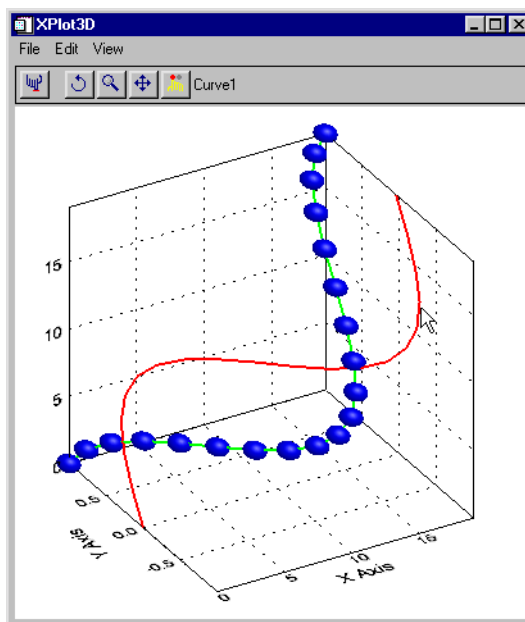


Figure 3-13: Two curves displayed in XPLOT3D

XROI

The XROI procedure is a utility for interactively defining regions of interest (ROIs), and obtaining geometry and statistical data about these ROIs.

This routine is written in the IDL language. Its source code can be found in the file `xroi.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Syntax

```
XROI [, ImageData] [, R] [, G] [, B] [, /BLOCK]
[ [, /FLOATING] , GROUP=widget_ID] [, /MODAL] [, REGIONS_IN=value]
[, REGIONS_OUT=value] [, REJECTED=variable] [, RENDERER={0 | 1}]
[, ROI_COLOR=[r, g, b] or variable] [, ROI_GEOMETRY=variable]
[, ROI_SELECT_COLOR=[r, g, b] or variable] [, STATISTICS=variable]
[, TITLE=string] [, TOOLS=string/string array {valid values are 'Freehand Draw',
'Polygon Draw', and 'Selection'}]
```

Arguments

ImageData

ImageData is both an input and output argument. It is an array representing an 8-bit or 24-bit image to be displayed. *ImageData* can be any of the following:

- [*m, n*] — 8-bit image
- [3, *m, n*] — 24-bit image
- [*m, 3, n*] — 24-bit image
- [*m, n, 3*] — 24-bit image

If *ImageData* is not supplied, the user will be prompted for a file via `DIALOG_PICKFILE`. On output, *ImageData* will be set to the current image data. (The current image data can be different than the input image data if the user imported an image via the **File** → **Import Image** menu item.)

R, G, B

R, *G*, and *B* are arrays of bytes representing red, green, or blue color table values, respectively. *R*, *G*, and *B* are both input and output arguments. On input, these values are applied to the image if the image is 8-bit. To get the red, green, or blue color table values for the image on output from XROI, specify a named variable for the appropriate argument. (If the image is 24-bit, this argument will output a 256-element

byte array containing the values given at input, or BINDGEN(256) if the argument was undefined on input.)

Keywords

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, BLOCK is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the BLOCK keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have XROI block, any earlier calls to XMANAGER must have been called with the NO_BLOCK keyword. See the documentation for the NO_BLOCK keyword to XMANAGER for an example.

FLOATING

Set this keyword, along with the GROUP keyword, to create a floating top-level base widget. If the windowing system provides Z-order control, floating base widgets appear above the base specified as their group leader. If the windowing system does not provide Z-order control, the FLOATING keyword has no effect.

Note

Floating widgets must have a group leader. Setting this keyword without also setting the GROUP keyword causes an error.

GROUP

Set this keyword to the widget ID of the widget that calls XROI. When this keyword is specified, the death of the caller results in the death of XROI.

MODAL

Set this keyword to block other IDL widgets from receiving events while XROI is active.

REGIONS_IN

Set this keyword to an array of IDLgrROI references. This allows you to open XROI with ROIs already defined. This is also useful when using a loop to open multiple images in XROI. By using the same named variable for both the `REGIONS_IN` and `REGIONS_OUT` keywords, you can reuse the same ROIs in multiple images (see [Example 2](#)). This keyword also accepts `-1`, or `OBJ_NEW()` (Null object) to indicate that there are no ROIs to read in. This allows you to assign the result of a previous `REGIONS_OUT` to `REGIONS_IN` without worrying about the case where the previous `REGIONS_OUT` is undefined.

REGIONS_OUT

Set this keyword to a named variable that will contain an array of IDLgrROI references. This keyword is assigned the null object reference if there are no ROIs defined. By using the same named variable for both the `REGIONS_IN` and `REGIONS_OUT` keywords, you can reuse the same ROIs in multiple images (see [Example 2](#)).

REJECTED

Set this keyword to a named variable that will contain those `REGIONS_IN` that are not in `REGIONS_OUT`. The objects defined in the variable specified for `REJECTED` can be destroyed with a call to `OBJ_DESTROY`, allowing you to perform cleanup on objects that are not required (see [Example 2](#)). This keyword is assigned the null object reference if no `REGIONS_IN` are rejected by the user.

RENDERER

Set this keyword to an integer value to indicate which graphics renderer to use when drawing objects within the window. Valid values are:

- 0 = Platform native OpenGL
- 1 = IDL's software implementation (the default)

ROI_COLOR

This keyword is both an input and an output parameter. Set this keyword to a 3-element byte array, $[r, g, b]$, indicating the color of ROI outlines when they are not selected. This color will be used by XROI unless and until the color is changed by the user via the “Unselected Outline Color” portion of the “ROI Outline Colors” dialog (which is accessed by selecting **Edit** → **Outline Colors**). If this keyword is assigned a named variable, that variable will be set to the current $[r, g, b]$ value at the time that XROI returns.

ROI_GEOMETRY

Set this keyword to a named variable that will contain an array of anonymous structures, one for each ROI that is valid when this routine returns. The structures will contain the following fields:

Field	Description
area	The area of the region of interest, in square pixels.
centroid	The coordinates (x , y , z) of the centroid of the region of interest, in pixels.
perimeter	The perimeter of the region of interest, in pixels.

Table 3-14: Fields of the structure returned by ROI_GEOMETRY

If there are no valid regions of interest when this routine returns, ROI_GEOMETRY will be undefined.

Note

If there are no REGIONS_IN, XROI must either be modal or must block control flow in order for ROI_GEOMETRY to be defined upon exit from XROI. Otherwise, XROI will return before an ROI can be defined, and ROI_GEOMETRY will therefore be undefined.

ROI_SELECT_COLOR

This keyword is both an input and an output parameter. Set this keyword to a 3-element byte array, $[r, g, b]$, indicating the color of ROI outlines when they are selected. This color will be used by XROI unless and until the color is changed by the user via the “Selected Outline Color” portion of the “ROI Outline Colors” dialog (which is accessed by selecting **Edit** → **Outline Colors**). If this keyword is assigned a named variable, that variable will be set to the current $[r, g, b]$ value at the time that XROI returns.

STATISTICS

Set this keyword to a named variable to receive an array of anonymous structures, one for each ROI that is valid when this routine returns. The structures will contain the following fields:

Field	Description
count	Number of pixels in region.
minimum	Minimum pixel value.
maximum	Maximum pixel value.
mean	Mean pixel value.
stddev	Standard deviation of pixel values.

Table 3-15: Fields of the structure returned by STATISTICS

If *ImageData* is 24-bit, or if there are no valid regions of interest when the routine exits, STATISTICS will be undefined.

Note

If there are no REGIONS_IN, XROI must either be modal or must block control flow in order for STATISTICS to be defined upon exit from XROI. Otherwise, XROI will return before an ROI can be defined, and STATISTICS will therefore be undefined.

TITLE

Set this keyword to a string to appear in the XROI title bar.

TOOLS

Set this keyword a string or vector of strings from the following list to indicate which ROI manipulation tools should be supported when XROI is run:

- 'Freehand Draw' — Freehand ROI drawing. Mouse down begins a region, mouse motion adds vertices to the region (following the path of the mouse), mouse up finishes the region.
- 'Polygon Draw' — Polygon ROI drawing. Mouse down begins a region, subsequent mouse clicks add vertices, double-click finishes the region.

- 'Selection' — ROI selection. Mouse down/up selects the nearest region. The nearest vertex in that region is identified with a crosshair symbol.

If more than one string is specified, a series of bitmap buttons will appear at the top of the XROI widget in the order specified (to the right of the fixed set of bitmap buttons used for saving regions, displaying region information, copying to clipboard, and flipping the image). If only one string is specified, no additional bitmap buttons will appear, and the manipulation mode is implied by the given string. If this keyword is not specified, bitmap buttons for all three manipulation tools are included on the XROI toolbar.

Using XROI

XROI displays a top-level base with a menu, toolbar and draw widget. After defining an ROI, the **ROI Information** window appears, as shown in the following figure:

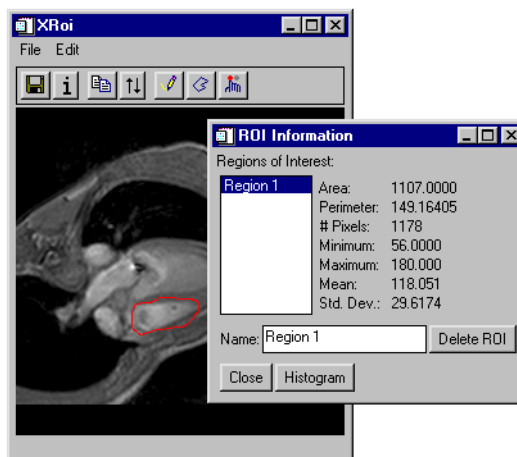






Figure 3-14: The XROI Utility




As you move the mouse over an image, the x and y pixel locations are shown in the status line on the bottom of the XROI window. For 8-bit images, the data value (z) is also shown. If an ROI is defined, the status line also indicates the mouse position relative to the ROI using the text “Inside”, “Outside”, “On Edge,” or “On Vertex.”

The XROI Toolbar

The XROI toolbar contains the following buttons:

	Save:	Opens a file selection dialog for saving the currently defined ROIs to a save file.
	Info:	Opens the ROI Information window.
	Copy:	Copies the contents of the display area to the clipboard.
	Flip:	Flips image vertically. Note that only the image is flipped; any ROIs that have been defined do not move.

Depending on the value of the **TOOLS** keyword, the XROI toolbar may also contain the following buttons:

	Draw Freehand:	Click this button to draw freehand ROIs. Mouse down begins a region, mouse motion adds vertices to the region (following the path of the mouse), mouse up finishes the region.
	Draw Polygon:	Click this button to draw polygon ROIs. Mouse down begins a region, subsequent mouse clicks add vertices, double-click finishes the region.
	Select:	Click this button to select an ROI region. Clicking the image causes a cross hairs symbol to be drawn at the nearest vertex of the selected ROI.

Importing an Image into XROI

To import an image into XROI, select **File** → **Import Image**. This opens a **DIALOG_READ_IMAGE** dialog, which can be used to preview and select an image.

Changing the Image Color Table

To change the color table properties for the current image, select **Edit** → **Image Color Table**. This opens the **CW_PALETTE_EDITOR** dialog, which is a compound widget used to edit color palettes. See [CW_PALETTE_EDITOR](#) for more information. This menu item is grayed out if the image does not have a color palette.

Changing the ROI Outline Colors

To change the outline colors for selected and unselected ROIs, select **Edit** → **Outline Colors**. This opens the **ROI Outline Colors** dialog, which consists of two CW_RGBSLIDER widgets for interactively adjusting the ROI outline colors. The left widget is used to define the color for the selected ROI, and the right widget is used to define the color of unselected ROIs. You can select the RGB, CMY, HSV, or HLS color system from the **Color System** drop-down list.

Viewing ROI Information

To view geometry and statistical data about the currently selected ROI, click the **Info** button or select **Edit** → **ROI Information**. This opens the **ROI Information** dialog, which displays area, perimeter, number of pixels, minimum and maximum pixel values, and standard deviation. Values for statistical information (minimum, maximum, mean, and standard deviation) appear as “N/A” for 24-bit images.

To view a histogram for the currently selected ROI, click the **Histogram** button. This opens a LIVE_PLOT dialog, which can be used to interactively control the plot properties.

Note

The **Histogram** button is enabled only for 8-bit images.

Deleting an ROI

To delete an ROI, do the following:

1. Click the **Info** button or select **Edit** → **ROI Information**. This opens the **ROI Information** dialog.
2. In the **ROI Information** dialog, select the ROI you wish to delete from the list of ROIs. You can also select an ROI by clicking the **Select** button on the XROI toolbar, then clicking on an ROI on the image.
3. Click the **Delete ROI** button.

Examples

Example 1

This example opens a single image in XROI:

```
image = READ_PNG(FILEPATH('mineral.png', $
    SUBDIR=['examples','data']))
XROI, image
```

Example 2

This example reads 3 images from the file `mr_abdomen.dcm`, and calls `XROI` for each image. A single list of regions is maintained, saving the user from having to redefine regions on each image:

```
;Read 3 images from mr_abdomen.dcm and open each one in XROI:
FOR i=0,2 DO BEGIN
    image = READ_DICOM(FILEPATH('mr_abdomen.dcm',$
        SUBDIR=['examples','data']), IMAGE_INDEX=i)
    XROI, image, r, g, b, REGIONS_IN=regions,$
        REGIONS_OUT=regions, ROI_SELECT_COLOR=roi_select_color,$
        ROI_COLOR=roi_color, REJECTED=rejected, /BLOCK
    OBJ_DESTROY, rejected
ENDFOR

OBJ_DESTROY, regions
```

Perform the following steps:

1. Draw an ROI on the first image, then close that `XROI` window. Note that the next image contains the ROI defined in the first image. This is accomplished by setting `REGIONS_IN` and `REGIONS_OUT` to the same named variable in the `FOR` loop of the above code.
2. Draw another ROI on the second image.
3. Click the **Select** button and select the first ROI. Then click the **Info** button to open the **ROI Information** window, and click the **Delete ROI** button.
4. Close the second `XROI` window. Note that the third image contains the ROI defined in the second image, but not the ROI deleted on the second image. This example sets the `REJECTED` keyword to a named variable, and calls `OBJ_DESTROY` on that variable. Use of the `REJECTED` keyword is not necessary to prevent deleted ROIs from appearing on subsequent images, but allows you perform cleanup on objects that are no longer required.

XVOLUME

The XVOLUME procedure is a utility for viewing and interactively manipulating volumes and isosurfaces.

This routine is written in the IDL language. Its source code can be found in the file `xvolume.pro` in the `lib/utilities` subdirectory of the IDL distribution.

Tip

The `XVOLUME_ROTATE` and `XVOLUME_WRITE_IMAGE` procedures, which can be called only after a call to `XVOLUME`, can be used to easily create animations of volumes and isosurfaces displayed in `XVOLUME`. See `XVOLUME_ROTATE` for an example.

Syntax

```
XVOLUME, Vol, [, /BLOCK] [, GROUP=widget_id] [, /INTERPOLATE]  
[, /MODAL] [, RENDERER={0 | 1}] [, /REPLACE] [, SCALE=value] [, /TEST]  
[, XSIZE=pixels] [, YSIZE=pixels]
```

Arguments

Vol

A 3-element array of the form `[x, y, z]` that specifies a data volume.

Keywords

BLOCK

Set this keyword to have XMANAGER block when this application is registered. By default, `BLOCK` is set equal to zero, providing access to the command line if active command line processing is available. Note that setting the `BLOCK` keyword causes all widget applications to block, not just this application. For more information, see the documentation for the [NO_BLOCK](#) keyword to XMANAGER.

Note

Only the outermost call to XMANAGER can block. Therefore, to have `XVOLUME` block, any earlier calls to XMANAGER must have been called with the

NO_BLOCK keyword. See the documentation for the [NO_BLOCK](#) keyword to XMANAGER for an example.

GROUP

Set this keyword to the widget ID of the widget that calls XVOLUME. When this keyword is specified, the death of the caller results in the death of XVOLUME.

INTERPOLATE

Set this keyword to indicate that trilinear interpolation is to be used when rendering the volume and the image planes. Setting this keyword improves the quality of images produced, at the cost of more computing time, especially when the volume has low resolution with respect to the size of the viewing plane. Nearest neighbor sampling is used by default.

MODAL

Set this keyword to block processing of events from other widgets until the user quits XVOLUME. The MODAL keyword does not require a group leader to be specified. If no group leader is specified, and the MODAL keyword is set, XVOLUME fabricates an invisible group leader for you.

Note

To be modal, XVOLUME does not require that its caller specify a group leader. This is unlike other IDL widget procedures such as XLOADCT, which, to be modal, do require that their caller specify a group leader. These other procedures were implemented this way to encourage the caller to create a modal widget that will be well-behaved with respect to layering and iconizing. (See [“Iconizing, Layering, and Destroying Groups of Top-Level Bases”](#) on page 1536 for more information.)

To provide a simple means of invoking XVOLUME as a modal widget in applications that contain no other widgets, XVOLUME can be invoked as MODAL without specifying a group leader, in which case XVOLUME fabricates an invisible group leader for you. For applications that contain multiple widgets, however, it is good programming practice to supply an appropriate group leader when invoking XVOLUME, /MODAL. As with other IDL widget procedures with names prefixed with “X”, specify the group leader via the GROUP keyword.

RENDERER

Set this keyword to an integer value indicating which graphics renderer to use when drawing objects within the window. Valid values are:

- 0 = Platform native OpenGL (the default)
- 1 = IDL's software implementation

REPLACE

If this keyword is set, and there is a current instance of `XVOLUME` running, the volume displayed in `XVOLUME` is replaced with the volume specified by *Vol*. For example, display volume1 using the command

```
XVOLUME, volume1
```

To replace volume1 with volume2, you would use the command

```
XVOLUME, volume2, /REPLACE
```

SCALE

Set this keyword to the zoom factor for the initial view. The default is $1/\text{SQRT}(3)$. This default value provides the largest possible view of the volume, while ensuring that no portion of the volume will be clipped by the `XVOLUME` window, regardless of the volume's orientation.

TEST

If set, the *Vol* argument is not required (and is ignored if provided). A volume of random numbers is displayed instead. This allows you to test code that uses `XVOLUME` without having to specify volume data.

XSIZE

The width of the drawable area in pixels.

YSIZE

The height of the drawable area in pixels.

Using XVOLUME

XVOLUME displays a resizable top-level base with a toolbar, a menu, a graphical interface for controlling volume and isosurface properties, and a draw widget for displaying and manipulating the volume, as shown in the following figure:

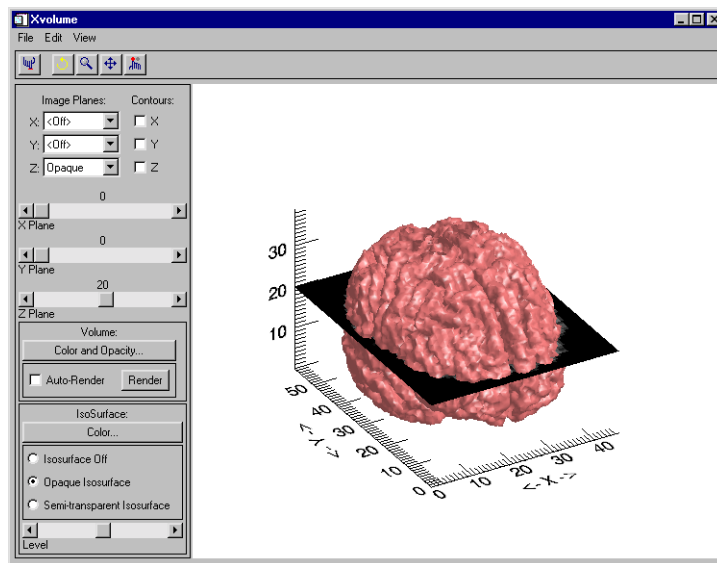


Figure 3-15: The XVOLUME Utility

The XVOLUME Toolbar

The XVOLUME toolbar contains the following buttons.

Note

If you have the **Auto-Render** option selected, the Rotate, Zoom, and Pan features may be more difficult to use. For the best performance while manipulating the orientation of a volume using these features, deselect the **Auto-Render** option.



Reset: Resets rotation, scaling, and panning.



Rotate: Click the left mouse button on the volume and drag to rotate.



Zoom: Click the left mouse button on the volume and drag to zoom in or out.



Pan: Click the left mouse button on the volume and drag to pan.



Select: Click in the draw widget to identify the selected item. A name identifying the selected item is displayed next to the Select button.

The XVOLUME Interface

The XVOLUME interface provides the following elements for controlling the display of image planes and contours, volumes, and isosurfaces:

Image Planes and Contours

Image planes and contours allow you to visualize the values associated with the volume or isosurface at a specified X, Y, or Z plane.

- **Image Planes:** Select one of the following options from the dropdown list for each dimension to control the display of image planes:
 - **Off:** Turns off the image plane display.
 - **Opaque:** Displays an opaque image plane at the location specified by the corresponding plane slider.
 - **Transparent:** Displays a transparent image plane at the location specified by the corresponding plane slider. The transparency value of the plane is taken from the volume at the current location of the image plane.
- **Contours:** Check this option to display contours on the specified plane at the location specified by corresponding the plane slider.
- **Plane Sliders:** Move these sliders to change the position of the plane in each dimension.

Volume

- **Color and Opacity:** Click this button to change the color and/or opacity of the current volume. This opens a `CW_PALETTE_EDITOR` dialog, which is a compound widget used to edit color palettes. See [CW_PALETTE_EDITOR](#) for more information.
- **Auto-Render:** Select this option to have rendering executed automatically after each change you make to the volume. If **Auto-Render** is unchecked, you

must manually click the **Render** button to see changes you have made to the volume. If **Auto-Render** is checked, the **Render** button will be grayed out.

- **Render:** Click on this button to execute rendering computations and display the current volume. If **Auto-Render** is checked, this button will be grayed out.

Isosurface

An isosurface is a 3D surface on which the data values are constant along the entire surface. Use the following elements to control the appearance of the isosurface:

- **Color:** Click this button to change the color system and/or values for the current isosurface. This opens a CW_RGBSLIDER dialog, which is a compound widget that provides a drop-down list for selecting the RGB, CMY, HSV, or HLS color system, and three sliders for adjusting the values associated with each color system.
- **Isosurface Off:** Select this option to turn off the isosurface display.
- **Opaque Isosurface:** Select this option to display an opaque isosurface.
- **Semi-transparent Isosurface:** Select this option to display a semi-transparent isosurface.
- **Level:** Use this slider to adjust the threshold value of the isosurface.

Example

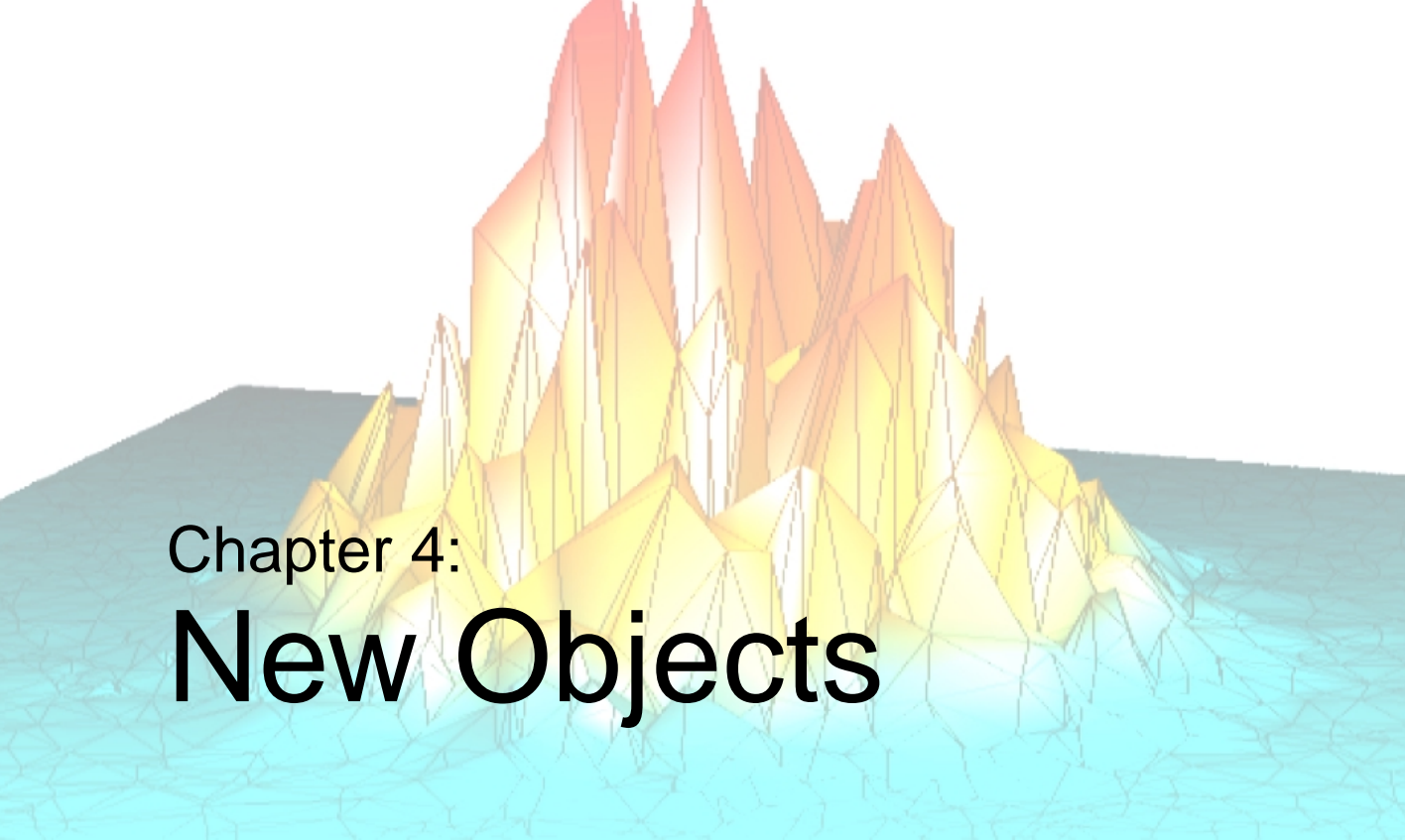
Create a volume and display using XVOLUME:

```
; Create a volume:
vol = BYTSCL(RANDOMU((SEED=0),5,5,5))
vol = CONGRID(vol, 30,30,30)

; Display volume:
XVOLUME, vol
```

See Also

[XVOLUME_ROTATE](#), [XVOLUME_WRITE_IMAGE](#), [IDLgrVolume](#), [ISOSURFACE](#), [SHADE_VOLUME](#), [SLICER3](#), “Volume Objects” in Chapter 26 of *Using IDL*.



Chapter 4: New Objects

This chapter describes IDL Objects introduced in IDL 5.4.

IDLffShape	280
----------------------------------	-----

IDLffShape

An IDLffShape object contains geometry, connectivity and attributes for graphics primitives accessed from ESRI Shapefiles.

Superclasses

This class has no superclass.

Subclasses

This class has no subclasses.

Creation

See IDLffShape::Init

Methods

Intrinsic Methods

This class has the following methods:

- [IDLffShape::AddAttribute](#)
- [IDLffShape::Cleanup](#)
- [IDLffShape::Close](#)
- [IDLffShape::DestroyEntity](#)
- [IDLffShape::GetAttributes](#)
- [IDLffShape::GetEntity](#)
- [IDLffShape::GetProperty](#)
- [IDLffShape::Init](#)
- [IDLffShape::Open](#)
- [IDLffShape::PutEntity](#)
- [IDLffShape::SetAttributes](#)

Overview

An ESRI Shapefile stores nontopological geometry and attribute information for the spatial features in a data set.

A Shapefile consists of a main file (.shp), an index file (.shx), and a dBASE table (.dbf). For example, the Shapefile “states” would have the following files:

- states.shp
- states.shx
- states.dbf

Naming Conventions for a Shapefile

All the files that comprise an ESRI Shapefile must adhere to the 8.3 filename convention and must be lower case. The main file, index file, and dBASE file must all have the same prefix. The prefix must start with an alphanumeric character and can contain any alphanumeric, underscore (_), or hyphen (-). The main file suffix must use the .shp extension, the index file the .shx extension, and the dBASE table the .dbf extension.

Major Elements of a Shapefile

A Shapefile consists of the following elements that you can access through the IDLffShape class:

- Entities
- Attributes

Entities

The geometry for a feature is stored as a shape comprising a set of vector coordinates (referred to as ‘entities’). The entities in a Shapefile must all be of the same type. The following are the possible types for entities in a Shapefile:

Shape Type	Type Code
Point	1
PolyLine	3
Polygon	5

Table 4-1: Entity Types

Shape Type	Type Code
MultiPoint	8
PointZ	11
PolyLineZ	13
PolygonZ	15
MultiPointZ	18
PointM	21
PolyLineM	23
PolygonM	25
MultiPointM	28
MultiPatch	31

Table 4-1: Entity Types (Continued)

When retrieving entities using the `IDLffShape::GetEntity` method, an IDL structure is returned. This structure has the following fields:

Field	Data Type
SHAPE_TYPE	IDL_LONG
ISHAPE	IDL_LONG
BOUNDS	Double[8]
N_VERTICES	IDL_LONG
VERTICES	Pointer (to Vertices array)
MEASURE	Pointer (to Measure array)
N_PARTS	IDL_LONG
PARTS	Pointer (to Parts array).
PART_TYPES	Pointer (to part types)
ATTRIBUTES	Pointer to attribute array.

Table 4-2: Entity Structure Field Data Types

The following table describes each field in the structure:

Field	Description
SHAPE_TYPE	The entity type.
ISHAPE	The identifier of the specific entity in the shape object.
BOUNDS	<p>A bounding box that specifies the range limits of the entity. This eight element array contains the following information:</p> <ul style="list-style-type: none"> • Index 0 — X minimum value • Index 1 — Y minimum value • Index 2 — Z minimum value (if Z is supported by type) • Index 3 — Measure minimum value (if measure is supported by entity type). • Index4 — X maximum value. • Index5 — Y maximum value. • Index6 — Z maximum value (if Z is supported by the entity type). • Index7 — Measure maximum value (if measure is supported by entity type). <p>Note - If the entity is a point type, the values contained in the bounds array are also the values of the entity.</p>
N_VERTICES	The number of vertices in the entity. If this value is one and the entity is a POINT type (POINT, POINTM, POINTZ), the vertices pointer will be set to NULL and the entity value will be maintained in the BOUNDS field.

Table 4-3: Entity Structure Field Descriptions

Field	Description
VERTICES	<p>An IDL pointer that contains the vertices of the entity. This pointer contains a double array that has one of the following formats:</p> <ul style="list-style-type: none"> • [2, N] - If Z data is not present • [3, N] - If Z data is present. <p>where N is the number of vertices. These array formats can be passed to the polygon and polyline objects of IDL Object Graphics.</p> <p>Note - This pointer will be null if the entity is a point type, with the values maintained in the BOUNDS array.</p>
MEASURE	<p>If the entity has a measure value (this is dependent on the entity type), this IDL pointer will contain a vector array of measure values. The length of this vector is N_VERTICES.</p> <p>Note - This pointer will be null if the entity is of type POINTM, with the values contained in the BOUNDS array.</p>
N_PARTS	<p>If the values of the entity are separated into parts, the break points are enumerated in the parts array. This field lists the number of parts in this entity. If this value is 0, the entity is one part and the PARTS pointer will be NULL.</p>
PARTS	<p>An IDL pointer that contains an array of indices into the vertex/measure arrays. These values represent the start of each part of the entity. The index range of each entity part is defined by the following:</p> <ul style="list-style-type: none"> • Start = Parts[I] • End = Parts[I+1]-1 or the end of the array
PART_TYPES	<p>This IDL pointer is only valid for entities of type MultiPatch and defines the type of the particular part. If the entity type is not MultiPatch, part types are assumed to be type RING (SHPP_RING).</p> <p>Note - This pointer is NULL if the entity is not type MultiPatch.</p>

Table 4-3: Entity Structure Field Descriptions (Continued)

Field	Description
ATTRIBUTES	If the attributes for an entity were requested, this field contains an IDL pointer that contains a structure of attributes for the entity. For more information on this structure, see “Attributes” on page 285.

Table 4-3: Entity Structure Field Descriptions (Continued)

Attributes

A Shapefile provides the ability to associate information describing each entity (a geometric element) contained in the file. This descriptive information, called attributes, consists of a set of named data elements for each geometric entity contained in the file. The set of available attributes is the same for every entity contained in a Shapefile, with each entity having its own set of attribute values.

An attribute consists of two components:

- A name
- A data value

The name consists of an 11 character string that is used to identify the data value. The data value is not limited to any specific format.

The two components that form an attribute are accessed differently using the shape object. To get the name of attributes for the specific file, the `ATTRIBUTE_NAMES` keyword to the `IDLffShape::GetProperty` method is used. This returns a string array that contains the names for the attributes defined for the file.

To get the attribute values for an entity, the `IDLffShape::GetAttributes` method is called or the `ATTRIBUTES` keyword of the `IDLffShape::GetEntity` method is set. In each case, the attribute values for the specified entity is returned as an anonymous IDL structure. The numeric order of the fields in the returned structure map to the numeric order of the attributes defined for the file. The actual format of the returned structure is:

```
ATTRIBUTE_0 : VALUE,
ATTRIBUTE_1 : VALUE,
ATTRIBUTE_2 : VALUE,
...
ATTRIBUTE_<N-1> : VALUE
```

To access the values in the returned structure, you can either hardcode the structure field names or use the structure indexing feature of IDL.

Accessing Shapefiles

The following example shows how to access data in a Shapefile. This example sets up a map to display parts of a Shapefile, opens a Shapefile, reads the entities from the Shapefile, and then plots only the state of Colorado:

```
PRO ex_shapefile

DEVICE, RETAIN=2, DECOMPOSED=0
!P.BACKGROUND=255

;Define a color table
r=BYTARR(256) & g=BYTARR(256) & b=BYTARR(256)
r[0]=0 & g[0]=0 & b[0]=0           ;Definition of black
r[1]=100 & g[1]=100 & b[1]=255      ;Definition of blue
r[2]=0 & g[2]=255 & b[2]=0          ;Definition of green
r[3]=255 & g[3]=255 & b[3]=0        ;Definition of yellow
r[255]=255 & g[255]=255 & b[255]=255 ;Definition of white

TVLCT, r, g, b
black=0 & blue=1 & green=2 & yellow=3 & white=255

; Set up map to plot Shapefile on
MAP_SET, /ORTHO,45, -120, /ISOTROPIC, $
/HORIZON, E_HORIZON={FILL:1, COLOR:blue}, $
/GRID, COLOR=black, /NOBORDER

; Fill the continent boundaries:
MAP_CONTINENTS, /FILL_CONTINENTS, COLOR=green

; Overplot coastline data:
MAP_CONTINENTS, /COASTS, COLOR=black

; Show national borders:
MAP_CONTINENTS, /COUNTRIES, COLOR=black

;Open the states Shapefile in the examples directory
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
SUBDIR=['examples', 'data']))

;Get the number of entities so we can parse through them
myshape -> IDLffShape::GetProperty, N_ENTITIES=num_ent

;Parsing through the entities and only plotting the state of
;Colorado
```

```

FOR x=1, (num_ent-1) DO BEGIN
    ;Get the Attributes for entity x
    attr = myshape -> IDLffShape::GetAttributes(x)
    ;See if 'Colorado' is in ATTRIBUTE_1 of the attributes for
    ;entity x
    IF attr.ATTRIBUTE_1 EQ 'Colorado' THEN BEGIN
        ;Get entity
        ent = myshape -> IDLffShape::GetEntity(x)
        ;Plot entity
        POLYFILL, (*ent.vertices)[0,*], (*ent.vertices)[1,*],
        COLOR=yellow
        ;Clean-up of pointers
        myshape -> IDLffShape::DestroyEntity, ent
    ENDIF
ENDFOR

;Close the Shapefile
OBJ_DESTROY, myshape

END

```

This results in the following:

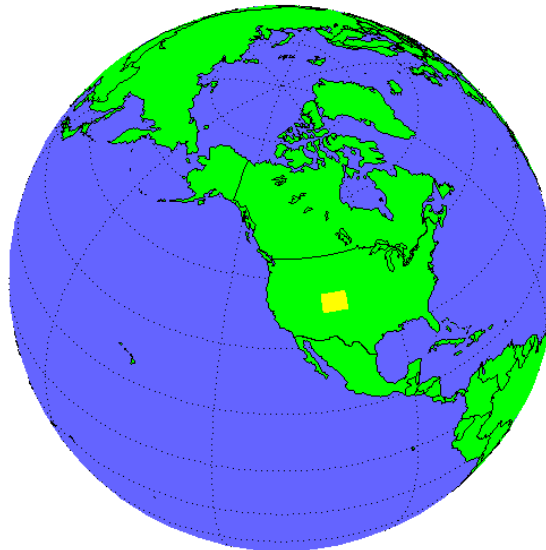


Figure 4-1: Example Use of Shapefiles

Creating New Shapefiles

To create a Shapefile, you need to create a new Shapefile object, define the entity and attributes definitions, and then add your data to the file. For example, the following program creates a new Shapefile (`cities.shp`), defines the entity type to be “Point”, defines 2 attributes (`CITY_NAME` and `STATE_NAME`), and then adds an entity to the new file:

```
PRO ex_shapefile_newfile

;Create the new shapefile and define the entity type to Point
mynewshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
    SUBDIR=['examples', 'data']), /UPDATE, ENTITY_TYPE=1)

;Set the attribute definitions for the new Shapefile
mynewshape->IDLffShape::AddAttribute, 'CITY_NAME', 7, 25, $
    PRECISION=0
mynewshape->IDLffShape::AddAttribute, 'STAT_NAME', 7, 25, $
    PRECISION=0

;Create structure for new entity
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity
entNew.SHAPE_TYPE = 1
entNew.ISHAPE = 1458
entNew.BOUNDS[0] = -104.87270
entNew.BOUNDS[1] = 39.768040
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -104.87270
entNew.BOUNDS[5] = 39.768040
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

;Create structure for new attributes
attrNew = mynewshape ->IDLffShape::GetAttributes( $
    /ATTRIBUTE_STRUCTURE)

;Define the values for the new attributes
attrNew.ATTRIBUTE_0 = 'Denver'
attrNew.ATTRIBUTE_1 = 'Colorado'

;Add the new entity to new shapefile
mynewshape -> IDLffShape::PutEntity, entNew

;Add the Colorado attributes to new shapefile
mynewshape -> IDLffShape::SetAttributes, 0, attrNew
```



```

;Close the shapefile
OBJ_DESTROY, mynewshape

END

```

Updating Existing Shapefiles

You can modify existing Shapefiles with the following:

- Adding new entities
- Adding new attributes (only to Shapefiles without any existing values in any attributes)
- Modifying existing attributes

Note

You cannot modify existing entities.

For example, the following program adds an entity and attributes for the city of Boulder to the `cities.shp` file we created in the previous example:

```

PRO ex_shapefile_modify

;Open the cities Shapefile
myshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
    SUBDIR=['examples', 'data']), /UPDATE)

;Create structure for new entity
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity
entNew.SHAPE_TYPE = 1
entNew.ISHAPE = 1380
entNew.BOUNDS[0] = -105.25100
entNew.BOUNDS[1] = 40.026878
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -105.25100
entNew.BOUNDS[5] = 40.026878
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

;Create structure for new attributes
attrNew = myshape ->IDLffShape::GetAttributes( $
    /ATTRIBUTE_STRUCTURE)

```

```
;Define the values for the new attributes
attrNew.ATTRIBUTE_0 = 'Boulder'
attrNew.ATTRIBUTE_1 = 'Colorado'

;Add the new entity to new shapefile
myshape -> IDLffShape::PutEntity, entNew

;Add the Colorado attributes to new shapefile
myshape -> IDLffShape::SetAttributes, 0, attrNew

;Close the shapefile
OBJ_DESTROY, myshape

END
```

IDLffShape::AddAttribute

The IDLffShape::AddAttribute method adds an attribute definition to a Shapefile. Adding a the attribute definition is required before adding the actual attribute data to a file. For more information on attributes, see [“Attributes”](#) on page 285.

Note

You can only define new attributes to Shapefiles that do not have any existing values in any attributes.

Syntax

Obj->[IDLffShape::]AddAttribute, *Name*, *Type*, *Width* [, PRECISION=*integer*]

Arguments

Name

Set to a string that contains the attribute name. Name values are limited to 11 characters. Arguments longer than 11 characters will be truncated.

Type

Set to the IDL type code that corresponds to the data type that will be stored in the attribute. The valid types are:

Code	Description
3	Longword Integer
5	Double-precision floating-point
7	String

Table 4-4: Type Code Descriptions

Width

Set to the width of the field for the data value of the attribute. The following table describes the possible values depending on the defined Type:

Field Type	Valid Values
Longword Integer	Maximum size of the field.
Double-precision floating-point	Maximum size of the field.
String	Maximum length of the string.

Table 4-5: Width Values

Keywords

PRECISION

Set this keyword to the number of positions to be included after the decimal point. The default is 8. This keyword is only valid for fields defined as double-precision floating-point.

Example

In the following example, we add the attribute “ELEVATION” to an existing Shapefile. Note that if the file already contains data in an attribute for any of the entities defined in the file, this operation will fail.

```
PRO ex_addattr_shapefile

;Open a shapefile
myshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
    SUBDIR=['examples', 'data']), /UPDATE)

;Define a new attribute for the Shapefile
myshape->IDLffShape::AddAttribute, 'ELEVATION', 3, 4, $
    PRECISION=0

;Close the shapefile
OBJ_DESTROY, myshape

END
```

IDLffShape::Cleanup

The IDLffShape::Cleanup method performs all cleanup on a Shapefile object. If the Shapefile being accessed by the object is open and the file has been modified, the new information is written to the file if one of the following conditions is met:

- The file was opened with write permissions using the UPDATE keyword to the IDLffShape::Open method
- It is a newly created file that has not been written previously.

Note

Cleanup methods are special lifecycle methods, and as such cannot be called outside the context of object destruction. This means that in most cases, you cannot call the Cleanup method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Cleanup method from within the Cleanup method of the subclass.

Syntax

OBJ_DESTROY, *Obj*

or

Obj -> [IDLffShape::]Cleanup (Only in subclass' Cleanup method.)

Arguments

None

Keywords

None

IDLffShape::Close

The IDLffShape::Close method closes a Shapefile. If the file has been modified, it is also written to the disk if neither of the following conditions is met:

- The file was opened with write permissions using the UPDATE keyword to the IDLffShape::Open method
- It is a newly created file that has not been written previously.

If the file has been modified and one of the previous conditions is not met, the file is closed and the changes are not written to disk.

Syntax

Obj->[IDLffShape::]Close

Arguments

None.

Keywords

None.

IDLffShape::DestroyEntity

The IDLffShape::DestroyEntity method frees memory associated with the entity structure. For more information on the entity structure, see [“Entities”](#) on page 281.

Syntax

Obj->[IDLffShape::]DestroyEntity, *Entity*

Arguments

Entity

This argument specifies a scalar or array of entities to destroy.

Keywords

None.

Example

In the following example, all of the entities from the `states.shp` Shapefile are read and then the DestroyEntity method is called to clean up all pointers:

```
PRO ex_shapefile

;Open the states Shapefile in the examples directory
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))

;Get the number of entities so we can parse through them
myshape -> IDLffShape::GetProperty, N_ENTITIES=num_ent

;Read all the entities
FOR x=1, (num_ent-1) DO BEGIN
    ;Read the entity x
    ent = myshape -> IDLffShape::GetEntity(x)
    ;Clean-up of pointers
    myshape -> IDLffShape::DestroyEntity, ent
ENDFOR

;Close the Shapefile
OBJ_DESTROY, myshape

END
```

IDLffShape::GetAttributes

The IDLffShape::GetAttributes method retrieves the attributes for the entities you specify from a Shapefile.

Syntax

```
Result = Obj->[IDLffShape::]GetAttributes([Index] [, /ALL]  
[, /ATTRIBUTE_STRUCTURE] )
```

Return Value

This method returns an anonymous structure array. For more information on the structure, see “[Attributes](#)” on page 285.

Arguments

Index

A scalar or array of longs specifying the entities for which you want to retrieve the attributes, with 0 being the first entity in the Shapefile.

Note

If you do not specify *Index* and the ALL keyword is not set, the attributes for the first entity (0) are returned.

Keywords

ALL

Set this keyword to retrieve the attributes for all entities in a Shapefile. If you set this keyword, the *Index* argument is not required.

ATTRIBUTE_STRUCTURE

Set this keyword to return an empty attribute structure that can then be used with the [IDLffShape::SetAttributes](#) method to add attributes to a Shapefile.

Examples

In the first example, we retrieve the attributes associated with entity at location 0 (the first entity in the file):

```
attr = myShape->getAttributes( 0)
```


In the next example, we retrieve the attributes associated with entities 10 through 20:

```
attr = myShape->getAttributes( 10+indgen(11) )
```

In the next example, we retrieve the attributes for entities 1,4, 9 and 70:

```
attr = myShape->getAttributes( [1, 4, 9, 70] )
```

In the next example, we retrieve all the attributes for a Shapefile:

```
attr = myShape->getAttributes( /ALL )
```

IDLffShape::GetEntity

The IDLffShape::GetEntity method returns the entities you specify from a Shapefile.

Syntax

```
Result = Obj->[IDLffShape::]GetEntity( [Index] [, /ALL] [, /ATTRIBUTES] )
```

Return Value

This method returns a type {IDL_SHAPE_ENTITY} structure array. For more information on the structure, see “Entities” on page 281.

Note

Since an entity structure contains IDL pointers, you must free all the pointers returned in these structures when the entity is no longer needed using the [IDLffShape::DestroyEntity](#) method.

Note

Since entities cannot be modified in a Shapefile, an entity is read directly from the Shapefile each time you use the IDLffShape::GetEntity method even if you have already read that entity. If you modify the structure array returned by this method for a given entity and then use IDLffShape::GetEntity on that same entity, the modified data will NOT be returned, the data that is actually written in the file is returned.

Arguments

Index

A scalar or array of longs specifying the entities for which you want to retrieve with 0 being the first entity in the Shapefile. If the ALL keyword is set, this argument is not required. If you do not specify any entities and the ALL keyword is not set, the first entity (0) is returned.

Keywords

ALL

Set this keyword to retrieve all entities from the Shapefile. If this keyword is set, the Index argument is not required.

ATTRIBUTES

Set this keyword to return the attributes in the entity structure. If not set, the ATTRIBUTES tag in the entity structure will be a null IDL pointer.

Example

In the following example, all of the entities from the `states.shp` Shapefile are read:

```
PRO ex_shapefile

;Open the states Shapefile in the examples directory
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))

;Get the number of entities so we can parse through them
myshape -> IDLffShape::GetProperty, N_ENTITIES=num_ent

;Read all the entities
FOR x=1, (num_ent-1) DO BEGIN
    ;Read the entity x
    ent = myshape -> IDLffShape::GetEntity(x)
    ;Clean-up of pointers
    myshape -> IDLffShape::DestroyEntity, ent
ENDFOR

;Close the Shapefile
OBJ_DESTROY, myshape

END
```

IDLffShape::GetProperty

The IDLffShape::GetProperty method returns the values of properties associated with a Shapefile object. These properties are:

- Number of entities
- The type of the entities
- The number of attributes associated with each entity
- The names of the attributes
- The name, type, width, and precision of the attributes
- The status of a Shapefile
- The filename of the Shapefile object

Syntax

```
Obj->[IDLffShape::]GetProperty [, N_ENTITIES=variable]  
[, ENTITY_TYPE=variable] [, N_ATTRIBUTES=variable]  
[, ATTRIBUTE_NAMES=variable] [, ATTRIBUTE_INFO=variable]  
[, IS_OPEN=variable] [, FILENAME=variable]
```

Arguments

None.

Keywords

N_ENTITIES

Set this keyword to a named variable to return the number of entities contained in Shapefile object. If the value is unknown, this method returns 0.

ENTITY_TYPE

Set this keyword to a named variable to return the integer type code for the entities contained in the Shapefile object. If the value is unknown, this method returns -1. For more information on entity type codes, see [“Entities”](#) on page 281.

N_ATTRIBUTES

Set this keyword to a named variable to return the number of attributes associated with a Shapefile object. If the value is unknown, this method returns 0.

ATTRIBUTE_NAMES

Set this keyword to a named variable to return the names of each attribute in the Shapefile object. These names are returned as a string array.

ATTRIBUTE_INFO

Set this keyword to a named variable to return the attribute information for each attribute. This consists of an array of attribute information structures that have the following fields:

Field	Description
NAME	A string that contains the name of the attribute.
TYPE	The IDL type code of the attribute.
WIDTH	The width of the attribute.
PRECISION	The precision of the attribute.

Table 4-6: ATTRIBUTE_INFO Fields

The file must be open to obtain this information.

IS_OPEN

Set this keyword to a named variable to return information about the status of a Shapefile. The following values can be returned:

Value	Description
0	File is not open
1	File is open in read-only mode.
3	File is open in update mode.

Table 4-7: IS_OPEN Values

FILENAME

Set this keyword to a named variable to return the fully qualified path name of the Shapefile in the current Shapefile object.

Examples

In the following example, the number of entities and the entity type is returned:

```
PRO entity_info
;Open the states Shapefile in the examples directory
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))

;Get the number of entities and the entity type
myshape -> IDLffShape::GetProperty, N_ENTITIES=num_ent, $
    ENTITY_TYPE=ent_type

;Print the number of entities and the type
PRINT, 'Number of Entities: ', num_ent
PRINT, 'Entity Type: ', ent_type

;Close the Shapefile
OBJ_DESTROY, myshape

END
```

This results in the following:

```
Number of Entities:      51
Entity Type:            5
```

In the next example, the definitions for attribute 1 are returned:

```
PRO attribute_info
;Open the states Shapefile in the examples directory
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $
    SUBDIR=['examples', 'data']))

;Get the info for all attribute
myshape -> IDLffShape::GetProperty, ATTRIBUTE_INFO=attr_info

;Print Attribute Info
PRINT, 'Attribute Number: ', '1'
PRINT, 'Attribute Name: ', attr_info[1].name
PRINT, 'Attribute Type: ', attr_info[1].type
PRINT, 'Attribute Width: ', attr_info[1].width
PRINT, 'Attribute Precision: ', attr_info[1].precision

;Close the Shapefile
OBJ_DESTROY, myshape

END
```

This results in the following:

```
Attribute Number: 1
Attribute Name:  STATE_NAME
Attribute Type:      7
Attribute Width:     25
Attribute Precision:      0
```

IDLffShape::Init

The IDLffShape::Init function method initializes or constructs a Shapefile object.

Note

Init methods are special *lifecycle methods*, and as such cannot be called outside the context of object creation. This means that in most cases, you cannot call the Init method directly. There is one exception to this rule: If you write your own subclass of this class, you can call the Init method from within the Init method of the subclass.

Syntax

```
Result = OBJ_NEW('IDLffShape' [, Filename] [, /UPDATE]  
[, ENTITY_TYPE='Value'])
```

Return Value

This method returns a Shapefile object.

Arguments

Filename

Set this argument to a scalar string containing the full path and filename of a Shapefile (.shp) to open. If this file exists, it is opened. If the file does not exist, a new Shapefile object is constructed. You do not need to use [IDLffShape::Open](#) to open an existing file when specifying this keyword.

Note

The .shp, .shx, and .dbx files must exist in the same directory for you to be able to open and access the file unless the UPDATE keyword is set.

Keywords

UPDATE

Set this keyword to have the file opened for writing. The default is read-only.

ENTITY_TYPE

Set this keyword to the entity type of a new Shapefile. Use this keyword only when creating a new Shapefile. For more information on entity types, see [“Entities”](#) on page 281.

Example

In the following example, we create a new Shapefile object and open the `examples/data/states.shp` file:

```
myshape=OBJ_NEW('IDLffShape', FILEPATH('states.shp', $  
    SUBDIR=['examples', 'data']))
```

IDLffShape::Open

The IDLffShape::Open method opens a specified Shapefile.

Syntax

```
Result = Obj->[IDLffShape::]Open( 'Filename' [, /UPDATE]  
[, ENTITY_TYPE='value'] )
```

Return Value

This method returns 1 if the file can be read successfully. If not able to open the file, it returns 0.

Arguments

Filename

Set this argument to a scalar string containing the full path and filename of a Shapefile (.shp) to open. Note that the .shp, .shx, and .dbx files must exist in the same directory for you to be able to open and access the file unless the UPDATE keyword is set.

Keywords

UPDATE

Set this keyword to have the file opened for writing. The default is read-only.

ENTITY_TYPE

Set this keyword to the entity type of a new Shapefile. Use this keyword only when creating a new Shapefile. For more information on entity types, see [“Entities”](#) on page 281

Example

In the following example, the file `examples/data/states.shp` is opened for reading and writing:

```
status = myShape->Open(FILEPATH('states.shp', $  
    SUBDIR=['examples', 'data']), /UPDATE)
```

IDLffShape::PutEntity

The IDLffShape::PutEntity method inserts an entity into the Shapefile object. The entity must be in the proper structure. For more information on the structure, see “Entities” on page 281.

Note

The shape type of the new entity must be the same as the shape type defined for the Shapefile. If the shape type has not been defined for the Shapefile using the ENTITY_TYPE keyword for the [IDLffShape::Open](#) or [IDLffShape::Init](#) methods, the first entity that is inserted into the Shapefile defines the type.

Note

Only new entities can be inserted into a Shapefile. Existing entities cannot be updated.

Syntax

Obj->[IDLffShape::]PutEntity, *Data*

Arguments

Data

Set this argument to a scalar or an array of entity structures.

Keywords

None.

Example

In the following example, we create a new shapefile, define a new entity, and then use the PutEntity method to insert it into the new file:

```
PRO ex_shapefile_newfile

;Create the new shapefile and define the entity type to Point
mynewshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
    SUBDIR=['examples', 'data']), /UPDATE, ENTITY_TYPE=1)
```

```
;Create structure for new entity
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity
entNew.SHAPE_TYPE = 1
entNew.ISHAPE = 1458
entNew.BOUNDS[0] = -104.87270
entNew.BOUNDS[1] = 39.768040
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -104.87270
entNew.BOUNDS[5] = 39.768040
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

;Add the new entity to new shapefile
mynewshape -> IDLffShape::PutEntity, entNew

;Close the shapefile
OBJ_DESTROY, mynewshape

END
```

IDLffShape::SetAttributes

The IDLffShape::SetAttributes method sets the attributes for a specified entity in a Shapefile object.

Syntax

Obj->[IDLffShape::]SetAttributes, *Index*, *Attribute_Num*, *Value*

or

Obj->[IDLffShape::]SetAttributes, *Index*, *Attributes*

Arguments

Index

A scalar specifying the entity in which you want to set the attributes. The first entity in the Shapefile object is 0.

Attribute_Num

The field number for the attribute whose value is being set. This value is 0-based.

Value

The value that the attribute is being set to. If the value is not of the correct type, type conversion is attempted.

If *Value* is an array and *Index* is a scalar, the value of record is treated as a starting point. Using this feature, all the attribute values of a specific field can be set for a Shapefile.

Attributes

An Attribute structure whose fields match the fields in the attribute table. If *Attributes* is an array, the entities specified in *Index*, up to the size of the Attributes array, are set. Using this feature, all the attribute values of a set of entities can be set for a Shapefile.

The type of this Attribute structure must match the type that is generated internally for Attribute table. To get a copy of this structure, either get the attribute set for an entity or get the definition using the `ATTRIBUTE_STRUCTURE` keyword of the [IDLffShape::GetProperty](#) method.

Keywords

None.

Example

In the following example, we create a new shapefile, define the attributes for the new file, define a new entity, define some attributes, insert the new entity, and then use the SetAttributes method to insert the attributes into the new file:

```
PRO ex_shapefile_newfile

;Create the new shapefile and define the entity type to Point
mynewshape=OBJ_NEW('IDLffShape', FILEPATH('cities.shp', $
    SUBDIR=['examples', 'data']), /UPDATE, ENTITY_TYPE=1)

;Set the attribute definitions for the new Shapefile
mynewshape->IDLffShape::AddAttribute, 'CITY_NAME', 7, 25, $
    PRECISION=0
mynewshape->IDLffShape::AddAttribute, 'STAT_NAME', 7, 25, $
    PRECISION=0

;Create structure for new entity
entNew = {IDL_SHAPE_ENTITY}

; Define the values for the new entity
entNew.SHAPE_TYPE = 1
entNew.ISHAPE = 1458
entNew.BOUNDS[0] = -104.87270
entNew.BOUNDS[1] = 39.768040
entNew.BOUNDS[2] = 0.00000000
entNew.BOUNDS[3] = 0.00000000
entNew.BOUNDS[4] = -104.87270
entNew.BOUNDS[5] = 39.768040
entNew.BOUNDS[6] = 0.00000000
entNew.BOUNDS[7] = 0.00000000

;Create structure for new attributes
attrNew = mynewshape ->IDLffShape::GetAttributes( $
    /ATTRIBUTE_STRUCTURE)

;Define the values for the new attributes
attrNew.ATTRIBUTE_0 = 'Denver'
attrNew.ATTRIBUTE_1 = 'Colorado'

;Add the new entity to new shapefile
mynewshape -> IDLffShape::PutEntity, entNew
```

```
;Add the Colorado attributes to new shapefile  
mynewshape -> IDLffShape::SetAttributes, 0, attrNew  
  
;Close the shapefile  
OBJ_DESTROY, mynewshape  
  
END
```




Index

Symbols

!MAKE_DLL system variable, [196](#)

!P.COLOR system variable, [256](#)

Numerics

3D plots

viewing, [257](#)

64-bit memory support, [32](#)

A

Altivec support, [38](#)

always on top, [49](#)

animation

XVOLUME, [273](#)

ANNOTATE routine, [41](#)

ARRAY_EQUAL function, [158](#)

arrays

comparing to scalars, [158](#)

comparing values, [158](#)

multiplying transposed arrays, [35](#)

testing equality, [35](#)

ASSOC enhancements, [33](#)

associated variables

opening compressed files, [33](#)

attributes

adding to a Shapefile, [291](#)

of a Shapefile, [285](#)

axes

changing type, [262](#)

date/time data, [141](#)

multiple-level date/time, [142](#)

reverse plotting, [15](#)

azimuth
mapping points, 22, 201

B

backprojection
Hough inverse transform, 18, 179
Radon inverse transform, 19, 210
BESELK function, 159
Bessel functions
BESELK, 159
enhancements, 23
big endian byte order, 225
BREAK statement, 161
byte order, 225

C

calendar dates in Julian, 136
CALL_EXTERNAL
intermediate glue code, 36
changing access permissions, 164
changing modes on all platforms, 164
chmod, 164
closing
Shapefiles, 294
COLORMAP_APPLICABLE function, 162
comparing array values, 158
compressed files
opening, 33
CONTINUE statement, 163
continuous wavelet transform, 237
contours
displaying date/time data, 143, 152
number of levels, 13
curve-fitting routines
LINFIT, 24
LMFIT, 24
POLY_FIT, 26
REGRESS, 27

SVDFIT, 29

D

date/time
contour plots use, 143
data generation, 138
displaying in direct graphics, 140
displaying in object graphics, 148
double-precision versus Julian, 137
generate using TIMEGEN, 138
keywords versus system variables, 146
plotting, 12
precision, 137
using system variables, 145
deleting
files or directories, 168
region of interest, 271
denoising
WV_DENOISE function, 239
DIALOG_PICKFILE enhancements, 57
DIALOG_READ_IMAGE routine, 41
DIALOG_WRITE_IMAGE routine, 41
digital smoothing polynomial, 219
directories
changing permissions, 164
creating, 172
deleting, 168
expanding pathnames, 170
making, 172
searching for files, 177
displaying
date/time data on contours, 144, 152
date/time data on plots, 141
date/time with system variables, 145
distance
between points, 201
DLM
building sharable libraries, 192
double-precision support
IBETA, 22

IGAMMA, [22](#)
 introduction, [8](#)
 objects, [10](#)
 routines, [10](#)
 system variables, [9](#)
 utilities, [11](#)
 DXF object
 displaying, [252](#)
 manipulation, [252](#)
 dynamic memory
 usage, [207](#)
 dynamically loadable modules. *See* DLM

E

Encapsulated PostScript
 preview, [13](#)
 endian
 big, [225](#)
 little, [225](#)
 entities
 inserting into a Shapefile, [307](#)
 retrieving from a Shapefile, [298](#)
 entity, in a Shapefile, [281](#)
 EPS file
 preview, [13](#)
 ESRI Shapefiles, [43](#)
 expanding pathnames, [170](#)

F

FACTORIAL function
 input enhancements, [24](#)
 fast Fourier transform. *See* FFT.
 FFT
 performance improvement, [17](#)
 file handling routines, [34](#)
 file status attributes, [34](#)
 FILE_CHMOD procedure, [164](#)
 FILE_DELETE procedure, [168](#)

FILE_EXPAND_PATH function, [170](#)
 FILE_MKDIR procedure, [172](#)
 FILE_TEST function, [173](#)
 FILE_WHICH function, [177](#)
 files
 changing permissions, [164](#)
 deleting, [168](#)
 expanding pathnames, [170](#)
 protection classes, [164](#)
 searching directories, [177](#)
 foreground color, [256](#)

G

Gaussian, *see* wavelet functions
 GIF support, [41](#)
 great circle, [22](#), [201](#)

H

histogram
 view of ROI, [271](#)
 histograms
 number of bins, [24](#)
 Hough
 backprojection, [179](#)
 transform, [179](#)
 HOUGH function, [179](#)

I

IDL Insight, [42](#)
 IDL Projects enhancements, [46](#)
 IDL Wavelet Toolkit, [59](#)
 IDLanROI vertices, [23](#)
 IDLffShape
 AddAttribute method, [291](#)
 class, [280](#)
 Cleanup method, [293](#)
 Close method, [294](#)

- DestroyEntity method, 295
- GetAttributes method, 296
- GetEntity method, 298
- GetProperty method, 300
- Init method, 304
- Open method, 306
- PutEntity method, 307
- SetAttributes method, 309
- importing preferences, 48
- Internet socket support, 223

J

- Julian date definition, 232
- Julian dates/time
 - CALDAT conversion, 138
 - calendar conversion, 136
 - displaying using LABEL_DATE, 140
 - generating, 232
 - IDL use, 136
 - precision using MACHAR, 137

K

- keywords, new and updated, 67

L

- LAGUERRE function, 187
- Laguerre polynomials, 21, 187
- large file support
 - Windows platform, 31
- least squares filtering, 21
- LEGENDRE function, 189
- Legendre polynomials, 21, 189
- library updates, 43
- license management utilities, 54
- licensing wizard, 54
- LINFIT enhancements, 24

- linking
 - C code with IDL, 192
 - dynamically, 192
- little endian byte order, 225
- LIVE_EXPORT routine, 42
- LMFIT enhancements, 24
- logical unit number
 - SOCKET procedure, 224
- LUN
 - TCP/IP socket, 223
- LZW support, 41

M

- Macintosh platform
 - Altivec support, 38
 - changing file permissions, 164
 - Error Window, 49
- macros
 - importing from previous releases, 48
- main window preferences
 - Macintosh platform, 53
- MAKE_DLL procedure, 192
- MAP_2POINTS function, 201
- mapping
 - points on a sphere, 22
- Marr, *see* WV_FN_GAUSSIAN
- matrices
 - MATRIX_MULTIPLY, 205
- MATRIX_MULTIPLY function, 205
- MEMORY function, 207
- minimum curvature surface interpolation, 23
- Morlet, *see* wavelet functions
- MPEG movie enhancements, 44
- multiplication of matrices, 205

O

- object class enhancements, 66
- object method enhancements, 66

- obsolete routines, [128](#)
- obsoleted features, [128](#)
- ONLINE_HELP enhancements, [58](#)
- opening
 - Shapefiles, [306](#)

P

- Paul, *see* wavelet functions
- platforms supported, [134](#)
- plots
 - viewing in 3D, [257](#)
- plotting
 - color, [256](#)
 - contour levels, [13](#)
 - displaying date/time data, [140](#)
 - Julian dates/time, [136](#)
 - multiple-level date/time axes, [142](#)
 - reverse axis, [15](#)
- PNG file order, [44](#)
- point values in patterns, [16](#)
- POLY_FIT enhancements, [26](#)
- polynomials
 - digital smoothing, [21](#), [219](#)
 - Laguerre, [21](#), [187](#)
 - least-squares fit, [219](#)
 - Legendre, [21](#), [189](#)
- PostScript
 - preview, [13](#)
- previewing PostScript files, [13](#)
- printer
 - support on UNIX, [14](#)
- printf format support, [39](#)
- probability functions
 - enhancements, [23](#)
- program control statements, [36](#)
- projections
 - 3D plots on walls, [262](#)

Q

- QUERY_GIF routine, [41](#)
- QUERY_IMAGE routine, [41](#)
- QUEUE startup switch, [55](#)
- quoted string format code, [39](#)

R

- Radon backprojection, [210](#)
- RADON function, [210](#)
- Radon transform, [210](#)
- READ_GIF procedure, [41](#)
- READ_IMAGE routine, [41](#)
- READ_PNG routine, [41](#)
- READ_TIFF routine, [41](#)
- reading
 - GIF files, [41](#)
- region of interest
 - XROI, [264](#)
- REGRESS enhancements, [27](#)
- RESOLVE_ROUTINE enhancements, [36](#)
- retrieving
 - attributes of a Shapefile, [296](#)
- reverse axis plotting, [15](#)
- rhumb line, [22](#), [201](#)
- ROI
 - deleting, [271](#)
 - geometric and statistical data, [264](#)
 - histogram view, [271](#)
- routine enhancements, [88](#)
- routines obsoleted, [128](#)

S

- SAVGOL function, [219](#)
- Savitzky-Golay smoothing filter, [21](#), [219](#)
- Shapefile
 - adding attributes, [291](#)
 - attribute structure, [285](#)
 - attributes, [285](#)

- closing, [294](#)
- entity, [281](#)
- entity structure, [282](#)
- included files, [281](#)
- inserting entities, [307](#)
- naming conventions, [281](#)
- object properties, [300](#)
- opening, [306](#)
- retrieving attributes, [296](#)
- retrieving entities, [298](#)
- setting attributes, [309](#)
- Shapefile object, [43](#)
- sharable library
 - building, [192](#)
- SOCKET procedure, [223](#)
- SPAWN enhancements, [37](#)
- SPHER_HARM function, [227](#)
- spherical harmonic
 - relation to Legendre polynomial, [227](#)
 - See also* Legendre polynomials
- startup switch option, [55](#)
- structures
 - concatenation and assignment, [38](#)
- supported platforms, [134](#)
- SVDFIT enhancements, [29](#)
- SWITCH statement, [230](#)
- system time conversion, [34](#)
- system variable enhancements, [126](#)
- system variables
 - !P.COLOR, [256](#)

T

- TCP/IP client side socket support, [37](#), [223](#)
- TIFF support, [41](#)
- TIMEGEN function, [232](#)
- transforms
 - FFT improvements, [17](#)
 - Hough, [18](#), [179](#)
 - Hough inverse, [18](#)
 - Radon, [18](#), [210](#)

- Radon backprojection, [19](#)
- TRIGRID
 - irregularly spaced rectangular output grids, [23](#)

U

- UNIX platform
 - changing file permissions, [164](#)
 - licensing Wizard, [54](#)
- utility enhancements, [62](#)

W

- wavelet functions
 - built-in
 - Gaussian, [243](#)
 - Morlet, [246](#)
 - Paul, [249](#)
- Wavelet Toolkit enhancements, [59](#)
- wavelet transform
 - continuous, [237](#)
- Windows Metafile Format, [14](#)
- Windows platform
 - always on top, [49](#)
 - changing file permissions, [164](#)
 - large file support, [31](#)
 - licensing wizard, [54](#)
- WMF. *See* Windows Metafile Format.
- WRITE_GIF procedure, [41](#)
- WRITE_IMAGE routine, [41](#)
- WRITE_TIFF routine, [41](#)
- writing
 - GIF files, [41](#)
- WV_CWT function, [237](#)
- WV_DENOISE function, [239](#)
- WV_FN_GAUSSIAN function, [243](#)
- WV_FN_MORLET function, [246](#)
- WV_FN_PAUL function, [249](#)

X

XDXF procedure, [252](#)

XPCOLOR procedure, [256](#)

XPLOT3D procedure, [257](#)

XROI

importing images, [270](#)

XROI procedure, [264](#)

XVOLUME procedure, [273](#)

