

What's New in IDL 5.5

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

WASTE text engine © 1993-1996 Marco Piovaneli

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

Use of this software for providing LZW capability for any purpose is not authorized unless user first enters into a license agreement with Unisys under U.S. Patent No. 4,558,302 and foreign counterparts. For information concerning licensing, please contact: Unisys Corporation, Welch Licensing Department - C1SW19, Township Line & Union Meeting Roads, P.O. Box 500, Blue Bell, PA 19424.

Portions of this computer program are copyright © 1995-1999 LizardTech, Inc. All rights reserved. MrSID is protected by U.S. Patent No. 5,710,835. Foreign Patents Pending.

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



Contents

Chapter 1:	
Overview of New Features in IDL 5.5	11
Visualization Enhancements	12
High-Resolution Textures Supported by IDLgrSurface	12
New Enhancements to XOBJVIEW	13
New XOBJVIEW_ROTATE Procedure	13
New XOBJVIEW_WRITE_IMAGE Procedure	13
New Procedure for Generating Tetrahedral Data	13
New Support for Region Growing	14
New XROI Functionality	14
New TrueColor Support for Any Depth on UNIX	14
New Support for Resolving Stitching Artifacts in Object Graphics	16
New QUIET Keyword for RECON3	19
New Keyword for Smoother Results Using WARP_TRI	19

Analysis Enhancements	20
The IDL Thread Pool and Multi-Threading	20
New Functionality for Gridding and Interpolation	21
New Examples Using the AUTO_GLUE Keyword to CALL_EXTERNAL	21
New REAL_PART Function	22
New ERF, ERFC, and ERFCX Functions	22
Support for SIMPLEX Method for Linear Programming	22
BESELJ, BESELK and BESELY Functionality Improvements	22
New NaN Support for SMOOTH and CONVOL	22
New LNORM Keyword for COND and NORM	23
New DOUBLE Keyword for POLY_AREA	23
New STATUS Keyword for POLYWARP Support	23
New ACOS, ASIN, ATAN Support for Complex Input	23
New Minimum/Maximum Operator Support for Complex Data	24
New SMOOTH Function Multidimensional Width Support	24
New Dimension-specific Transforming for FFT	25
New Dimension-setting functionality for Arrays	25
Source Code for CLUSTER, CLUST_WTS, EIGENQL, PCOMP	25
New Histogram Cumulative Probability Distribution Functionality	26
Language Enhancements	27
Maximum String Length Limit Increased for 32-Bit IDL	27
New MESSAGE Keywords and Message Block Support	27
Relaxed Formatted Input/Output Record Length Limits	31
New and Enhanced File Handling Routines	31
New Functionality Frees Dynamic Resources	33
New Ability to Check for Keyword Inheritance Errors	33
Enhancements to IDL Path Expansion	34
New Support for REFORM-Style Dimension Array	35
New DOUBLE Keyword for COMPLEX	36
New CENTER Keyword for CONGRID	37
New SIGN Keyword for FINITE	37
Improvements to Files Created with SAVE	37
Improvements to UNIX Filename Expansion	38
Pre-IDL 4.0 C Internals Compatibility Library Removed	38
User Interface Toolkit Enhancements	40
New COM and ActiveX Functionality for IDL	40

New Shortcut Menu Widget	40
Emulating System Colors in Application Widgets	41
New Functionality to Specify Slider Increments in IDL Widgets	43
File Access Enhancements	44
New PATH_SEP Function	44
Enhanced TIFF Support	44
New Support for MrSID	47
Development Environment Enhancements	48
Improved Project Exporting	48
Scientific Data Formats Enhancements	49
HDF-EOS Data Output Enhancements	49
New HDF Vdata Attribute Routines	50
IDL ActiveX Control Enhancements	51
IDL DataMiner Enhancements	52
Platform Specific Information	52
Documentation Enhancements	56
Enhanced IDL Utilities	57
Enhanced IDL Utilities	57
New Keywords/Arguments to Existing IDL Utilities	58
New and Enhanced IDL Objects	60
New Object Classes	60
IDL Object Method Enhancements	60
New and Enhanced IDL Routines	72
New IDL Routines	72
IDL Routine Enhancements	80
Updates to Executive Commands	120
New and Updated System Variables	121
Features Obsoleted	122
Obsoleted Routines	122
Obsoleted Keywords and Arguments	122
Platforms Supported in this Release	124
Chapter 2:	
Multi-Threading in IDL	125
The IDL Thread Pool	126
Benefits of the IDL Thread Pool	126

Possible Drawbacks to the Use of the IDL Thread Pool	126
Controlling the Thread Pool in IDL	128
Using the Initial Settings of the Thread Pool	128
Programmatically Controlling the Settings of the Thread Pool	128
Disabling the Thread Pool	133
Routines Supporting the Thread Pool	134

Chapter 3:

Using COM Objects in IDL 137

Introduction to IDL COM Objects	138
Skills Required to Use COM Objects	139
IDL COM Naming Schemes	140
About Obtaining COM Class Identifiers	140
Using IDL IDispatch COM Objects	142
IDL IDispatch Naming Schemes	142
IDispatch Object Creation	143
IDispatch Method Dispatching	143
IDispatch COM Object Destruction	144
IDispatch Property Management	144
COM Objects Returning IDispatch Pointers to Other Objects	145
Example: Creating an IDispatch COM Object in IDL	145
Using ActiveX Controls in IDL	149
ActiveX-based COM Naming Schemes	149
ActiveX Control Creation	150
ActiveX Control Access and Dispatching	150
Freeing Dynamic Resources	151
ActiveX Control Destruction	151
Example: Embedding an ActiveX Control in IDL	152
Access to ActiveX Methods and Properties	155
Event Propagation	156

Chapter 4:

Using the Shortcut Menu Widget 157

Introduction to the Shortcut Menu Widget	158
Using WIDGET_DISPLAYCONTEXTMENU	159
Creating a Base Widget Shortcut Menu	160
Creating a Draw Widget Shortcut Menu	162

Creating a List Widget Shortcut Menu	166
Creating a Text Widget Shortcut Menu	170

Chapter 5: New Objects 175

IDLcomIDispatch	176
IDLcomIDispatch::Init	177
IDLcomIDispatch::GetProperty	179
IDLcomIDispatch::SetProperty	180
IDLffMrSID	181
IDLffMrSID::Cleanup	182
IDLffMrSID::GetDimsAtLevel	183
IDLffMrSID::GetImageData	185
IDLffMrSID::GetProperty	188
IDLffMrSID::Init	191

Chapter 6: New IDL Routines 193

CPU	194
DEFINE_MSGBLK	197
DEFINE_MSGBLK_FROM_FILE	200
ERF	203
ERFC	204
ERFCX	205
FILE_INFO	206
FILE_SEARCH	210
GRID_INPUT	224
GRIDDATA	228
HDF_VD_ATTRFIND	253
HDF_VD_ATTRINFO	254
HDF_VD_ATTRSET	256
HDF_VD_ISATTR	262
HDF_VD_NATTRS	263
HEAP_FREE	264
INTERVAL_VOLUME	267
PATH_SEP	270
QGRID3	271

QHULL	276
QUERY_MRSID	279
READ_MRSID	281
REAL_PART	283
REGION_GROW	284
SIMPLEX	287
WIDGET_ACTIVEX	291
WIDGET_DISPLAYCONTEXTMENU	298
XOBJVIEW_ROTATE	300
XOBJVIEW_WRITE_IMAGE	302
XROI	303

Chapter 7:

New Examples 319

Overview of New Examples	320
Mapping an Image Onto a Surface	322
Centering an Image Object	325
Alpha Blending: Creating a Transparent Image Object	328
Working with Mesh Objects and Routines	332
Clipping a Mesh	333
Decimating a Mesh	336
Merging Meshes	339
Smoothing a Mesh	342
Advanced Meshing: Combining Meshing Routines	345
Copying and Printing Objects	351
Copying a Plot Display to the Clipboard	351
Printing a Plot Display	353
Copying an Image Display to the Clipboard	355
Printing an Image Display	357
Capturing IDL Direct Graphics Displays	359
Capturing Direct Graphics Displays on PseudoColor Devices	359
Capturing Direct Graphics Displays on TrueColor Devices	360
Creating and Restoring .sav Files	363
Customizing and Saving an ASCII Template	363
Saving and Restoring the XROI Utility and Image ROI Data	365
Handling Table Widgets in GUIs	368

Finding Straight Lines in Images	374
Color Density Contrasting in an Image	376
Removing Noise from an Image with FFT	379
Using Double and Triple Integration	381
Integrating to Determine the Volume Under a Surface (Double Integration)	381
Integrating to Determine the Mass of a Volume (Triple Integration)	382
Obtaining Irregular Grid Intervals	385
Calculating Incomplete Beta and Gamma Functions	387
Working With Tolerances in the Incomplete Beta Function	387
Working With Iteration Controls in the Incomplete Gamma Function	388
Determining Bessel Function Accuracy	390
Analyzing the Bessel Function of the First Kind	390
Analyzing the Bessel Function of the Second Kind	392
Analyzing the Modified Bessel Function of the First Kind	394
Analyzing the Modified Bessel Function of the Second Kind	396
Index	399



Chapter 1: Overview of New Features in IDL 5.5

This chapter contains the following topics:

Visualization Enhancements	12	Documentation Enhancements	56
Analysis Enhancements	20	Enhanced IDL Utilities	57
Language Enhancements	27	New and Enhanced IDL Objects	60
User Interface Toolkit Enhancements	40	New and Enhanced IDL Routines	72
Development Environment Enhancements ..	48	New and Updated System Variables	121
File Access Enhancements	44	Features Obsoleted	122
Scientific Data Formats Enhancements	49	Platforms Supported in this Release	124
IDL DataMiner Enhancements	52		

Visualization Enhancements

The following enhancements have been made in the area of Visualization in the IDL 5.5 release:

- [High-Resolution Textures Supported by IDLgrSurface](#)
- [New Enhancements to XOBJVIEW](#)
- [New XOBJVIEW_ROTATE Procedure](#)
- [New XOBJVIEW_WRITE_IMAGE Procedure](#)
- [New Procedure for Generating Tetrahedral Data](#)
- [New Support for Region Growing](#)
- [New XROI Functionality](#)
- [New TrueColor Support for Any Depth on UNIX](#)
- [New Support for Resolving Stitching Artifacts in Object Graphics](#)
- [New QUIET Keyword for RECON3](#)
- [New Keyword for Smoother Results Using WARP_TRI](#)

High-Resolution Textures Supported by IDLgrSurface

Different 3D hardware platforms support different maximum texture resolutions. For example, OpenGL only guarantees that the maximum resolution will be at least 64-by-64 pixels. This presents a problem if a high pixel resolution image needs to be mapped onto a 3D surface. Previously, IDL solved this problem by scaling the image down to the maximum texture size supported by the hardware. This resulted in a loss of data that was particularly noticeable when zooming in on the surface. In some cases, magnification of the low-resolution texture resulted in an unrecognizable image.

IDL 5.5 addresses this problem with the new `TEXTURE_HIGHRES` keyword to `IDLgrSurface`. Using this new keyword tiles multiple textures across the surface and may also divide the surface geometry to fit the texture tiles. Although IDL tiles the texture and surface, the original data is unaltered. Use of the `TEXTURE_HIGHRES` keyword thus preserves fine detail by allowing a high-resolution image to be mapped onto a surface.

Note

Because of the way in which high-resolution textures require modified texture coordinates, if the TEXTURE_COORD keyword is used, TEXTURE_HIGHRES will be disabled.

New Enhancements to XOBJVIEW

A new JUST_REG keyword has been added to the XOBJVIEW utility in IDL 5.5. You can set this keyword to indicate that the XOBJVIEW utility should just be registered and return immediately. This keyword is useful if you want to register XOBJVIEW before beginning event processing and either:

- your command-processing front-end does not support an active command line, or
- one or more of the registered widgets requests that XMANAGER block event processing. (Note that in this case a later call to XMANAGER without the JUST_REG keyword is necessary to begin blocking.)

Also in IDL 5.5 a new RENDERER keyword has been added to the XOBJVIEW utility. You can set this keyword to an integer value indicating which graphics renderer to use when drawing objects in the XOBJVIEW draw window. Valid values can be given for either platform-native OpenGL or for IDL's software implementation.

New XOBJVIEW_ROTATE Procedure

The new XOBJVIEW_ROTATE procedure is used to programmatically rotate the object currently displayed in XOBJVIEW. For more information about the new XOBJVIEW_ROTATE procedure, see [“XOBJVIEW_ROTATE”](#) in Chapter 6 of this book.

New XOBJVIEW_WRITE_IMAGE Procedure

The new XOBJVIEW_WRITE_IMAGE procedure is used to write the object currently displayed in XOBJVIEW to an image file with the specified name and file format. For more information about the new XOBJVIEW_WRITE_IMAGE procedure, see [“XOBJVIEW_WRITE_IMAGE”](#) in Chapter 6 of this book.

New Procedure for Generating Tetrahedral Data

The new INTERVAL_VOLUME procedure can be used to generate a tetrahedral mesh from volumetric data. The mesh generated by this procedure spans the portion

of the volume where the volume data samples fall between two constant data values. This can also be thought of as a mesh constructed to fill the volume between two isosurfaces where the isosurfaces are drawn at the two supplied constant data values. For more information about the new `INTERVAL_VOLUME` procedure, see “[INTERVAL_VOLUME](#)” in Chapter 6 of this book.

New Support for Region Growing

IDL 5.5 now supports region growing, an image processing technique that extends the boundaries of a specified region to include neighboring pixels that share a common trait. The new `REGION_GROW` function takes a given region within an N-dimensional array and expands the region to include all connected, neighboring pixels that fall within the specified limits. For more information about the `REGION_GROW` function, see “[REGION_GROW](#)” in Chapter 6 of this book. The `XROI` utility also offers an interactive implementation of `REGION_GROW`. See “[Growing an ROI](#)” on page 312 for more information.

New XROI Functionality

The `XROI` utility has been improved in 5.5, offering several new interactive ROI definition tools including Rectangle and Ellipse drawing tools. Additionally, any ROI selected in the drawing window can be translated or scaled using the Translate/Scale tool. `XROI` also includes the functionality of the new IDL routine, `REGION_GROW`. An ROI defined in `XROI` can be grown to include all neighboring pixels which match specified threshold conditions. The Region Grow Properties dialog allows you to precisely control the properties associated with a region growing process. Support for RGB images has been added to the histogram plot feature and is also a part of the Region Grow properties dialog, allowing you to select the channel used when growing a region of an RGB image. For more information about the `XROI` utility, see “[XROI](#)” in Chapter 6 of this book.

New TrueColor Support for Any Depth on UNIX

In previous releases of IDL, the X Windows device only supported TrueColor with a visual depth of 24. In IDL 5.5, TrueColor visuals of any depth are now supported.

How IDL Selects a Visual Class

With the new support for TrueColor visuals of any depth, the following is now the order in which IDL will query the display to find the first available visual class:

1. DirectColor, 24-bit
2. TrueColor, 24-bit

3. TrueColor, 16-bit (on Linux platforms only)
4. PseudoColor, 8-bit, then 4-bit
5. StaticColor, 8-bit, then 4-bit
6. GrayScale, any depth
7. StaticGray, any depth

Setting a Visual Class with the DEVICE Routine

You can manually set the visual class (instead of having IDL determine the visual class) by using the `DEVICE` routine to specify the desired visual class and depth before you create a window. For the `TRUE_COLOR` keyword, you can now specify any value (the most common being 15, 16, and 24). For example:

```
DEVICE, TRUE_COLOR = 16
```

Setting a Default Visual Class in Your .Xdefaults File

You can set the initial default value of the visual class and color depth by setting resources in the `.Xdefaults` file in your home directory. For example, to set the default visual class to TrueColor and the visual depth to 24, insert the following lines in your `.Xdefaults` file:

```
idl.gr_visual: TrueColor  
idl.gr_depth: 24
```

How Color is Interpreted for a TrueColor Visual

How a color (such as `!P.COLOR`) is interpreted by IDL (when a TrueColor visual is being utilized) depends in part upon the decomposed setting for the device.

To retrieve the decomposed setting:

```
DEVICE, GET_DECOMPOSED = currentDecomposed
```

To set the decomposed setting:

```
DEVICE, DECOMPOSED = newDecomposed
```

If the decomposed value is zero, colors (like `!P.COLOR`) are interpreted as indices into IDL's color table. A color should be in the range from 0 to `!D.TABLE_SIZE - 1`. The IDL color table contains a red, green, and blue component at a given index; each of these components is in the range of 0 up to 255.

Note

IDL's color table does not map directly to a hardware color table for a TrueColor visual. If IDL's color table is modified, for example using the `LOADCT` or `TVLCT`

routines, then the new color table will only take effect for graphics that are drawn after it has been modified.

If the decomposed value is non-zero, colors (like !P.COLOR) are interpreted as a combination of red, green, and blue settings. The least significant 8 bits contain the red component, the next 8 bits contain the green component, and the most significant 8 bits contain the blue component.

In either case, the most significant bits of each of the resulting red, green, and blue components are utilized. The number of bits utilized per component depends upon the red, green, and blue masks for the visual. On UNIX systems, a new field (Bits Per RGB) has been added to the output from HELP, /DEVICE. This Bits Per RGB field indicates the amount of bits utilized for each component.

Tip

The UNIX command, `xdpyinfo`, also provides information about each of the visuals.

New Support for Resolving Stitching Artifacts in Object Graphics

In previous releases of IDL, it was very difficult to reduce or remove a common visual artifact called *stitching*. Stitching may occur when multiple graphic primitives are rendered at the same depth, or distance from the eye in view space. If the primitives overlap each other at the same depth, parts of some of the primitives may *poke through* other primitives, creating a stitching effect. These artifacts are caused by unavoidable rounding in rasterization calculations, Z-buffer limitations and by different algorithms used to rasterize different primitives.

One of the most common examples of this effect is caused by trying to draw lines "on top" of a surface, using the same vertex data. Even though the lines may be drawn last, the surface still pokes through the lines, leaving a stitched appearance. An attempt to correct the situation by moving the lines up or away from the surface in world coordinates usually fails because rotating the objects with a trackball or other mechanism fails to keep the lines above the surface.

In IDL 5.5, this problem has been addressed by allowing the specification of a DEPTH_OFFSET value that is used to displace polygons away from the eye in view space as the polygons are rendered. This displacement is applied in the view, after the model transforms have been applied. If two objects overlap at the same depth, one of them can be rendered with a non-zero DEPTH_OFFSET to force a separation between them in view space. For example, if one object is a set of lines, and the other

is a surface, the surface can be rendered with a `DEPTH_OFFSET` greater than zero to "push" it back away from the eye and allow the lines to appear without interference from the surface. Even if the objects are rotated with a model transform, the surface will always be drawn slightly farther away from the eye. `DEPTH_OFFSET` has no effect on the drawing order of objects, and vice-versa.

Note

RSI suggests using this feature to remove stitching artifacts and not as a means for layering complex scenes with multiple `DEPTH_OFFSET` values. It is safest to use only a `DEPTH_OFFSET` value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of `DEPTH_OFFSET` values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of `DEPTH_OFFSET` values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use `DEPTH_OFFSET`.

The new `DEPTH_OFFSET` keyword has been added to the following methods:.

Object Class	Method
IDLgrContour	GetProperty
	Init
	SetProperty
IDLgrPolygon	GetProperty
	Init
	SetProperty
IDLgrSurface	GetProperty
	Init
	SetProperty

Table 1-1: Methods That Support the New `DEPTH_OFFSET` Keyword

As an example, the following program displays a surface. When you run the program, you can see the "stitching" in the surface.

```

PRO stitch_ex

; Create data.
x = 5.*SIN(10*FINDGEN(37)*!DTOR)
y = 5.*COS(10*FINDGEN(37)*!DTOR)
data = x ## y

; Initialize model to contain surface and
; mesh.
oModel = OBJ_NEW('IDLgrModel')

; Initialize surface object.
oSurface = OBJ_NEW('IDLgrSurface', data, $
    STYLE = 2, COLOR = [200, 200, 200])

; Initialize mesh object.
oMesh = OBJ_NEW('IDLgrSurface', data, $
    COLOR = [0, 0, 0])

; Add surface and mesh to model.
oModel -> Add, oSurface
oModel -> Add, oMesh

; Rotate model for better initial perspective.
oModel -> Rotate, [-1, 0, 1], 45

; Display model in XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Example of Line Stitching'

END

```

Now, modify the program to specify the `DEPTH_OFFSET` keyword. Change the lines that initialized the surface object:

```

; Initialize surface object.
oSurface = OBJ_NEW('IDLgrSurface', data, $
    STYLE = 2, COLOR = [200, 200, 200], DEPTH_OFFSET = 1)

```

When you run the example again, you will not see the “stitching”.

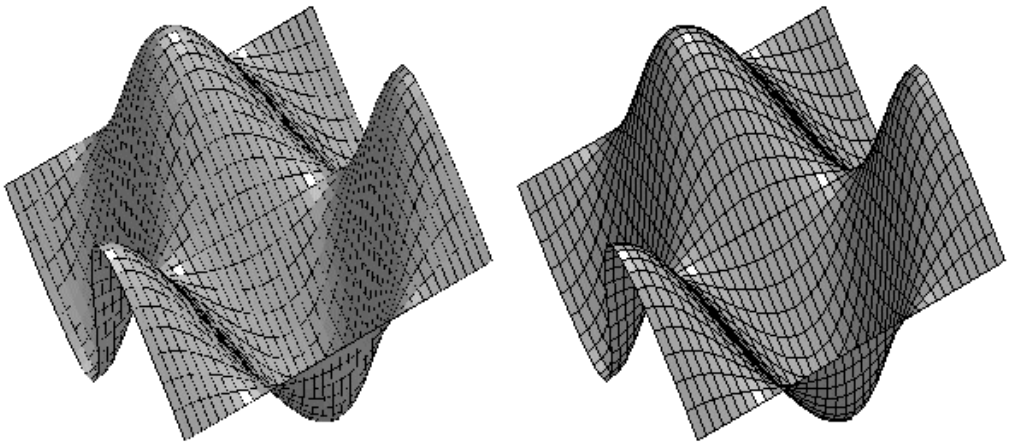


Figure 1-1: Surface Without the `DEPTH_OFFSET` Keyword (Left) and Using the `DEPTH_OFFSET` Keyword (Right)

New QUIET Keyword for RECON3

A new QUIET keyword has been added to the RECON3 function in IDL 5.5. By default (QUIET = 0), the RECON3 function outputs an informational message when the processing of each image has been completed. This keyword, when set, allows you to suppress the output of this message.

New Keyword for Smoother Results Using WARP_TRI

The new TPS keyword to WARP_TRI uses Thin Plate Spline interpolation. The Thin Plate Spline interpolation is ideal for modeling functions with complex local distortions, such as warping functions, which are too complex to be fit with polynomials.

Analysis Enhancements

The following enhancements have been made in the area of Analysis in the IDL 5.5 release:

- [The IDL Thread Pool and Multi-Threading](#)
- [New Functionality for Gridding and Interpolation](#)
- [New Examples Using the AUTO_GLUE Keyword to CALL_EXTERNAL](#)
- [New REAL_PART Function](#)
- [New ERF, ERFC, and ERFCX Functions](#)
- [Support for SIMPLEX Method for Linear Programming](#)
- [BESELI, BESELJ, BESELK and BESELY Functionality Improvements](#)
- [New NaN Support for SMOOTH and CONVOL](#)
- [New LNORM Keyword for COND and NORM](#)
- [New DOUBLE Keyword for POLY_AREA](#)
- [New STATUS Keyword for POLYWARP Support](#)
- [New ACOS, ASIN, ATAN Support for Complex Input](#)
- [New Minimum/Maximum Operator Support for Complex Data](#)
- [New SMOOTH Function Multidimensional Width Support](#)
- [New Dimension-specific Transforming for FFT](#)
- [New Dimension-setting functionality for Arrays](#)
- [Source Code for CLUSTER, CLUST_WTS, EIGENQL, PCOMP](#)
- [New Histogram Cumulative Probability Distribution Functionality](#)

The IDL Thread Pool and Multi-Threading

With this release, IDL for Windows and IDL for UNIX have the ability to use multiple threads of execution in a user transparent manner when performing some numeric computations on multi-CPU hardware. This can greatly increase the speed at which calculations are accomplished on large data sets; however, it can also hinder analysis time in certain cases. Developers are able to control the default use of multi-threading by using the !CPU system variable, the new CPU procedure, and the new multi-threading keywords in each routine supporting multi-threading.

What is Multi-Threading?

On systems equipped with multiple processors, IDL automatically evaluates the advantages and disadvantages of using the processors in parallel to accomplish the calculation. Unless otherwise overridden by using the new CPU procedure to change the new !CPU system variable, IDL may decide to perform calculations using a thread pool for routines which support this capability. See [Chapter 2, “Multi-Threading in IDL”](#) for a complete description of multi-threading, and a listing of all routines currently supporting this capability.

Platform Support for Multi-Threading

IDL supports the use of the thread pool on all platforms except AIX and Macintosh.

New Functionality for Gridding and Interpolation

Four new routines have been added to the gridding and interpolation functionality in this release: GRID_INPUT, GRIDDATA, QGRID3 and QHULL.

- [GRID_INPUT](#) preprocesses and sorts two-dimensional scattered data sets, and removes duplicate points.
- [GRIDDATA](#) interpolates data to a regular grid from scattered data values and locations.
- [QGRID3](#) linearly interpolates dependent variable values to points in a regularly sampled volume.
- [QHULL](#) is used to construct convex hulls, Delaunay triangulations, and Voronoi diagrams for a set of points two-dimensional or higher.

New Examples Using the AUTO_GLUE Keyword to CALL_EXTERNAL

The IDL distribution now includes two new examples of how to use the AUTO_GLUE keyword to the CALL_EXTERNAL function. The AUTO_GLUE keyword, introduced in IDL 5.4, allows you to easily access routines within other programming libraries.

Two new examples show how to use AUTO_GLUE to access routines within the IMSL C Numerical Library. The examples are located in the `examples/ims1` directory. This directory also includes a `readme.txt` text file, which explains how to use these examples.

These examples are implemented as IDL functions. The first example computes the Airy function using the Visual Numerics IMSL C Numerical Library. This example is

called `IMSL_AIRY` and is in the `imsl_airy.pro` file. The second example computes the singular value decomposition of an input array using the Visual Numerics IMSL C Numerical Library. This example is called `IMSL_SVDC` and is in the `imsl_svdc.pro` file.

New `REAL_PART` Function

The new `REAL_PART` function returns the real part of its complex-valued argument. For more information about the new `REAL_PART` function, see [“`REAL_PART`”](#) in Chapter 6 of this book.

New `ERF`, `ERFC`, and `ERFCX` Functions

The new `ERF`, `ERFC`, and `ERFCX` functions return the value of the error function, the complimentary error function, and the scaled complimentary error function, respectively. For more information about these new functions, see [“`ERF`”](#), [“`ERFC`”](#), and [“`ERFCX`”](#) in Chapter 6 of this book.

Support for `SIMPLEX` Method for Linear Programming

The new `SIMPLEX` function uses the simplex method to solve linear programming problems and is modeled on the `simplex` routine found in Numerical Recipes. For more information about the new `SIMPLEX` function, see [“`SIMPLEX`”](#) in Chapter 6 of this book.

`BESEL`, `BESELJ`, `BESELK` and `BESELY` Functionality Improvements

The `BESEL` functions now accept any order greater than or equal to zero (within memory limitations), and also return arrays of the correct dimensions.

New NaN Support for `SMOOTH` and `CONVOL`

IDL’s `CONVOL` and `SMOOTH` functions now support the handling of NaNs.

When using `CONVOL` and `SMOOTH`, the new `NAN` keyword may be set to cause the routine to check for occurrences of the IEEE floating-point value NaN in the input data. Elements with this value are treated as missing data, and are ignored when computing the convolution for neighboring elements. In the Result, missing elements are replaced by the convolution of all other valid points within the kernel. If all points within the kernel are missing, then the result at that point is given by the `MISSING` keyword.

New LNORM Keyword for COND and NORM

A new LNORM keyword has been added to the COND and NORM functions in IDL 5.5. This keyword allows you choose which norm is used in the computation of the COND and NORM functions. For NORM with a vector input argument, you can choose L_∞ norm, L_1 norm, L_2 norm, ..., L_n norm where n is any number. The default for vectors is L_2 norm. For COND and NORM with a two-dimensional array input, you can choose L_∞ norm (the maximum absolute row sum norm), L_1 norm (the maximum absolute column sum norm), or L_2 norm (the spectral norm). The default for two-dimensional arrays is L_∞ norm.

New DOUBLE Keyword for POLY_AREA

In IDL 5.5, a new DOUBLE keyword has been added to the POLY_AREA function. You can set this keyword to force the computation of the POLY_AREA function to be performed using double-precision arithmetic.

New STATUS Keyword for POLYWARP Support

When calculating polynomial coefficients for 45-degree rotations with POLYWARP certain inputs may cause singular matrices. This rarely happens with real data, but does happen with more idealized data (such as squares or regular shapes). In this type of case, it is very easy to get a failure in the INVERT. In IDL 5.5, a STATUS keyword has been introduced for feedback on such rare occurrences.

New ACOS, ASIN, ATAN Support for Complex Input

In IDL 5.5, new support has been added allowing complex input to ACOS, ASIN, and ATAN. Previously, the inverse transcendental functions ACOS and ASIN did not accept complex input. The ATAN function accepted complex input, $Z=X+iY$, but incorrectly converted the complex number into the 2-argument $ATAN(y, x)$ form and returned a real result. For ATAN, support has been added for input of two complex arguments.

ATAN Function Support

The ATAN function now computes the complex arctangent for complex input. Previously, for a complex number $Z=X+iY$, internally $ATAN(Z)$ would split Z into its real and imaginary components and compute $ATAN(Y, X)$. IDL code that uses this undocumented behavior should be changed by replacing calls to $ATAN(Z)$ with $ATAN(IMAGINARY(Z), REAL_PART(Z))$.

For example, in IDL 5.4, to compute the argument (or angle) of a complex number:

```
z = COMPLEX(2, 1)
print, ATAN(z)*180/!PI      ; undocumented behavior
```

IDL prints:

```
26.5651
```

Now, in IDL 5.5, to compute the argument:

```
z = COMPLEX(2, 1)
print, ATAN( IMAGINARY(z), REAL_PART(z) ) * 180 / !PI
```

IDL prints:

```
26.5651
```

New Minimum/Maximum Operator Support for Complex Data

Complex data types now work with `<`, `>`, `LT`, `LE`, `GT`, and `GE` operators, utilizing the absolute value (or modulus) for all comparisons. Behavior is unchanged for `EQ` and `NE`.

New SMOOTH Function Multidimensional Width Support

Since `SMOOTH` allows n-dimensional input arrays, IDL 5.5 now allows an n-dimensional smoothing window (the *Width* input argument can now have more than one dimension).

Example

This example shows the use of `SMOOTH` with the new multidimensional width argument on an RGB image.

```
; Determine the path to the file.
file = FILEPATH('rose.jpg', $
    SUBDIRECTORY = ['examples', 'data'])

; Import in the RGB image from the file.
image = READ_IMAGE(file)

; Initialize the image size parameter.
imageSize = SIZE(image, /DIMENSIONS)

; Initialize the display.
DEVICE, DECOMPOSED = 1
WINDOW, 0, XSIZE = imageSize[1], YSIZE = imageSize[2], $
    TITLE = 'Original Rose Image'

; Display the original image on the left side.
TV, image, TRUE = 1
```

```

; Initialize another display
WINDOW, 1, XSIZE = 3*imageSize[1], YSIZE = imageSize[2], $
    TITLE = 'Vertically Smoothed (left), Horizontally ' + $
    'Smoothed (middle), and Both (right)'

; Smooth the RGB image in just the width dimension.
smoothed = SMOOTH(image, [1, 1, 21])

; Display the results.
TV, smoothed, 0, TRUE = 1

; Smooth the RGB image in just the height dimension.
smoothed = SMOOTH(image, [1, 21, 1])

; Display the results.
TV, smoothed, 1, TRUE = 1

; Smooth the RGB image in just the width and height dimensions.
smoothed = SMOOTH(image, [1, 5, 5])

; Display the results.
TV, smoothed, 2, TRUE = 1

```

New Dimension-specific Transforming for FFT

Previously, the FFT function accepted multi-dimensional arguments but did not allow specification of which dimension to transform, but instead transformed along all dimensions. Now in IDL 5.5, the new **DIMENSION** keyword allows you to transform only along one dimension.

New Dimension-setting functionality for Arrays

The new **DIMENSION** keyword to the **MIN** and **MAX** functions allows you to set the dimension over which to find the minimum or maximum values (respectively) of an array of data. If not present or set to zero, the minimum or maximum (respectively) values are found over the entire array.

Source Code for CLUSTER, CLUST_WTS, EIGENQL, PCOMP

The IDL source code for the **CLUSTER**, **CLUST_WTS**, **EIGENQL**, and **PCOMP** routines is now available. They can be accessed in the **lib** subdirectory of the IDL distribution in the following files: **cluster.pro**, **clust_wts.pro**, **eigenql.pro**, and **pcomp.pro**.

New Histogram Cumulative Probability Distribution Functionality

The new FCN keyword to HIST_EQUAL and ADAPT_HIST_EQUAL allow you to set the resulting histogram's desired cumulative probability distribution function by specifying a 256 element vector. If omitted, a linear ramp, which yields equal probability bins will result. This function is later normalized, so magnitude is not important, though it should increase monotonically.

Language Enhancements

The following enhancements have been made in the area of Language in the IDL 5.5 release:

- [Maximum String Length Limit Increased for 32-Bit IDL](#)
- [New MESSAGE Keywords and Message Block Support](#)
- [Relaxed Formatted Input/Output Record Length Limits](#)
- [New and Enhanced File Handling Routines](#)
- [New Functionality Frees Dynamic Resources](#)
- [New Ability to Check for Keyword Inheritance Errors](#)
- [Enhancements to IDL Path Expansion](#)
- [New Support for REFORM-Style Dimension Array](#)
- [New DOUBLE Keyword for COMPLEX](#)
- [New CENTER Keyword for CONGRID](#)
- [New SIGN Keyword for FINITE](#)
- [Improvements to Files Created with SAVE](#)
- [Improvements to UNIX Filename Expansion](#)
- [Pre-IDL 4.0 C Internals Compatibility Library Removed](#)

Maximum String Length Limit Increased for 32-Bit IDL

Prior to IDL 5.5, 32-bit IDL had a maximum string length limit of 64K (65534 characters) while 64-bit IDL allowed strings to be up to 2.1GB (2147483647 characters) in length. With IDL 5.5, this limit has been raised to 2.1GB for both types of IDL.

New MESSAGE Keywords and Message Block Support

The new message block support in IDL 5.5 allows the MESSAGE routine to issue any IDL error instead of the single IDL_M_USER_ERR message previously supported. IDL `printf`-style formatting is supported, using the `printf`-style formatting added to explicit formatting in IDL 5.4. For more information on the `printf`-style formatting, see “C `printf`-Style Quoted String Format Code” on page 187 of the *Building IDL Applications* manual.

Two new procedures have been added in IDL 5.5 to further provide message block support: the [DEFINE_MSGBLK](#) and [DEFINE_MSGBLK_FROM_FILE](#) procedures. These new procedures allow the user to define new message blocks within large applications built on IDL which must manage their own errors. When a message block is loaded, the messages can be issued to the user-level using the **BLOCK** and **NAME** keywords to the **MESSAGE** procedure.

The **MESSAGE** procedure has been changed by implementing three new keywords: **BLOCK**, **LEVEL**, and **NAME** to allow you to issue any IDL error.

Example Using MESSAGE (Pre-IDL 5.5)

In previous releases of IDL, messages were issued by programs using the **MESSAGE** procedure. The following simple program illustrates how this was done.

This program randomly chooses a number between 0 and 10. It outputs that number to let you know if the messages from your guesses are correct. Then, the program prompts the user to guess the number. If the user's guess is lower than the number, the message "Too Low!" appears in the Output Log. If the user's guess is higher than the number, the message "Too High!" appears in the Output Log. And if the user guesses the number correctly, the message tells the user their guess is correct.

```
PRO guessANumber
; Derive a number in-between 0 and 10.
number = LONG(10.*RANDOMU(seed, 1))
; Output the number.
PRINT, ''
PRINT, 'The number is ' + STRTRIM(number, 2) + '.'
; Initialize variable as a float-point value outside
; of the 0 to 10 range.
guess = -1.
; Loop over guesses until the correct number is inputted.
WHILE (number[0] NE ROUND(guess)) DO BEGIN
; Prompt user to guess.
PRINT, ''
READ, guess, $
    PROMPT = 'Guess a number between 0 and 10: '
; Output whether user is below or above the
; correct number.
PRINT, ''
IF (number[0] GT ROUND(guess)) THEN MESSAGE, $
    'Too Low!', /INFORMATIONAL, /NONAME
IF (number[0] LT ROUND(guess)) THEN MESSAGE, $
    'Too High!', /INFORMATIONAL, /NONAME
; Loop until correct number is inputted.
ENDWHILE
```

```

; Output correct number.
MESSAGE, STRTRIM(number[0], 2) + ' is the number!', $
      /INFORMATIONAL, /NONAME
END

```

New Message Block Support in IDL 5.5

The same example program in IDL 5.5 can now use the new message block support. In the two examples that follow, you will see how to use the `DEFINE_MSGBLK` and the `DEFINE_MSGBLK_FROM_FILE` procedures, respectively to improve the guess a number program.

Message blocks can be defined for an IDL session, or in a main routine of an application using the `DEFINE_MSGBLK` procedure. For large message blocks, it may be easier to maintain a message text file and access it using the `DEFINE_MSGBLK_FROM_FILE` procedure.

These examples establish message blocks for the IDL session. The message blocks are defined from the IDL command line. If you were using either of these procedures in an application, you would define the message block within the main routine of the application, instead from the IDL command line.

DEFINE_MSGBLK Example

This example uses the same program as before with a few modifications. For this example the message block is defined using the `DEFINE_MSGBLK` procedure entered at the IDL command line.

The program must first be modified as follows before defining the new message block as follows.

```

PRO guessANumber
; Derive a number in-between 0 and 10.
number = LONG(10.*RANDOMU(seed, 1))
; Output the number.
PRINT, ""
PRINT, "The number is " + STRTRIM(number, 2) + "."
; Initialize variable as a float-point value outside
; of the 0 to 10 range.
guess = -1.
; Loop over guesses until the correct number is inputed.
WHILE (number[0] NE ROUND(guess)) DO BEGIN
    ; Prompt user to guess.
    PRINT, ""
    READ, guess, $
    PROMPT = "Guess a number between 0 and 10: "
    ; Output whether user is below or above the correct
    ; number.

```

```

PRINT, " "
  IF (number[0] GT ROUND(guess)) THEN MESSAGE, $
    BLOCK = "GUESSING", NAME = "GUESS_MSG_LOW", $
    /INFORMATIONAL
  IF (number[0] LT ROUND(guess)) THEN MESSAGE, $
    BLOCK = "GUESSING", NAME = "GUESS_MSG_HIGH", $
    /INFORMATIONAL
; Loop until correct number is inputed.
ENDWHILE
; Output correct number.
MESSAGE, STRTRIM(number[0], 2), BLOCK = "GUESSING", $
  NAME = "GUESS_MSG_CORRECT", /INFORMATIONAL
END

```

Now define the message block (named GUESSING) to associate the message “Too Low!” with the name GUESS_MSG_LOW, the message “Too High!” with the name GUESS_MSG_HIGH, and the message “%s is the number” with the name GUESS_MSG_CORRECT by entering the following lines of code at the command line.

```

name = ["LOW", "HIGH", "CORRECT"]
format = ["Too Low!", "Too High!", "%s is the number!"]

```

These names and formats are now used to create the message block using the new `DEFINE_MSGBLK` procedure as follows.

```

DEFINE_MSGBLK, "GUESSING", name, format, PREFIX = "GUESS_MSG_"

```

The message block has now been established for the remainder of the IDL session. Now when you run the program, this message block supplies the messages as needed. Once the message block is defined, it exists for the entire session.

Example Using `DEFINE_MSGBLK_FROM_FILE`

This example uses the same message block but defines it as a separate message text file rather than entering it at the IDL command line.

Note

Since the same block of messages is used, exit out of IDL before continuing with this example.

For this example, create a message text file by opening a new file in a text editor. Copy and paste the following text into that file:

```

@IDENT GUESSING
@PREFIX GUESS_MSG_
@      LOW      "Too Low!"
@      HIGH     "Too High!"
@      CORRECT  "%s is the number!"

```

Save this file as `guessANumber.msg` in your IDL working directory.

Start up IDL. At the IDL command line define the message block with the `DEFINE_MSGBLK_FROM_FILE` procedure:

```
DEFINE_MSGBLK_FROM_FILE, "guessANumber.msg"
```

Now you can run the previous example program to see the messages applied using the new `DEFINE_MSGBLK_FROM_FILE` procedure.

Relaxed Formatted Input/Output Record Length Limits

Several IDL record length limits have been relaxed in IDL 5.5.

- The 32K limit for default or explicitly formatted Input/Output has been removed. Now, the only limit on the length of a line is the maximum length allowed in an IDL string variable (2.1GB).
- The A format code used to require that the width parameter be in the range ($1 \leq w \leq 256$). This requirement has been relaxed to ($1 \leq w$).
- The A, F, D, E, G, I, O, Z, X, C(), and open parenthesis (format codes all allow you to specify a repetition count, n , controlling how many times each format element is processed before moving on to the next format element. Previous versions of IDL required this repetition count to fall in the range ($1 \leq n \leq 32767$). This requirement has been relaxed to ($1 \leq n$).
- The T, TL, and TR format codes all require a parameter n , that specifies the column to move to, either directly or as an offset, depending on the format code used. Previous versions of IDL required that n be in the range ($1 \leq n \leq 32767$). This requirement has been relaxed to ($1 \leq n$).

New and Enhanced File Handling Routines

The following table describes new and enhanced routines in IDL 5.5 that improve IDL's ability to perform file handling operations:

New/Enhanced Routine	Description
FILE_CHMOD	New NOEXPAND_PATH keyword allows you to use <i>File</i> exactly as specified, without applying the usual file path expansion.

Table 1-2: New File Handling Routines in IDL 5.5

New/Enhanced Routine	Description
FILE_DELETE	New NOEXPAND_PATH keyword allows you to use <i>File</i> exactly as specified, without applying the usual file path expansion.
FILE_INFO	The new FILE_INFO function provides file status information based on a filename, without opening the file. This differs from FSTAT because FSTAT requires the file to be open, and much of the information FSTAT provides is only relevant for open files. FILE_INFO returns file access, type, and size information, and together with FSTAT and FILE_TEST, provides a complete set of file query operations in IDL. See “ FILE_INFO ” in Chapter 6 for more information.
FILE_MKDIR	New NOEXPAND_PATH keyword allows you to use <i>File</i> exactly as specified, without applying the usual file path expansion.
FILE_SEARCH	<p>The new FILE_SEARCH function returns a string array containing the names of all files matching the input path specification. Input path specifications may contain wildcard characters, enabling them to match multiple files. All matched filenames are returned in a string array, one file name per array element. In comparison to the existing FINDFILE function, FILE_SEARCH is more powerful and provides full cross-platform compatibility. See “FILE_SEARCH” in Chapter 6 for more information.</p> <p>Note - Research Systems strongly recommends the FILE_SEARCH function be used rather than the FINDFILE function. FILE_SEARCH is intended as a replacement for FINDFILE.</p>
FILE_TEST	New NOEXPAND_PATH keyword which allows you to use <i>File</i> exactly as specified, without applying the usual file path expansion.

Table 1-2: New File Handling Routines in IDL 5.5 (Continued)

New Functionality Frees Dynamic Resources

The `HEAP_FREE` routine recursively frees all heap variables associated with the argument which is passed to the routine. This routine will examine the variable data, traversing arrays and structures, pointer, and object references. When an object value is encountered, it is released using the `OBJ_DESTROY` routine. When a pointer value is encountered, its contents are scanned, freeing any dynamic resources, and then the pointer itself is released using the `PTR_FREE` routine.

`HEAP_FREE` may be used:

- To release the dynamic resources contained in a structure returned from the `GetRecord` method of an `IDLdbRecordset` object.
- To release any dynamic resources associated with an event generated by an ActiveX control that is embedded in an IDL Widget hierarchy using `Widget_ActiveX()`.

However, `HEAP_FREE` does have some disadvantages, see “[HEAP_FREE](#)” in Chapter 6 for more information.

New Ability to Check for Keyword Inheritance Errors

When passing inherited keywords to a routine, the `_EXTRA` keyword quietly ignores any keywords not accepted by the routine you are calling. Although this is often the desired behavior, this can allow incorrect usage to go undetected under some circumstances. For example, consider the following two routines:

```
PRO PRINT_HELLO_WORLD, UPCASE = upcase
    PRINT, KEYWORD_SET(upcase) ? 'Hello World!' : 'Hello World!'
END

PRO HELLO_WORLD, number, _EXTRA = extra
    FOR I = 1, number DO PRINT_HELLO_WORLD, _EXTRA = extra
END
```

This generally works as desired, but will not report an error for any inherited keywords that are not understood by the `PRINT_HELLO_WORLD` procedure. For example, if you called the `HELLO_WORLD` procedure using a non-existent keyword (`LOWCASE`), the routine would quietly ignore the incorrect usage:

```
HELLO_WORLD, 2, /LOWCASE
```

You would receive the results:

```
Hello World!
```

Also, if you called the `HELLO_WORLD` procedure with the following (notice that the `UPCASE` keyword is misspelled):

```
HELLO_WORLD, 2, /UCASE
```

You would receive the same results as the previous example since the incorrect keyword would be quietly ignored.

The new `_STRICT_EXTRA` keyword restricts the use of keywords not accepted by the routine you are calling. You can use this keyword to provide error checking. For example, if you changed the `_EXTRA` keyword to the `_STRICT_EXTRA` keyword in the `HELLO_WORLD` procedure:

```
FOR I = 1, number DO PRINT_HELLO_WORLD, _STRICT_EXTRA = extra
```

and run the example again:

```
HELLO_WORLD, 2, /UCASE
```

You would receive the following error message:

```
% Keyword UCASE not allowed in call to: PRINT_HELLO_WORLD
```

Enhancements to IDL Path Expansion

The following enhancements have been made to the expansion of the `IDL_PATH`, `IDL_DLM_PATH`, and `IDL_HELP_PATH` environment variables. IDL expands these variables when they are translated at startup time.

- **Using `<IDL_BIN_DIRNAME>`** — When IDL gets the value of the `IDL_PATH`, `IDL_DLM_PATH`, and `IDL_HELP_PATH` environment variables, it replaces any instances of the string `<IDL_BIN_DIRNAME>` with the name of the subdirectory within the installed IDL distribution where binaries for the current system are kept. This feature is useful for distributing packages of Dynamically Loadable Modules (DLMs) with support for multiple operating system and hardware combinations.

For example, on UNIX, assume that you have your DLMs installed in `/usr/local/mydml`, with support for each platform in a subdirectory using the same naming convention that IDL uses for the platform dependant subdirectories underneath the `bin` directory of the IDL distribution. The following line, which might be located in a file executed by your shell when you log in (your `.cshrc` or `.login` file) will add the location of the proper DLM for your current system to IDL's `!DLM_PATH` at startup:

```
% setenv IDL_DLM_PATH "/usr/local/mydml/<IDL_BIN_DIRNAME>
: <IDL_DEFAULT>"
```

On Windows, you would set the appropriate environment variable, and then exit and restart IDL to update the path.

- **Using <IDL_VERSION_DIRNAME>** — When IDL gets the value of the IDL_PATH, IDL_DLM_PATH, and IDL_HELP_PATH environment variables, it replaces any instances of the string <IDL_VERSION_DIRNAME> with a unique name for the IDL version that is currently running. This feature can be combined with <IDL_BIN_DIRNAME> to easily distribute packages of DLMs with support for multiple IDL versions, operating systems, and hardware platforms.

For example, on UNIX, assume that you have your DLMs installed in /usr/local/mydml. Within the mydml subdirectory would be a directory for each supported version of IDL. Within each of those subdirectories would be a subdirectory for each operating system and hardware combination supported by that version of IDL. The following line, which might be located in a file executed by your shell when you log in (your .cshrc or .login file) will add the location of the proper DLM for your current system to IDL's !DLM_PATH at startup:

```
% setenv IDL_DLM_PATH
    "/usr/local/mydml/<IDL_VERSION_DIRNAME>/
    <IDL_BIN_DIRNAME>:<IDL_DEFAULT>"
```

On Windows, you would set the appropriate environment variable, and then exit and restart IDL to update the path.

New Support for REFORM-Style Dimension Array

The REFORM function in IDL allows you to specify the resulting dimensions (the D_i argument) of an array as separate arguments, or as a single array argument containing the dimensions. For example, if a variable, *a*, is defined as a 20 x 10 x 5 array:

```
a = FINDGEN(20, 10, 5)
```

Then, the following statements are equivalent:

```
b = REFORM(a, 200, 5)
b = REFORM(a, [200, 5])
```

This syntax, which was unique to REFORM, allows code to easily handle data of arbitrary dimensionality. IDL 5.5 extends this notation to the following routines that accept dimension arguments:

BINDGEN	FLTARR	REFORM
BYTARR	INDGEN	REPLICATE
BYTE	INTARR	SHIFT
CINDGEN	L64INDGEN	SINDGEN
COMPLEX	LINDGEN	STRARR
COMPLEXARR	LON64ARR	UINDGEN
DBLARR	LONARR	UINT
DCINDGEN	LONG	UINTARR
DCOMPLEX	LONG64	UL64INDGEN
DCOMPLEXARR	MAKE_ARRAY	ULINDGEN
DINDGEN	OBJARR	ULON64ARR
DOUBLE	PTRARR	ULONARR
FINDGEN	RANDOMN	ULONG
FIX	RANDOMU	ULONG64
FLOAT	REBIN	

Note

The SHIFT function accepts shift parameters (S_i arguments), and not dimensions (D_i argument), but the syntax is identical.

New DOUBLE Keyword for COMPLEX

A new DOUBLE keyword has been added to the COMPLEX function in IDL 5.5. You can set this keyword to return a double-precision complex result. This is equivalent to using the DCOMPLEX function. This keyword is provided as a programming convenience.

New CENTER Keyword for CONGRID

A new CENTER keyword has been added to the CONGRID function in IDL 5.5. If you set this keyword, the interpolation is shifted so that points in the input and output arrays are assumed to lie at the midpoint of their coordinates rather than at their lower-left corner.

New SIGN Keyword for FINITE

A new SIGN keyword has been added to the FINITE function in IDL 5.5. You can use this keyword with the INFINITY and NAN keywords to determine if an infinite or NaN value is positive or negative. By default (SIGN = 0), the FINITE function ignores the sign of infinite and NaN values.

Improvements to Files Created with SAVE

With IDL 5.4, Research Systems released a version of IDL that was 64-bit capable. The original IDL SAVE/RESTORE format used 32-bit offsets. In order to support 64-bit memory access, the IDL SAVE/RESTORE file format was modified to allow the use of 64-bit offsets within the file, while retaining the ability to read old files that use the 32-bit offsets.

The SAVE command always begins reading any .sav file using 32-bit offsets. If the 64-bit offset command is detected, 64-bit offsets are then used for any subsequent commands.

- In IDL versions capable of writing large files (!VERSION.FILE_OFFSET_BITS EQ 64), SAVE writes a special command at the beginning of the file that switches the format from 32 to 64-bit.
- SAVE always starts reading any .sav file using 32-bit offsets. If it sees the 64-bit offset command, it switches to 64-bit offsets for any commands following that one.

This configuration is fully backward compatible, in that any IDL program can read any .sav file it has created, or by any earlier IDL version. Note however that files produced in IDL 5.4 using 64-bit offsets are not readable by older versions of IDL.

It has come to our attention that IDL users commonly transfer SAVE/RESTORE data files written by newer IDL versions to sites where they are restored by older versions of IDL (that is new files being input by old programs). It is not generally reasonable to expect this sort of forward compatibility, and it does not fit the usual definition of backwards compatibility. Research Systems has always strived to maintain this compatibility. However, in IDL 5.4 this was not the case. The following steps have

been taken in IDL 5.5 to minimize the problems that have been caused by the IDL 5.4 save format:

- 64-bit offsets encoding has been improved. The .sav files written within IDL 5.5 and subsequently should be readable by any previous version of IDL, if the file data does not exceed 2.1 GB in length.
- IDL 5.5 and subsequent versions will retain the ability to read the 64-bit offset files produced by IDL 5.4.x, thus ensuring backwards compatibility.
- The .sav files written within IDL 5.5 or subsequent versions, which contain file data exceeding 2.1GB in length are not readable by older versions of IDL, but will be readable by IDL 5.5 and subsequent versions of IDL that have !VERSION.MEMORY_BITS equal to 64.
- The CONVERT_SR54 procedure, a part of the IDL 5.5 user library, can be used to convert .sav files written within IDL 5.4 into the newer IDL 5.5 format. This allows existing data files to become readable by previous IDL versions. The CONVERT_SR54 procedure is located in the `RSI-Directory/lib/obsolete`.

Improvements to UNIX Filename Expansion

IDL for UNIX expands wildcard characters within file names in executive commands (such as `.compile` and `.run`) and in routines that accept file names as arguments (such as `OPEN`, `FILE_TEST`, `FILE_INFO`, and so on). Previous to IDL 5.5, this expansion was done by a child process running the C-shell (`/bin/csh`). Now, this expansion is done by IDL's internal file searching engine, which is also the heart of the new `FILE_SEARCH` function. The wildcard characters accepted remain the same (`~`, `*`, `?`, `[]`, `{ }`, and environment variables), and any change should be negligible. However, expansion of C-shell variables such as `$path` or `$shell` are no longer expanded. Instead, they are treated as environment variables, and since most environments do not contain lower-case names, they expand to null replacement text. In this case, the desired effect can usually be obtained by instead using the equivalent environment variables (for example `$SHELL`, or `$PATH`).

Pre-IDL 4.0 C Internals Compatibility Library Removed

The sharable library `libobsolete.so` (known as `libobsolete.a` under AIX, and `libobsolete.sl` under HP-UX) has been removed from IDL. This library, which first appeared within IDL 4.0, supplied implementations of the older non-IDL_ prefixed IDL internal API (application programming interface) written in terms of the API documented in the IDL External Development Guide.

Historical Note: IDL 4.0 (released in 1995) offered Callable IDL, which allows IDL to be called from other compiled programs. From that time, the names of all externally visible functions and data structures have had a standard `IDL_` prefix. This prevented internal IDL names from conflicting with names in the calling user program. In order to ease the transition for UNIX and VMS customers with existing code, a sharable library (`libobsolete.so` for UNIX, and `OBSOLETE.EXE` for VMS) was included in the `bin` subdirectory that contained an implementation of the old non-prefixed API written in terms of the new. This code consists largely of functions with the old names each making a single call to the corresponding function in the IDL sharable library. It has always been recommended that user code be revised to utilize the newer supported API instead of the older API. For a time however, the option of linking against `libobsolete` has been available during the transition. The amount of code which relied on this library has never been large, and after six years, any code that relied on this library should have had ample time to be converted to the new prefixed API. Therefore, the obsolete library is no longer included in the IDL distribution. If you have existing code that relies on this library, it is recommended that it be converted to the supported version of the API, as documented in the [*External Development Guide*](#).

User Interface Toolkit Enhancements

The following enhancements have been made in the area of the User Interface Toolkit in the IDL 5.5 release:

- [New COM and ActiveX Functionality for IDL](#)
- [New Shortcut Menu Widget](#)
- [Emulating System Colors in Application Widgets](#)
- [New Functionality to Specify Slider Increments in IDL Widgets](#)

New COM and ActiveX Functionality for IDL

IDL for Windows now supports the use of COM objects. COM (Component Object Model) objects, regardless of type or method of creation, are treated as IDL objects.

There are two main uses for COM functionality in IDL:

- Using the `IDLcomIDispatch` object to instantiate a desired COM object by using a provided class or program ID. This method is ideal for COM objects that do not utilize a graphical-user interface.
- Using the `WIDGET_ACTIVEX` function to embed an ActiveX control in an IDL widget hierarchy.

The primary differences in IDL between using `IDLcomIDispatch`-based objects and using an ActiveX control are the methods by which they are created and managed. These methods of creation and management as well as more in-depth information on COM objects are detailed in [Chapter 3, “Using COM Objects in IDL”](#).

New Shortcut Menu Widget

In IDL 5.5 for Windows and IDL 5.5 for UNIX, a shortcut menu widget (otherwise known as a context sensitive or pop-up menu) has been added to enhance the IDL widget system. These menus are available for:

- Base widgets
- Text widgets
- Draw widgets
- List widgets

An example of a shortcut menu widget is shown in the following figure.

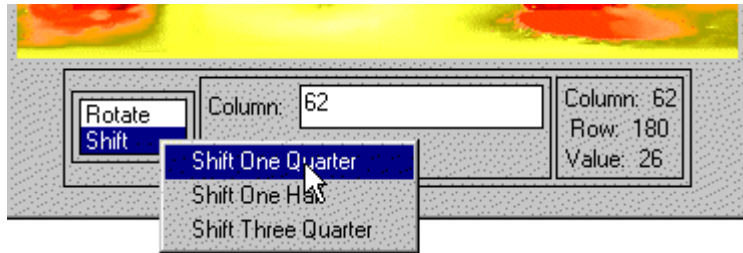


Figure 1-2: Shortcut Menu Widget

For more information, see [Chapter 4, “Using the Shortcut Menu Widget”](#).

Emulating System Colors in Application Widgets

A new [SYSTEM_COLORS](#) keyword has been added to the `WIDGET_INFO` routine for the Windows and UNIX operating systems. This new keyword enables an application developer to determine what colors are used in IDL application widgets so they can design widgets for their application with the same look and feel as the supplied IDL widgets.

The `WIDGET_SYSTEM_COLORS` Structure

When the new `SYSTEM_COLORS` keyword is used in a `WIDGET_INFO` call with a valid IDL widget identifier, an IDL structure is returned. The `WIDGET_SYSTEM_COLORS` structure contains 25 fields holding the 3 element vector values for the corresponding RGB colors. The vector elements range between 0 and 255 or are assigned a value of -1 if unavailable. The field names and meaning on the Windows and UNIX operating systems are shown in the following table.

Field Names	Windows Platform	UNIX Platform
<code>DARK_SHADOW_3D</code>	Dark shadow color for 3D display elements.	N/A
<code>FACE_3D</code>	Face color for 3D display elements and dialog boxes.	Base background color for all widgets.

Table 1-3: `WIDGET_SYSTEM_COLORS` Structure Fields

Field Names	Windows Platform	UNIX Platform
LIGHT_EDGE_3D	Highlight color for 3D edges that face the light source.	Color of top and left edges of 3D widgets.
LIGHT_3D	Light color for 3D display elements.	Color of highlight rectangle around widgets with the keyboard focus.
SHADOW_3D	Color for 3D edges that face away from the light source.	Color of bottom and right edges of 3D widgets.
ACTIVE_BORDER	Active window's border color.	Push button background color when button is armed.
ACTIVE_CAPTION	Active window's caption color.	N/A
APP_WORKSPACE	Background color of MDI applications.	N/A
DESKTOP	Desktop color.	N/A
BUTTON_TEXT	Text color on push buttons.	Widget text color.
CAPTION_TEXT	Color of text in caption, size box, and scroll bar arrow box.	Widget text color.
GRAY_TEXT	Color of disabled text.	N/A
HIGHLIGHT	Color of item(s) selected in a widget.	Toggle button fill color.
HIGHLIGHT_TEXT	Color of text of item(s) selected in a widget.	N/A
INACTIVE_BORDER	Inactive window's border color.	N/A

Table 1-3: WIDGET_SYSTEM_COLORS Structure Fields (Continued)

Field Names	Windows Platform	UNIX Platform
INACTIVE_CAPTION	Inactive window's caption color.	N/A
INACTIVE_CAPTION_TEXT	Inactive window's caption text color.	N/A
TOOLTIP_BK	Background color for tooltip controls.	N/A
TOOLTIP_TEXT	Text color for tooltip controls.	N/A
MENU	Menu background color.	N/A
MENU_TEXT	Menu text color.	N/A
SCROLLBAR	Color of scroll bar "gray" area.	Color of scroll bar "gray" area.
WINDOW_BK	Window background color.	Base background color for all widgets.
WINDOW_FRAME	Window frame color.	Widget border color.
WINDOW_TEXT	Text color in windows.	Widget text color.

Table 1-3: WIDGET_SYSTEM_COLORS Structure Fields (Continued)

Note

This feature is currently not available on the Macintosh platform.

New Functionality to Specify Slider Increments in IDL Widgets

The WIDGET_SLIDER and CW_FSLIDER widgets in IDL for Windows and Macintosh have right and left arrow buttons that increment the sliders. In IDL 5.5, you may now specify the amount the slider is incremented when the arrow buttons are pressed. The SCROLL keyword to WIDGET_SLIDER (increments by integer values) and CW_FSLIDER (increments by floating point/decimal values) now causes the slider to be incremented by the correct amount each time the slider arrows are pressed.

File Access Enhancements

The following enhancements have been made in the area of File Access in the IDL 5.5 release:

- [New PATH_SEP Function](#)
- [Enhanced TIFF Support](#)
- [New Support for MrSID](#)

New PATH_SEP Function

The new PATH_SEP function returns the proper segment separator character in the file path for the current operating system. This is the same character used by the host operating system for delimiting subdirectory names in a path specification. This new function enables code to be more flexible and portable as opposed to hardwiring the separators in the code.

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

Enhanced TIFF Support

Enhanced Support for 1-bit and 4-bit TIFF Images

IDL 5.5 now supports reading and writing 1-bit (black and white) and 4-bit TIFF files. The WRITE_TIFF procedure can write TIFF files with one or more channels, where each channel can contain 1, 4, 8, 16, or 32-bit integer pixels, or floating-point values. For black and white images, writing out the image as a 1-bit TIFF will take approximately 1/8 of the disk space compared to an 8-bit grayscale image. For 4-bit images (pixel values 0 through 15), writing out the image as a 4-bit TIFF will take approximately 1/2 the disk space compared to an 8-bit grayscale image.

New Returned Information for TIFF Queries

The Info argument to QUERY_TIFF returns an anonymous structure containing information about the image in the file. In IDL 5.5 the following new QUERY_TIFF fields have been added:

Field	IDL data type	Description
BITS_PER_SAMPLE	Long	This new field indicates the number of bits per sample or channel. Possible values are 1, 4, 8, 16, or 32.
ORIENTATION	Long	This new field indicates image orientation (by columns and rows): <ul style="list-style-type: none"> • 1 = Left to right, top to bottom (default) • 2 = Right to left, top to bottom • 3 = Right to left, bottom to top • 0 or 4 = Left to right, bottom to top • 5 = Top to bottom, left to right • 6 = Top to bottom, right to left • 7 = Bottom to top, right to left • 8 = Bottom to top, left to right
PLANAR_CONFIG	Long	This new field indicates how the components of each pixel are stored. Possible values are: <ul style="list-style-type: none"> • 0 = Pixel interleaved RGB image or a two-dimensional image (no interleaving exists). Pixel components (such as RGB) are stored contiguously. • 2 = Image interleaved. Pixel components are stored in separate planes.

Table 1-4: QUERY_TIFF Routine Info Structure Fields

Field	IDL data type	Description
PHOTOMETRIC	Long	<p>This new field indicates the color model used for the image data. Possible values are:</p> <ul style="list-style-type: none"> • 0 = White is zero • 1 = Black is zero • 2 = RGB color model • 3 = Palette color model • 4 = Transparency mask • 5 = Separated (usually CMYK - cyan-magenta-yellow-black)
RESOLUTION	Float array	<p>This new field is a two-element vector [x resolution, y resolution] giving the number of pixels per resolution unit in the width and height directions.</p>
UNITS	Long	<p>This new field is used to indicate the units of measurement for RESOLUTION:</p> <ul style="list-style-type: none"> • 1 = No units • 2 = Inches (the default) • 3 = Centimeters
TILE_SIZE	Long array	<p>This new field is used for images stored in separate tiles. This is a two-element vector [<i>tile width</i>, <i>tile height</i>] giving the width and height of each tile. For non-tiled images the TILE_SIZE will contain [Image width, 1].</p>

Table 1-4: QUERY_TIFF Routine Info Structure Fields (Continued)

Improved TIFF Orientation Functionality

In IDL 5.5, a new ORIENTATION keyword has been added for WRITE_TIFF as well as for READ_TIFF. The ORIENTATION keyword is set to indicate the orientation of the image with respect to the columns and rows of *Image*. This ORIENTATION keyword replaces the Order argument to WRITE_TIFF and the

ORDER keyword to READ_TIFF which are now both obsolete. Code that uses the Order argument or ORDER keyword will continue to work as before, but new code should use the ORIENTATION keyword.

New Unit-setting Functionality for WRITE_TIFF

The new UNITS keyword to WRITE_TIFF can be set to indicate the units of the XRESOL and YRESOL keywords (which define the horizontal and vertical resolutions). Possible values are; 1 = No units, 2 = Inches (the default), or 3 = Centimeters.

New Support for MrSID

In IDL 5.5 for Windows, functionality has been added for MrSID. The MrSID (Multi-Resolution Seamless Image Database) file format is a wavelet compressed, multi-resolution raster image format. The multi-resolution nature of the format allows the image to be opened using *selective decompression* with only the required portion of an image being opened at once. Using this method, the image may be viewed at the highest detail while never being fully decompressed. The memory requirements and time delays associated with opening a full image into memory are thus avoided, and an image, irrespective of size, may be viewed quickly at any resolution. IDL 5.5 now provides support for MrSID through use of the IDLffMrSID object and through the READ_MRSID and QUERY_MRSID methods. The IDLffMrSID object encapsulates all functionality that is required to access MrSID files.

For more information on the new IDLffMrSID class, see [Chapter 5, “New Objects”](#).

Development Environment Enhancements

Improved Project Exporting

IDL 5.5 for Windows features enhanced project exporting capabilities. The export feature assists users in packaging up their IDL programs for distribution.

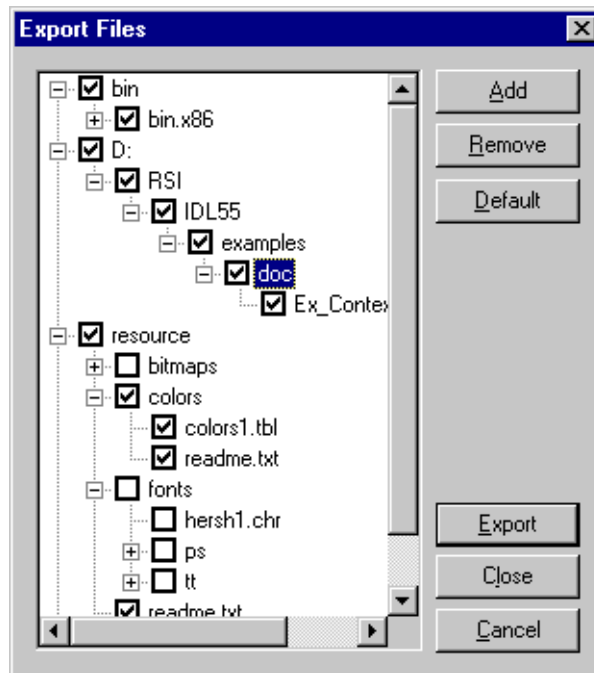


Figure 1-3: The New Export Files Dialog

For more information on this feature and how to export and distribute an IDL application, contact your RSI sales representative.

Scientific Data Formats Enhancements

Enhancements have been made to the following Scientific Data Formats in the IDL 5.5 release:

- [HDF-EOS Data Output Enhancements](#)
- [New HDF Vdata Attribute Routines](#)

HDF-EOS Data Output Enhancements

IDL HDF-EOS routines now consistently handle the array ordering between IDL and C used by the HDF-EOS library routines. In addition, dimension size vectors and dimension name lists are also now in IDL order rather than in C order. This was done so that IDL order is maintained in the reading and writing of data arrays with the HDF-EOS routines.

Enhanced routines include:

EOS_SW_DEFDATAFIELD	EOS_GD_DEFFIELD
EOS_SW_DEFGEOFIELD	EOS_GD_DEFTILE
EOS_SW_EXTRACTPERIOD	EOS_GD_READFIELD
EOS_SW_EXTRACTREGION	EOS_GD_READTILE
EOS_SW_PERIODINFO	EOS_GD_REGIONINFO
EOS_SW_READFIELD	EOS_GD_TILEINFO
EOS_SW_REGIONINFO	EOS_GD_WRITEFIELD
EOS_SW_WRITEDATAMETA	EOS_GD_WRITEFIELDMETA
EOS_SW_WRITEFIELD	EOS_GD_WRITETILE
EOS_SW_WRITEGEOMETA	

Note

For the EOS_GD_READFIELD, EOS_SW_READFIELD, EOS_GD_WRITEFIELD, and EOS_SW_WRITEFIELD routines, the START, STRIDE, and EDGE keywords should also be specified in the IDL dimension order.

Note

EOS_GD_INQDIMS and EOS_SW_INQDIMS return dimension size and name information without consideration of order.

Note

Programs written with previous versions of the IDL HDF-EOS routines may have been created to intentionally compensate for the previous behavior. Due to the array-handling enhancements in IDL 5.5, this work-around may now generate incorrect results.

New HDF Vdata Attribute Routines

New IDL versions of the HDF vdata attribute routines have been created. HDF has seven routines dealing with vdata attributes, whose functionality have been built into five new IDL routines. Vdata attributes are scalars, vectors or strings and can be associated with a vdata (like a data table) or with a specific field (column of the table) in a vdata. These new IDL routines are:

- **HDF_VD_ATTRSET** - Creates an attribute for a given vdata or vdata/field pair.
- **HDF_VD_ATTRINFO** - Gets information about a particular vdata attribute, including its value(s).
- **HDF_VD_ATTRFIND** - Returns the attribute index number for a given attribute name.
- **HDF_VD_NATTRS** - Returns the number of attributes associated with a vdata or a vdata/field pair.
- **HDF_VD_ISATTR** - Indicates whether the specified vdata is being used to store an attribute (in HDF, HDF structures are used to store internal HDF information).

IDL ActiveX Control Enhancements

IDL 5.5 includes a new version of the IDLDrawX ActiveX control. The control is now named IDLDrawX3. This control has added a method to allow specification of IDL_Init options for use in developing external applications.

Why Was a New Version of the Control Created?

One of the features of COM is that interfaces are immutable. That is to say that when an interface is created you “contractually” agree that the interface won’t change. Changes require that a new interface (or version) be created. Since the IDL ActiveX control is a COM object it is bound by this agreement. Because we have made improvements to the ActiveX control interface by adding new methods and properties, it was necessary that we create a new ActiveX control with the new interface.

What Must You Change to Take Advantage of the Control?

If you are a Visual Basic user, you need to add the “IDLDrawX3 ActiveX Control Module” to your project and remove the “IDLDrawX ActiveX Control Module” or “IDLDrawX2 ActiveX Control Module” from your project. The source code need not change.

What About the Previous ActiveX Control?

While previous versions of the IDLDrawX control will continue to work with new versions of IDL, it is no longer supported and will not be shipped with IDL. It is recommended that you upgrade to the new version to take advantage of new features and bug fixes.

Why Should You Upgrade?

The new control has a number of new features including printing support, dual interface control, and new memory improvements. The rest of this section details the improvements in the new version of the IDL ActiveX control.

IDL DataMiner Enhancements

In IDL 5.5, the ODBC support for IDL DataMiner has been upgraded. This upgrade affects the following platforms:

- Solaris (Sparc base platforms)
- AIX
- HP-UX
- Windows
- Linux (new platform support), see [“Platform Specific Information”](#) for more information.

Other supported platforms remain at the current level of ODBC support. These platforms are:

- MacOS
- SGI IRIX

Platform Specific Information

ODBC drivers are installed with IDL if you have selected the IDL DataMiner option. For more information on installing IDL, see the [Installing and Licensing IDL 5.5](#) manual.

For more information on specific platform requirements, issues, and how to configure the ODBC driver for use with your database, see the Merant *DataDirect Connect ODBC Reference* manual.

- For IRIX and Macintosh, see the 3.11 version of the [DataDirect Connect ODBC Reference](#) manual.
- For all other platforms, see the 3.7 version of the [DataDirect Connect ODBC Reference](#) manual. Both manuals are located in the `info/docs/odbc` directory of your product CD-ROM.

The following table describes the drivers that are included with and supported by IDL DataMiner:

Note

The following table is for support of ODBC drivers on specific platforms, which maybe different from support of IDL 5.5 (including IDL DataMiner) on specific platforms. IDL platform support may supersede the listed OS levels for ODBC drivers. See [“Platforms Supported in this Release”](#) on page 124 for more information.

Supported Databases	Driver Name	Supported Platforms
INFORMIX 7.x or 9.x	INFORMIX 9	Windows 98, Me, NT 4.0, 2000 Sun Solaris 8 AIX 4.3 IRIX 6.5 HP-UX 11 Red Hat Linux 6.2, Caldera OpenLinux 2.3, SuSE Linux 6.4
INFORMIX Dynamic Server 9.x, 2000	INFORMIX WP	Windows 98, Me, NT 4.0, 2000 Sun Solaris 8 HP-UX 11 Red Hat Linux 6.2, Caldera OpenLinux 2.3, SuSE Linux 6.4
Oracle 7.x (7.x functionality via SQL*Net 2.x)	Oracle7	Windows 98, Me, NT 4.0, 2000 Solaris 8 AIX 4.3 IRIX 6.5 Red Hat Linux 6.2, Caldera OpenLinux 2.3, SuSE Linux 6.4 Mac OS 8.1

Table 1-5: Supported ODBC Drivers for DataMiner

Supported Databases	Driver Name	Supported Platforms
Oracle 8.0.5+, 7.3, 8I (via Net 8 8.0.5+)	Oracle 8	Windows 98, Me, NT 4.0, 2000 Solaris 8 AIX 4.3 HP-UX 11 IRIX 6.5 (requires Oracle N32 Client Development Kit, Version 8.0.5.0.0 (Oracle Part Number Z24604-02) or later) Red Hat Linux 6.2, Caldera OpenLinux 2.3, SuSE Linux 6.4 Mac OS 8.1 (SQL*Net 2.x)
Sybase Adaptive Server 11.0 +	SybaseASE	Windows 98, Me, NT 4.0, 2000 Solaris 8 AIX 4.3 HP-UX 11 Red Hat Linux 6.2, Caldera OpenLinux 2.3, SuSE Linux 6.4
SQL Server 4.9.2	Sybase	IRIX 6.5 MacOS 8.1 (System 10 and 11 only)
Sybase System 10.11, Adaptive Server 11.x, 12.0	Sybase	IRIX 6.5
MS SQL Server 6.5, 7.0, 2000 (UNIX support on SQL Server 2000 is via 7.0 functionality)	MS_SQLServer7	Windows 98, Me, NT 4.0, 2000 Solaris 8 AIX 4.3 HP-UX 11 Red Hat Linux 6.2, Caldera OpenLinux 2.3, SuSE Linux 6.4

Table 1-5: Supported ODBC Drivers for DataMiner

Supported Databases	Driver Name	Supported Platforms
ASCII text files	Text	Windows 98, Me, NT 4.0, 2000 Solaris 8 AIX 4.3 HP-UX 11 IRIX 6.5 MacOS 8.1 Red Hat Linux 6.2, Caldera OpenLinux 2.3, SuSE Linux 6.4

Table 1-5: Supported ODBC Drivers for DataMiner

Note

For more information on specific platform requirements, issues, and how to configure the ODBC driver for use with your database, see the Merant *DataDirect Connect ODBC Reference* manual. For IRIX and Macintosh, see the 3.11 version of the *DataDirect Connect ODBC Reference* manual, for all other platforms, see the 3.7 version of the *DataDirect Connect ODBC Reference* manual. Both manuals are located in the `/info/docs/odbc` directory of your product CD-ROM.

Documentation Enhancements

Many new examples highlighting a wide range of functionality in IDL have been added in this release. These examples provide code that can be easily followed and adapted when developing your own routines using the covered functionality. Areas that have new examples are:

- Object Graphics
- Language
- Visualization
- Analysis

For more information, see [Chapter 7, “New Examples”](#).

Enhanced IDL Utilities

IDL 5.5 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.

Enhanced 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
XOBJVIEW	The XOBJVIEW_ROTATE and XOBJVIEW_WRITE_IMAGE procedures, which can be called only after a call to XOBJVIEW, can be used to easily create animations of volumes and isosurfaces displayed in XOBJVIEW. For more information, see XOBJVIEW_ROTATE and XOBJVIEW_WRITE_IMAGE .

Table 1-6: Enhanced IDL Utilities

New Keywords/Arguments to Existing IDL Utilities

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

XOBJVIEW

Item	Description
RENDERER	<p>Set this keyword to an integer value indicating which graphics renderer to use when drawing objects in the XOBJVIEW draw window. Valid values are:</p> <ul style="list-style-type: none"> • 0 = Platform native OpenGL • 1 = IDL's software implementation <p>By default, your platform's native OpenGL implementation is used. If your platform does not have a native OpenGL implementation, IDL's software implementation is used regardless of the value of this property.</p>
JUST_REG	<p>Set this keyword to indicate that the XOBJVIEW utility should just be registered and return immediately.</p>
XOFFSET	<p>The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.</p> <p>Specifying an offset relative to a row-major or column-major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.</p>

Item	Description
YOFFSET	<p>The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the upper left corner of the parent widget.</p> <p>Specifying an offset relative to a row-major or column-major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.</p>

XROI

Keyword/Argument	Description
TOOLS	<p>The values for the TOOLS keyword indicate the buttons to be included on an XROI toolbar. New values to the TOOLS keyword are:</p> <ul style="list-style-type: none"> • 'Translate Scale' — Enables translation and scaling of ROIs. Mouse down on the bounding box selects a region, mouse motion translates (repositions) the region. Mouse down on a scale handle of the bounding box enables scaling (stretching, enlarging and shrinking) of the region according to mouse motion. Mouse up finishes the translation or scaling. • 'Rectangle' — Enables rectangular ROI drawing. Mouse down positions one corner of the rectangle, mouse motions creates the rectangle, positioning the rectangle's opposite corner, mouse up finishes the rectangular region. • 'Ellipse' — Enables elliptical ROI drawing. Mouse down positions the center of the ellipse, mouse motion positions the corner of the ellipse's imaginary bounding box, mouse up finishes the elliptical region.

New and Enhanced IDL Objects

This section describes the following:

- [New Object Classes](#)
- [IDL Object Method Enhancements](#)

New Object Classes

The following table describes the new object classes in IDL 5.5 for Windows.

New Object Class	Description
IDLcomIDispatch	Used to create and utilize an IDispatch COM object in IDL which implements the IDispatch interface.
IDLffMrSID	Used to query information about and load image data from a MrSID (.sid) image file.

IDL Object Method Enhancements

The following table describes new and updated keywords and arguments to IDL object methods.

IDLgrBuffer::Pickdata

Item	Description
DIMENSIONS	<p>Set this keyword to a two-element array $[w, h]$ to specify data picking should occur for all device locations that fall within a pick box of these dimensions. The pick box will be centered about the coordinates $[x, y]$ specified in the <i>Location</i> argument, and will occupy the rectangle defined by:</p> $(x-(w/2), y-(h/2)) - (x+(w/2), y+(h/2))$ <p>By default, the pick box covers a single pixel. The return value of the Pickdata method will match the dimensions of the pick box. Likewise, the array returned via the <i>XYZLocation</i> argument will have dimensions $[3, w, h]$.</p>

IDLgrContour::GetProperty

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the FILL keyword is set.</p>

IDLgrContour::Init

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the FILL keyword is set.</p>

IDLgrContour:: SetProperty

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the FILL keyword is set.</p>

IDLgrPolygon::GetProperty

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the value of the STYLE keyword is 2 (Filled).</p>

IDLgrPolygon::Init

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the value of the STYLE keyword is 2 (Filled).</p>

IDLgrPolygon::SetProperty

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the value of the STYLE keyword is 2 (Filled).</p>

IDLgrSurface::GetProperty

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the value of the STYLE keyword is 2 or 6 (Filled or LegoFilled).</p>

IDLgrSurface::Init

Enhancement	Description
TEXTURE_HIGHRES	<p>Set this keyword to cause texture tiling to be used as necessary to maintain the full pixel resolution of the original texture image. This is recommended if IDL is running on modern 3D hardware and resolution loss due to downscaling becomes problematic. If not set, and the texture map is larger than the maximum resolution supported by the 3D hardware, the texture is scaled down to the maximum resolution supported by the 3D hardware on your system. The default value is 0.</p> <p>Note - Because of the way in which high-resolution textures require modified texture coordinates, if you specify the TEXTURE_COORD keyword, high resolution textures (TEXTURE_HIGHRES) will be disabled.</p>

Enhancement	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the value of the STYLE keyword is 2 or 6 (Filled or LegoFilled).</p>

IDLgrSurface:: SetProperty

Item	Description
DEPTH_OFFSET	<p>An integer value that specifies an offset in depth to be used when rendering filled primitives. This offset is applied along the viewing axis, with positive values moving the primitive away from the viewer.</p> <p>The units are "Z-Buffer units", where a value of 1 is used to specify a distance that corresponds to a single step in the device's Z-Buffer.</p> <p>Use DEPTH_OFFSET to always cause a filled primitive to be rendered slightly deeper than other primitives, independent of model transforms.</p> <p>This is useful for avoiding stitching artifacts caused by rendering lines or polygons on top of other polygons at the same depth.</p> <p>Note - RSI suggests using this feature to remove stitching artifacts and not as a means for "layering" complex scenes with multiple DEPTH_OFFSET values. It is safest to use only a DEPTH_OFFSET value of 0, the default, and one other non-zero value such as 1. Many system-level graphics drivers are not consistent in their handling of DEPTH_OFFSET values, particularly when multiple non-zero values are used. This can lead to portability problems because one set of DEPTH_OFFSET values may produce better results on one machine as compared to another. Using IDL's software renderer will help improve the cross-platform consistency of scenes that use DEPTH_OFFSET.</p> <p>Note - DEPTH_OFFSET has no effect unless the value of the STYLE keyword is 2 or 6 (Filled or LegoFilled).</p>

IDLgrWindow::Pickdata

Item	Description
DIMENSIONS	<p>Set this keyword to a two-element array $[w, h]$ to specify data picking should occur for all device locations that fall within a pick box of these dimensions. The pick box will be centered about the coordinates $[x, y]$ specified in the <i>Location</i> argument, and will occupy the rectangle defined by: $(x-(w/2), y-(h/2)) - (x+(w/2), y+(h/2))$</p> <p>By default, the pick box covers a single pixel. The return value of the Pickdata method will match the dimensions of the pick box. Likewise, the array returned via the <i>XYZLocation</i> argument will have dimensions $[3, w, h]$.</p>

New and Enhanced IDL Routines

This section describes the following:

- [New IDL Routines](#)
- [IDL Routine Enhancements](#)
- [Updates to Executive Commands](#)

New IDL Routines

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

New Routine	Description
CPU	Controls the way IDL uses the system processor for calculations. The results of using the CPU procedure are reflected in the state of the !CPU system variable.
DEFINE_MSGBLK	Defines and loads a new message block into the currently running IDL session. Once loaded, the MESSAGE procedure can be used to issue messages from this block.
DEFINE_MSGBLK_FROM_FILE	Reads the definition of a message block from a file, and uses DEFINE_MSGBLK to load it into the currently running IDL session. Once loaded, the MESSAGE procedure can be used to issue messages from this block.
ERF	Returns the value of the error function.
ERFC	Returns the value of the complimentary error function.
ERFCX	Returns the value of the scaled complimentary error function.
FILE_INFO	Returns status information about a specified file, without opening the file.

New Routine	Description
FILE_SEARCH	<p>Returns a string array containing the names of all files matching the input path specification. Input path specifications may contain wildcard characters, enabling them to match multiple files. All matched filenames are returned in a string array, one file name per array element.</p> <p>Note - Research Systems strongly recommends the FILE_SEARCH function be used rather than the FINDFILE function. FILE_SEARCH is intended as a replacement for FINDFILE.</p>
GRID_INPUT	This new procedure preprocesses and sorts two-dimensional data sets and removes duplicate points.
GRIDDATA	This new function interpolates data to a regular grid from scattered data values and locations using one of several available interpolation methods. Computations are preformed in single precision floating point.
HDF_VD_ATTRFIND	This new function returns an attribute's index number given the name of an attribute associated with the specified vdata or vdata/field pair. If the attribute cannot be located, -1 is returned.
HDF_VD_ATTRINFO	This new procedure reads or retrieves information about a vdata attribute or a vdata field attribute from the currently attached HDF vdata structure. If the attribute is not present, an error message is printed.

New Routine	Description
HDF_VD_ATTRSET	This new procedure writes a vdata attribute or a vdata field attribute to the currently attached HDF vdata structure. If no data type keyword is specified, the data type of the attribute value is used.
HDF_VD_ISATTR	This new function returns TRUE (1) if the vdata is storing an attribute, FALSE (0) otherwise. HDF stores attributes as vdatas, so this routine provides a means to test whether or not a particular vdata contains an attribute.
HDF_VD_NATTRS	This new function returns the number of attributes associated with the specified vdata or vdata/field pair if successful. Otherwise, -1 is returned.

New Routine	Description
HEAP_FREE	<p>This new routine frees all dynamic resources associated with the argument which is passed to the routine. This routine will traverse the data represented by the variable, traversing arrays and structures. When an object value is encountered, it is released using the OBJ_DESTROY routine. When a pointer value is encountered, its contents are scanned, freeing any dynamic resources, and then the pointer itself is released using the PTR_FREE routine. This is especially helpful with routines that return dynamically allocated information.</p> <p>HEAP_FREE may be used:</p> <ul style="list-style-type: none"> • To release the dynamic resources contained in a structure returned from the GetRecord method of an IDLdbRecordset object. • To release any dynamic resources associated with an event generated by an ActiveX control that is embedded in an IDL Widget hierarchy using Widget_ActiveX(). <p>However, HEAP_FREE does have some disadvantages, see “HEAP_FREE” in Chapter 6 for more information.</p>

New Routine	Description
INTERVAL_VOLUME	This new procedure can be used to generate a tetrahedral mesh from volumetric data. The mesh generated by this procedure spans the portion of the volume where the volume data samples fall between two constant data values. This can also be thought of as a mesh constructed to fill the volume between two isosurfaces where the isosurfaces are drawn at the two supplied constant data values.
PATH_SEP	This new function returns the proper file path segment separator character for the current operating system.
QGRID3	Linearly interpolates the dependent variable values to points in a regularly sampled volume, given a triangulation of scattered data points in three dimensions, and the value of a dependent variable for each point.
QHULL	This new function constructs convex hulls, Delaunay triangulations, and Voronoi diagrams of a set of points of 2 or more dimensions. It uses and is based on the program QHULL, which is described in Barber, Dobkin and Huhdanpaa, "The Quickhull Algorithm for Convex Hulls," <i>ACM Transactions on Mathematical Software</i> , Vol. 22, No 4, December 1996, Pages 469-483.

New Routine	Description
QUERY_MRSID (Windows only)	This new method allows you to obtain information about a MrSID image file without having to import in an image from the file. This wrapper around the object interface presents MrSID image loading in a familiar way to users of the QUERY_* image routine. However this function is not as efficient as the object interface and the object interface should be used whenever possible.
READ_MRSID (Windows only)	This new method extracts and returns image data from a MrSID file at the specified level and location. This wrapper around the object interface presents MrSID image loading in a familiar way to users of the READ_* image routine. However this function is not as efficient as the object interface and the object interface should be used whenever possible.
REAL_PART	This new function returns the real part of its complex-valued argument.

New Routine	Description
REGION_GROW	This new function performs region growing for a given region within an N-dimensional array by expanding the region to include all connected neighboring pixels that fall within the specified limits. The limits are specified either as a threshold range (a minimum and maximum pixel value) or as a multiple of the standard deviation of the original region pixel values. If the threshold is used (this is the default), the region is grown to include all connected neighboring pixels that fall within the given threshold range. If the standard deviation multiple is used, the region is grown to include all connected neighboring pixels that fall within the range of the mean (of the region's pixel values) plus or minus the given multiplier times the sample standard deviation. REGION_GROW returns the vector of array indices that represent pixels within the grown region.
SIMPLEX	The new SIMPLEX function uses the simplex method to solve linear programming problems.
WIDGET_ACTIVEX (Windows only)	The new WIDGET_ACTIVEX function creates an ActiveX control in IDL and places it into an IDL widget hierarchy.

New Routine	Description
WIDGET_DISPLAYCONTEXTMENU (Windows, UNIX only)	The new WIDGET_DISPLAYCONTEXTMENU function displays a context menu. After buttons for the context menu have been created, a context menu can be displayed using WIDGET_DISPLAYCONTEXTMENU . This is normally called in an event handler that has processed a context menu event. This procedure takes the ID of the widget that is the parent of the context menu, the x and y location to display the menu, and the ID of the context menu base.
XOBJVIEW_ROTATE	This procedure can be used to programmatically rotate the object currently displayed in XOBJVIEW .
XOBJVIEW_WRITE_IMAGE	This procedure can be used to write the object currently displayed in XOBJVIEW to an image file using the specified name and file format.

IDL Routine Enhancements

The following is a list of new and updated keywords, arguments, and/or return values to existing IDL routines.

ABS

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The ABS function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ACOS

Item	Description
Complex Input	ACOS now supports complex input.
Thread Pool Keywords (Windows, UNIX only)	The ACOS function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ADAPT_HIST_EQUAL

Item	Description
FCN	Set this keyword to the desired cumulative probability distribution function in the form of a 256 element vector. If omitted, a linear ramp, which yields equal probability bins results. This function is later normalized, so magnitude is inconsequential, though it should increase monotonically.

ALOG

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The ALOG function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ALOG10

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The ALOG10 function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ASIN

Item	Description
Complex Input	ASIN now supports complex input.
Thread Pool Keywords (Windows, UNIX only)	The ASIN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ATAN

Item	Description
Complex Input	ATAN now supports complex input as well as input of two complex arguments.
Thread Pool Keywords (Windows, UNIX only)	The ATAN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

BINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The BINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

BREAKPOINT

Item	Description
ON_RECOMPILE	This new keyword allows you to specify that a breakpoint will not take effect until the next time the file containing it is compiled.

BYTARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

BYTE

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The BYTE function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

BYTEORDER

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The BYTEORDER function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

BYTSCL

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The BYTSCL function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

CEIL

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The CEIL function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

CINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The CINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

COMPLEX

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
DOUBLE	Set this keyword to return a double-precision complex result. This is equivalent to using the DCOMPLEX function, and is provided as a programming convenience.
Thread Pool Keywords (Windows, UNIX only)	The COMPLEX function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

COMPLEXARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

COND

Item	Description
LNORM	<p>Set this keyword to indicate which norm to use for the computation. The possible values of this keyword are:</p> <ul style="list-style-type: none"> • LNORM = 0 Use the L_∞ norm (the maximum absolute row sum norm). • LNORM = 1 Use the L_1 norm (the maximum absolute column sum norm). • LNORM = 2 Use the L_2 norm (the spectral norm). For LNORM = 2, A cannot be complex. <p>LNORM is set to 0 by default.</p>

CONGRID

Item	Description
CENTER	If this keyword is set, the interpolation is shifted so that points in the input and output arrays are assumed to lie at the midpoint of their coordinates rather than at their lower-left corner.

CONJ

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The CONJ function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

CONVOL

Item	Description
MISSING	The value to return for elements that contain no valid points within the kernel. The default is the IEEE floating-point value NaN. This keyword is only used if the NAN keyword is set.
NAN	Set this keyword to cause the routine to check for occurrences of the IEEE floating-point value NaN in the input data. Elements with the value NaN are treated as missing data, and are ignored when computing the convolution for neighboring elements. In the Result, missing elements are replaced by the convolution of all other valid points within the kernel. If all points within the kernel are missing, then the result at that point is given by the MISSING keyword. Note that CONVOL should never be called without the NAN keyword if the input array may possibly contain NaN values.

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The CONVOL function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

COS

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The COS function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

CW_FIELD

Item	Description
TEXT_FRAME	Set this keyword to the width in pixels of a frame to be drawn around the text field. This keyword is only a "hint" to the toolkit, and may be ignored in some instances. Under Microsoft Windows, text widgets always have a frame of width 1 pixel.

CW_FSLIDER

Item	Description
DOUBLE	Set this keyword to return double-precision values for the GET_VALUE keyword to WIDGET_CONTROL, and for the VALUE field in widget events. If DOUBLE=0 then the GET_VALUE keyword and the VALUE field will return single-precision values. The default is /DOUBLE if one of the MINIMUM, MAXIMUM, or VALUE keywords is double precision, otherwise the default is DOUBLE=0.

Item	Description
SCROLL	Under the Motif window manager, the SCROLL value specifies how many units the scroll bar should move when the user clicks the left mouse button inside the slider area, but not on the slider itself. On Macintosh and Microsoft Windows, the SCROLL value specifies how many units the scroll bar should move when the user clicks the left mouse button on the slider arrows, but not within the slider area or on the slider itself. The default SCROLL value is 1% of the slider width.

CW_PDMENU

Item	Description
CONTEXT_MENU (Windows, UNIX only)	Set this new keyword to create a context menu pulldown. If CONTEXT_MENU is set, Parent must be the widget ID of a context menu base, and the return value of CW_PDMENU is this widget ID. Also see the CONTEXT_MENU keyword to WIDGET_BASE.

DBLARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

DCINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The DCINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

DCOMPLEX

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The DCOMPLEX function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

DCOMPLEXARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

DEVICE

Item	Description
LANGUAGE_LEVEL	Set this keyword to indicate the language level of the PostScript output that is to be generated by the device. Valid values include 1 (the default) and 2 (required for some features, such as filled patterns for polygons).
TRUE_COLOR	You can now use this keyword to specify any TrueColor visual depth. The most common are 15, 16, and 24.

DINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The DINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

DOUBLE

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The DOUBLE function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

EXP

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The EXP function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

EXPINT

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The EXPINT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

FFT

Item	Description
DIMENSION	Set this keyword to the dimension across which to calculate the FFT. If this keyword is not present or is zero, then the FFT is computed across all dimensions of the input array. If this keyword is present, then the FFT is only calculated only across a single dimension. For example, if the dimensions of Array are N1, N2, N3, and DIMENSION is 2, the FFT is calculated only across the second dimension.
Thread Pool Keywords (Windows, UNIX only)	The FFT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

FILE_CHMOD

Item	Description
NOEXPAND_PATH	If specified, FILE_CHMOD uses <i>File</i> exactly as specified, without applying the usual file path expansion.

FILE_DELETE

Item	Description
NOEXPAND_PATH	If specified, FILE_DELETE uses <i>File</i> exactly as specified, without applying the usual file path expansion.

FILE_MKDIR

Item	Description
NOEXPAND_PATH	If specified, FILE_MKDIR uses <i>File</i> exactly as specified, without applying the usual file path expansion.

FILE_TEST

Item	Description
NOEXPAND_PATH	If specified, FILE_TEST uses <i>File</i> exactly as specified, without applying the usual file path expansion.

FINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The FINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

FINITE

Item	Description
SIGN	<p>If the INFINITY or NAN keyword is set, then set this keyword to one of the following values:</p> <ul style="list-style-type: none"> • SIGN > 0: For /INFINITY, return True (1) if <i>X</i> is positive infinity, False (0) otherwise. For /NAN, return True (1) if <i>X</i> is +NaN (negative sign bit is not set), False (0) otherwise. • SIGN = 0 (the default): The sign of <i>X</i> (positive or negative) is ignored. • SIGN < 0: For /INFINITY, return True (1) if <i>X</i> is negative infinity, False (0) otherwise. For /NAN, return True (1) if <i>X</i> is -NaN (negative sign bit is set), False (0) otherwise. <p>If neither the INFINITY nor NAN keyword is set, then this keyword is ignored.</p>

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The FINITE function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

FIX

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The FIX function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

FLOAT

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The FLOAT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

FLOOR

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The FLOOR function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

FLTARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

GAMMA

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The GAMMA function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

GAUSSINIT

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The GAUSSINIT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

GAUSSFIT

Item	Description
ESTIMATES	The way the estimates are constructed in GAUSSFIT if not provided by the user has been improved. If the ESTIMATES array is not specified, estimates are calculated by first subtracting a polynomial of degree NTERMS-4 (only if NTERMS is greater than 3) and then forming a simple estimate of the Gaussian coefficients.

GET_DRIVE_LIST

Item	Description
COUNT	This new keyword is named variable into which the number of drives/volumes found is placed. If no drives/volumes are found, a value of 0 is returned.
CDROM	If set by this new keyword, compact disk drives are reported. Note that although CDROM devices are removable, they are treated as a special case, and the REMOVABLE keyword does not apply to them. Note - This is a Windows only keyword.
FIXED	If set by this new keyword, hard drives physically attached to the current system are reported. Note - This is a Windows only keyword.
REMOTE	This new keyword specifies that remote (i.e. network) drives should be reported. Note - This is a Windows only keyword.
REMOVABLE	This new keyword reports removable media devices (e.g. floppy, zip drive) other than CDROMs. Note - This is a Windows only keyword.

GETENV

Item	Description
Return Value	Returns the equivalence string for <i>Name</i> from the environment of the IDL process, or a null string if <i>Name</i> does not exist in the environment. If <i>Name</i> is an array, the result has the same structure, with each element containing the equivalence string for the corresponding element of <i>Name</i> .
Name	The string variable for which equivalence strings from the environment is desired.

HELP

Item	Description
DEVICE	On UNIX systems, a new field (Bits Per RGB) has been added to the output from the DEVICE keyword. This Bits Per RGB field indicates the amount of bits utilized for each RGB component.

HIST_EQUAL

Item	Description
FCN	Set this keyword to the desired cumulative probability distribution function in the form of a 256 element vector. If omitted, a linear ramp, which yields equal probability bins results. This function is later normalized, so magnitude is inconsequential, though it should increase monotonically.

IMAGINARY

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The IMAGINARY function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

INDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The INDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

INTERPOLATE

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The INTERPOLATE function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ISHFT

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The ISHFT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

L64INDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The L64INDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

LINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The LINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

LNGAMMA

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The LNGAMMA function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

LONARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

LONG

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The ERRORF function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

LONG64

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The LONG64 function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

MAKE_ARRAY

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
DIMENSION	This modified keyword represents a vector of 1 to 8 elements specifying the dimensions of the result. This is equivalent to the array form of the D_i plain arguments.
Thread Pool Keywords (Windows, UNIX only)	The MAKE_ARRAY function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

MATRIX_MULTIPLY

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The MATRIX_MULTIPLY function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

MAX

Item	Description
DIMENSION	Set this new keyword to the dimension over which to find the maximum values for an array. If this keyword is not present or is zero, then the maximum is found over the entire array. If this keyword is present, then the return values for Result, Max_Subscript, MIN, and SUBSCRIPT_MIN will all be arrays of one dimension less than the input array. For example, if the dimensions of Array are N1, N2, N3, and DIMENSION is 2, the dimensions of the result are (N1, N3), and element (i,j) of the result contains the maximum value of Array[i, *, j].

Item	Description
SUBSCRIPT_MIN	A named variable that, if supplied, is converted to an integer containing the one-dimensional subscript of the minimum element, the value of which is available via the MIN keyword.
Thread Pool Keywords (Windows, UNIX only)	The MAX function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

MESH_OBJ

Item	Description
CLOSED	This new keyword, if set, “closes” the polygonal mesh topologically by using the first vertex in a row for both the first and last polygons in that row. This keyword parameter is only applicable to the CYLINDRICAL, SPHERICAL, REVOLUTION, and EXTRUSION surface types. This keyword parameter removes the discontinuity where the mesh wraps back around on itself, which can improve the mesh's appearance when viewing it as a shaded object. For the EXTRUSION surface type, this procedure handles input polygons that form a closed loop with the last vertex being a copy of the first vertex, as well as those that do not.

MESSAGE

Item	Description
BLOCK	If specified, BLOCK supplies the name of the message block to use. The BLOCK keyword is ignored unless the NAME keyword is also specified.
LEVEL	The LEVEL keyword is used to indicate that the name of a routine further up in the current call chain should be used instead.

Item	Description
NAME	If specified, NAME supplies the name of the message to throw. NAME is often used in conjunction with the BLOCK keyword.

MIN

Item	Description
DIMENSION	Set this new keyword to the dimension over which to find the minimum values of an array. If this keyword is not present or is zero, then the minimum is found over the entire array. If this keyword is present, then the return values for Result, Min_Subscript, MAX, and SUBSCRIPT_MAX will all be arrays of one dimension less than the input array. For example, if the dimensions of Array are N1, N2, N3, and DIMENSION is 2, the dimensions of the result are (N1, N3), and element (i,j) of the result contains the minimum value of Array[i, *, j].
SUBSCRIPT_MAX	A named variable that, if supplied, is converted to an integer containing the one-dimensional subscript of the maximum element, the value of which is available via the MAX keyword.
Thread Pool Keywords (Windows, UNIX only)	The MIN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

N_TAGS

Item	Description
DATA_LENGTH	Set this new keyword to return the length of the data fields contained within the structure, in bytes. This differs from LENGTH in that it does not include any alignment padding required by the structure. The length of the data for a given structure will be the same on any system.

NORM

Item	Description
LNORM	<p>Set this keyword to indicate which norm to compute. If A is a vector, then the possible values of this keyword are:</p> <ul style="list-style-type: none"> • LNORM = 0 Compute the L_{∞} norm, defined as $\text{MAX}(\text{ABS}(A))$. • LNORM = 1 Compute the L_1 norm, defined as $\text{TOTAL}(\text{ABS}(A))$. • LNORM = 2 Compute the L_2 norm, defined as $\text{SQRT}(\text{TOTAL}(\text{ABS}(A)^2))$. • LNORM = n Compute the L_n norm, defined as $(\text{TOTAL}(\text{ABS}(A)^n))^{(1/n)}$ where n is any number, float-point or integer. <p>LNORM for vectors is set to 2 by default.</p> <p>If A is a two-dimensional array, then the possible values of this keyword are:</p> <ul style="list-style-type: none"> • LNORM = 0 Compute the L_{∞} norm (the maximum absolute row sum norm), defined as $\text{MAX}(\text{TOTAL}(\text{ABS}(A), 1))$. • LNORM = 1 Compute the L_1 norm (the maximum absolute column sum norm), defined as $\text{MAX}(\text{TOTAL}(\text{ABS}(A), 2))$. • LNORM = 2 Compute the L_2 norm (the spectral norm) defined as the largest singular value, computed from SVDC. For LNORM = 2, A cannot be complex. <p>LNORM for two-dimensional arrays is set to 0 by default.</p>

OBJARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

OPENR, OPENU, OPENW

Item	Description
NOEXPAND_PATH	If specified, <i>File</i> is used exactly as specified, without applying the usual file path expansion.

POLYWARP

Item	Description
DOUBLE	Set this keyword to use double-precision for computations and to return a double-precision result. Set DOUBLE=0 to use single-precision for computations and to return a single-precision result. The default is /DOUBLE if any of the inputs are double precision, otherwise the default is DOUBLE=0.
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). 2 = Warning that a small pivot element was used and that significant accuracy was probably lost. Note - If STATUS is not specified, any warning messages will be output to the screen.

POLY_2D

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The POLY_2D function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

POLY_AREA

Item	Description
DOUBLE	Set this keyword to use double-precision for computations and to return a double-precision result. Set <code>DOUBLE = 0</code> to use single-precision for computations and to return a single-precision result. If either of the inputs are double-precision, the default is <code>/DOUBLE</code> (<code>DOUBLE = 1</code>), otherwise the default is <code>DOUBLE = 0</code> .

PTRARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

QUERY_TIFF

Item	Description
Info	The Info argument to <code>QUERY_TIFF</code> returns an anonymous structure containing information about the image in the file. New Info structure fields have been added. See “New Returned Information for TIFF Queries” on page 45.

RANDOMN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

RANDOMU

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

READ_TIFF

Item	Description
ORIENTATION	<p>Set this keyword to a named variable that will contain the orientation value from the TIFF file. Possible return values are:</p> <ul style="list-style-type: none"> • 1 = Column 0 represents the left-hand side, and row 0 represents the top. • 2 = Column 0 represents the right-hand side, and row 0 represents the top. • 3 = Column 0 represents the right-hand side, and row 0 represents the bottom. • 0 or 4 = Column 0 represents the left-hand side, and row 0 represents the bottom. • 5 = Column 0 represents the top, and row 0 represents the left-hand side. • 6 = Column 0 represents the top, and row 0 represents the right-hand side. • 7 = Column 0 represents the bottom, and row 0 represents the right-hand side. • 8 = Column 0 represents the bottom, and row 0 represents the left-hand side. <p>If an orientation value does not appear in the TIFF file, an orientation of 0 is returned.</p>
Return Value	<p>READ_TIFF now imports 1- and 4-bit images from TIFF files. For 1-bit (bi-level) images, the image values are 0 or 1. For 4-bit grayscale images, the image values are in the range 0 to 15.</p>

REBIN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

RECON3

Item	Description
QUIET	Set this keyword to suppress the output of informational messages when the processing of each image is completed.

REFORM

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

REPLICATE

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The REPLICATE function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

REPLICATE_INPLACE

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The REPLICATE_INPLACE function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ROUND

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The ROUND function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

SETENV

Item	Description
Environment_Expression	This argument may now be either a scalar or array string variable containing environment expressions to be added to the environment.

SHIFT

Item	Description
S_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

SIN

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The SIN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

SINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

SINH

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The SINH function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

SMOOTH

Item	Description
MISSING	The value to return for elements that contain no valid points within the kernel. The default is the IEEE floating-point value NaN. This keyword is only used if the NAN keyword is set.

Item	Description
NAN	<p>Set this keyword to cause the routine to check for occurrences of the IEEE floating-point value NaN in the input data. Elements with the value NaN are treated as missing data, and are ignored when computing the smooth value for neighboring elements. In the Result, missing elements are replaced by the smoothed value of all other valid points within the smoothing window. If all points within the window are missing, then the result at that point is given by the MISSING keyword. Note that SMOOTH should never be called without the NAN keyword if the input array may possibly contain NaN values.</p>
Width	<p>This modified argument defines the width of the smoothing window. Width can either be a scalar or a vector with length equal to the number of dimensions of Array. If Width is a scalar then the same width is applied for each dimension that has length greater than 1 (dimensions of length 1 are skipped). If Width is a vector, then each element of Width is used to specify the smoothing width for each dimension of Array. Values for Width must be smaller than the corresponding Array dimension. If a Width value is even, then Width+1 will be used instead. The value of Width does not affect the running time of SMOOTH to a great extent.</p> <p>Note - A Width value of zero or 1 implies no smoothing. However, if the NAN keyword is set, then any NaN values within the Array will be treated as missing data and will be replaced.</p> <p>Tip - For a multidimensional array, set widths to 1 within the Width vector for dimensions that you don't want smoothed.</p>

SQRT

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The SQRT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

STRARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

TAN

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The TAN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

TANH

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The TANH function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

TOTAL

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The TOTAL function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

TVSCL

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The TVSCL procedure supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

UINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The UINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

UINT

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The UINT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

UINTARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

UL64INDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The UL64INDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ULINDGEN

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The ULINDGEN function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2, “Multi-Threading in IDL” .

ULON64ARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

ULONARR

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.

ULONG

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The UNLONG function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

ULONG64

Item	Description
D_i	This modified argument can now specify dimensions as a single array as well as a sequence of scalar values.
Thread Pool Keywords (Windows, UNIX only)	The ULONG64 function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

VOIGT

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The VOIGT function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

WARP_TRI

Item	Description
TPS	This new keyword uses Thin Plate Spline interpolation which is ideal for modeling functions with complex local distortions, such as warping functions, which are too complex to be fit with polynomials.

WHERE

Item	Description
Thread Pool Keywords (Windows, UNIX only)	The WHERE function supports the new thread pool keywords. For more information, see “Multi-Threading Keywords” on page 119 and Chapter 2 , “Multi-Threading in IDL” .

WIDGET_BASE

Item	Description
CONTEXT_EVENTS (Windows, UNIX only)	Set this new keyword to generate context events when the right mouse button is pressed over the widget. To request right mouse button events in a draw widget use the BUTTON_EVENTS keyword to WIDGET_DRAW at creation or the DRAW_BUTTON_EVENTS keyword to WIDGET_CONTROL for an existing draw widget. A right button press generates a WIDGET_DRAW event with the EVENT.TYPE field equal to 0 and the EVENT.RELEASE field equal to 4.

Item	Description
CONTEXT_MENU (Windows, UNIX only)	Set this new keyword to cause a context menu to be created. The context menu base must be a child of one of the following types of widgets: <ul style="list-style-type: none"> • WIDGET_BASE • WIDGET_DRAW • WIDGET_TEXT • WIDGET_LIST

WIDGET_CONTROL

Item	Description
CONTEXT_EVENTS (Windows, UNIX only)	Set this new keyword to enable context menu events generated by right mouse button clicks. Setting a zero value disables such events. This keyword applies to widgets created with WIDGET_BASE, WIDGET_TEXT, or WIDGET_LIST.

WIDGET_INFO

Item	Description
CONTEXT_EVENTS (Windows, UNIX only)	Set this new keyword to return 1 if Widget_ID is a widget with the CONTEXT_EVENTS attribute set. Otherwise, 0 is returned. This keyword applies to widgets created with WIDGET_BASE, WIDGET_TEXT, or WIDGET_LIST.

Item	Description
SYSTEM_COLORS (Windows, UNIX only)	<p>This new keyword requires a valid IDL widget identifier and returns an IDL structure named WIDGET_SYSTEM_COLORS. The structure contains RGB values for 25 display elements. Each RGB value is a three-dimensional array of integers representing the red, green, blue values in the range 0 to 255 or a value of -1 if unavailable.</p> <p>For more detailed information on the WIDGET_SYSTEM_COLORS structure fields and their meaning see the “Emulating System Colors in Application Widgets” on page 41.</p>

WIDGET_LIST

Item	Description
CONTEXT_EVENTS (Windows, UNIX only)	<p>Set this new keyword to generate context events when the right mouse button is pressed over the widget. To request right mouse button events in a draw widget use the BUTTON_EVENTS keyword to WIDGET_DRAW at creation or the DRAW_BUTTON_EVENTS keyword to WIDGET_CONTROL for an existing draw widget. A right button press generates a WIDGET_DRAW event with the EVENT.TYPE field equal to 0 and the EVENT.RELEASE field equal to 4.</p>

WIDGET_SLIDER

Item	Description
SCROLL	Under the Motif window manager, the SCROLL value specifies how many units the scroll bar should move when the user clicks the left mouse button inside the slider area, but not on the slider itself. The default on Motif is 10% of the slider width. On Macintosh and Microsoft Windows, the SCROLL value specifies how many units the scroll bar should move when the user clicks the left mouse button on the slider arrows, but not within the slider area or on the slider itself. The default on Macintosh and Microsoft Windows is 1 unit.

WIDGET_TEXT

Item	Description
CONTEXT_EVENTS (Windows, UNIX only)	Set this new keyword to generate context events when the right mouse button is pressed over the widget. To request right mouse button events in a draw widget use the BUTTON_EVENTS keyword to WIDGET_DRAW at creation or the DRAW_BUTTON_EVENTS keyword to WIDGET_CONTROL for an existing draw widget. A right button press generates a WIDGET_DRAW event with the EVENT.TYPE field equal to 0 and the EVENT.RELEASE field equal to 4.

WRITE_TIFF

Item	Description
BITS_PER_SAMPLE	This new keyword can be used for a grayscale image, by being set to either 1, 4, or 8 to indicate the bits per sample to write. For 1-bit (bi-level) images, an output bit is assigned the value 1 if the corresponding input pixel is nonzero. For 4-bit grayscale images, the input pixel values should be in the range 0 through 15. The default is BITS_PER_SAMPLE = 8. This keyword is ignored if an RGB image or color palette is present, or if one of the FLOAT, LONG, or SHORT keywords is set.

Item	Description
ORIENTATION	<p>Set this new keyword to indicate the orientation of the image with respect to the columns and rows of Image. Possible values are:</p> <ul style="list-style-type: none"> • 1 = Column 0 represents the left-hand side, and row 0 represents the top. • 2 = Column 0 represents the right-hand side, and row 0 represents the top. • 3 = Column 0 represents the right-hand side, and row 0 represents the bottom. • 0 or 4 = Column 0 represents the left-hand side, and row 0 represents the bottom. • 5 = Column 0 represents the top, and row 0 represents the left-hand side. • 6 = Column 0 represents the top, and row 0 represents the right-hand side. • 7 = Column 0 represents the bottom, and row 0 represents the right-hand side. • 8 = Column 0 represents the bottom, and row 0 represents the left-hand side. <p>The default is ORIENTATION=1.</p> <p>Warning - Not all TIFF readers honor the value of the ORIENTATION field. IDL writes the value into the file, but many known readers ignore this value. In such cases, it is recommended that the image be converted to top to bottom order with the REVERSE function and then ORIENTATION be set to 1.</p>
UNITS	<p>Set this new keyword to indicate the units of the XRESOL and YRESOL keywords. Possible values are:</p> <ul style="list-style-type: none"> • 1 = No units • 2 = Inches (the default) • 3 = Centimeters

Item	Description
XRESOL	This existing keyword sets the horizontal resolution. Units may now be set for XRESOL using the UNITS keyword.
YRESOL	This existing keyword sets vertical resolution. Units may now be specified for YRESOL using the UNITS keyword.

Multi-Threading Keywords

These keywords can be used to modify IDL's use of the IDL Thread Pool to perform calculations. See [Chapter 2, “Multi-Threading in IDL”](#) for a complete listing of the operators and routines which support multi-threading in this release.

Keyword	Description
TPOOL_MAX_ELTS (Windows, UNIX only)	<p>Use this keyword to override the default and use a different upper limit for a given computation call without altering the !CPU system variable.</p> <p>If !CPU.TPOOL_MAX_ELTS is non-zero, IDL will use the single threaded version of any routine with more than !CPU.TPOOL_MAX_ELTS elements to avoid situations where use of the thread pool can be slower than the single threaded case because the threads end up fighting each other for access to system memory.</p>
TPOOL_MIN_ELTS (Windows, UNIX only)	<p>Use this keyword to override the default and use a different lower limit for a given computation call without altering the !CPU system variable.</p> <p>Use of the thread pool requires some overhead. If a given computation does not involve enough data points to make it worthwhile, the threaded version of a routine can be slower than the non-threaded version. To avoid this pitfall, IDL does not use the thread pool for computations involving fewer than !CPU.TPOOL_MIN_ELTS elements.</p>

Keyword	Description
TPOOL_NOTHREAD (Windows, UNIX only)	If TPOOL_NOTHREAD is set, the routine will not use the thread pool, and instead uses the non-threaded implementation of the routine. Normally, IDL decides whether to use the thread pool for a given computation based on the current setting of the !CPU system variable.

Updates to Executive Commands

The following list of executive commands have been updated as indicated.

Executive Command	Update Description
.SKIP	<p>The .SKIP command skips one or more statements and stops. It is useful for moving past a program statement that caused an error. If the optional argument <i>n</i> is present, it gives the number of statements to skip; otherwise, a single statement is skipped.</p> <p>Note - .SKIP does not skip into a called routine.</p>

New and Updated System Variables

The following system variables have been added or updated in IDL 5.5:

System Variable	Description
!CPU	Supplies information about the state of the system processor, and of IDL's use of it. !CPU is read-only, and cannot be modified directly.
!ERROR_STATE	<p>A new field has been added to the returned structure called <code>SYS_CODE_TYPE</code>. The new field follows the <code>SYS_CODE</code> field and comes before the <code>MSG</code> field.</p> <p>The <code>SYS_CODE_TYPE</code> field is a string describing the type of system code contained in the <code>SYS_CODE</code> field. Possible values are:</p> <ul style="list-style-type: none"> • <code>errno</code> — Unix/Posix system error. • <code>win32</code> — Microsoft Windows Win32 system error. • <code>winsock</code> — Microsoft Windows sockets library error. • <code>macos</code> — Macintosh system error. <p>A null string in this field indicates that there is no system code corresponding to the current error.</p>
!VERSION	This variable has been changed by the addition of an <code>OS_NAME</code> field.
!WARN	The !WARN system variable no longer contains the <code>TRUNCATED_FILENAME</code> field.

Features Obsoleted

Obsoleted Routines

The following routines were present in IDL Version 5.4 but became obsolete in IDL Version 5.5. 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
ERRORF	ERF

Note

ERF and ERFC are the standard mathematical names for the error function and complimentary error function. However, because of their short length, users should be aware of conflicts with their existing code which might have variables or functions named *erf* or *erfc*. Existing uses of the name *erf* or *erfc* should be replaced.

Obsoleted Keywords and Arguments

The following keywords and arguments became obsolete in IDL Version 5.5. 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	Item	Description
WRITE_TIFF	Order	The Order argument is obsolete, and has been replaced by the ORIENTATION keyword. Code that uses the Order argument will continue to work as before, but new code should use the ORIENTATION keyword instead.

Routine	Item	Description
READ_TIFF	ORDER	The ORDER keyword is obsolete, and has been replaced by the ORIENTATION keyword. Code that uses the ORDER keyword will continue to work as before, but new code should use the ORIENTATION keyword instead.

Platforms Supported in this Release

IDL 5.5 supports the following platforms and operating systems:

Platform	Vendor	Hardware	Operating System	Supported Versions
UNIX†	Compaq	Alpha	Tru64 UNIX	5.1
	Compaq	Alpha	Linux	Red Hat 6.2††
	HP	PA-RISC	HP-UX	11.0
	IBM	RS/6000	AIX	4.3
	Intel	Intel x86	Linux	Red Hat 6.0, 7.1††
	SGI	Mips	IRIX	6.5.1
	SUN	SPARC	Solaris	8
	SUN	SPARC (64-bit Ultra)	Solaris	8
	SUN	Intel x86	Solaris	8
Windows	Microsoft	Intel x86	Windows	98, NT 4.0, 2000
Macintosh	Apple	PowerMAC†††	MacOS	8.6, 9.x

Table 1-7: Platforms Supported in IDL 5.5

† For UNIX, the supported versions indicate that IDL was either built on (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 5.5 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: Multi-Threading in IDL

This chapter describes the implementation of the IDL Thread Pool and how it can be used to accelerate your computations.

The IDL Thread Pool	126	Routines Supporting the Thread Pool . . .	134
Controlling the Thread Pool in IDL	128		

The IDL Thread Pool

Multi-threading can be used to increase the speed of numeric computations by using multiple system processors to simultaneously carry out different parts of the computation. IDL uses a *thread pool*, a pool of multiple computation threads that are used as helpers to accelerate numerical computations, for this purpose. The implementation of the IDL thread pool allows IDL to automatically determine whether a specified computation can be accomplished using parallel processing to save time.

IDL automatically evaluates all computations to determine whether or not to use the thread pool to carry them out. This decision is based on attributes such as the number of data elements involved, the availability of multiple CPUs in the current system, and the applicability of the thread pool to the specific computation. The IDL user has the ability to alter the parameters used by IDL to make this decision, either on a global basis for the duration of the IDL session, or for an individual computation.

IDL supports the use of the thread pool on all platforms except AIX and Macintosh.

Benefits of the IDL Thread Pool

The IDL thread pool will increase processing performance on certain computations. When not involved in a calculation, the threads in the thread pool are inactive and consume little in the way of system resources. When IDL reaches a computation that can use the thread pool and which would benefit from parallel execution, it divides the task into sub-parts for each thread, enables the thread pool to do the computation, waits until the thread pool completes, and then continues. Other than the improved performance, the end result is virtually indistinguishable when compared to the same computation performed in the standard single-threaded manner.

Possible Drawbacks to the Use of the IDL Thread Pool

There are instances when allowing IDL to use its default thread pool settings can produce results which are less than optimal. For instance, the thread pool can actually take longer to complete a given job, or cause other undesirable effects if used in inappropriate situations.

The following situations describe when it is better to override the initial thread pool settings:

- **Computation of a relatively small number of data elements.** The IDL thread pool requires a small fixed overhead when compared to a non-threaded version of the same computation. Normally, computational speed efficiency is

achieved when the multiple CPUs work in parallel and the speed-up is much larger than the overhead required to use them. However, if the computation does not include enough data elements (each element being a data value of a particular data type), the overhead exceeds the benefit and the overall computation speed can be slower.

- **Large computation that requires virtual memory use.** If the desired computation is too large to fit into physical memory, the threads in the thread pool may cause page faults which will activate the virtual memory system. If more than one thread encounters this situation simultaneously, the threads will compete with each other for access to memory and performance will fall below that of a single-threaded approach to the computation.
- **Multiple users on a shared system competing for CPU use.** On a large multi-user system, an IDL application that uses the thread pool may consume all available CPUs, thus affecting other users of the system by reducing overall performance.
- **Sensitivity to numerical precision.** Algorithms that are sensitive to the order of operations may produce different results when performed by the thread pool. Such results are due to the use of finite precision floating point types, and are equally correct within the precision of the data type.

Controlling the Thread Pool in IDL

IDL allows you to programmatically control the use of thread pool. This section discusses the following aspects of thread pool use:

- [Using the Initial Settings of the Thread Pool](#)
- [Programatically Controlling the Settings of the Thread Pool](#)
- [Disabling the Thread Pool](#)

Note

For a list of the types of computations that support the thread pool, see [“Routines Supporting the Thread Pool”](#) on page 134.

Using the Initial Settings of the Thread Pool

The current values of the parameters that determine IDL’s use of the thread pool for computations are always available in the `!CPU` system variable. `!CPU` is initialized by IDL at startup with default values for the number of CPUs (threads) to use, as well as the minimum and maximum number of data elements. If you have more than one processor on your system, if your desired computation is able to use the thread pool, and if the number of data elements in your computation falls into the allowed range (neither too few, nor too many), then IDL will employ the thread pool in that calculation.

If the number of data elements is too low (is below the minimum allowed number), the overhead associated with the use of the thread pool will exceed the potential performance gain. If the number of data elements is too high (exceeds the maximum number of data elements), you may not have enough available memory on your system, requiring the use of virtual memory which degrades performance. For these reasons, IDL will not use the thread pool for computations that fall outside the specified number of elements.

Programmatically Controlling the Settings of the Thread Pool

There are two ways to control the settings for the thread pool in IDL:

- Use the `CPU` procedure to alter the global thread pool settings for a session or group of computations.
- Use the thread pool keywords supported by individual IDL routines to override the current global thread pool settings for the duration of that single call.

Controlling the Thread Pool Settings for a Session or Group of Computations

The global parameters that control IDL's use of the thread pool are always visible in the !CPU system variable. IDL initializes the defaults for these values at startup. The CPU procedure is used to modify these parameters to better fit individual needs. This procedure allows you to specify:

- The minimum number of data elements required before IDL will use the thread pool.
- The maximum number of data elements for which IDL will use the thread pool.
- The number of threads to use (Note that specifying the use of 1 thread disables the use of the thread pool).

For more information on the CPU procedure, see “CPU” on page 194.

The !CPU system variable supplies information about your system, including the current global thread pool parameters. !CPU is read-only, and cannot be modified directly. The CPU procedure is used to change the values in !CPU. The fields of !CPU are shown in the following table:

Field	Description
HW_VECTOR	Information on whether or not the system supports a vector unit (e.g. Macintosh Altivec/Velocity Engine). Possible values are: <ul style="list-style-type: none"> • 1 — True, the system supports a vector unit • 0 — False Note - This value is currently always 0 (False) on platforms other than Macintosh.
VECTOR_ENABLE	Information about whether or not the use of a vector unit is enabled in IDL. Possible values are: <ul style="list-style-type: none"> • 1 — True (IDL will use a vector unit, if such a unit is available on the current system) • 0 — False Note - This value is currently always 0 (False) on platforms other than Macintosh.

Table 2-1: Fields of the !CPU System Variable Structure

Field	Description
HW_NCPU	The number of CPUs on the system IDL is currently running on.
TPOOL_NTHREADS	The number of threads that IDL will use in thread pool computations. The initial value is equal to the value contained in HW_NCPU, so that each thread will have the potential to run in parallel with the others. For numerical computation, there is no benefit to using more threads than your system has CPUs. However, depending on the size of the problem and the number of other programs running on the system, there may be a performance advantage to using fewer CPUs.
TPOOL_MIN_ELTS	The number of data elements in a computation that are necessary before IDL will use the thread pool. If the number of elements is less than TPOOL_MIN_ELTS, IDL will perform the computation without using the thread pool. Use this parameter to prevent IDL from using the thread pool on tasks that are too small to benefit from it.
TPOOL_MAX_ELTS	If non-zero, the maximum number of elements in a computation that will be processed using the thread pool. Computations with more than this number of elements will not use the thread pool. Setting TPOOL_MAX_ELTS to 0 (the default) means that no limit is imposed and any computation with at least TPOOL_MIN_ELTS can use the thread pool. Set this parameter if large jobs are causing virtual memory paging on your system.

Table 2-1: Fields of the !CPU System Variable Structure

Note

The following examples will only work on systems with more than one processor. Do not try these examples on a single processor system.

As a first example, imagine that we want to make sure that the thread pool is not used unless there are at least 50,000 data elements and no more than 1,000,000. We set the minimum to 50,000 since we know, for our particular system, that at least 50,000

floating point data elements are required before the use of the thread pool will exceed the overhead required to use it. We set the maximum to 1,000,000 since we know that 1,000,000 floating point data elements will exceed the maximum amount of memory we want to use for this computation.

```
; Modify the thread pool settings
CPU, TPOOL_MAX_ELTS = 1000000, TPOOL_MIN_ELTS = 50000

; Create 65,341 elements of floating point data
theta = FINDGEN(361, 181)

; Perform computation
sineSquared = 1. - (COS(!DTOR*theta))^2
```

In this example, the thread pool will be used since we are performing a computation on an array of 65,341 data elements which falls between the minimum and maximum thresholds. Note that we altered the global thread pool parameters to achieve this. An alternative approach that does not change the global defaults is shown in [“Controlling the Thread Pool Settings for a Specific Computation”](#) on page 132.

In the next example, we will:

- Save the current thread pool settings from the !CPU system environment variable.
- Modify the thread pool settings so that IDL is configured, for our particular system, to efficiently perform a floating point computation.
- Perform a floating point computation.
- Modify the thread pool settings so that IDL is configured, for our particular system, to efficiently perform a double precision computation.
- Perform a double precision computation.
- Restore the thread pool settings to their original values.

The first computation will use the thread pool since it does not exceed any of the specified parameters. The second computation, since it exceeds the maximum number of data elements, will not use the thread pool:

```
; Retrieve the current thread pool settings
threadpool = !CPU

; Modify the thread pool settings
CPU, TPOOL_MAX_ELTS = 1000000, TPOOL_MIN_ELTS = 50000, $
    TPOOL_NTHREADS = 2

; Create 65,341 elements of floating point data
theta = FINDGEN(361, 181)
```

```

; Perform computation, using 2 threads
sineSquared = 1. - (COS(!DTOR*theta))^2

; Modify thread pool settings for new data type
CPU, TPOOL_MAX_ELTS = 50000, TPOOL_MIN_ELTS = 10000

; Create 65,341 elements of double precision data
theta = DINDGEN(361, 181)

; Perform computation
sineSquared = 1. - (COS(!DTOR*theta))^2

;Return thread pool settings to their initial values
CPU, TPOOL_MAX_ELTS = threadpool.TPOOL_MAX_ELTS, $
    TPOOL_MIN_ELTS = threadpool.TPOOL_MIN_ELTS, $
    TPOOL_NTHREADS = threadpool.HW_NCPU

```

Controlling the Thread Pool Settings for a Specific Computation

All routines that support the thread pool accept the following three keywords that allow you to override the thread pool settings for the duration of a single call. This allows you to modify the settings for a particular computation without affecting the global default settings of your session.

The three thread pool keywords are described in the following table.

Keyword	Description
TPOOL_MAX_ELTS	<p>If non-zero, this keyword sets the maximum number of data elements for a given computation. If the number of elements you specify is exceeded, IDL will not use the thread pool for this computation. Setting this value to 0 removes any limit on maximum number of elements, and any computation with at least TPOOL_MIN_ELTS will use the thread pool.</p> <p>This keyword overrides the default value, which is given by !CPU.TPOOL_MAX_ELTS.</p>

Table 2-2: The Thread Pool Keywords

Keyword	Description
TPOOL_MIN_ELTS	This keyword sets the minimum number of data elements for a given computation. If the number of elements is less than what you specified, IDL will not use the thread pool for this computation. Use this keyword to prevent IDL from using the thread pool on tasks that are too small to benefit from it. This keyword overrides the default value, which is given by !CPU.TPOOL_MIN_ELTS.
TPOOL_NOTHREAD	If set, the computation will not use the thread pool.

Table 2-2: The Thread Pool Keywords

We can use the TPOOL_MIN_ELTS and TPOOL_MAX_ELTS keywords to modify the example used in the previous section so that it does not alter the global default thread pool settings:

```
; Create 65,341 elements of floating point data
theta = FINDGEN(361, 181)

; Perform computation and override session settings for maximum
; and minimum number of elements
sineSquared = 1. - (COS(!DTOR*theta, TPOOL_MAX_ELTS = 1000000, $
    TPOOL_MIN_ELTS = 50000))^2
```

Disabling the Thread Pool

There are two ways to disable the thread pool in IDL:

- Employ the CPU procedure to alter the default global thread pool parameters, either for an entire IDL session, or just for a related group of computations.
- Use the thread pool keywords to a routine to disable the thread pool for a specific single computation.

In the first example, we will disable the thread pool for the session by setting the number of threads to use to 1:

```
CPU, TPOOL_NTHREADS = 1
```

In the next example, we will disable the thread pool for a specific computation using the TPOOL_NOTHREAD keyword:

```
sineSquared = 1. - (COS(!DTOR*theta, /TPOOL_NOTHREAD))^2
```

Routines Supporting the Thread Pool

The operators and routines currently supporting the thread pool in IDL are listed in the section that follows, grouped per the functional category (as listed in the *IDL Quick Reference*) to which the routines belong.

Binary and Unary Operators:

- | | |
|-------|-------|
| • − | • + |
| • NOT | • AND |
| • / | • * |
| • EQ | • NE |
| • GE | • LE |
| • GT | • LT |
| • > | • < |
| • OR | • XOR |
| • ^ | • MOD |
| • # | • ## |

Mathematical Routines:

- | | | |
|----------|-------------|-------------------|
| • ABS | • ERRORF | • MATRIX_MULTIPLY |
| • ACOS | • EXP | • ROUND |
| • ALOG | • EXPINT | • SIN |
| • ALOG10 | • FINITE | • SINH |
| • ASIN | • FLOOR | • SQRT |
| • ATAN | • GAMMA | • TAN |
| • CEIL | • GAUSSINT | • TANH |
| • CONJ | • IMAGINARY | • VOIGT |
| • COS | • ISHFT | |
| • COSH | • LNGAMMA | |

Image Processing Routines:

- BYTSCL
- CONVOL
- FFT
- INTERPOLATE
- POLY_2D
- TVSCL

Array Creation Routines:

- BINDGEN
- BYTARR
- CINDGEN
- DCINDGEN
- DCOMPLEXARR
- DINDGEN
- FINDGEN
- INDGEN
- LINDGEN
- L64INDGEN
- MAKE_ARRAY
- REPLICATE
- UINDGEN
- ULINDGEN
- UL64INDGEN

Non-string Data Type Conversion Routines:

- BYTE
- COMPLEX
- DCOMPLEX
- DOUBLE
- FIX
- FLOAT
- LONG
- LONG64
- UINT
- ULONG
- ULONG64

Array Manipulation Routines:

- MAX
- MIN
- REPLICATE_INPLACE
- TOTAL
- WHERE

Programming and IDL Control Routines:

- BYTEORDER



Chapter 3: Using COM Objects in IDL

This chapter describes the following topics:

Introduction to IDL COM Objects	138	Using IDL IDispatch COM Objects	142
Skills Required to Use COM Objects	139	Using ActiveX Controls in IDL	149
IDL COM Naming Schemes	140		

Introduction to IDL COM Objects

COM (Component Object Model) objects are a specification and implementation for building software components that may be used to build programs or to add functionality to existing programs running on the Windows platform. COM components are written in a variety of programming languages (although most are written in C++) and are able to be utilized in a program at run time without having to recompile the program. In IDL, COM objects, regardless of type or method of creation, are treated as IDL objects. In order to call methods associated with a COM object, a user employs the arrow operator \rightarrow , just as would be done when calling any other object method in IDL. IDL will then internally recognize this COM-based object and will route the method call to the internal COM subsystem for dispatching.

When adding COM functionality in IDL, an IDispatch interface must be exposed on all COM objects accessed by IDL since this interface is used by IDL to call methods on each COM object. Although this may seem to be a limitation, it is a minimal one since it is common to scriptable objects, for which the interface is designed.

There are two main uses for COM functionality in IDL:

- Using the IDLcomIDispatch object to instantiate a desired COM object by using a provided class or program ID. This method is ideal for COM objects that do not utilize a graphical-user interface.
- Using the WIDGET_ACTIVEX function to embed an ActiveX control in an IDL widget hierarchy.

The primary differences in IDL between using IDLcomIDispatch-based objects and using an ActiveX control are the methods by which they are created and managed. These methods of creation and management are detailed in this chapter.

Note

IDL COM functionality is not accessible when in IDL demonstration mode.

Skills Required to Use COM Objects

Although IDL provides an abstracted interface to COM functionality, some knowledge of COM is required to use the functionality. There is a large difference between the level at which a typical user sees IDL compared to that of the internal programmer. To the user, IDL is an easy-to-use, array-oriented language that combines numerical and graphical abilities, and runs on many platforms. Internally, IDL is a large application that includes a compiler, an interpreter, graphics, mathematical computation, user interface, and a large amount of operating system-dependent code.

The amount of knowledge required to effectively write internal code for IDL can come as a surprise to the user who is only familiar with IDL's external face. To be successful, the programmer must have internal programming experience and proficiency.

ActiveX

To use the IDL ActiveX control, a level of understanding of ActiveX and COM is necessary. Although IDL provides an abstracted interface to COM functionality, some knowledge of COM is required to use the functionality.

IDL COM Naming Schemes

IDL uses the identifier for the underlying COM object to construct the IDL class name. This then ensures each particular type of COM object has a unique IDL class type. Since two types of class identifiers exist in COM (class ID and program ID) these must also be indicated during this class construction process. With this in mind the following naming scheme was devised:

<Base Class Name>\$<ID Type>\$<ID>

For IDispatch based objects, the class name takes the following form:

Using a COM Class ID

IDLcomIDispatch\$CLSID\$<the Class ID>

Using a COM Program ID

IDLcomIDispatch\$PROGID\$<the Program ID>

Note

All IDispatch-based objects created in IDL sub-class from the intrinsic IDL class IDLcomIDispatch.

For ActiveX based objects, the class name takes one of the following forms:

- Using a COM Class ID

IDLcomActiveX\$CLSID\$<the Class ID>

- Using a COM Program ID

IDLcomActiveX\$PROGID\$<the Program ID>

Note

All ActiveX based objects created in IDL sub-class from the intrinsic IDL class IDLcomActiveX, which is a sub-class from IDLcomIDispatch.

It should be noted that the COM Class ID separator (-) or the Program ID separator (.) should be indicated using an underscore (_) when constructing the class name for the particular object name in IDL.

About Obtaining COM Class Identifiers

The COM system depends on COM class identifiers and program identifiers to instantiate or reference a particular control. These are often obtained from the control

being used or documentation provided by a given control. This information can be difficult to obtain, but Microsoft provides a tool to determine the controls available on a particular computer and to retrieve the Class Identifier for that particular control, object, or type library. This downloadable tool (which has also been known also as the OLE/COM Object Viewer) can be found at:

<http://www.microsoft.com/com>

Using IDL IDispatch COM Objects

Creation, management, and destruction of IDispatch-based COM objects that are not being placed in an IDL Widget GUI are carried-out using standard IDL object-management routines.

You can create an IDispatch COM object by using the OBJ_NEW() function. Either the class identifier or the program identifier is provided to indicate which object will be created. For information on creating an IDispatch COM object, see “[IDispatch Object Creation](#)” on page 143.

Once creation is complete, the object is then usable and may be manipulated like any other IDL object. Method calls are identical to any other IDL object. For information on dispatching methods for an IDispatch COM object, see “[IDispatch Method Dispatching](#)” on page 143 and for information on IDispatch COM Object Property Management see “[IDispatch Property Management](#)” on page 144.

You can destroy the object by using the OBJ_DESTROY procedure. This will release the internal COM object and free any resources associated with it. For information on destroying an IDispatch COM object, see “[IDispatch COM Object Destruction](#)” on page 144.

IDL IDispatch Naming Schemes

IDL uses the identifier for the underlying COM object to construct the IDL class name. This ensures each particular type of COM object has a unique IDL class type. Since two types of class identifiers exist in COM, those must also be included during this class construction process. With this in mind, the following naming scheme is used:

<Base Class Name>\$<ID Type>\$<ID>

For IDispatch-based objects, the class name takes the following form:

- Using a COM Class ID:

IDLcomIDispatch\$CLSID\$<the Class ID>

- Using a COM Program ID:

IDLcomIDispatch\$PROGID\$<the Program ID>

Note

All IDispatch-based objects created in IDL subclass from the intrinsic IDL class IDLcomIDispatch.

Note

The COM Class ID separator (-) or the Program ID separator (.) should be indicated using an underscore (_) when constructing the class name for the particular object name in IDL.

Note

The curly braces ({ }) for COM Class IDs should not be included in the name of the object. They are invalid characters for IDL Class names.

IDispatch Object Creation

When working with IDispatch COM objects in IDL, it is first necessary to learn the method used to create an IDL object which represents a COM object which in turn implements the IDispatch interface.

As with any IDL object, an IDispatch COM object is created using the intrinsic IDL function, OBJ_NEW(). Using the provided class or program ID, the underlying implementation then employs the internal IDL COM sub-system to instantiate the desired COM object.

Note

OBJ_NEW should only be used to create non-ActiveX COM objects. WIDGET_ACTIVEX is the only method used to create an IDL object that represents an ActiveX control. Creating an ActiveX control (an object based off the class name prefix IDLcomActiveX\$) using OBJ_NEW() is not supported and the results are undefined.

IDispatch Method Dispatching

The → operator is used to invoke an IDispatch method as it is with other IDL object methods. The general pattern is:

```
IDLcomIDispatch-><MethodName>
```

There is no distinction between a procedure or a function in COM, so only the IDL procedure interface is supported when calling IDispatch methods.

When a method is called on a COM-based IDL object, the method name and arguments are passed to the internal IDL COM subsystem and are used to construct the equivalent pair of calls IDispatch → GetIDsOfNames() and IDispatch → Invoke() on the underlying COM object.

Note

Aside from other COM-based objects, no complex types are supported as parameters to procedure calls. This is due to the limitations imposed by the internal data representations used in COM (VARIANTS).

Note

IDL objects use method names to identify and call object life cycle methods (INIT and CLEANUP). As such, these method names should be considered reserved. If an underlying ActiveX or IDispatch object implements a method using either INIT or CLEANUP, those methods will be overridden by the IDL life cycle methods and will not be accessible from IDL. Also, these ActiveX or IDispatch object cannot have their own GetProperty or SetProperty method, since IDL uses these methods to manage properties.

IDispatch COM Object Destruction

The OBJ_DESTROY procedure is used to destroy an IDispatch COM object.

When OBJ_DESTROY is called with a COM-based object as an argument, the underlying reference to the COM object is released and IDL resources relating to that object are freed.

Destruction of the IDL object does not automatically cause the destruction of the underlying COM object. Due to the method by which COM objects are implemented, object destruction is left to the component itself. A reference-counting methodology is used in COM. Therefore, when the IDL COM object is destroyed, IDL will decrement the reference count on the underlying object. It is then left to the underlying object to determine when to destroy itself based on other outstanding reference counts.

IDispatch Property Management

The ability to set and get properties is also provided by the IDispatch interface. In order to do these tasks, the following methods are defined:

IDLcomIDispatch -> GetProperty, <PROPERTY_NAME> = Value, [arg0, arg1, ...]

IDLcomIDispatch -> SetProperty, <PROPERTY_NAME> = Value

As is the convention with other IDL objects, IDispatch property names are mapped to IDL keywords and the underlying property values are treated as IDL keyword values.

It is also important to realize that the provided keywords must map directly to a property name or an error will be shown. Any keyword that is passed into either of

the property routines is assumed to be a fully-qualified IDispatch property name. As such, the partial keyword name functionality provided by IDL is not valid with IDL COM-based objects.

Some *gettable* properties also require input parameters. Therefore, the `GetProperty` method can take parameters. If parameters are provided, only one property (keyword) can be provided.

COM Objects Returning IDispatch Pointers to Other Objects

It is not uncommon for COM objects to return references to other COM objects. This is done either through accessing a property or a method call. If an `IDLcomIDispatch` object returns a reference to another COM object's IDispatch interface, then the IDL COM subsystem automatically converts the returned IDispatch pointer into an `IDLcomIDispatch` object for immediate use. For example:

`obj1 → GetOtherObject, obj2`

`obj2 → DoSomeMethod`

The `GetOtherObject()` method for `obj1` returns a reference to the IDispatch interface for `obj2`. The IDL COM subsystem takes the IDispatch reference and creates an `IDLcomIDispatch` object for `obj2`.

Note

If an IDispatch reference is returned and an `IDLcomIDispatch` object is automatically created, the newly created object must be explicitly destroyed by calling `OBJ_DESTROY`. For example, after using `obj2` from the above example, it must be destroyed by calling:

`OBJ_DESTROY, obj2`

Example: Creating an IDispatch COM Object in IDL

In this example, an IDispatch COM object is used in IDL.

All COM components and ActiveX controls must be registered on a machine before they can be used by any client. A component (`.dll` or `.exe`) or a control (`.ocx`) can be registered using the command line program `regsvr32`, supplying it with name of the component or control to register.

For example, if you had a COM component named `RSIDemoComponent` and it was contained in a file named `rsidemo.dll`. To install it in a directory called

C:\IDL_DIR\Demo and then use it, you would first need to register this component by performing either of the following actions:

- Open a command prompt window and type in the following:

```
regsvr32 'c:\idl_dir\demo\rsidemo.dll'
```

- Similarly, you could open a command prompt window, change directories to C:\idl_dir, then just say:

```
regsvr32 rsidemo.dll
```

Note

The “/s” parameter means to be silent during the registration. If the “/s” is not specified, then a pop-up dialog is presented saying the component was registered correctly.

Now, an object called RSIDemoObj1 could be created. This object could be created using either the Program ID or the Class ID. However, if the Class ID is used, the hyphens (-) must be replaced with underscores (_) since hyphens are not valid symbols for IDL identifiers.

1. The procedure would begin by creating an object from that component.

```
pro IDispatchDemo
```

```
obj1 = $
```

```
OBJ_NEW( 'IDLCOMIDispatch$PROGID$RSIDemoComponent.RSIDemoObj1' )
```

or (with Class ID):

```
obj1 = OBJ_NEW( $
```

```
'IDLCOMIDispatch$CLSID$A77BC2B2_88EC_4D2A_B2B3_F556ACB52E52' )
```

2. Next, the following line of code would be added to call the GetCLSID method, which returns the Class ID for the component. (This should be: '{A77BC2B2-88EC-4D2A-B2B3-F556ACB52E52}')

```
obj1 -> GetCLSID, strCLSID
```

```
PRINT, strCLSID
```

Note

The GetCLSID returns the class identifier of the object using the standard COM separators (-).

3. Next, to get the current value of the MessageStr property, you would enter:

```
obj1 -> GetProperty, MessageStr = outStr
```

```
PRINT, outStr
```

4. You could also set the `MessageStr` property of the object and display it:

```
obj1 -> SetProperty, MessageStr = 'Hello, world'
obj1 -> DisplayMessageStr
```

5. The `Msg2InParams` method can be used to take two input parameters and concatenates them into the resultant string (the Output string should be: String part of input25):

```
instr = 'String part of input'
val = 25L
obj1 -> Msg2InParams, instr, val, outStr
PRINT, outStr
```

6. The `GetIndexObject()` method may return an object reference to three possible objects (If the index is not 1, 2, or 3, it will return an error).

The three possible objects are:

- `RSIDemoObj1`, WHERE index = 1
- `RSIDemoObj2`, WHERE index = 2
- `RSIDemoObj3`, WHERE index = 3

7. You could get a reference to one of these objects, `RSIDemoObj3` for example:

```
obj1 -> GetIndexObject, 3, obj3
```

8. Since all three objects have the '`GetCLSID`' method, they could now be used to verify that the desired object was returned (The output should be: {13AB135D-A361-4A14-B165-785B03AB5023}):

```
obj3 -> GetCLSID, obj3CLSID
PRINT, obj3CLSID
```

9. Always destroy a retrieved object when you are finished with it:

```
OBJ_DESTROY, obj3
```

10. Next, the `GetArrayOfObjects()` method could be used to return a vector of object references to `RSIDemoObj1`, `RSIDemoObj2`, `RSIDemoObj3`, respectively (The number of elements in the vector is returned in the first parameter and should be 3):

```
obj1 -> GetArrayOfObjects, cItems, objs
PRINT, cItems
```

11. Since each object implements the 'GetCLSID()' method, you could loop through all the object references and get its class ID:

```
FOR i = 0, cItems-1 do begin
    objs[i] -> GetCLSID, objCLSID
    PRINT, 'Object[' ,i,'] CLSID: ', objCLSID
ENDFOR
```

12. Always destroy object references when you are finished with them, and end the procedure:

```
OBJ_DESTROY, objs
OBJ_DESTROY, obj1
END
```

Using ActiveX Controls in IDL

The instantiation of an ActiveX control in IDL is very different than typical IDispatch-based object instantiation. This is because ActiveX controls must be placed in an IDL Widget hierarchy. In addition, events generated by the ActiveX control are carried over into the IDL event model. Aside from these important differences, the user then calls methods as they would with any other IDL object.

Note

IDL ActiveX control creation is available on the Windows NT/Windows 2000 platforms only.

ActiveX-based COM Naming Schemes

IDL uses the identifier for the underlying COM object to construct the IDL class name. This ensures each particular type of COM object has a unique IDL class type. Since two types of class identifiers exist in COM, those must also be indented during this class construction process. With this in mind, the following naming scheme is used:

`<Base Class Name>$<ID Type>$<ID>`

For ActiveX based objects, the class name takes the following form:

- Using a COM Class ID:

`IDLcomActiveX$CLSID$<the Class ID>`

- Using a COM Program ID:

`IDLcomActiveX$PROGID$<the Program ID>`

Note

All ActiveX-based objects created in IDL subclass from the intrinsic IDL class `IDLcomActiveX`, which is a sub-class of `IDLcomIDispatch`.

Note

The COM Class ID separator (-) or the Program ID separator (.) should be indicated using an underscore (_) when constructing the class name for the particular object name in IDL.

ActiveX Control Creation

The creation of an ActiveX control in IDL follows the model used with Object Graphics in draw widgets. The control is created using a procedural interface that is exposed as part of the IDL Widget system. Methods of the ActiveX control are accessed via the underlying IDL object that represents the control. Essentially, all IDL Widget-related functionality is managed using the IDL Widget interface, while COM method dispatching is handled using the underlying IDL object that represents the ActiveX control.

The `WIDGET_ACTIVEX` function is used to create an ActiveX control in IDL and also to place it into an IDL widget hierarchy. The Program ID or Class ID of the underlying IDL object that represents the ActiveX control is retrieved using the `GET_VALUE` keyword to the `WIDGET_CONTROL`. This is similar to the operations used to get the window object from an IDL draw widget.

Note

If you specify the class ID of a non-ActiveX component using `WIDGET_ACTIVEX()`, the results are unpredictable (this is not recommended since it may or may not work depending on the actual COM object.)

Note

`WIDGET_ACTIVEX` is the only method used to create an IDL object that represents an ActiveX control. `OBJ_NEW` should only be used to create non-ActiveX COM objects. Creating an ActiveX control (an object based off the class name prefix `IDLcomActiveX$`) using `OBJ_NEW()` is not supported and the results are undefined.

ActiveX Control Access and Dispatching

Access to the IDL object that represents the control is gained using the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure after Widget realization. Once the underlying IDL object is retrieved from the Widget that represents the ActiveX control, methods are called using the same methodology and underlying technology as IDispatch-based COM objects in IDL.

Events generated by the ActiveX control are also dispatched using the standard IDL widget methodology. When an ActiveX event is passed into IDL, it is packaged into an IDL structure that contains the ActiveX event parameters, and is dispatched using the standard IDL widget event-dispatching methodologies. As such, user event-handling routines are called with a structure that contains the ActiveX event parameters.

Note

IDL objects use method names to identify and call object life cycle methods (INIT and CLEANUP). As such, these method names should be considered reserved. If an underlying ActiveX or IDispatch object implements a method using either INIT or CLEANUP those methods will be overridden by the IDL life cycle methods and not accessible from IDL. The ActiveX or IDispatch object also cannot have a GetProperty or SetProperty method, since IDL uses these to manage properties.

Freeing Dynamic Resources

The HEAP_FREE routine frees all dynamic resources associated with the argument which is passed to the routine. This routine will traverse the data represented by the variable, traversing arrays and structures. When an object value is encountered, it is released using the OBJ_DESTROY routine. When a pointer value is encountered, its contents are scanned, freeing any dynamic resources, and then the pointer itself is released using the PTR_FREE routine. This is especially helpful with routines that return dynamically allocated information.

HEAP_FREE may be used:

- To release the dynamic resources contained a structure returned from the GetRecord method of an IDLdbRecordset object.
- To release any dynamic resources associated with an event generated by an ActiveX control that is embedded in an IDL Widget hierarchy using Widget_ActiveX().

Arrays can be contained in the events that are propagated from the ActiveX control. If an event contains an array, the array is placed in an IDL pointer and that pointer is contained in the event structure. Since this memory is in an IDL Pointer, it is the user's responsibility to free the pointer using PTR_FREE or HEAP_FREE.

If it is unclear if the event structure will contain dynamic elements (objects or pointers) it is best to pass the ActiveX event structure to the HEAP_FREE routine when finished. This will ensure that all dynamic portions of the structure are released.

ActiveX Control Destruction

Destruction of an ActiveX control takes places in any of the following cases:

- When the widget hierarchy that it belongs to is destroyed.
- When a call to WIDGET_CONTROL, /DESTROY is made.
- When the underlying IDL object is destroyed using OBJ_DESTROY.

Example: Embedding an ActiveX Control in IDL

The following example demonstrates just how you can embed and ActiveX control in an IDL widget. This example creates a base widget that calls an ActiveX calendar control (obtained from the Microsoft Office 2000 package). The result is a clickable desktop IDL calendar.

Copy and paste the following text into an IDL Editor window. After saving the file as `cal.pro`, compile and run the program.

1. Give your program an identifying header:

```
pro cal_event, ev
```

2. Prepare the base widget:

```
WIDGET_CONTROL, ev.id, GET_VALUE = oCal
WIDGET_CONTROL, ev.top, GET_UVALUE = state
ocal->GetProperty, day=day, year=year, month = month
WIDGET_CONTROL, state.Day , SET_VALUE = STRTRIM(day,2)
WIDGET_CONTROL, state.year , SET_VALUE = STRTRIM(year,2)
WIDGET_CONTROL, state.month , SET_VALUE = STRTRIM(month,2)

HEAP_FREE, ev
end
```

3. Now create an ActiveX control:

```
pro cal
wBase = WIDGET_BASE(COLUMN = 1, SCR_XSIZE = 400)
wSubBase = WIDGET_BASE(wBase, /ROW)
wVoid = WIDGET_LABEL(wSubBase, value = 'Month: ')
wMonth = WIDGET_LABEL(wSubBase, value = 'October')
wSubBase = WIDGET_BASE(wBase, /ROW)
wVoid = WIDGET_LABEL(wSubBase, VALUE = 'Day: ')
wDay = WIDGET_LABEL(wSubBase, VALUE = '22')
wSubBase = WIDGET_BASE(wBase, /ROW)
wVoid = WIDGET_LABEL(wSubBase, VALUE = 'Year: ')
wYear = WIDGET_LABEL(wSubBase, VALUE = '1999')
wAx=WIDGET_ACTIVEX(WBASE, $
    '{8E27C92B-1264-101C-8A2F-040224009C02}')

WIDGET_CONTROL, wBase, /REALIZE

WIDGET_CONTROL, wBase, $
    SET_UVALUE = {month:wMonth, day:wDay, year:wYear}

; Should be IDispatch object for ActiveX control
WIDGET_CONTROL, wAx, GET_VALUE = oAx
oAx->GetProperty, day = day, year = year, month = month
```

```

WIDGET_CONTROL, wDay , SET_VALUE = STRTRIM(day, 2)
WIDGET_CONTROL, wyear , SET_VALUE = STRTRIM(year, 2)
WIDGET_CONTROL, wmonth , SET_VALUE = STRTRIM(month, 2)

XMANAGER, 'cal', wBase
END

```

4. Now run this example. You should see the following results:

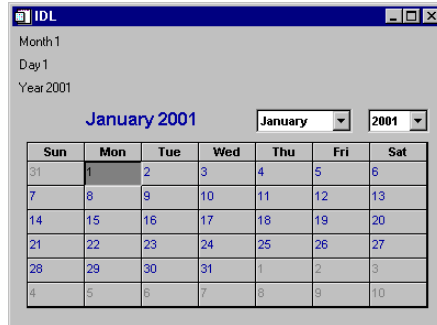


Figure 3-1: A Simple and Functional Calendar Created in IDL with an ActiveX Widget

Example: Creating an Excel Spreadsheet in IDL

In the next example, `WIDGET_ACTIVEX` is used to create a widget which calls an Excel spreadsheet (obtained from the Microsoft Office 2000 package) in IDL.

```

; excel_getSelection

; Purpose:
; Grab the data out of the current selection.
; Return 1 on success, 0 on error.
FUNCTION EXCEL_GETSELECTION, oExcel, aData

; Get the Selection collection of cells.
oExcel -> GetProperty, SELECTION = oSel
oSel -> GetProperty, COUNT = nCells
IF(nCells lt 1)THEN BEGIN
    OBJ_DESTROY, oSel
    RETURN, 0
ENDIF

; Now get the size of the selection.
oSel -> GetProperty, COLUMNS = oCols, ROWS = oRows

```

```

oCols -> GetProperty, COUNT = nCols
OBJ_DESTROY, oCols
oRows -> GetProperty, COUNT = nRows
OBJ_DESTROY, oRows
aData = FLTARR (nCols, nRows,/NOZERO)
  FOR i = 1, nCells DO BEGIN
    oSel -> GetProperty, ITEM = oItem, i
    oItem -> GetProperty, VALUE = vValue
    aData[i-1] = vValue
    OBJ_DESTROY, oItem
  endfor
OBJ_DESTROY, oSel
RETURN,1
END

; excel_setData

; Purpose:
; Set or initialize the values in the spreadsheet.
PRO excel_setData , oExcel
; size of data
nX = 20
oExcel -> GetProperty, ActiveSheet=oSheet
im = BESELJ (dist(nX))
  for i = 0, nx-1 do begin
    for j = 0, nx-1 do begin
      oSheet -> GetProperty, cells = oCell, i+1, j+1
      oCell -> SetProperty, value = im(i,j)
      OBJ_DESTROY, oCell
    ENDFOR
  ENDFOR
OBJ_DESTROY, oSheet
end

; excel_event

; Purpose:
; Event Handler for the excel component.

pro excel_event, ev
WIDGET_CONTROL ,ev.top, GET_UVALUE = sState, /NO_COPY
  IF(ev.dispid eq 1513)THEN BEGIN; Selection is changing
    ; Get the data for the selection
    IF(excel_getSelection(sState.oExcel, aData) NE 0)THEN BEGIN
      szData = SIZE (aData)
      ; Are we 2d?
      IF(szData[0] GT 1 AND szData[1] GT 1 AND szData[2] GT 1)THEN $
        SURFACE, aData $
      ELSE $

```

```

        PLOT, aData ; nope, 1 D
    ENDIF
ENDIF

; Reset our state variable.
WIDGET_CONTROL, ev.top, SET_UVALUE = sState,/NO_COPY
HEAP_FREE, ev
END

; Excel

; Purpose:
; Example that places the excel like spreadsheet
; control in an IDL widget and then plots the selected data.
PRO Excel
; Makes an ActiveX control.
!Except = 0
wBase = WIDGET_BASE(COLUMN = 1, TITLE = "IDL Excel Example")
wAx = WIDGET_ACTIVEX (WBASE,$
    '{0002E510-0000-0000-C000-000000000046}', $
    SCR_XSIZE = 800, SCR_YSIZE = 600)
wTxt = WIDGET_TEXT(wBase, value = '
                                ')
WIDGET_CONTROL, wBase,/REALIZE, SET_UVALUE = {wAX:wAX, wTXT:wTxt}
WIDGET_CONTROL, wAX, GET_VALUE = oExcel
oExcel->SetProperty, DisplayTitleBar = 0
excel_setData, oExcel
WIDGET_CONTROL, wBase, SET_UVALUE = {oExcel:oExcel, wText:wTxt}
XMANAGER, 'excel', wBase, /NO_BLOCK
END

```

Access to ActiveX Methods and Properties

In IDL, an ActiveX control is represented in a similar method as an IDispatch COM object, as an IDL object. The user gains access to the object using `WIDGET_CONTROL` with the `GET_VALUE` keyword, passing in the widget id returned from `WIDGET_ACTIVEX()`. The `GET_VALUE` keyword returns an IDL object that represents the ActiveX control. For example:

```

idAX = Widget_ActiveX(idParent, idClass)

WIDGET_CONTROL, idTLB, /REALIZE
WIDGET_CONTROL, idAX, GET_VALUE=oAX
oAX -> ActiveXMethod

```

Once the object is retrieved, methods are called just like any other IDL object methods.

The object is destroyed by either calling OBJ_DESTROY on it, or when the Widget is destroyed.

Note

If the initialization method for IDispatch object creation uses a standard class name, the class returned from GET_VALUE should also have a standard name. One possibility is IDLcomActiveX.

Event Propagation

Events generated by an ActiveX control are propagated to the IDL user as with any other IDL Widget event; a user event handler is called with an event structure.

For ActiveX controls, events are signaled by the control calling methods on the ActiveX container that holds the control. The parameters to the called method contain the attributes associated with the triggered event. To propagate this information to IDL, this method call is converted into an IDL event structure.

As with other IDL Widget event structures, the first three fields contain the Widget ID, Top ID and the Handler ID for the event. For event typing, the DISPID and method name of the ActiveX event callback method are also included in the structure. As for the parameter information, it is placed in fields of the structure. The parameter name is used to construct the field name and the associated data is placed in the field. Because this is dynamic, an anonymous structure is used for this event.

The following gives an idea of the basic format of an ActiveX event structure:

```
{ ID           : 0L,
  TOP          : 0L,
  HANDLER      : 0L,
  DISPID       : 0L, ; The DISPID of the callback method
  METHOD        : "", ; The name of the callback method
  <Param1 name> : <Param1 value>,
  <Param2 name> : <Param2 value>,

  <ParamN name> : <ParamN value>
}
```



Chapter 4: Using the Shortcut Menu Widget

This chapter describes the implementation of shortcut menus for use with the IDL Widget system.

Introduction to the Shortcut Menu Widget	158	Creating a List Widget Shortcut Menu . . .	166
Creating a Base Widget Shortcut Menu . .	160	Creating a Text Widget Shortcut Menu . .	170
Creating a Draw Widget Shortcut Menu . .	162		

Introduction to the Shortcut Menu Widget

In IDL 5.5, a new shortcut menu widget (otherwise known as a context sensitive or pop-up menu) has been added to enhance the IDL widget system. These menus are available for:

- Base widgets
- Text widgets
- Draw widgets
- List widgets

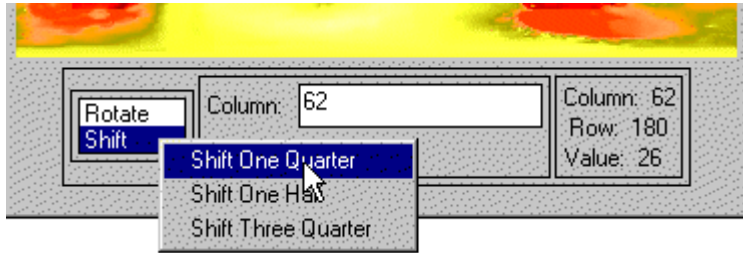


Figure 4-1: Widget Shortcut Menu

Shortcut menus are made available to the user in two separate components. The first is a shortcut menu event and the second is the creation and display of a shortcut menu for a particular widget.

Shortcut menu events can be requested by setting the `CONTEXT_EVENTS` keyword at the time of widget creation using `WIDGET_CONTROL`. The events can be turned off for a particular widget by calling `WIDGET_CONTROL` with the `CONTEXT_EVENTS` keyword set to 0.

To create a shortcut menu, use the `CONTEXT_MENU` keyword when creating a widget base. The shortcut menu base must be a child of one of the widget types listed previously. The shortcut menu base is a special base widget that can be used as a parent to add menu buttons or regular push buttons. The use of this widget is similar to the way a menu bar base is used as a parent for menu buttons. Multiple shortcut menu bases may be associated with a single widget.

Note

For shortcut menus, both plain buttons and menu buttons are allowed while for menu bars only menu buttons are allowed.

Using WIDGET_DISPLAYCONTEXTMENU

The new WIDGET_DISPLAYCONTEXTMENU procedure displays a shortcut (context sensitive or pop-up) menu. After creating buttons for the shortcut menu, it can be displayed using WIDGET_DISPLAYCONTEXTMENU. This is normally called in an event handler that has processed a shortcut menu event or a button event from a draw widget. This procedure takes the ID of the widget that is the parent of the shortcut menu, the x and y location to display the menu, and the ID of the shortcut menu base. The ID would normally be the event.id value of the shortcut menu event, and the x and y locations also come from the shortcut event. As stated above, there may be multiple shortcut menus for a particular widget. The last parameter of WIDGET_DISPLAYCONTEXTMENU allows the user to specify which menu to display. In the case of a draw widget that is the parent of a shortcut menu, the x and y locations can be obtained from the button event structure.

When WIDGET_DISPLAYCONTEXTMENU is called it displays the shortcut menu and handles the native event if the user selects a button. If a button is selected, a button event is generated and the menu is dismissed. If no button is selected (the user clicks elsewhere on the screen) then the menu is dismissed and no event is generated. Normally no further processing would be done in the shortcut event or draw event handler after calling WIDGET_DISPLAYCONTEXTMENU. The new user event is queued and will be handled in a new call to the event handler.

Creating a Base Widget Shortcut Menu

A base widget allows you to create the base upon which you can incorporate other widgets. With new functionality in IDL, you can add a shortcut menu to your base widget. Since a base widget does not usually cause events, you do not need to specify when a context event occurs as shown in the following example:

```

; Event handler routine for the "Selection 1" button in
; the context menu of the top level base.
PRO FirstEvent, event

; Output that the "Selection 1" button has been pressed.
PRINT, ' '
PRINT, 'Selection 1 Pressed'

END

; Event handler routine for the "Selection 2" button in
; the context menu of the top level base.
PRO SecondEvent, event

; Output that the "Selection 1" button has been pressed.
PRINT, ' '
PRINT, 'Selection 2 Pressed'

END

; Event handler routine for the "Done" button in
; the context menu of the top level base.
PRO DoneEvent, event

; Output that the "Done" button has been pressed.
PRINT, ' '
PRINT, 'Done Pressed'

; Destroy the top level base.
WIDGET_CONTROL, event.top, /DESTROY

END

; Event handler routine for the context menu of the
; top level base. This event handler routine is called
; when the user right-clicks on the top level base.
PRO ContextTLBaseExample_Event, event

```

```

; Obtain the widget ID of the context menu base.
contextBase = WIDGET_INFO(event.id, $
    FIND_BY_UNAME = 'contextMenu')

; Display the context menu and send its events to the
; other event handler routines.
WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, $
    event.y, contextBase

END

; Main Routine: GUI creation routine.
PRO ContextTLBaseExample

; Initialize the top level (background) base. This base
; contains context events. In other words, the user
; can right-click on the base to obtain a context
; menu.
topLevelBase = WIDGET_BASE(/COLUMN, XSIZE = 100, $
    YSIZE = 100, /CONTEXT_EVENTS)

; Initialize the base for the context menu.
contextBase = WIDGET_BASE(topLevelBase, /CONTEXT_MENU, $
    UNAME = 'contextMenu')

; Initialize the buttons of the context menu.
firstButton = WIDGET_BUTTON(contextBase, $
    VALUE = 'Selection 1', EVENT_PRO = 'FirstEvent')
secondButton = WIDGET_BUTTON(contextBase, $
    VALUE = 'Selection 2', EVENT_PRO = 'SecondEvent')
doneButton = WIDGET_BUTTON(contextBase, VALUE = 'Done', $
    /SEPARATOR, EVENT_PRO = 'DoneEvent')

; Display the GUI.
WIDGET_CONTROL, topLevelBase, /REALIZE

; Handle the events from the GUI.
XMANAGER, 'ContextTLBaseExample', topLevelBase

END

```

Creating a Draw Widget Shortcut Menu

A draw widget contains a rectangular area which functions as a standard IDL graphics window. Using the new IDL functionality, shortcut menu options can be added to a draw widget allowing for such choices as changing the color tables of an image (as is demonstrated in the following example).

Note

The `CONTEXT_EVENTS` keyword may not be used with `WIDGET_DRAW`. Draw widgets already support button events. When using a draw widget, enable button events and then check for `EVENT.RELEASE EQ 4` to indicate a right mouse button release event.

```

; Event handler routine for the "XLOADCT" button in
; the context menu of the draw widget.
PRO LoadCTEvent, event

; Display the XLOADCT utility to allow the user to
; change the current color table.
XLOADCT, /BLOCK, GROUP = event.id

; Obtain the window ID of the draw widget.
imageDraw = WIDGET_INFO(event.top, $
    FIND_BY_UNAME = 'imageDisplay')
WIDGET_CONTROL, imageDraw, GET_VALUE = windowDraw

; Obtain the image to redisplay it with the updated
; color table from XLOADCT utility.
WIDGET_CONTROL, event.top, GET_UVALUE = image

; Redisplay the image with the updated color table.
WSET, windowDraw
TV, image

END

; Event handler routine for the "XPALETTE" button in
; the context menu of the draw widget.
PRO PaletteEvent, event

; Display the XPALETTE utility to allow the user to
; modify some or all of the current color table.
XPALETTE, /BLOCK, GROUP = event.id

```

```

; Obtain the window ID of the draw widget.
imageDraw = WIDGET_INFO(event.top, $
    FIND_BY_UNAME = 'imageDisplay')
WIDGET_CONTROL, imageDraw, GET_VALUE = windowDraw

; Obtain the image to redisplay it with the updated
; color table from XPALETTE utility.
WIDGET_CONTROL, event.top, GET_UVALUE = image

; Redisplay the image with the updated color table.
WSET, windowDraw
TV, image

END

; Event handler routine for the "Done" button in
; the context menu of the top level base.
PRO DoneEvent, event

; Destroy the top level base.
WIDGET_CONTROL, event.top, /DESTROY

END

; Event handler routine for the events of the draw
; widget. This event handler routine is called
; when the user left- or right-clicks on the draw widget.
PRO DrawEvents, event

; If either a left- or right-click occurs, obtain the image to
; determine the value of the pixel at the location under the
; cursor.
WIDGET_CONTROL, event.top, GET_UVALUE = image

; If either a left- or right-click occurs, output the location
; and value of the pixel under the cursor.
PRINT, ' '
PRINT, 'Column: ', event.x
PRINT, 'Row: ', event.y
PRINT, 'Value: ', image[event.x, event.y]

```

```

; If a right-click occurs, display the context menu and send
; its events to the other event handler routines.
IF (event.release EQ 4) THEN BEGIN
    ; Obtain the widget ID of the context menu base.
    contextBase = WIDGET_INFO(event.top, $
        FIND_BY_UNAME = 'drawContext')
    ; Display the context menu and send its events to the
    ; other event handler routines.
    WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, event.y, $
        contextBase
ENDIF

END

; Main Routine: GUI creation routine.
PRO ContextDrawExample

; Determine the path to the file containing the image.
file = FILEPATH('worldelv.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize the image size parameter.
imageSize = [360, 360]

; Import in the image from the file.
image = READ_BINARY(file, DATA_DIMS = imageSize)

; Initialize the top level (background) base.
topLevelBase = WIDGET_BASE(/COLUMN)

; Initialize the draw widget to contain the display
; of the image. This draw widget enables buttons events. In
; other words, the user can left- or right-click on the image
; display to obtain the location of the pixel under the
; cursor or to obtain a context menu, respectively.
imageDraw = WIDGET_DRAW(topLevelBase, /BUTTON_EVENTS, $
    XSIZE = imageSize[0], YSIZE = imageSize[1], $
    EVENT_PRO = 'DrawEvents', UNAME = 'imageDisplay')

; Initialize the base for the context menu.
contextBase = WIDGET_BASE(topLevelBase, /CONTEXT_MENU, $
    UNAME = 'drawContext')

```

```

; Initialize the buttons of the context menu.
loadCTButton = WIDGET_BUTTON(contextBase, $
    VALUE = 'XLOADCT', EVENT_PRO = 'LoadCTEvent')
paletteButton = WIDGET_BUTTON(contextBase, $
    VALUE = 'XPALETTE', EVENT_PRO = 'PaletteEvent')
doneButton = WIDGET_BUTTON(contextBase, VALUE = 'Done', $
    /SEPARATOR, EVENT_PRO = 'DoneEvent')

; Display the GUI.
WIDGET_CONTROL, topLevelBase, /REALIZE

; Set the UVALUE of the top level base to the image so
; it can be accessed within the event handler routines.
WIDGET_CONTROL, topLevelBase, SET_UVALUE = image

; Obtain the window ID of the draw widget.
WIDGET_CONTROL, imageDraw, GET_VALUE = windowDraw

; Set the display to the window within the draw
; widget.
WSET, windowDraw

; Initialize the display.
DEVICE, DECOMPOSED = 0
LOADCT, 5

; Display the image in the window of the draw
; widget.
TV, image

; Determine the center location of the image display.
column = imageSize[0]/2
row = imageSize[1]/2

; Initially show the cursor in the center of the image
; display.
TVCRS, column, row

; Handle the events from the GUI.
XMANAGER, 'ContextDrawExample', topLevelBase, $
    /NO_BLOCK

END

```

Creating a List Widget Shortcut Menu

A list widget allows the use of a list of selectable text elements. An item can be selected by pointing with the mouse cursor and selecting a text element. With shortcut menu functionality, a right mouse click on this text element can allow for further choices as is shown in the following example widget:

Note

You can determine if a context menu event occurred with a list widget by the name of the event structure as in the following statement fragment:

```
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT')...

; Event handler routine for the "Rotate 90 Degrees" button in
; the context menu of the top level base.
PRO Rotate90Event, event

; Output that the "Rotate 90 Degrees" button has been pressed.
PRINT, ' '
PRINT, 'Rotate 90 Degrees Pressed'

END

; Event handler routine for the "Rotate 180 Degrees" button in
; the context menu of the top level base.
PRO Rotate180Event, event

; Output that the "Rotate 180 Degrees" button has been pressed.
PRINT, ' '
PRINT, 'Rotate 180 Degrees Pressed'

END

; Event handler routine for the "Rotate 270 Degrees" button in
; the context menu of the top level base.
PRO Rotate270Event, event

; Output that the "Rotate 270 Degrees" button has been pressed.
PRINT, ' '
PRINT, 'Rotate 270 Degrees Pressed'

END
```

```
; Event handler routine for the "Shift One Quarter" button in
; the context menu of the top level base.
PRO Shift025Event, event

; Output that the "Shift One Quarter" button has been pressed.
PRINT, ' '
PRINT, 'Shift One Quarter Pressed'

END

; Event handler routine for the "Shift One Half" button in
; the context menu of the top level base.
PRO Shift050Event, event

; Output that the "Shift One Half" button has been pressed.
PRINT, ' '
PRINT, 'Shift One Half Pressed'

END

; Event handler routine for the "Shift Three Quarters" button in
; the context menu of the top level base.
PRO Shift075Event, event

; Output that the "Shift Three Quarters" button has been pressed.
PRINT, ' '
PRINT, 'Shift Three Quarters Pressed'

END

; Event handler routine for the "Done" button in
; the context menu of the top level base.
PRO DoneEvent, event

; Output that the "Done" button has been pressed.
PRINT, ' '
PRINT, 'Done Pressed'

; Destroy the top level base.
WIDGET_CONTROL, event.top, /DESTROY

END
```

```

; Event handler routine for the events of the draw
; widget. This event handler routine is called
; when the user left- or right-clicks on the draw widget.
PRO ListEvents, event

; If either a left- or right-click occurs, obtain the selection
; index to determine the type of geometry change to occur.
selection = WIDGET_INFO(event.id, /LIST_SELECT)

; Output resulting selection.
PRINT, ' '
PRINT, 'Selection = ', selection

; If a right-click occurs display the appropriate context menu.
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT') THEN $
BEGIN
; If "Rotate" is selected, then use the rotate context menu.
IF (selection EQ 0) THEN BEGIN
; Obtain the widget ID of the rotate context menu base.
contextBase = WIDGET_INFO(event.top, $
FIND_BY_UNAME = 'contextRotate')
; Display the context menu and send its events to the
; other event handler routines.
WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, $
event.y, contextBase
ENDIF
; If "Shift" is selected, then use the shift context menu.
IF (selection EQ 1) THEN BEGIN
; Obtain the widget ID of the shift context menu base.
contextBase = WIDGET_INFO(event.top, $
FIND_BY_UNAME = 'contextShift')
; Display the context menu and send its events to the
; other event handler routines.
WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, $
event.y, contextBase
ENDIF
ENDIF

END

; Main Routine: GUI creation routine.
PRO ContextListExample

; Initialize the top level (background) base.
topLevelBase = WIDGET_BASE(/COLUMN)

```

```

; Initialize the geometry transform list. This list widget enables
; context events. In other words, the user can left- or right-click
; on the list to obtain a general selection or to make a specific
; selection, respectively.
list = ['Rotate', 'Shift']
geometryList = WIDGET_LIST(topLevelBase, VALUE = list, $
    /CONTEXT_EVENTS, EVENT_PRO = 'ListEvents')

; Initialize the base for the rotate context menu.
contextRotateBase = WIDGET_BASE(topLevelBase, /CONTEXT_MENU, $
    UNAME = 'contextRotate')

; Initialize the buttons of the rotate context menu.
rotate90Button = WIDGET_BUTTON(contextRotateBase, $
    VALUE = 'Rotate 90 Degrees', EVENT_PRO = 'Rotate90Event')
rotate180Button = WIDGET_BUTTON(contextRotateBase, $
    VALUE = 'Rotate 180 Degrees', EVENT_PRO = 'Rotate180Event')
rotate270Button = WIDGET_BUTTON(contextRotateBase, $
    VALUE = 'Rotate 270 Degrees', EVENT_PRO = 'Rotate270Event')
doneButton = WIDGET_BUTTON(contextRotateBase, VALUE = 'Done', $
    /SEPARATOR, EVENT_PRO = 'DoneEvent')

; Initialize the base for the shift context menu.
contextShiftBase = WIDGET_BASE(topLevelBase, /CONTEXT_MENU, $
    UNAME = 'contextShift')

; Initialize the buttons of the shift context menu.
shift025Button = WIDGET_BUTTON(contextShiftBase, $
    VALUE = 'Shift One Quarter', EVENT_PRO = 'Shift025Event')
shift050Button = WIDGET_BUTTON(contextShiftBase, $
    VALUE = 'Shift One Half', EVENT_PRO = 'Shift050Event')
shift075Button = WIDGET_BUTTON(contextShiftBase, $
    VALUE = 'Shift Three Quarter', EVENT_PRO = 'Shift075Event')
doneButton = WIDGET_BUTTON(contextShiftBase, VALUE = 'Done', $
    /SEPARATOR, EVENT_PRO = 'DoneEvent')

; Display the GUI.
WIDGET_CONTROL, topLevelBase, /REALIZE

; Handle the events from the GUI.
XMANAGER, 'ContextListExample', topLevelBase

END

```

Creating a Text Widget Shortcut Menu

Text widgets are used to display text and to get text input from the user. Text widgets can be one or more lines and can even contain scroll bars. An example of incorporating a shortcut menu into a text widget follows:

Note

You can determine if a context menu event occurred with a text widget by the name of the event structure as in the following statement fragment:

```
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT')...

; Event handler routine for the "Column" button in
; the context menu of the text widget.
PRO ColumnEvent, event

; Obtain the location variable from the UVALUE of the
; text widget.
locationText = WIDGET_INFO(event.top, FIND_BY_UNAME = 'xyText')
WIDGET_CONTROL, locationText, GET_UVALUE = location

; If location index is set to "Row" change it to "Column".
IF (location[2] EQ 1) THEN BEGIN
    titleLabel = WIDGET_INFO(event.top, FIND_BY_UNAME = 'xyLabel')
    WIDGET_CONTROL, titleLabel, SET_VALUE = 'Column: '
    location[2] = 0
ENDIF ELSE RETURN

; Store updated location variable in the UVALUE of the
; text widget.
WIDGET_CONTROL, locationText, SET_UVALUE = location

END

; Event handler routine for the "Row" button in
; the context menu of the text widget.
PRO RowEvent, event

; Obtain the location variable from the UVALUE of the
; text widget.
locationText = WIDGET_INFO(event.top, FIND_BY_UNAME = 'xyText')
WIDGET_CONTROL, locationText, GET_UVALUE = location
```

```

; If location index is set to "Column" change it to "Row".
IF (location[2] EQ 0) THEN BEGIN
    titleLabel = WIDGET_INFO(event.top, FIND_BY_UNAME = 'xyLabel')
    WIDGET_CONTROL, titleLabel, SET_VALUE = 'Row:  '
    location[2] = 1
ENDIF ELSE RETURN

; Store updated location variable in the UVALUE of the
; text widget.
WIDGET_CONTROL, locationText, SET_UVALUE = location

END

; Event handler routine for the "Done" button in
; the context menu of the text widget.
PRO DoneEvent, event

; Destroy the top level base.
WIDGET_CONTROL, event.top, /DESTROY

END

; Event handler routine for the events of the text
; widget. This event handler routine is called
; when the user left- or right-clicks in the text widget.
PRO TextEvents, event

; If a right-click occurs display the context menu.
IF (TAG_NAMES(event, /STRUCTURE_NAME) EQ 'WIDGET_CONTEXT') THEN $
    BEGIN

        ; Obtain the widget ID of the context menu base.
        contextBase = WIDGET_INFO(event.top, $
            FIND_BY_UNAME = 'contextMenu')

        ; Display the context menu and send its events to
        ; the other event handler routines.
        WIDGET_DISPLAYCONTEXTMENU, event.id, event.x, $
            event.y, contextBase
    ENDIF

; If text is edited, obtain new text inputted into widget.
WIDGET_CONTROL, event.id, GET_VALUE = textString
IF ((FIX(textString) GE 0) AND (FIX(textString) LE 360)) $
    THEN textValue = FIX(textString) ELSE RETURN $
textValue = textValue[0]

```

```

; Output resulting inputed value.
PRINT, ' '
PRINT, 'Text Value = ', textValue

; Obtain the location variable from the UVALUE of the
; text widget.
WIDGET_CONTROL, event.id, GET_UVALUE = location

; Determine if inputed value should be column or row.
IF(location[2] EQ 0) THEN location[0] = textValue $
    ELSE location[1] = textValue

; Output resulting location.
PRINT, ' '
PRINT, 'Column = ', location[0]
PRINT, 'Row = ', location[1]

; Store updated location variable in the UVALUE of the
; text widget.
WIDGET_CONTROL, event.id, SET_UVALUE = location

END

; Main Routine: GUI creation routine.
PRO ContextTextExample

; Initialize the top level (background) base.
topLevelBase = WIDGET_BASE(/COLUMN)

; Initialize location variable. This variable contains
; the column value, the row value, and a location index.
; The location index determines if the text value represents
; a column value, or it represents a row value.
column = 180
row = 180
locationIndex = 0
location = [column, row, locationIndex]

; Set initial title of the label for the text widget.
title = 'Column: '

; Initialize a base to contain the text widget and its label.
textBase = WIDGET_BASE(topLevelBase, /ROW, /FRAME)

; Initialize the label of the text widget.
titleLabel = WIDGET_LABEL(textBase, VALUE = title, $
    /DYNAMIC_RESIZE, UNAME = 'xyLabel')

```

```
; Initialize the text widget.
locationText = WIDGET_TEXT(textBase, VALUE = STRTRIM(column, 2), $
    /EDITABLE, UNAME = 'xyText', /CONTEXT_EVENTS, $
    UVALUE = location, EVENT_PRO = 'TextEvents')

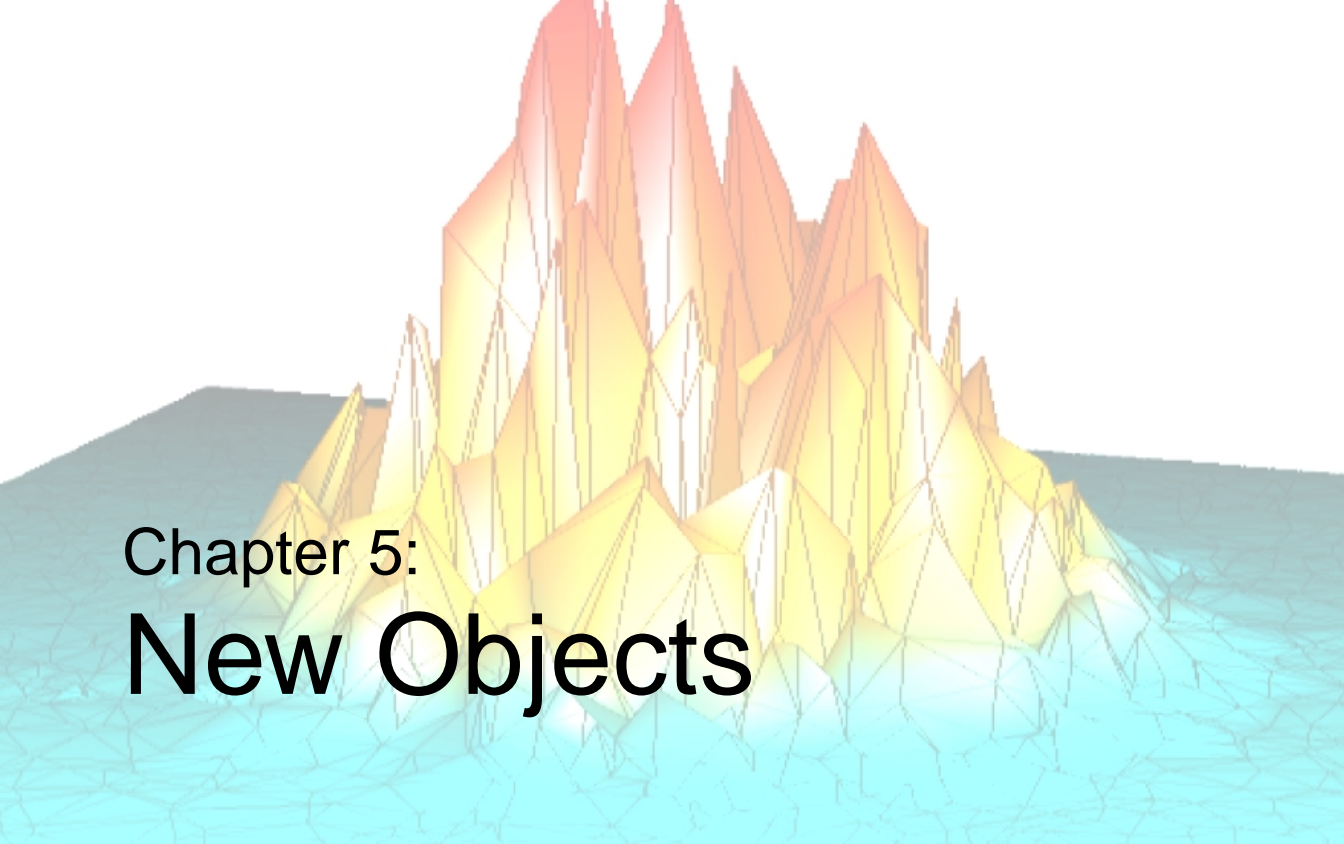
; Initialize the base for the context menu.
contextBase = WIDGET_BASE(topLevelBase, /CONTEXT_MENU, $
    UNAME = 'contextMenu')

; Initialize the buttons of the context menu.
columnButton = WIDGET_BUTTON(contextBase, $
    VALUE = 'Column', EVENT_PRO = 'ColumnEvent')
rowButton = WIDGET_BUTTON(contextBase, $
    VALUE = 'Row', EVENT_PRO = 'RowEvent')
doneButton = WIDGET_BUTTON(contextBase, VALUE = 'Done', $
    /SEPARATOR, EVENT_PRO = 'DoneEvent')

; Display the GUI.
WIDGET_CONTROL, topLevelBase, /REALIZE

; Handle the events from the GUI.
XMANAGER, 'ContextTextExample', topLevelBase

END
```

Chapter 5: New Objects

This chapter provides documentation for IDL Objects introduced in IDL 5.5.

IDLcomIDispatch	176	IDLffMrSID	181
---------------------------------------	-----	----------------------------------	-----

IDLcomIDDispatch

The IDLcomIDDispatch object class creates a COM object that implements an IDispatch interface. Using the provided class or program ID, the underlying implementation will utilize the internal IDL COM sub-system to instantiate the desired COM object.

Note

IDL objects use method names to identify and call object life cycle methods (INIT and CLEANUP). As such, these method names should be considered reserved. If an underlying ActiveX or IDispatch object implements a method using either INIT or CLEANUP those methods will be overridden by the IDL life cycle methods and not accessible from IDL.

Subclasses

A dynamic sub-class of IDLcomIDDispatch is created when the object is instantiated. A dynamic class name is created to provide a unique name for each component type, while providing the same super-class across all IDispatch components.

Creation

See [IDLcomIDDispatch::Init](#)

Methods

- [IDLcomIDDispatch::Init](#)
- [IDLcomIDDispatch::GetProperty](#)
- [IDLcomIDDispatch::SetProperty](#)

IDLcomIDDispatch::Init

The IDLcomIDDispatch::Init function method is used to initialize a given COM object and establish a link between the resulting IDL object and the IDispatch interface of the underlying COM object.

Syntax

Obj = OBJ_NEW('IDLcomIDDispatch\$<IDTYPE>\$ID')

Arguments

None

Class Names

To ensure that each particular type of COM object has a unique IDL class type, the identifier for the underlying COM object is utilized to construct the IDL class name. Since two types of class identifiers exist in COM, those must also be indicted during this class construction process. With this in mind the following naming scheme was devised:

<Base Class Name>\$<ID Type>\$<ID>

For IDispatch based objects, the class name takes the following form:

Using a COM Class ID

IDLcomIDDispatch\$CLSID\$<the Class ID>

Using a COM Program ID

IDLcomIDDispatch\$PROGID\$<the Program ID>

Note

All IDispatch based objects created in IDL sub-class from the intrinsic IDL class IDLcomIDDispatch.

The COM Class ID separator (-) or the Program ID separator (.) should be indicated using an underscore (_) when constructing the class name for the particular object name. For example:

If the CLSID of an object is:

A77BC2B2-88EC-4D2A-B2B3-FS56ACB52ES2

then using IDLcomIDDispatch to create an instance of the object would appear as:

```
demobj = OBJ_NEW $  
  ( 'IDLcomIDispatch$CLSID$A77BC2B2-88EC_4D2A_B2B3_FS56ACB52ES2' )
```

Note

The curly braces ({ }) for COM Class IDs should not be included in the name of the object. They are invalid characters for IDL Class names.

IDLcomIDispatch::GetProperty

The IDLcomIDispatch::GetProperty function method is used to get properties for a particular IDispatch interface. The IDispatch property names are mapped to IDL keywords. The underlying property values are treated as IDL keyword values. This follows conventions set forth by other IDL objects.

Note

The provided keywords must map directly to a property name or an error will be thrown. Any keyword that is passed into either of the property routines is assumed to be a fully-qualified IDispatch property name. As such, the partial keyword name functionality provided by IDL is not valid with IDL COM based objects.

Note

Some *gettable* properties require input parameters. As such, the GetProperty method can take parameters. If parameters are provided, only one property can be provided.

Syntax

IDLcomIDispatch -> GetProperty, <PROPERTY_NAME> = *Value*, [*arg0*, *arg1*, ...]

Arguments

Note

Some IDLcomIDispatch GetProperty calls take arguments. The argument used, if any, is dependent on the individual property.

IDLcomIDDispatch:: SetProperty

The IDLcomIDDispatch::SetProperty function method is used to set properties for a particular IDispatch interface. The IDispatch property names are mapped to IDL keywords. The underlying property values are treated as IDL keyword values. This follows conventions set forth by other IDL objects.

Note

The provided keywords must map directly to a property name or an error will be thrown. Any keyword that is passed into either of the property routines is assumed to be a fully-qualified IDispatch property name. As such, the partial keyword name functionality provided by IDL is not valid with IDL COM based objects.

Syntax

IDLcomIDDispatch -> SetProperty, <PROPERTY_NAME> = *Value*

Arguments

None

IDLffMrSID

An IDLffMrSID object class is used to query information about and load image data from a MrSID (.sid) image file.

Superclasses

This class has no superclasses.

Subclasses

This class has no subclasses.

Creation

See [IDLffMrSID::Init](#)

Methods

This class has the following methods:

- [IDLffMrSID::Cleanup](#)
- [IDLffMrSID::GetDimsAtLevel](#)
- [IDLffMrSID::GetImageData](#)
- [IDLffMrSID::GetProperty](#)
- [IDLffMrSID::Init](#)

IDLffMrSID::Cleanup

The IDLffMrSID::Cleanup procedure method deletes all MrSID objects, closing the MrSID file in the process. It also deletes the IDL objects used to communicate with the MrSID library.

Syntax

OBJ_DESTROY, *Obj*

or

Obj -> [IDLffMrSID::]Cleanup(*Only in subclass' Cleanup method.*)

Arguments

None.

Keywords

None.

IDLffMrSID::GetDimsAtLevel

The IDLffMrSID::GetDimsAtLevel function method is used to retrieve the dimensions of the image at a given level. This can be used, for example, to determine what level is required to fit the image into a certain area.

Syntax

Dims = Obj -> [IDLffMrSID::]GetDimsAtLevel (Level)

Arguments

Level

Set this argument to a scalar integer that specifies the level at which the dimensions are to be determined. This level must be in the range returned by the LEVELS keyword of IDLffMrSID::GetProperty.

Keywords

None.

Example

```
PRO MrSID_GetDimsAtLevel

; Initialize the MrSID file object.
oFile = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $
    SUBDIRECTORY = ['examples', 'data']))

; Get the range of levels of resolution contained within the file.
oFile -> GetProperty, LEVELS = lvls
PRINT, lvls
; IDL prints, -9, 4

; Print the image dimensions at the lowest image resolution
; where image level = 4.
imgLevelA = MAX(lvls)
dimsAtA = oFile-> GetDimsAtLevel(imgLevelA)
PRINT, 'Dimensions of lowest resolution image is', dimsAtA
;IDL prints, 32, 32
```

```
; Print the image dimensions at full resolution
; where image level = 0
dimsAtFull = oFile -> GetDimsAtLevel(0)
PRINT, 'Dimensions of full resolution image is', dimsAtFull
;IDL prints, 512, 512

;Print the image dimensions at the highest resolution
; where image level = -9
highestLvl = MIN(lvls)
dimsAtHighest = oFile -> GetDimsAtLevel(highestLvl)
PRINT, 'Dimensions of highest resolution image is', dimsAtHighest
;IDL prints, 262144, 262144

; Clean up object references.
OBJ_DESTROY, [oFile]

END
```

IDLffMrSID::GetImageData

The IDLffMrSID::GetImageData function method extracts and returns the image data from the MrSID file at the specified level and location.

Syntax

```
ImageData = Obj->[IDLMrSID::]GetImageData ( [, LEVEL = lvl]  
[, SUB_RECT = rect] )
```

Return Value

ImageData returns an n -by- w -by- h array containing the image data where n is 1 for grayscale or 3 for RGB images, w is the width and h is the height.

Note

The returned image is ordered bottom-up, the first pixel returned is located at the bottom-left corner of the image. This differs from how data is stored in the MrSID file where the image is top-down, meaning the pixel at the start of the file is located at the top-left corner of the image.

Arguments

None.

Keywords

LEVEL

Set this keyword to an integer that specifies the level at which to read the image.

If this keyword is not set, the maximum level is used which returns the minimum resolution (see the LEVELS keyword to IDLffMrSID::GetProperty).

SUB_RECT

Set this keyword to a four-element vector [x, y, xdim, ydim] specifying the position of the lower left-hand corner and the dimensions of the sub-rectangle of the MrSID image to return. This is useful for displaying portions of a high-resolution image.

If this keyword is not set, the whole image will be returned. This may require significant memory if a high-resolution image level is selected.

If the sub-rectangle is greater than the bounds of the image at the selected level the area outside the image bounds will be set to black.

Note

The elements of SUB_RECT are measured in pixels at the current level. This means the point $x = 10$, $y = 10$ at level 1 will be located at $x = 20$, $y = 20$ at level 0 and $x = 5$, $y = 5$ at level 2.

Example

```
PRO MrSID_GetImageData

; Initialize the MrSID file object.
oFile = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $
    SUBDIRECTORY = ['examples', 'data']))

; Get the range of levels of resolution contained within the file.
oFile -> GetProperty, LEVELS = lvls
PRINT, lvls
; IDL prints, -9, 4

; Get the image data at level 0.
imgDataA = oFile -> GetImageData(LEVEL = 0)
HELP, 'image array data at full resolution', imgDataA
; IDL prints, Array[1, 512, 512] indicating a grayscale 512 x 512
array.

; Display the full resolution image.
oImgA = OBJ_NEW('IDLgrImage', imgDataA)
oModelA = OBJ_NEW('IDLgrModel')
oModelA -> Add, oImgA
XOBJVIEW, oModelA, BACKGROUND = [0,0,0], $
    TITLE = 'Full Resolution Image', /BLOCK

; Get the image data of a higher resolution image,
imgDataB = oFile -> GetImageData(LEVEL = -2)
HELP, imgDataB
; IDL returns [1,2048,2048] indicating a grayscale 2048 x 2048
array.
```

```
; To save processing time, display only a 1024 x 1024 portion of
; the high resolution, using 512,512 as the origin..
imgDataSelect = oFile -> GetImageData(LEVEL = -2,$
    SUB_RECT = [512, 512, 1024, 1024])
oImgSelect = OBJ_NEW('IDLgrImage', imgDataSelect)
oModel = OBJ_NEW('IDLgrModel')
oModel -> Add, oImgSelect

XOBJVIEW, oModel, BACKGROUND = [0,0,0], $
    TITLE = 'Detail of High Resolution Image', /BLOCK

; Clean up object references.
OBJ_DESTROY, [oFile, oImgA, oModelA, oImgSelect, oModel]

END
```

IDLffMrSID::GetProperty

The IDLffMrSID::GetProperty function method is used to query properties associated with the MrSID image.

Syntax

```
Obj->[IDLffMrSID::]GetProperty [, CHANNELS=nChannels]  
[, DIMENSIONS=Dims] [, LEVELS=Levels] [, PIXEL_TYPE=pixelType]  
[, TYPE=strType] [, GEO_VALID=geoValid] [, GEO_PROJTYPE=geoProjType]  
[, GEO_ORIGIN=geoOrigin] [, GEO_RESOLUTION=geoRes]
```

Arguments

None.

Keywords

CHANNELS

Set this keyword to a named variable that will contain the number of image bands. For RGB images this is 3, for grayscale it is 1.

DIMENSIONS

Set this keyword equal to a named variable that will contain a two-element long integer array of the form [*width*, *height*] that specifies the dimensions of the MrSID image at level 0 (full resolution).

LEVELS

Set this keyword equal to a named variable that will contain a two-element long integer array of the form [*minlvl*, *maxlvl*] that specifies the range of levels within the current image. Higher levels are lower resolution. A level of 0 equals full resolution. Negative values specify higher levels of resolution.

PIXEL_TYPE

Set this keyword to a named variable that will contain the IDL basic type code for a pixel sample. For a list of the data types indicated by each type code, see “IDL Type Codes” in the *IDL Reference Guide*.

TYPE

Set this keyword to a named variable that will contain a string identifying the file format. This should always be MrSID.

GEO_VALID

Set this keyword to a named variable that will contain a long integer that is set to:

- 1 - If the MrSID image contains valid georeferencing data.
- 0 - If the MrSID image does not contain georeferencing data or the data is in an unsupported format.

Note

Always verify that this keyword returns 1 before using the data returned by any other GEO_* keyword.

GEO_PROJTYPE

Set this keyword to a named variable that will contain an unsigned integer that specifies the geoTIFF projected coordinate system type code. For example, type code 32613 corresponds to PCS_WGS84_UTM_zone_13N.

For more information on the geoTIFF file type and available type codes see:

<http://www.remotesensing.org/geotiff/geotiff.html>

GEO_ORIGIN

Set this keyword to a named variable that will contain a two-element double precision array of the form [x, y] that specifies the location of the center of the upper-left pixel.

GEO_RESOLUTION

Set this keyword to a named variable that will contain a two-element double precision array of the form [xres, yres] that specifies the pixel resolution.

Example

```
PRO MrSID_GetProperty

; Initialize the MrSID object.
oFile = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $
    SUBDIRECTORY = ['examples', 'data']))
```

```
; Get the property information of the MrSID file
oFile -> GetProperty, CHANNELS = chan, LEVELS = $
      lvls, Pixel_Type = pType, TYPE = fileType, GEO_VALID = geoQuery

; Print MrSID file information.
PRINT, 'Number of image channels = ', chan
; IDL returns 1 indicating one image band.

PRINT, 'Range of image levels = ', lvls
; IDL returns -9, 4, the minimum and maximum level values.

PRINT, 'Type code of image pixels = ', pType
; IDL returns 1 indicating byte data type.

PRINT, 'Image file type = ', FileType
; IDL returns "MrSID"

PRINT, 'Result of georeferencing data query = ', geoQuery
; IDL returns 0 indicating that the image does not contain
; georeferencing data.

; Destroy object references.
OBJ_DESTROY, [oFile]

END
```

IDLffMrSID::Init

The IDLffMrSID::Init function method initializes an IDLffMrSID object containing the image data from a MrSID image file.

Syntax

```
Result = OBJ_NEW('IDLffMrSID', Filename [, /QUIET] )
```

Arguments

Filename

Set this argument to a scalar string containing the full path and filename of a MrSID file to be accessed through this IDLffMrSID object.

Note

This is a required argument; it is not possible to create an IDLffMrSID object without specifying a valid MrSID file.

Keywords

QUIET

Set this keyword to suppress error messages while constructing the IDLffMrSID object.

Example

```
oMrSID = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $  
    SUBDIRECTORY = ['examples', 'data']))
```




Chapter 6: New IDL Routines

This chapter describes IDL Routines introduced in IDL version 5.5.

CPU

The CPU procedure controls the way IDL uses the system processor for calculations. The results of using the CPU procedure are reflected in the state of the !CPU system variable.

Syntax

```
CPU [,TPOOL_MAX_ELTS = NumMaxElts] [,TPOOL_MIN_ELTS = NumMinElts]  
[,TPOOL_NTHREADS = NumThreads] [,/VECTOR_ENABLE]
```

Keywords

TPOOL_MAX_ELTS

This keyword specifies the maximum number of data elements for computations that use the thread pool. This keyword changes the value returned by !CPU.TPOOL_MAX_ELTS. If the memory required for a given computation fits in physical memory, using the thread pool typically provides an increase in speed compared to the single-threaded case. However, once the computation exceeds the ability of the system's physical memory to contain it, use of the thread pool can be slower than the single-threaded case as the threads end up vying for access to system memory. If the system variable !CPU.TPOOL_MAX_ELTS is non-zero, IDL will not use the thread pool for any computation involving more than that number of elements. The default for this value is 0, meaning no imposed limit.

TPOOL_MIN_ELTS

This keyword sets the minimum number of data elements for a computation that are necessary before IDL will use the thread pool. For fewer than TPOOL_MIN_ELTS, the main IDL thread will perform the computation without using the thread pool. It is important not to use the thread pool for small tasks since the overhead of using the thread pool will not be offset by the overhead incurred by operation of the thread pool, and the performance of the computation will be slower than if the thread pool was not used.

TPOOL_NTHREADS

This keyword sets the number of threads that IDL will use in thread pool computations. The default is to use !CPU.HW_NCPU threads, so that each thread will have the potential to run in parallel with the others. If you set TPOOL_NTHREADS to 0, !CPU.HW_NCPU threads will be used. Setting this keyword to 1 disables threading. For numerical computation, there is no benefit to

using more threads than your system has CPUs. However, depending on the size of the problem and the number of other programs running on the system, there may be a performance advantage to using fewer CPUs.

VECTOR_ENABLE

Set this keyword to enable use of the system's vector unit (e.g. Macintosh Altivec/Velocity Engine). Set it to zero to disable such use. This keyword is ignored if the current system does not support a vector unit, which can be determined by the value of the !CPU.HW_VECTOR system variable.

Example

In the following example, we will:

- Save the current thread pool settings from the !CPU system environment variable.
- Modify the thread pool settings so that IDL is configured, for our particular system, to efficiently perform a floating point computation.
- Perform a floating point computation.
- Modify the thread pool settings so that IDL is configured, for our particular system, to efficiently perform a double precision computation.
- Perform a double precision computation.
- Restore the thread pool settings to their original values.

The first computation will use the thread pool since it does not exceed any of the specified parameters. The second computation, since it exceeds the maximum number of data elements, will not use the thread pool:

```
; Retrieve the current thread pool settings.
threadpool = !CPU

; Modify the thread pool settings.
CPU, TPOOL_MAX_ELTS = 1000000, TPOOL_MIN_ELTS = 50000, $
    TPOOL_NTHREADS = 2

; Create 65,341 elements of floating point data.
theta = FINDGEN(361, 181)

; Perform computation, using 2 threads.
sineSquared = 1. - (COS(!DTOR*theta))^2

; Modify thread pool settings for new data type.
CPU, TPOOL_MAX_ELTS = 50000, TPOOL_MIN_ELTS = 10000
```

```
; Create 65,341 elements of double precision data
theta = DINDGEN(361, 181)

; Perform computation.
sineSquared = 1. - (COS(!DTOR*theta))^2

;Return thread pool settings to their initial values.
CPU, TPOOL_MAX_ELTS = threadpool.TPOOL_MAX_ELTS, $
    TPOOL_MIN_ELTS = threadpool.TPOOL_MIN_ELTS, $
    TPOOL_NTHREADS = threadpool.HW_NCPU
```

See Also

[!CPU, “Controlling the Thread Pool Settings for a Session or Group of Computations”](#) on page 129

DEFINE_MSGBLK

The `DEFINE_MSGBLK` procedure defines and loads a new message block into the currently running IDL session. Once loaded, the `MESSAGE` procedure can be used to issue messages from this block.

A message block is a collection of messages that are loaded into IDL as a single unit. Each block contains the messages required for a specific application. At startup, IDL contains a single internal message block named `IDL_MBLK_CORE`, which contains the standard messages required by the IDL system. Dynamically loadable modules (DLMs) usually define additional message blocks for their own needs when they are loaded. At the IDL programming level, the `DEFINE_MSGBLK` or `DEFINE_MSGBLK_FROM_FILE` procedures can be used to define message blocks. You can use the `HELP, /MESSAGES` command to see the currently defined message blocks.

Syntax

```
DEFINE_MSGBLK, BlockName, ErrorNames, ErrorFormats  
[./IGNORE_DUPLICATE] [,PREFIX = PrefixStr]
```

Arguments

BlockName

A string giving the name of the message block to be defined. Block names must be unique within the IDL system. We recommend that you follow the advice given in [“Advice for Library Authors”](#) in Chapter 12 of the *Building IDL Applications* manual when selecting this name in order to avoid name conflicts. Use of the `PREFIX` keyword is also recommended to enforce a consistent naming convention.

ErrorNames

An array of strings giving the names of the messages to be defined with the message block.

ErrorFormats

An array of strings giving the formats for the messages to be defined with the message block. Each format is matched with the corresponding name in *ErrorNames*. For this reason, *ErrorFormats* should have the same number of elements as *ErrorNames*. We recommend the use of the `PREFIX` keyword to enforce a consistent naming scheme for your messages.

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

Keywords

IGNORE_DUPLICATE

Attempts to define a given *BlockName* more than once in the same IDL session usually cause `DEFINE_MSGBLK` to issue an error and stop execution of the IDL program. Specify `IGNORE_DUPLICATE` to cause `DEFINE_MSGBLK` to quietly ignore attempts to redefine a message block. In this case, no error is issued and execution continues. The original message block remains installed and available for use.

PREFIX

It is a common and recommended practice to give each message name defined in *ErrorNames* a common unique prefix that identifies it as an error from a specific message block. However, specifying this prefix in each entry of *ErrorNames* is tedious and error prone. The `PREFIX` keyword can be used to specify a prefix string that will be added to each element of *ErrorNames*.

Example

This example defines a message block called `ROADRUNNER` that contains 2 messages:

```
name = ['BADPLAN', 'RRNOTCAUGHT']
fmt   = ['Bad plan detected: %s.', 'Road Runner not captured.']
DEFINE_MSGBLK, prefix = 'ACME_M_', 'ROADRUNNER', name, fmt
```

Once this message block is loaded, the `ACME_M_BADPLAN` message can be issued using the following statement:

```
MESSAGE, NAME = 'acme_m_badplan', BLOCK = 'roadrunner', $
    'Exploding bridge while standing underneath'
```

This `MESSAGE` statement produces the output similar to:

```
% Bad plan detected: Exploding bridge while standing underneath.
% Execution halted at: $MAIN$
```

The IDL command:

```
HELP, /STRUCTURES, !ERROR_STATE
```

can be used to examine the effect of this message on IDL’s error state.

See Also

[DEFINE_MSGBLK_FROM_FILE](#), [MESSAGE](#)

DEFINE_MSGBLK_FROM_FILE

The `DEFINE_MSGBLK_FROM_FILE` procedure reads the definition of a message block from a file, and uses `DEFINE_MSGBLK` to load it into the currently running IDL session. Once loaded, the `MESSAGE` procedure can be used to issue messages from this block.

`DEFINE_MSGBLK_FROM_FILE` can be more convenient than `DEFINE_MSGBLK` for large message blocks.

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

Syntax

```
DEFINE_MSGBLK_FROM_FILE, Filename [,BLOCK = BlockName]  
[, /IGNORE_DUPLICATE] [,PREFIX = PrefixStr] [, /VERBOSE]
```

Arguments

Filename

The name of the file containing the message block definition. The contents of this file must be formatted as described in the section [“Format of Message Definition Files”](#) which follows.

Keywords

BLOCK

If present, specifies the name of the message block. Normally, this keyword is not specified, and an `@IDENT` line in the message file specifies the name of the block. We recommend that you follow the advice given in [“Advice for Library Authors”](#) in Chapter 12 of the *Building IDL Applications* manual when selecting this name in order to avoid name clashes. Use of a prefix is also recommended to enforce a consistent naming convention.

IGNORE_DUPLICATE

Attempts to define a given *BlockName* more than once in the same IDL session usually cause `DEFINE_MSGBLK` to issue an error and stop execution of the IDL program. Specify `IGNORE_DUPLICATE` to cause `DEFINE_MSGBLK` to quietly ignore attempts to redefine a message block. In this case, no error is issued and

execution continues. The original message block remains installed and available for use.

PREFIX

If present, specifies a prefix string to be applied to the beginning of each message name in the message block. Normally, this keyword is not specified, and an `@PREFIX` line in the message file specifies the prefix string. We recommend the use of a prefix to enforce a consistent naming scheme for your messages.

VERBOSE

If set, causes `DEFINE_MSGBLK_FROM_FILE` to print informational messages describing the message block loaded.

Format of Message Definition Files

A message definition file has a simple structure designed to ease the specification of message blocks. Any line starting with the character `@` specifies information about the message block. Any line not starting with an `@` character is ignored, and can be used for comments, documentation, notes, or other human readable information. All such text is ignored by `DEFINE_MSGBLK_FROM_FILE`.

There are three different types of lines starting with `@` allowed in a message definition file:

@IDENT name

Specifies the name of the message block being defined. There should be exactly one such line in every message definition file. If the `BLOCK` keyword to `DEFINE_MSGBLK_FROM_FILE` is specified, the `@IDENT` line is ignored and can be omitted. RSI recommends always specifying an `@IDENT` line.

@PREFIX PrefixStr

If present, specifies a prefix string to be applied to the beginning of each message name in the message block. There should be at most one such line in every message definition file. If the `PREFIX` keyword to `DEFINE_MSGBLK_FROM_FILE` is specified, the `@PREFIX` line is ignored and can be omitted. RSI recommends always specifying an `@PREFIX` line.

@ MessageName MessageFormat

Specifies a single message name and format string pair. The format string should be delimited with double quotes. A message definition file should contain one such line for every message it defines.

Example

The following example uses the same message block as in the example given for “[DEFINE_MSGBLK](#)” on page 197, but uses a message definition file to create the message block. The first step is to create a message definition file called `roadrunner.msg` containing the following lines:

```
; Message definition file for ROADRUNNER message block
@IDENT roadrunner
@PREFIX ACME_M_
@      BADPLAN "Bad plan detected: %s."
@      RRNOTCAUGHT "Road Runner not captured."
```

If you are currently in IDL, exit out and restart. Then, within IDL, you can use the following statement to load in the message block:

```
DEFINE_MSGBLK_FROM_FILE, 'roadrunner.msg'
```

Once this message block is loaded, the `ACME_M_BADPLAN` message can be issued using the following statement:

```
MESSAGE, NAME = 'acme_m_badplan', BLOCK='roadrunner', $
      'Exploding bridge while standing underneath'
```

This MESSAGE statement produces the output similar to:

```
% Bad plan detected: Exploding bridge while standing underneath.
% Execution halted at: $MAIN$
```

The IDL command:

```
HELP, /STRUCTURES, !ERROR_STATE
```

can be used to examine the effect of this message on IDL’s error state.

See Also

[DEFINE_MSGBLK](#), [MESSAGE](#)

ERF

The ERF function returns the value of the error function:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

The result is double-precision if the argument is double-precision, otherwise the result is floating-point. The result always has the same structure as *X*. The ERF function does not work with complex arguments.

Syntax

Result = ERF(*X*)

Arguments

X

The expression for which the error function is to be evaluated.

Example

To find the error function of 0.4 and print the result, enter:

```
PRINT, ERF(0.4D)
```

IDL prints:

```
0.42839236
```

See Also

[ERFC](#), [ERFCX](#), [GAMMA](#), [IGAMMA](#), [EXPINT](#)

ERFC

The ERFC function returns the value of the complimentary error function:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$$

The result is double-precision if the argument is double-precision, otherwise the result is floating-point. The result always has the same structure as *X*. The ERFC function does not work with complex arguments.

Syntax

Result = ERFC(*X*)

Arguments

X

The expression for which the complimentary error function is to be evaluated.

Example

To find the complimentary error function of 0.4 and print the result, enter:

```
PRINT, ERFC(0.4D)
```

IDL prints:

```
0.57160764
```

See Also

[ERF](#), [ERFCX](#)

ERFCX

The ERFCX function returns the value of the scaled complimentary error function:

$$erfcx(x) = e^{x^2} erfc(x)$$

The result is double-precision if the argument is double-precision, otherwise the result is floating-point. The result always has the same structure as *X*. The ERFCX function does not work with complex arguments.

Syntax

Result = ERFCX(*X*)

Arguments

X

The expression for which the complimentary error function is to be evaluated.

Example

To find the scaled complimentary error function of 0.4 and print the result, enter:

```
PRINT, ERFCX(0.4D)
```

IDL prints:

```
0.67078779
```

See Also

[ERF](#), [ERFC](#)

FILE_INFO

The FILE_INFO function returns status information about a specified file.

Syntax

Result = FILE_INFO(Path, /NOEXPAND_PATH)

Return Value

The FILE_INFO function returns a structure expression of type FILE_INFO containing status information about a specified file or files. The result will contain one structure for each input element.

Fields of the FILE_INFO Structure

The following descriptions are of fields in the structure returned by the FILE_INFO function. They are not keywords to FILE_INFO.

- **NAME** — The name of the file.
- **EXISTS** — True (1) if the file exists. False (0) if it does not exist.
- **READ** — True (1) if the file exists and is readable by the user. False (0) if it is not readable.
- **WRITE** — True (1) if the file exists and is writable by the user. False (0) if it is not writable.
- **EXECUTE** — True (1) if the file exists and is executable by the user. False (0) if it is not 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.

- **REGULAR** — True (1) if the file exists and is a regular disk file and not a directory, pipe, socket, or other special file type. False (0) if it is not a regular disk file (it maybe a directory, pipe, socket, or other special file type).

- **DIRECTORY** — True (1) if the file exists and is a directory. False (0) if it is not a directory.
- **BLOCK_SPECIAL** — True (1) if the file exists and is a UNIX block special device. On non-UNIX operating systems, this field will always be False (0).
- **CHARACTER_SPECIAL** — True (1) if the file exists and is a UNIX character special device. On non-UNIX operating systems, this field will always be False (0).
- **NAMED_PIPE** — True (1) if the file exists and is a UNIX named pipe (fifo) device. On non-UNIX operating systems, this field will always be False (0).
- **SETGID** — True (1) if the file exists and has its Set-Group-ID bit set. On non-UNIX operating systems, this field will always be False (0).
- **SETUID** — True (1) if the file exists and has its Set-User-ID bit set. On non-UNIX operating systems, this field will always be False (0).
- **SOCKET** — True (1) if the file exists and is a UNIX domain socket. On non-UNIX operating systems, this field will always be False (0).
- **STICKY_BIT** — True (1) if the file exists and has its sticky bit set. On non-UNIX operating systems, this field will always be False (0).
- **SYMLINK** — True (1) if the file exists and is a UNIX symbolic link. On non-UNIX operating systems, this field will always be False (0).
- **DANGLING_SYMLINK** — True (1) if the file exists and is a UNIX symbolic link that points at a non-existent file. On non-UNIX operating systems, this field will always be False (0).
- **ATIME, CTIME, MTIME** — The date of last access, date of creation, and date of last modification given in seconds since 1 January 1970 UTC. Use the `SYSTIME` function to convert these dates into a textual representation.

Note

Some file systems do not maintain all of these dates (e.g. MS DOS FAT file systems), and may return 0. On some non-UNIX operating systems, access time is not maintained, and `ATIME` and `MTIME` will always return the same date.

- **SIZE** — The current length of the file in bytes. If *Path* is not to a regular file (possibly to a directory, pipe, socket, or other special file type), the value of `SIZE` will not contain any useful information.

Arguments

Path

The path of the file about which information is required. This parameter can be a scalar or array of type string.

Keywords

NOEXPAND_PATH

If specified, FILE_INFO uses *Path* exactly as specified, without applying the usual file path expansion.

Examples

To get information on the file `dist.pro` within the IDL User Library:

```
HELP, /STRUCTURE, FILE_INFO(FILEPATH('dist.pro', $
    SUBDIRECTORY = 'lib'))
```

Executing the above command will produce output similar to:

```
** Structure FILE_INFO, 21 tags, length=72:
    NAME                STRING      '/usr/local/rsi/idl/lib/dist.pro'
    EXISTS              BYTE         1
    READ                BYTE         1
    WRITE              BYTE         0
    EXECUTE            BYTE         0
    REGULAR            BYTE         1
    DIRECTORY          BYTE         0
    BLOCK_SPECIAL      BYTE         0
    CHARACTER_SPECIAL  BYTE         0
    NAMED_PIPE         BYTE         0
    SETGID             BYTE         0
    SETUID             BYTE         0
    SOCKET             BYTE         0
    STICKY_BIT         BYTE         0
    SYMLINK            BYTE         0
    DANGLING_SYMLINK   BYTE         0
    MODE               LONG          420
    ATIME              LONG64        970241431
    CTIME              LONG64        970241595
    MTIME              LONG64        969980845
    SIZE               LONG64        1717
```

See Also

[FILE_TEST](#), [FSTAT](#)

FILE_SEARCH

The `FILE_SEARCH` function returns a string array containing the names of all files matching the input path specification. Input path specifications may contain wildcard characters, enabling them to match multiple files. All matched filenames are returned in a string array, one file name per array element. If no files exist with names matching the input arguments, a null scalar string is returned instead of a string array. `FILE_SEARCH` has the ability to perform standard, or recursive searching:

- **Standard:** When called with a single *Path_Specification* argument, `FILE_SEARCH` returns all files that match that specification. This is the same operation, sometimes referred to as *file globbing*, performed by most operating system command interpreters when wildcard characters are used in file specifications.
- **Recursive:** When called with two arguments, `FILE_SEARCH` performs recursive searching of directory hierarchies. In a recursive search, `FILE_SEARCH` looks recursively for any and all subdirectories in the file hierarchy rooted at the *Dir_Specification* argument. Within each of these subdirectories, it returns the names of all files that match the pattern in the *Recur_Pattern* argument. This operation is similar to that performed by the UNIX `find(1)` command.

Note

To avoid going into an infinite loop, the `FILE_SEARCH` routine does not search for files designated by symbolic links.

A relative path is a file path that can only be unambiguously interpreted by basing it relative to some other known location. Usually, this location is the current working directory for the process. A fully qualified path is a complete and unambiguous path that can be interpreted directly. For example, `bin/idl` is a relative path, while `/usr/local/rsi/idl/bin/idl` is a fully qualified path. By default, `FILE_SEARCH` follows the format of the input to decide the form of returned paths. If the input is relative, the results will be relative. If the input is fully qualified, the results will also be fully qualified. If you specify the `FULLY_QUALIFY_PATH` keyword, the results will be fully qualified no matter which form of input is used.

The wildcards understood by `FILE_SEARCH` are based on those used by standard UNIX tools. They are described in the “[Supported Wildcards and Expansions](#)” on page 218.

Note

Research Systems strongly recommends the `FILE_SEARCH` function be used rather than the `FINDFILE` function. `FILE_SEARCH` is ultimately intended as a replacement for `FINDFILE`.

Syntax

Result = `FILE_SEARCH(Path_Specification)`

or for recursive searching,

Result = `FILE_SEARCH(Dir_Specification, Recur_Pattern)`

Keywords: [, `COUNT=variable`] [, `/EXPAND_ENVIRONMENT`] [, `/EXPAND_TILDE`] [, `/FOLD_CASE`] [, `/FULLY_QUALIFY_PATH`] [, `/ISSUE_ACCESS_ERROR`] [, `/MARK_DIRECTORY`] [, `/MATCH_INITIAL_DOT`] [, `/MATCH_ALL_INITIAL_DOT`] [, `/NOSORT`] [, `/QUOTE`] [, `/TEST_DIRECTORY`] [, `/TEST_EXECUTABLE`] [, `/TEST_READ`] [, `/TEST_REGULAR`] [, `/TEST_WRITE`] [, `/TEST_ZERO_LENGTH`]

UNIX-Only Keywords: [, `/TEST_BLOCK_SPECIAL`] [, `/TEST_CHARACTER_SPECIAL`] [, `/TEST_DANGLING_SYMLINK`] [, `/TEST_GROUP`] [, `/TEST_NAMED_PIPE`] [, `/TEST_SETGID`] [, `/TEST_SETUID`] [, `/TEST_SOCKET`] [, `/TEST_STICKY_BIT`] [, `/TEST_SYMLINK`] [, `/TEST_USER`]

Arguments

Any of the arguments described in this section can contain wildcard characters, as described in the Supported Wildcards and Expansions section below.

Path_Specification

A scalar or array variable of string type, containing file paths to match. If *Path_Specification* is not supplied, or if it is supplied as a null string, `FILE_SEARCH` uses a default pattern of `*` and matches all files in the current directory.

Dir_Specification

A scalar or array variable of string type, containing directory paths within which `FILE_SEARCH` will perform recursive searching for files matching the *Recur_Pattern* argument. `FILE_SEARCH` examines *Dir_Specification*, and any directory found below it, and returns the paths of any files in those directories that

match *Recur_Pattern*. If *Dir_Specification* is supplied as a null string, FILE_SEARCH searches the current directory.

Recur_Pattern

A scalar string containing a pattern for files to match in any of the directories specified by the *Dir_Specification* argument. If *Recur_Pattern* is supplied as a null string, FILE_SEARCH uses a default pattern of * and matches all files in the specified directories.

Keywords

COUNT

A named variable into which the number of files found is placed. If no files are found, a value of 0 is returned.

EXPAND_ENVIRONMENT

By default, FILE_SEARCH follows the conventions of the underlying operating system to determine if it expands environment variable references in input file specification patterns. The default is to do such expansions under UNIX, and not to do them on the Macintosh or Microsoft Windows. The EXPAND_ENVIRONMENT keyword is used to change this behavior. Set it to a non-zero value to cause FILE_SEARCH to perform environment variable expansion on all platforms. Set it to zero to disable such expansion.

Note

Macintosh users should note that the Macintosh operating system does not support the concept of an environment, and as such, environment variable expansion is likely to be of little use. One significant exception to this is to use the IDL_TMPDIR environment variable to generate paths to temporary files. See the description of the GETENV function for further details.

The syntax for expanding environment variables in an input file pattern is based on that supported by the standard UNIX shell (/bin/sh), as described in the [Supported Wildcards and Expansions](#) section below.

EXPAND_TILDE

Users of the UNIX C-shell (/bin/csh), and other tools influenced by it, are familiar with the use of a tilde (~) character at the beginning of a path to denote a home directory. A tilde by itself at the beginning of the path (e.g. ~/directory/file) is

equivalent to the home directory of the user executing the command, while a tilde followed by the name of a user (e.g. `~user/directory/file`) is expanded to the home directory of the named user.

By default, `FILE_SEARCH` follows the conventions of the underlying operating system in deciding whether to expand a leading tilde or to treat it as a literal character. Hence, the default is to expand them under UNIX, and not on Macintosh or Microsoft Windows. The `EXPAND_TILDE` keyword is used to change this behavior.

Set it to zero to disable tilde expansion on all platforms. Set it to a non-zero value to enable tilde expansion.

Note

Microsoft Windows users should note that only the plain form of tilde is recognized by Windows IDL. Attempts to use the `~user` form will cause IDL to issue an error. IDL uses the `HOME` and `HOMEPath` environment variables to obtain a home directory for the current Windows user.

Note

Macintosh users should note that the `FILE_SEARCH` quietly ignores the `EXPAND_TILDE` keyword. There is no support for tilde expansion on that platform.

FOLD_CASE

By default, `FILE_SEARCH` follows the case sensitivity policy of the underlying operating system. Matches are case sensitive on UNIX platforms, and case insensitive on Macintosh and Microsoft Windows platforms. The `FOLD_CASE` keyword is used to change this behavior. Set it to a non-zero value to cause `FILE_SEARCH` to do all file matching case insensitively. Set to zero to cause all file matching to be case sensitive.

FULLY_QUALIFY_PATH

If set, `FILE_SEARCH` expands all returned file paths so that they are complete. Under UNIX, this means that all files are specified relative to the root of the file system. On Macintosh and Windows platforms, it means that all files are specified relative to the Drive/Volume on which they are located. By default, `FILE_SEARCH` returns fully qualified paths when the input specification is fully qualified, and returns relative paths otherwise. For example:

```
CD, '/usr/local/rsi/idl/bin'
PRINT, FILE_SEARCH('idl')
idl
PRINT, FILE_SEARCH('idl',/FULLY_QUALIFY_PATH)
/usr/local/rsi/idl/bin/idl
```

Under Microsoft Windows, any use of a drive letter colon (:) character implies full qualification, even if the path following the colon does not start with a slash character.

ISSUE_ACCESS_ERROR

If the IDL process lacks the necessary permission to access a directory included in the input specification, `FILE_SEARCH` will normally skip over it quietly and not include it in the generated results. Set `ISSUE_ACCESS_ERROR` to cause an error to be issued instead.

MARK_DIRECTORY

If set, all directory paths are returned with a path separator character appended to the end. This allows the caller to concatenate a file name directly to the end without having to supply a separator character first. This is convenient for cross-platform programming, as the separator characters differ between operating systems:

```
PRINT, FILE_SEARCH(!DIR)
/usr/local/rsi/idl
PRINT, FILE_SEARCH(!DIR, /MARK_DIRECTORY)
/usr/local/rsi/idl/
```

MATCH_ALL_INITIAL_DOT

By default, wildcards do not match leading dot (.) characters, and `FILE_SEARCH` does not return the names of files that start with the dot (.) character unless the leading dot is actually contained within the search string. Set `MATCH_ALL_INITIAL_DOT` to change this policy so that wildcards will match all files starting with a dot, including the special “.” (current directory) and “..” (parent directory) entries. RSI recommends the use of the `MATCH_INITIAL_DOT` keyword instead of `MATCH_ALL_INITIAL_DOT` for most purposes.

MATCH_INITIAL_DOT

`MATCH_INITIAL_DOT` serves the same function as `MATCH_ALL_INITIAL_DOT`, except that the special “.” (current directory) and “..” (parent directory) directories are not included.

NOSORT

If set, `FILE_SEARCH` will not sort the resulting files. On some operating systems, particularly UNIX, this can make `FILE_SEARCH` execute faster. By default, `FILE_SEARCH` sorts the results from each element of the input file specification together, and places the results from each input element into the result in the order they are found. Hence, the statement:

```
Result = FILE_SEARCH(['*.c', '*.h'])
```

returns all of the C files in the current directory in lexical order, followed by all of the H files, also sorted lexically among themselves. In contrast, the statement:

```
Result = FILE_SEARCH('*. [ch]')
```

returns all of the C and H files sorted together into lexical order. This version is more efficient than the previous one, because the directory is only searched once.

QUOTE

`FILE_SEARCH` usually treats all wildcards found in the input specification as having the special meanings described in [“Supported Wildcards and Expansions”](#) on page 218. This means that such characters cannot normally be used as plain literal characters in file names. For example, it is not possible to match a file that contains a literal asterisk character in its name because asterisk is interpreted as the “match zero or more characters” wildcard.

If the `QUOTE` keyword is set, the backslash character can be used to escape any character so that it is treated as a plain character with no special meaning. In this mode, `FILE_SEARCH` replaces any two character sequence starting with a backslash with the second character of the pair. In the process, any special wildcard meaning that character might have had disappears, and the character is treated as a literal.

If `QUOTE` is set, any literal backslash characters in your path must themselves be escaped with a backslash character. This is especially important for Microsoft Windows users, because the directory separator character for that platform is the backslash. Windows IDL also accepts UNIX-style forward slashes for directory separators, so Windows users have two choices in handling this issue:

```
Result = FILE_SEARCH('C:\\home\\bob\\*.dat', /QUOTE)  
Result = FILE_SEARCH('C:/home/bob/*.dat', /QUOTE)
```

TEST_DIRECTORY

Only include a matching file if it is a directory.

TEST_EXECUTABLE

Only include a matching file if it is executable. The source of this information differs between operating systems:

UNIX: 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.

TEST_READ

Only include a matching file if it is readable by the user.

Note

This keyword does not support Access Control Listing (ACL) settings for files.

TEST_REGULAR

Only include a matching file if it is a regular disk file and not a directory, pipe, socket, or other special file type.

TEST_WRITE

Only include a matching file if it is writable by the user.

Note

This keyword does not support Access Control Listing (ACL) settings for files.

TEST_ZERO_LENGTH

Only include a matching file if it 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 `TEST_ZERO_LENGTH` keyword on directories, as the information returned cannot be used in a meaningful way.

UNIX-Only Keywords

TEST_BLOCK_SPECIAL

Only include a matching file if it is a block special device.

TEST_CHARACTER_SPECIAL

Only include a matching file if it is a character special device.

TEST_DANGLING_SYMLINK

Only include a matching file if it is a symbolic link that points at a non-existent file.

TEST_GROUP

Only include a matching file if it belongs to the same effective group ID (GID) as the IDL process.

TEST_NAMED_PIPE

Only include a matching file if it is a named pipe (fifo) device.

TEST_SETGID

Only include a matching file if it has its Set-Group-ID bit set.

TEST_SETUID

Only include a matching file if it has its Set-User-ID bit set.

TEST_SOCKET

Only include a matching file if it is a UNIX domain socket.

TEST_STICKY_BIT

Only include a matching file if it has its sticky bit set.

TEST_SYMLINK

Only include a matching file if it is a symbolic link that points at an existing file.

TEST_USER

Only include a matching file if it belongs to the same effective user ID (UID) as the IDL process.

Supported Wildcards and Expansions

The wildcards understood by `FILE_SEARCH` are based on those used by the standard UNIX shell `/bin/sh` (`*?[]`, environment variables) with some enhancements commonly found in the C-shell `/bin/csh` (`~` and `{}`). These wildcards are processed identically across all IDL supported platforms. The supported wildcards are shown in the following table:

Wildcard	Description
<code>*</code>	Matches any string, including the null string.
<code>?</code>	Matches any single character.
<code>[...]</code>	Matches any one of the enclosed characters. A pair of characters separated by “-” matches any character lexically between the pair, inclusive. If the first character following the opening bracket (<code>[</code>) is a <code>!</code> or <code>^</code> , any character not enclosed is matched.
<code>{str, str, ...}</code>	Expand to each string (or filename-matching pattern) in the comma-separated list.
<code>~</code> <code>~user</code>	If used at start of input file specification, is replaced with the path to the appropriate home directory. See the description of the <code>EXPAND_TILDE</code> keyword for details.
<code>\$var</code>	Replace with value of named environment variable. See the description of the <code>EXPAND_ENVIRONMENT</code> keyword for full details.
<code>\${var}</code>	Replace <code>\${var}</code> with the value of the <code>var</code> environment variable. If <code>var</code> is not found in the environment, <code>\${var}</code> is replaced with a null string. This format is useful when the environment variable reference sits directly next to unrelated text, as the use of the <code>{}</code> brackets make it possible for IDL to determine where the environment variable ends and the remaining text starts (e.g. <code>\${mydir} other text</code>).
<code>\${var:-alttext}</code>	If environment variable <code>var</code> is present in the environment and has a non-NULL value, then substitute that value. If <code>var</code> is not present, or has a NULL value, then substitute the alternative text (<code>alttext</code>) provided instead.

Table 6-1: Supported Wildcards and Expansions

Wildcard	Description
<code>\${var-alttext}</code>	If environment variable <code>var</code> is present in the environment (even if it has a NULL value) then substitute that value. If <code>var</code> is not present, then substitute the alternative text (<code>alttext</code>) provided instead.

Table 6-1: Supported Wildcards and Expansions (Continued)

These wildcards can appear anywhere in an input file specification, with the following exceptions:

Tilde (~)

The tilde character is only considered to be a wildcard if it is the first character in the input file specification, and only if allowed by the `EXPAND_TILDE` keyword. Otherwise, it is treated as a regular character.

Microsoft Windows UNC Paths

On a local area network, Microsoft Windows offers an alternative to the drive letter syntax for accessing files. The Universal Naming Convention allows specifying paths on other hosts, using the syntax:

```
\\hostname\sharename\dir\dir\file
```

UNC paths are distinguished from normal paths by the use of two initial slashes in the path. `FILE_SEARCH` can process such paths, but wildcard characters are not allowed in the `hostname` or `sharename` segments. Wildcards are allowed for specifying directories and files. For performance reasons, RSI does not recommend using the recursive form of `FILE_SEARCH` with UNC paths on very large directory trees.

When using `FILE_SEARCH`, you should be aware of the following issues:

Initial Dot Character

The default is for wildcards not to match the dot (.) character if it occurs as the first character of a directory or file name. This follows the convention of UNIX shells, which treat such names as hidden files. In order to match such files, you can take any of the following actions:

- Explicitly include the dot character at the start of your pattern (e.g. `“.*”`).

- Specify the `MATCH_INITIAL_DOT` keyword, which changes the dot matching policy so that wildcards will match any names starting with dot (except for the special “.” and “..” directories).
- Specify the `MATCH_ALL_INITIAL_DOT` keyword, which changes the dot matching policy so that wildcards will match any names starting with dot (including the special “.” and “..” directories).

File Path Syntax

The syntax allowed for file paths differs between operating systems. `FILE_SEARCH` always processes file paths using the syntax rules for the platform on which the IDL session is running. As a convenience for Microsoft Windows users, Windows IDL accepts UNIX style forward slashes as well as the usual backslashes as path separators.

Differing Defaults Between Platforms

The different operating systems supported by IDL have some conventions for processing file paths that are inherently incompatible. If `FILE_SEARCH` attempted to force an identical default policy for these features across all platforms, the resulting routine would be inconvenient to use on all platforms. `FILE_SEARCH` resolves this inherent tension between convenience and control in the following way:

- These features are controlled by keywords which are listed in the table below. If a keyword is not explicitly specified, `FILE_SEARCH` will determine an appropriate default for that feature based on the conventions of the underlying operating system. Hence, `FILE_SEARCH` will by default behave in a way that is reasonable on the platform it is used on.
- If one of these keywords is explicitly specified, `FILE_SEARCH` will use its value to determine support for that feature. Hence, if the keyword is used, `FILE_SEARCH` will behave identically on all platforms. If maximum cross-platform control is desired, you can achieve it by specifying all the relevant keywords.

The keywords that have different defaults on different platforms are listed in the following table:

Wildcard	Keyword	Default Mac	Default UNIX	Default Win
\$var \${var} \${var:-alttext} \${var-alttext}	EXPAND_ENVIRONMENT	no	yes	no
~	EXPAND_TILDE	no	yes	no
	FOLD_CASE	yes	no	yes

Table 6-2: Differing Defaults on Different Platforms

TEST_* Keywords

The keywords with names that start with the TEST_ prefix allow you to filter the list of resulting file paths based on various criteria. If you remove the TEST_ prefix from these keywords, they correspond directly to the same keywords to the FILE_TEST function, and are internally implemented by the same test code. One could therefore use FILE_TEST instead of the TEST_ keywords to FILE_SEARCH. For example, the following statement locates all subdirectories of the current directory:

```
Result = FILE_SEARCH(/TEST_DIRECTORY)
```

It is equivalent to the following statements, using FILE_TEST:

```
result = FILE_SEARCH()
idx = where(FILE_TEST(result, /DIRECTORY), count)
result = (count eq 0) ? '' : result[idx]
```

The TEST_* keywords are more succinct, and can be more efficient in the common case in which FILE_SEARCH generates a long list of results, only to have FILE_TEST discard most of them.

Examples

Example 1

Find all files in the current working directory:

```
Result = FILE_SEARCH()
```

Example 2

Find all IDL program (*.pro) files in the current working directory:

```
Result = FILE_SEARCH('*.pro')
```

Example 3

Under Microsoft Windows, find all files in the top level directories of all drives other than the floppy drives:

```
Result=FILE_SEARCH('[!ab]:*')
```

This example relies on the following:

- FILE_SEARCH allows wildcards within the drive letter part of an input file specification.
- Drives A and B are always floppies, and are not used by Windows for any other type of drive.

Example 4

Find all files in the user's home directory that start with the letters A-D. Match both upper and lowercase letters:

```
Result = FILE_SEARCH('~/[a-d]*', /EXPAND_TILDE, /FOLD_CASE)
```

Example 5

Find all directories in the user's home directory that start with the letters A-D. Match both upper and lowercase letters:

```
Result = FILE_SEARCH('~/[a-d]*', /EXPAND_TILDE, /FOLD_CASE, $  
/TEST_DIRECTORY)
```

Example 6

Recursively find all subdirectories found underneath the user's home directory that do not start with a dot character:

```
Result = FILE_SEARCH('$HOME', '*', /EXPAND_ENVIRONMENT, $  
/TEST_DIRECTORY)
```

Example 7

Recursively find all subdirectories found underneath the user's home directory, including those that start with a dot character, but excluding the special "." and ".." directories:

```
Result = FILE_SEARCH('$HOME', '*', /MATCH_INITIAL_DOT, $  
/EXPAND_ENVIRONMENT, /TEST_DIRECTORY)
```

Example 8

Find all .pro and .sav files in a UNIX IDL library search path, sorted by directory, in the order IDL searches for them:

```
Result = FILE_SEARCH(STRSPLIT(!PATH, ':', /EXTRACT) + $  
'/*.{pro,sav}')
```

Colon (:) is the UNIX path separator character, so the call to `STRSPLIT` breaks the IDL search path into an array of directories. To each directory name, we concatenate the wildcards necessary to match any .pro or .sav files in that directory. When this array is passed to `FILE_SEARCH`, it locates all files that match these specifications. `FILE_SEARCH` sorts all of the files found by each input string. The files for each string are then placed into the output array in the order they were searched for.

Example 9

Recursively find all directories in your IDL distribution:

```
Result = FILE_SEARCH(!DIR, '*', /TEST_DIRECTORY)
```

See Also

[FILE_TEST](#), [FILEPATH](#), [FINDFILE](#), [GETENV](#)

GRID_INPUT

The GRID_INPUT procedure preprocesses and sorts two-dimensional scattered data points, and removes duplicate values. This procedure is also used for converting spherical coordinates to Cartesian coordinates.

Syntax

```
GRID_INPUT, X, Y, F, X1, Y1, F1 [, Duplicates=string ] [, EPSILON=value ]  
[, EXCLUDE=vector ]
```

or

```
GRID_INPUT, Lon, Lat, F, XYZ, F1, /SPHERE [, /DEGREES]  
[, Duplicates=string ] [, EPSILON=value ] [, EXCLUDE=vector ]
```

or

```
GRID_INPUT, R, Theta, F, X1, Y1, F1, /POLAR [, /DEGREES]  
[, Duplicates=string ] [, EPSILON=value ] [, EXCLUDE=vector ]
```

Arguments

X, Y

These are input arguments for scattered data points, where *X*, and *Y* are location. All of these arguments are N point vectors.

F

The function value at each location in the form of an N point vector.

Lon, Lat

These are input arguments representing scattered data points on a sphere, specifying location (longitude and latitude). All are N point vectors. *Lon*, *Lat* are in degrees or radians (default).

R, Theta

These are scattered data point input arguments representing the *R* and *Theta* polar coordinate location in degrees or radians (default). All arguments are N point vectors.

X1, Y1, F1

These output arguments are processed and sorted single precision floating point data which are passed as the input points to the GRIDDATA function.

Xyz

Upon return, a named variable that contains a 3-by-*n* array of Cartesian coordinates representing points on a sphere.

Keywords

DEGREES

By default, all angular inputs and keywords are assumed to be in radian units. Set the DEGREES keyword to change the angular input units to degrees.

DUPLICATES

Set this keyword to a string indicating how duplicate data points are handled per the following table. The case (upper or lower) is ignored. The default setting for DUPLICATES is “First”.

String	Meaning
“First”	Retain only the first encounter of the duplicate locations.
“Last”	Retain only the last encounter of the duplicate locations.
“All”	Retains all locations, which is invalid for any gridding technique that requires a TRIANGULATION. Some methods, such as Inverse Distance or Polynomial Regression with no search criteria can handle duplicates.
“Avg”	Retain the average F value of the duplicate locations.
“Midrange”	Retain the average of the minimum and maximum duplicate locations $((\text{Max}(F) + \text{Min}(F)) / 2)$.
“Min”	Retain the minimum of the duplicate locations ($\text{Min}(F)$).
“Max”	Retain the maximum of the duplicate locations ($\text{Max}(F)$).

EPSILON

The tolerance for finding duplicates. Points within EPSILON distance of each other are considered duplicates. For spherical coordinates, EPSILON is in units of angular distance, as set by the DEGREES keyword.

EXCLUDE

An N -point vector specifying the indices of the points to exclude.

POLAR

Set to indicate inputs are in polar coordinates.

SPHERE

Set to indicate inputs are in spherical coordinates. In this case, the output argument *Xyz* is set to a 3-by- n array containing the spherical coordinates converted to 3-dimensional Cartesian points on a sphere.

Example

The following example uses the data from the `irreg_grid1.txt` ASCII file. This file contains scattered elevation data of a model of an inlet. This scattered elevation data contains two duplicate locations. The `GRID_INPUT` procedure is used to omit the duplicate locations.

```
; Import the Data:

; Determine the path to the file.
file = FILEPATH('irreg_grid1.txt', $
    SUBDIRECTORY = ['examples', 'data'])

; Import the data from the file into a structure.
dataStructure = READ_ASCII(file)

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)

; Initialize the variables of this example from
; the imported array.
x = dataArray[:, 0]
y = dataArray[:, 1]
data = dataArray[:, 2]

; Display the Data:
```

```

; Scale the data to range from 1 to 253 so a color table can be
; applied. The values of 0, 254, and 255 are reserved as outliers.
scaled = BYTSCL(data, TOP = !D.TABLE_SIZE - 4) + 1B

; Load the color table. If you are on a TrueColor, set the
; DECOMPOSED keyword to the DEVICE command before running a
; color table related routine.
DEVICE, DECOMPOSED = 0
LOADCT, 38

; Open a display window and plot the data points.
WINDOW, 0
PLOT, x, y, /XSTYLE, /YSTYLE, LINESTYLE = 1, $
    TITLE = 'Original Data, Scaled (1 to 253)', $
    XTITLE = 'x', YTITLE = 'y'

; Now display the data values with respect to the color table.
FOR i = 0L, (N_ELEMENTS(x) - 1) DO PLOTS, x[i], y[i], PSYM = -1, $
    SYMSIZE = 2., COLOR = scaled[i]

; Preprocess and sort the data. GRID_INPUT will
; remove any duplicate locations.
GRID_INPUT, x, y, data, xSorted, ySorted, dataSorted

; Display the results from GRID_INPUT:

; Scale the resulting data.
scaled = BYTSCL(dataSorted, TOP = !D.TABLE_SIZE - 4) + 1B

; Open a display window and plot the resulting data points.
WINDOW, 1
PLOT, xSorted, ySorted, /XSTYLE, /YSTYLE, LINESTYLE = 1, $
    TITLE = 'The Data Preprocessed and Sorted, Scaled (1 to 253)', $
    XTITLE = 'x', YTITLE = 'y'

; Now display the resulting data values with respect to the color
; table.
FOR i = 0L, (N_ELEMENTS(xSorted) - 1) DO PLOTS, $
    xSorted[i], ySorted[i], PSYM = -1, COLOR = scaled[i], $
    SYMSIZE = 2.

```

See Also

[GRIDDATA](#)

GRIDDATA

The GRIDDATA function interpolates scattered data values and locations sampled on a plane or a sphere to a regular grid. This is accomplished using one of several available methods. The function result is a two-dimensional floating point array. Computations are performed in single precision floating point. Interpolation methods supported by this function are as follows:

- Inverse Distance (default)
- Kriging
- Linear
- Minimum Curvature
- Modified Shepard's
- Natural Neighbor
- Nearest Neighbor
- Polynomial Regression
- Quintic
- Radial Basis Function

Syntax

Interleaved

Result = GRIDDATA(*X*, *F*)

Planar

Result = GRIDDATA(*X*, *Y*, *F*)

Sphere From Cartesian Coordinates

Result = GRIDDATA(*X*, *Y*, *Z*, *F*, /SPHERE)

Sphere From Spherical Coordinates

Result = GRIDDATA(*Lon*, *Lat*, *F*, /SPHERE)

Inverse Distance Keywords:

[, METHOD='InverseDistance' | /INVERSE_DISTANCE]
 [, ANISOTROPY=*vector*] [, /DEGREES] [, DELTA=*vector*]
 [, DIMENSION=*vector*] [, TRIANGLES=*array* [, EMPTY_SECTORS=*value*]
 [, MAX_PER_SECTOR=*value*] [, MIN_POINTS=*value*]
 [, SEARCH_ELLIPSE=*vector*] [, FAULT_POLYGONS=*vector*]
 [, FAULT_XY=*array*] [, /GRID, XOUT=*vector*, YOUT=*vector*]
 [, MISSING=*value*] [, POWER=*value*] [, SECTORS={ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 }]
 [, SMOOTHING=*value*] [, /SPHERE] [, START=*vector*]

Kriging Keywords: METHOD='Kriging' | /KRIGING [, ANISOTROPY=*vector*]
 [, DELTA=*vector*] [, DIMENSION=*vector*]
 [, TRIANGLES=*array* [, EMPTY_SECTORS=*value*]
 [, MAX_PER_SECTOR=*value*] [, MIN_POINTS=*value*]
 [, SEARCH_ELLIPSE=*vector*]] [, FAULT_POLYGONS=*vector*]
 [, FAULT_XY=*array*] [, /GRID, XOUT=*vector*, YOUT=*vector*]
 [, MISSING=*value*] [, SECTORS={1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 }] [, /SPHERE]
 [, START=*vector*] [, VARIOGRAM=*vector*]

Linear Interpolation Keywords:

METHOD='Linear' | /LINEAR [, TRIANGLES=*array* [, DELTA=*vector*]
 [, DIMENSION=*vector*] [, /GRID, XOUT=*vector*, YOUT=*vector*]
 [, MISSING=*value*] [, START=*vector*]

Minimum Curvature Keywords:

METHOD='MinimumCurvature' | /MIN_CURVATURE [, DELTA=*vector*]
 [, DIMENSION=*vector*] [, START=*vector*]

Modified Shepard's Keywords: METHOD='ModifiedShepards' | /SHEPARDS,
 TRIANGLES=*array* [, ANISOTROPY=*vector*] [, DELTA=*vector*]
 [, DIMENSION=*vector*] [, EMPTY_SECTORS=*value*]
 [, FAULT_POLYGONS=*vector*] [, FAULT_XY=*array*] [, /GRID, XOUT=*vector*,
 YOUT=*vector*] [, MAX_PER_SECTOR=*value*] [, MIN_POINTS=*value*]
 [, MISSING=*value*] [, NEIGHBORHOOD=*array*] [, SEARCH_ELLIPSE=*vector*]
 [, SECTORS={1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 }] [, START=*vector*]

Natural Neighbor Keywords:

METHOD='NaturalNeighbor' | /NATURAL_NEIGHBOR, TRIANGLES=*array*
 [, /DEGREES] [, DELTA=*vector*] [, DIMENSION=*vector*]
 [, /GRID, XOUT=*vector*, YOUT=*vector*] [, MISSING=*value*]
 [, /SPHERE] [, START=*vector*]

Nearest Neighbor Keywords:

METHOD='NearestNeighbor' | /NEAREST_NEIGHBOR, TRIANGLES=*array*
 [, /DEGREES] [, DELTA=*vector*] [, DIMENSION=*vector*]
 [, FAULT_POLYGONS=*vector*] [, FAULT_XY=*array*] [, /GRID, XOUT=*vector*,
 YOUT=*vector*] [, MISSING=*value*] [, /SPHERE] [, START=*vector*]

Polynomial Regression Keywords:

```
METHOD='PolynomialRegression' | /POLYNOMIAL_REGRESSION,
[, DELTA=vector ] [, DIMENSION=vector ]
[, TRIANGLES=array [, EMPTY_SECTORS=value ]
[, MAX_PER_SECTOR=value ] [, MIN_POINTS=value ]
[, SEARCH_ELLIPSE=vector ] [, FAULT_POLYGONS=vector ]
[, FAULT_XY=array ] [, /GRID, XOUT=vector, YOUT=vector ]
[, MISSING=value ] [, POWER=value ] [, SECTORS={ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 } ]
[, START=vector ]
```

Quintic Keywords: METHOD='Quintic' | /QUINTIC, TRIANGLES=array

```
[, DELTA=vector ] [, DIMENSION=vector ] [, MISSING=value ]
[, START=vector ]
```

Radial Basis Function Keywords:

```
METHOD='RadialBasisFunction' | /RADIAL_BASIS_FUNCTION,
[, ANISOTROPY=vector ] [, /DEGREES ] [, DELTA=vector ]
[, DIMENSION=vector ] [, TRIANGLES=array [, EMPTY_SECTORS=value ]
[, MAX_PER_SECTOR=value ] [, MIN_POINTS=value ]
[, SEARCH_ELLIPSE=vector ] [, FAULT_POLYGONS=vector ]
[, FAULT_XY=array ] [, FUNCTION_TYPE={ 0 | 1 | 2 | 3 | 4 } ]
[, /GRID, XOUT=vector, YOUT=vector ] [, MISSING=value ]
[, SECTORS={ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 } ] [, SMOOTHING=value ] [, /SPHERE]
[, START=vector ]
```

Return Value

Result is a two-dimensional floating point array. Computations are preformed in single precision floating point.

Arguments

X [, Y [, Z]]

The point locations. If only one input coordinate parameter is supplied, the points are interleaved; for the Cartesian coordinate system the points are 2-by-*n* dimensions; and 3-by-*n* for a sphere in Cartesian coordinates.

F

The function value at each location in the form of an *n*-point vector.

Lon, Lat

These arguments contain the locations (on a sphere) of the data points (similar to *X*, and *Y*) but are in degrees or radians (default) depending on the use of the keyword **DEGREES**.

Keywords

ANISOTROPY

This keyword is a vector describing an ellipse (see the description for the **SEARCH_ELLIPSE** keyword). All points on the circumference of the ellipse have an equal influence on a point at the center of the ellipse.

For example, assume that atmospheric data are being interpolated, with one dimension being altitude, and the other dimension representing distance from a point. If the vertical mixing is half that of the horizontal mixing, a point 100 units from an interpolate and at the same level has the same influence as a point 50 units above or below the interpolate at the same horizontal location. This effect requires setting the **ANISOTROPY** keyword to `[2, 1, 0]` which forms an ellipse with an *X*-axis length twice as long as its *Y*-axis length.

DEGREES

By default, all angular inputs and keywords are assumed to be in radian units. Set the **DEGREES** keyword to change the angular input units to degrees.

DELTA

A two-element array specifying the grid spacing in *X*, and *Y*. If this keyword is not specified, then the grid spacing is determined from the values of the **DIMENSION** and **START** keywords. These keywords have default values of 25 and `[min(x), min(y)]`, respectively. The spacing derived from these keywords creates a grid of **DIMENSION** cells, enclosing a rectangle from **START**, to `[max(x), max(y)]`. This keyword can also be set to a scalar value to be used for the grid size in both *X* and *Y*.

This keyword is ignored if the **GRID**, **XOUT** and **YOUT** keywords are specified.

DIMENSION

A two element array specifying the grid dimensions in *X* and *Y*. Default value is 25 for each dimension. This keyword can also be set to a scalar value to be used for the grid spacing in both *X* and *Y*.

This keyword is ignored if the **GRID**, **XOUT** and **YOUT** keywords are specified.

EMPTY_SECTORS

This keyword defines the search rules for the maximum number of sectors that may be empty when interpolating at each point. If this number or more sectors contain no data points, considering the search ellipse and/or the fault polygons, the resulting interpolant is the missing data value.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

FAULT_POLYGONS

Set this keyword to an array containing one or more polygon descriptions. A polygon description is an integer or longword array of the form: $[n, i_0, i_1, \dots, i_{n-1}]$, where n is the number of vertices that define the polygon, and $i_0 \dots i_{n-1}$ are indices into the FAULT_XY vertices. The FAULT_POLYGON array may contain multiple polygon descriptions that have been concatenated. To have this keyword ignore an entry in the FAULT_POLYGONS array, set the vertex count, n , and all associated indices to 0. To end the drawing list, even if additional array space is available, set n to -1 . If this keyword is not specified, a single connected polygon is generated from FAULT_XY.

Note

FAULT_POLYGONS are not supported with spherical gridding.

FAULT_XY

The a 2-by- n array specifying the coordinates of points on the fault lines/polygons.

Note

FAULT_XY is not supported with spherical gridding.

FUNCTION_TYPE

Note

This keyword is only used with the Radial Basis Function method of interpolation.

Set this keyword to one of the values shown in the following table to indicate which basis function to use. Default is 0, the Inverse Multiquadric function.

Value	Function Type Used	Equation
0	Inverse Multiquadric	$B(h) = 1/(\sqrt{h^2 + R^2})$
1	Multilog	$B(h) = \log(h^2 + R^2)$
2	Multiquadric	$B(h) = \sqrt{h^2 + R^2}$
3	Natural Cubic Spline	$B(h) = (h^2 + R^2)^{3/2}$
4	Thin Plate Spline	$B(h) = (h^2 + R^2)\log(h^2 + R^2)$

Note - In the equations, h = the anisotropically scaled distance from the interpolant to the node, and R^2 = the value of the SMOOTHING keyword.

GRID

The GRID keyword controls how the XOUT and YOUT vectors specify where interpolates are desired.

If GRID is set, XOUT and YOUT must also be specified. Interpolation is performed on a regular or irregular grid specified by the vectors XOUT with m elements and YOUT with n elements. The *Result* is an m -by- n grid with point $[i, j]$ resulting from the interpolation at $(XOUT[i], YOUT[j])$. When XOUT and YOUT are used, the DELTA, DIMENSION and START keywords are ignored.

INVERSE_DISTANCE

Selects the Inverse Distance method of interpolation.

KRIGING

Selects the Kriging method of interpolation. The variogram type for the Kriging method is set by default, however the VARIOGRAM keyword can be used to set variogram parameters.

LINEAR

Selects the Linear method of interpolation. The TRIANGLES keyword is required when the LINEAR keyword is used.

MAX_PER_SECTOR

This keyword defines the search rules for the maximum number of data points to include in each sector when interpolating. Search rules effectively limit the number of data points used in computing each interpolate. For example, to use the nearest n nodes to compute each interpolant, specify `MAX_PER_SECTOR = n` and use the TRIANGLES keyword.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

METHOD

A string containing one of the method names as shown in the following table. The default for METHOD is “InverseDistance”.

Note

The interpolation method can be chosen using the METHOD keyword set to the specific string, or by setting the corresponding method name keyword.

Note

There are no spaces between words in the method strings and the strings are case insensitive.

Method String	Meaning
“InverseDistance”	Data points closer to the grid points have more effect than those which are further away.

Method String	Meaning
“Kriging”	Data points and their spatial variance are used to determine trends which are applied to the grid points.
“Linear”	Grid points are linearly interpolated from triangles formed by Delaunay triangulation.
“MinimumCurvature”	A plane of grid points is conformed to the data points while trying to minimize the amount of bending in the plane.
“ModifiedShepards”	Inverse Distance weighted with the least squares method.
“NaturalNeighbor”	Each interpolant is a linear combination of the three vertices of its enclosing Delaunay triangle and their adjacent vertices.
“NearestNeighbor”	The grid points have the same value as the nearest data point.
“PolynomialRegression”	Each interpolant is a least-squares fit of a polynomial in X and Y of the specified power to the specified data points.
“Quintic”	Grid points are interpolated with quintic polynomials from triangles formed by Delaunay triangulation.
“RadialBasisFunction”	The effects of data points are weighted by a function of their radial distance from a grid point.

MIN_CURVATURE

Selects the Minimum Curvature method of interpolation.

MIN_POINTS

If fewer than this number of data points are encountered in all sectors, the value of the resulting grid point is set to the value of the MISSING keyword.

The MIN_POINTS keyword also indicates the number of closest points used for each local fit, if SEARCH_ELLIPSE isn't specified.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

MISSING

Set this keyword to the value to use for missing data values. Default is 0.

NATURAL_NEIGHBOR

Selects the Natural Neighbor method of interpolation.

Note

The TRIANGLES keyword is required when the NATURAL_NEIGHBOR keyword is used.

NEAREST_NEIGHBOR

Selects the Nearest Neighbor method of interpolation.

Note

The TRIANGLES keyword is required when the NEAREST_NEIGHBOR keyword is used.

NEIGHBORHOOD

Note

The NEIGHBORHOOD keyword is only used for the Modified Shepard's method of interpolation.

A two-element array, $[Nq, Nw]$ defining the quadratic fit, Nq , and weighting, Nw , neighborhood sizes for the Modified Shepard's method. The default for Nq is the smaller of 13 and the number of points minus 1, with a minimum of 5. The default for Nw is the smaller of 19 and the number of points. The Modified Shepard's method first computes the coefficients of a quadratic fit for each input point, using its Nq closest neighbors.

When interpolating an output point, the quadratic fits from the N_w closest input points are weighted inversely by a function of distance and then combined. The size of the neighborhood used for Shepard's method interpolation may also be specified by the search rules keywords.

POLYNOMIAL_REGRESSION

Selects the Polynomial Regression method for interpolation. The power of the polynomial regression is set to 2 by default, however the **POWER** keyword can be used to change the power to 1 or 3.

The function fit to each interpolant corresponding to the **POWER** keyword set equal to 1, 2 (default), and 3 respectively is as follows:

$$F(x,y) = a_0 + a_1x + a_2y$$

$$F(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4y^2 + a_5xy \quad (\text{default})$$

$$F(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4y^2 + a_5xy + a_6x^3 + a_7y^3 + a_8x^2y + a_9xy^2$$

By inspection, a minimum of three data points are required to fit the linear polynomial, six data points for the second polynomial equation (where **POWER** = 2), and ten data points for the third polynomial (**POWER** = 3). If not enough data points exist for a given interpolant, the missing data values are set to the value of the **MISSING** keyword.

POWER

The weighting power of the distance, or the maximum order in the polynomial fitting function. For polynomial regression, this value is either 1, 2 (the default), or 3.

Note

The **POWER** keyword is only used for the Inverse Distance and Polynomial Regression methods of interpolation.

QUINTIC

Selects the triangulation with Quintic interpolation method.

Note

The **TRIANGLES** keyword is required when the **QUINTIC** keyword is used.

RADIAL_BASIS_FUNCTION

Selects the Radial Basis Function method of interpolation.

SEARCH_ELLIPSE

This keyword defines the search rules as a scalar or vector of from 1 to 3 elements that specify an ellipse or circle in the form $[R1]$, $[R1, R2]$, or $[R1, R2, Theta]$. $R1$ is one radius, $R2$ the other radius, and $Theta$ describes the angle between the X -axis to the $R1$ -axis, counterclockwise, in degrees or radians as specified by the `DEGREES` keyword. Only data points within this ellipse, centered on the location of the interpolate, are considered. If not specified, or 0, this distance test is not applied. Search rules effectively limit the number of data points used in computing each interpolate.

For example, to only consider data points within a distance of 5 units of each interpolant, specify the keyword as `SEARCH_ELLIPSE = 5`.

Note

The `TRIANGLES` keyword is required when the `EMPTY_SECTORS`, `MAX_PER_SECTOR`, `MIN_POINTS`, or `SEARCH_ELLIPSE` keywords are used.

SECTORS

This keyword defines the search rules for the number of sectors used in applying the `MAX_SECTOR`, `EMPTY_SECTORS`, and `MIN_POINTS` tests, an integer from 1 (the default setting) to 8.

SHEPARDS

Selects the Modified Shepard's method of interpolation. The parameters for the Modified Shepard's method are set by default, however the `NEIGHBORHOOD` keyword can be used to modify the parameters.

Note

The `TRIANGLES` keyword is required when the `SHEPARDS` keyword is used.

SMOOTHING

A scalar value defining the smoothing radius. For the Radial Basis Function method, if SMOOTHING is not specified, the default value is equal to the average point spacing, assuming a uniform distribution. For the Inverse Distance method, the default value is 0, implying no smoothing.

Note

The SMOOTHING keyword is used only for the Inverse Distance and Radial Basis Function methods of interpolation.

SPHERE

If set, data points lie on the surface of a sphere.

START

A scalar or a two-element array specifying the start of the grid in *X*, and *Y*. Default value is [*min(x)*, *min(y)*].

This keyword is ignored if the GRID, XOUT and YOUT keywords are specified.

TRIANGLES

A 3-by-*nt* longword array describing the connectivity of the input points, as returned by TRIANGULATE, where *nt* is the number of triangles. If duplicate point locations are input and the TRIANGLES keyword is present, only one of the points is considered.

Note

The TRIANGLES keyword is required for the Natural Neighbor, Nearest Neighbor, Modified Shepard's, Linear, and Quintic Interpolation methods.

Note

The TRIANGLES keyword is required when the EMPTY_SECTORS, MAX_PER_SECTOR, MIN_POINTS, or SEARCH_ELLIPSE keywords are used.

VARIOGRAM

Specifies the variogram type and parameters for the Kriging method. This parameter is a vector of one to four elements in the form of: [*Type*, *Range*, *Nugget*, *Scale*]. The *Type* is encoded as: 1 for linear, 2 for exponential, 3 for gaussian, 4 for spherical.

Defaults values are: *Type* is exponential, *Range* is 8 times the average point spacing assuming a uniform distribution, *Nugget* is zero, and *Scale* is 1.

Note

The VARIOGRAM keyword is only used with the Kriging method of interpolation.

XOUT

If the GRID keyword is set, use XOUT to specify irregularly spaced rectangular output grids. If XOUT is specified, YOUT must also be specified. When XOUT and YOUT are used, the DELTA, DIMENSION and START keywords are ignored.

If GRID is not set (the default), the location vectors XOUT and YOUT directly contain the X and Y values of the interpolates, and must have the same number of elements. The *Result* has the same structure and number of elements as XOUT and YOUT, with point $[i]$ resulting from the interpolation at (XOUT $[i]$, YOUT $[i]$).

YOUT

If the GRID keyword is set, use YOUT to specify irregularly spaced rectangular output grids. If YOUT is specified, XOUT must also be specified. When XOUT and YOUT are used, the DELTA, DIMENSION and START keywords are ignored.

If GRID is not set (the default), the location vectors XOUT and YOUT directly contain the X and Y values of the interpolates, and must have the same number of elements. The *Result* has the same structure and number of elements as XOUT and YOUT, with point $[i]$ resulting from the interpolation at (XOUT $[i]$, YOUT $[i]$).

Example 1

This example interpolates a data set measured on an irregular grid. Various types of the Inverse Distance interpolation method (the default method) are used in this example.

```

; Create a dataset of N points.
n = 100                                ;# of scattered points
seed = -121147L                        ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 768, TITLE = 'Inverse Distance'
!P.MULTI = [0, 1, 3, 0, 0]

; Inverse distance: Simplest default case which produces a 25 x
; 25 grid.
grid = GRIDDATA(x, y, f)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Simple Example'

; Default case, Inverse distance.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Larger Grid'

; Inverse distance + smoothing.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    SMOOTH = 0.05)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Smoothing'

; Set system variable back to default value.
!P.MULTI = 0

```

Example 2

This example uses the same data as the previous one, however in this example we use the Radial Basis Function and the Modified Shepard's interpolation methods.

```

; Create a dataset of N points.
n = 100                                ;# of scattered points
seed = -121147L                        ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form of a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 512, $
    TITLE = 'Different Methods of Gridding'
!P.MULTI = [0, 1, 2, 0, 0]

; Use radial basis function with multilog basis function.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    /RADIAL_BASIS_FUNCTION, FUNCTION_TYPE = 1)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Radial Basis Function'

; The following example requires triangulation.
TRIANGULATE, x, y, tr

; Use Modified Shepard's method.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /SHEPARDS)
SURFACE, grid, CHARSIZE = 3, TITLE = "Modified Shepard's Method"

; Set system variable back to default value.
!P.MULTI = 0

```

Example 3

This example uses the same data as the previous ones, however in this example we use various types of the Polynomial Regression interpolation method.

```
; Create a dataset of N points.
n = 100                                ;# of scattered points
seed = -121147L                        ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 768, $
    TITLE = 'Polynomial Regression'
!P.MULTI = [0, 1, 3, 0, 0]

; The following examples require the triangulation.
TRIANGULATE, x, y, tr

; Fit with a 2nd degree polynomial in x and y. This fit considers
; all points when fitting the surface, obliterating the individual
; peaks.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /POLYNOMIAL_REGRESSION)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Global Degree 2 Polynomial'

; Fit with a 2nd degree polynomial in x and y, but this time use
; only the 10 closest nodes to each interpolant. This provides a
; relatively smooth surface, but still shows the individual peaks.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /POLYNOMIAL_REGRESSION, MAX_PER_SECTOR = 10)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Local Polynomial, 10 Point'

; As above, but use only the nodes within a distance of 0.4 when
; fitting each interpolant.
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    TRIANGLES = tr, /POLYNOMIAL_REGRESSION, SEARCH_ELLIPSE = 0.4)
SURFACE, grid, CHARSIZE = 3, $
    TITLE = 'Local Polynomial, Radius = 0.4'

; Set system variable back to default value.
!P.MULTI = 0
```

Example 4

This example uses the same data as the previous ones, however in this example we show how to speed up the interpolation by limiting the interpolation to the local area around each interpolate.

```

; Create a dataset of N points.\.
n = 100                                ;# of scattered points
seed = -121147L                        ;For consistency
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)

; Create a dependent variable in the form a function of (x,y)
; with peaks & valleys.
f = 3 * EXP(-((9*x-2)^2 + (7-9*y)^2)/4) + $
    3 * EXP(-((9*x+1)^2)/49 - (1-0.9*y)) + $
    2 * EXP(-((9*x-7)^2 + (6-9*y)^2)/4) - $
    EXP(-(9*x-4)^2 - (2-9*y)^2)

; Note: the inverse distance, kriging, polynomial regression, and
; radial basis function methods are, by default, global methods in
; which each input node affects each output node. With these
; methods, large datasets can require a prohibitively long time to
; compute unless the scope of the interpolation is limited to a
; local area around each interpolate by specifying search rules.
; In fact, the radial basis function requires time proportional to
; the cube of the number of input points.

; For example, with 2,000 input points, a typical workstation
; required 500 seconds to interpolate a 10,000 point grid using
; radial basis functions. By limiting the size of the fit to the
; 20 closest points to each interpolate, via the MIN_POINTS
; keyword, the time required dropped to less than a second.

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 512, $
    TITLE = 'Radial Basis Function'
!P.MULTI = [0, 1, 2, 0, 0]

; Slow way:
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    /RADIAL_BASIS_FUNCTION)
SURFACE, grid, CHARSIZE = 3, TITLE = 'All Points'

; The following example requires triangulation.
TRIANGULATE, x, y, tr

```

```

; Faster way:
grid = GRIDDATA(x, y, f, START = 0, DELTA = 0.02, DIMENSION = 51, $
    /RADIAL_BASIS_FUNCTION, MIN_POINTS = 15, TRIANGLES = tr)
SURFACE, grid, CHARSIZE = 3, TITLE = 'Nearest 15 Points'

; Set system variable back to default value.
!P.MULTI = 0

```

Example 5

This example interpolates a spherical data set measured on an irregular grid. We use the Kriging and Natural Neighbors interpolation methods in this example.

```

; Create a 100 scattered points on a sphere and form a function
; of their latitude and longitude. Then grid them to a 2 degree
; grid over the sphere, display a Mollweide projection map, and
; overlay the contours of the result on the map.

; Create a dataset of N points.
n = 100
; A 2 degree grid with grid dimensions.
delta = 2
dims = [360, 180]/delta
; Longitude and latitudes
lon = RANDOMU(seed, n) * 360 - 180
lat = RANDOMU(seed, n) * 180 - 90
; The lon/lat grid locations
lon_grid = FINDGEN(dims[0]) * delta - 180
lat_grid = FINDGEN(dims[1]) * delta - 90

; Create a dependent variable in the form of a smoothly varying
; function.
f = SIN(2*lon*!DTOR) + COS(lat*!DTOR) ;

; Initialize display.
WINDOW, 0, XSIZE = 512, YSIZE = 768, TITLE = 'Spherical Gridding'
!P.MULTI = [0, 1, 3, 0, 0]

; Kriging: Simplest default case.
z = GRIDDATA(lon, lat, f, /KRIGING, /DEGREES, START = 0, /SPHERE, $
    DELTA = delta, DIMENSION = dims)
MAP_SET, /MOLLWEIDE, /ISOTROPIC, /HORIZON, /GRID, CHARSIZE = 3, $
    TITLE = 'Sphere: Kriging'
CONTOUR, z, lon_grid, lat_grid, /OVERPLOT, NLEVELS = 10, /FOLLOW

```

```

; This example is the same as above, but with the addition of a
; call to QHULL to triangulate the points on the sphere, and to
; then interpolate using the 10 closest points. The gridding
; portion of this example requires about one-fourth the time as
; above.
QHULL, lon, lat, tr, /DELAUNAY, SPHERE = s
z = GRIDDATA(lon, lat, f, /DEGREES, START = 0, DELTA = delta, $
    DIMENSION = dims, TRIANGLES = tr, MIN_POINTS = 10, /KRIGING, $
    /SPHERE)
MAP_SET, /MOLLWEIDE, /ISOTROPIC, /HORIZON, /GRID, /ADVANCE, $
    CHARSIZE = 3, TITLE = 'Sphere: Kriging, 10 Closest Points'
CONTOUR, z, lon_grid, lat_grid, /OVERPLOT, NLEVELS = 10, /FOLLOW

; This example uses the natural neighbor method, which is about
; four times faster than the above example but does not give as
; smooth a surface.
z = GRIDDATA(lon, lat, f, /DEGREES, START = 0, DELTA = delta, $
    DIMENSION = dims, /SPHERE, /NATURAL_NEIGHBOR, TRIANGLES = tr)
MAP_SET, /MOLLWEIDE, /ISOTROPIC, /HORIZON, /GRID, /ADVANCE, $
    CHARSIZE = 3, TITLE = 'Sphere: Natural Neighbor'
CONTOUR, z, lon_grid, lat_grid, /OVERPLOT, NLEVELS = 10, /FOLLOW

; Set system variable back to default value.
!P.MULTI = 0

```

Example 6

The following example uses the data from the `irreg_grid1.txt` ASCII file. This file contains scattered elevation data of a model of an inlet. This scattered elevation data contains two duplicate locations.

The `GRID_INPUT` procedure is used to omit the duplicate locations for the `GRIDDATA` function. The `GRIDDATA` function is then used to grid the data using the Radial Basis Function method. This method is specified by setting the `METHOD` keyword the `RadialBasisFunction` string, although it could easily be done using the `RADIAL_BASIS_FUNCTION` keyword.

```

; Import the Data:

; Determine the path to the file.
file = FILEPATH('irreg_grid1.txt', $
    SUBDIRECTORY = ['examples', 'data'])

; Import the data from the file into a structure.
dataStructure = READ_ASCII(file)

```

```

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)

; Initialize the variables of this example from
; the imported array.
x = dataArray[*, 0]
y = dataArray[*, 1]
data = dataArray[*, 2]

; Display the Data:

; Scale the data to range from 1 to 253 so a color table can be
; applied. The values of 0, 254, and 255 are reserved as outliers.
scaled = BYTSCL(data, TOP = !D.TABLE_SIZE - 4) + 1B

; Load the color table. If you are on a TrueColor, set the
; DECOMPOSED keyword to the DEVICE command before running a
; color table related routine.
DEVICE, DECOMPOSED = 0
LOADCT, 38

; Open a display window and plot the data points.
WINDOW, 0
PLOT, x, y, /XSTYLE, /YSTYLE, LINESTYLE = 1, $
    TITLE = 'Original Data, Scaled (1 to 253)', $
    XTITLE = 'x', YTITLE = 'y'

; Now display the data values with respect to the color table.
FOR i = 0L, (N_ELEMENTS(x) - 1) DO PLOTS, x[i], y[i], PSYM = -1, $
    SYMSIZE = 2., COLOR = scaled[i]

; Grid the Data and Display the Results:

; Preprocess and sort the data. GRID_INPUT will
; remove any duplicate locations.
GRID_INPUT, x, y, data, xSorted, ySorted, dataSorted

; Initialize the grid parameters.
gridSize = [51, 51]

; Use the equation of a straight line and the grid parameters to
; determine the x of the resulting grid.
slope = (MAX(xSorted) - MIN(xSorted))/(gridSize[0] - 1)
intercept = MIN(xSorted)
xGrid = (slope*FINDGEN(gridSize[0])) + intercept

```

```

; Use the equation of a straight line and the grid parameters to
; determine the y of the resulting grid.
slope = (MAX(ySorted) - MIN(ySorted))/(gridSize[1] - 1)
intercept = MIN(ySorted)
yGrid = (slope*FINDGEN(gridSize[1])) + intercept

; Grid the data with the Radial Basis Function method.
grid = GRIDDATA(xSorted, ySorted, dataSorted, $
    DIMENSION = gridSize, METHOD = 'RadialBasisFunction')

; Open a display window and contour the Radial Basis Function
; results.
WINDOW, 1
scaled = BYTSCL(grid, TOP = !D.TABLE_SIZE - 4) + 1B
CONTOUR, scaled, xGrid, YGrid, /XSTYLE, /YSTYLE, /FILL, $
    LEVELS = BYTSCL(INDGEN(18), TOP = !D.TABLE_SIZE - 4) + 1B, $
    C_COLORS = BYTSCL(INDGEN(18), TOP = !D.TABLE_SIZE - 4) + 1B, $
    TITLE = 'The Resulting Grid with Radial Basis Function', $
    XTITLE = 'x', YTITLE = 'y'

```

Example 7

The following example uses the data from the `irreg_grid1.txt` ASCII file. This file contains scattered elevation data of a model of an inlet. This scattered elevation data contains two duplicate locations. The same data is used in the previous example.

The `GRID_INPUT` procedure is used to omit the duplicate locations for the `GRIDDATA` function. The `GRIDDATA` function is then used to grid the data using the Radial Basis Function method. This method is specified by setting the `METHOD` keyword the `RadialBasisFunction` string, although it could easily be done using the `RADIAL_BASIS_FUNCTION` keyword.

Faulting is also applied in this example. First, a fault area is placed around the right side of the dataset. This fault area contains data points. The data points within this area are gridded separately from the points outside of the fault area.

Then, a fault area is defined within an region that does not contain any data points. Since this fault area does not contain any points, the grid within this region simply results to the value defined by the `MISSING` keyword. The points outside of the fault area are gridded independent of the fault region.

```

; Import the Data:

; Determine the path to the file.
file = FILEPATH('irreg_grid1.txt', $
    SUBDIRECTORY = ['examples', 'data'])

```

```

; Import the data from the file into a structure.
dataStructure = READ_ASCII(file)

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)

; Initialize the variables of this example from
; the imported array.
x = dataArray[*, 0]
y = dataArray[*, 1]
data = dataArray[*, 2]

; Grid the Data and Display the Results:

; Preprocess and sort the data. GRID_INPUT will
; remove any duplicate locations.
GRID_INPUT, x, y, data, xSorted, ySorted, dataSorted

; Initialize the grid parameters.
gridSize = [51, 51]

; Use the equation of a straight line and the grid parameters to
; determine the x of the resulting grid.
slope = (MAX(xSorted) - MIN(xSorted))/(gridSize[0] - 1)
intercept = MIN(xSorted)
xGrid = (slope*FINDGEN(gridSize[0])) + intercept

; Use the equation of a straight line and the grid parameters to
; determine the y of the resulting grid.
slope = (MAX(ySorted) - MIN(ySorted))/(gridSize[1] - 1)
intercept = MIN(ySorted)
yGrid = (slope*FINDGEN(gridSize[1])) + intercept

; Initialize display.
DEVICE, DECOMPOSED = 0
LOADCT, 38
WINDOW, 0, XSIZE = 600, YSIZE = 600, $
    TITLE = 'The Resulting Grid from the Radial Basis Function '+' $
    'Method with Faulting'
!P.MULTI = [0, 1, 2, 0, 0]

; Define a fault area, which contains data points.
faultVertices = [[2200, 4000], [2200, 3000], [2600, 2700], $
    [2600, -50], [5050, -50], [5050, 4000], [2200, 4000]]
faultConnectivity = [7, 0, 1, 2, 3, 4, 5, 6, -1]

```

```

; Grid the data with faulting using the Radial Basis Function
; method.
grid = GRIDDATA(xSorted, ySorted, dataSorted, $
    DIMENSION = gridSize, METHOD = 'RadialBasisFunction', $
    FAULT_XY = faultVertices, FAULT_POLYGONS = faultConnectivity, $
    MISSING = MIN(dataSorted))

; Display grid results.
CONTOUR, BYTSCL(grid), xGrid, YGrid, /XSTYLE, /YSTYLE, /FILL, $
    LEVELS = BYTSCL(INDGEN(18)), C_COLORS = BYTSCL(INDGEN(18)), $
    TITLE = 'Fault Area Contains Data ' + $
    '(Fault Area in Dashed Lines)', XTITLE = 'x', YTITLE = 'y'

; Display outline of fault area.
PLOTS, faultVertices, /DATA, LINESTYLE = 2, THICK = 2

; Define a fault area, which does not contain data points.
faultVertices = [[2600, -50], [2800, -50], [2800, 2700], $
    [2400, 3000], [2400, 4000], [2200, 4000], [2200, 3000], $
    [2600, 2700], [2600, -50]]
faultConnectivity = [9, 0, 1, 2, 3, 4, 5, 6, 7, 8, -1]

; Grid the data with faulting using the Radial Basis Function
; method.
grid = GRIDDATA(xSorted, ySorted, dataSorted, $
    DIMENSION = gridSize, METHOD = 'RadialBasisFunction', $
    FAULT_XY = faultVertices, FAULT_POLYGONS = faultConnectivity, $
    MISSING = MIN(dataSorted))

; Display grid results.
CONTOUR, BYTSCL(grid), xGrid, YGrid, /XSTYLE, /YSTYLE, /FILL, $
    LEVELS = BYTSCL(INDGEN(18)), C_COLORS = BYTSCL(INDGEN(18)), $
    TITLE = 'Fault Area Does Not Contain Data ' + $
    '(Fault Area in Dashed Lines)', XTITLE = 'x', YTITLE = 'y'

; Display outline of fault area.
PLOTS, faultVertices, /DATA, LINESTYLE = 2, THICK = 2

; Set system variable back to default value.
!P.MULTI = 0

```

References

Kriging

Isaaks, E. H., and Srivastava, R. M., *An Introduction to Applied Geostatistics*, Oxford University Press, New York, 1989.

Minimum Curvature

Barrodale, I., et al, "Warping Digital Images Using Thin Plate Splines", *Pattern Recognition*, Vol 26, No 2, pp. 375-376., 1993.

Powell, M.J.D., "Tabulation of thin plate splines on a very fine two-dimensional grid", Report No. DAMTP 1992/NA2, University of Cambridge, Cambridge, U.K. 1992.

Modified Shepard's

Franke, R., and Nielson, G. , "Smooth Interpolation of Large Sets of Scattered Data", *International Journal for Numerical Methods in Engineering*, v. 15, 1980, pp. 1691-1704.

Renka, R. J., Algorithm 790 - CSHEP2D: Cubic Shepard Method for Bivariate Interpolation of Scattered Data, Robert J. Renka, *ACM Trans. Math Softw.* 25, 1 (March 1999), pp. 70-73.

Shepard, D., "A Two Dimensional Interpolation Function for Irregularly Spaced Data", *Proc. 23rd Nat. Conf. ACM*, 1968, pp. 517-523.

Natural Neighbor

Watson, D. F., *Contouring: A Guide to the Analysis and Display of Spatial Data*, Pergamon Press, ISBN 0 08 040286 0, 1992.

Watson, D. F., *Nngridr - An Implementation of Natural Neighbor Interpolation*, David Watson, P.O. Box 734, Claremont, WA 6010, Australia, 1994.

Quintic

Akima, H., Algorithm 761 - Scattered-data Surface Fitting that has the Accuracy of a Cubic Polynomial, Hiroshi Akima, *ACM Trans. Math. Softw.* 22, 3 (Sep. 1996), pp. 362 - 371.

Renka, R.J., "A Triangle-based C1 Interpolation Method", *Rocky Mountain Journal of Mathematics*, Vol 14, No. 1, 1984.

Radial Basis Function

Franke, R., A Critical Comparison of Some Methods for Interpolation of Scattered Data, Naval Postgraduate School, Technical Report, NPS 53-79-003, 1979.

Hardy, R.L., "Theory and Applications of the Multiquadric-biharmonic Method", Computers Math. With Applic, v 19, no. 8/9, 1990, pp.163-208.

See Also

[GRID_INPUT](#)

HDF_VD_ATTRFIND

The `HDF_VD_ATTRFIND` function returns an attribute's index number given the name of an attribute associated with the specified `vdata` or `vdata/field` pair. If the attribute cannot be located, `-1` is returned.

Syntax

Result = `HDF_VD_ATTRFIND(VData, FieldID, Name)`

Arguments

VData

The `VData` handle returned by a previous call to `HDF_VD_ATTACH`.

FieldID

A zero-based index specifying the field, or a string containing the name of the field within the `VData` to which the attribute is attached. Setting `FieldID` to `-1` specifies that the attribute is attached to the `vdata` itself.

Name

A string containing the name of the attribute whose index is to be returned.

Example

For an example using this routine, see the documentation for `HDF_VD_ATTRSET`.

See Also

[HDF_VD_ATTRINFO](#), [HDF_VD_ATTRSET](#), [HDF_VD_ISATTR](#),
[HDF_VD_NATTRS](#)

HDF_VD_ATTRINFO

The `HDF_VD_ATTRINFO` procedure reads or retrieves information about a vdata attribute or a vdata field attribute from the currently attached HDF vdata structure. If the attribute is not present, an error message is printed.

Syntax

```
HDF_VD_ATTRINFO, VData, FieldID, AttrID, Values [, COUNT=variable]  
[, DATA=variable] [, HDF_TYPE=variable] [, NAME=variable ]  
[, TYPE=variable]
```

Arguments

VData

The VData handle returned by a previous call to `HDF_VD_ATTACH`.

FieldID

A zero-based index specifying the field, or a string containing the name of the field within the VData whose attribute is to be read. Setting `FieldID` to -1 specifies that the attribute to be read is attached to the vdata itself.

AttrID

A zero-based integer index specifying the attribute to be read, or a string containing the name of that attribute.

Values

The attribute value(s) to be written.

Keywords

COUNT

Set this keyword to a named variable in which the number of data values (order of the attribute) is returned.

DATA

Set this keyword to a named variable in which the attribute data is returned.

HDF_TYPE

Set this keyword to a named variable in which the HDF data type of the attribute is returned as a scalar string.

NAME

Set this keyword to a named variable in which the name of the attribute is returned.

TYPE

Set this keyword to a named variable in which the IDL type of the attribute is returned as a scalar string.

Example

For an example using this routine, see the documentation for `HDF_VD_ATTRSET`.

See Also

[HDF_VD_ATTRFIND](#), [HDF_VD_ATTRSET](#), [HDF_VD_ISATTR](#),
[HDF_VD_NATTRS](#)

HDF_VD_ATTRSET

The `HDF_VD_ATTRSET` procedure writes a vdata attribute or a vdata field attribute to the currently attached HDF vdata structure. If no data type keyword is specified, the data type of the attribute value is used.

Syntax

```
HDF_VD_ATTRSET, VData, FieldID, Attr_Name, Values [, Count] [, /BYTE]
[, /DFNT_CHAR8] [, /DFNT_FLOAT32] [, /DFNT_FLOAT64] [, /DFNT_INT8]
[, /DFNT_INT16] [, /DFNT_INT32] [, /DFNT_UCHAR8] [, /DFNT_UINT8]
[, /DFNT_UINT16] [, /DFNT_UINT32] [, /DOUBLE] [, /FLOAT] [, /INT]
[, /LONG] [, /SHORT] [, /STRING] [, /UINT ] [, /ULONG ]
```

Arguments

VData

The VData handle returned by a previous call to `HDF_VD_ATTACH`.

Note

The vdata structure must have been attached in write mode in order for attributes to be correctly associated with a vdata or one of its fields. If the vdata is not write accessible, HDF does not return an error; instead, the attribute information is written to the file but is not associated with the vdata.

FieldID

A zero-based index specifying the field, or a string containing the name of the field within the VData whose attribute is to be set. If `FieldID` is set to -1, the attribute will be attached to the vdata itself.

Attr_Name

A string containing the name of the attribute to be written.

Values

The attribute value(s) to be written.

Note

Attributes to be written as characters may not be a multi-dimensional array (e.g. if being converted from byte values) or an array of IDL strings.

Count

An optional integer argument specifying how many values are to be written. Count must be less than or equal to the number of elements in the Values argument. If not specified, the actual number of values present will be written.

Keywords

BYTE

Set this keyword to indicate that the attribute is composed of bytes. Data will be stored with the HDF DFNT_UINT8 data type. Setting this keyword is the same as setting the DFNT_UINT8 keyword.

DFNT_CHAR8

Set this keyword to create an attribute of HDF type DFNT_CHAR8. Setting this keyword is the same as setting the STRING keyword.

DFNT_FLOAT32

Set this keyword to create an attribute of HDF type DFNT_FLOAT32. Setting this keyword is the same as setting the FLOAT keyword.

DFNT_FLOAT64

Set this keyword to create an attribute of HDF type DFNT_FLOAT64. Setting this keyword is the same as setting the DOUBLE keyword.

DFNT_INT8

Set this keyword to create an attribute of HDF type DFNT_INT8.

DFNT_INT16

Set this keyword to create an attribute of HDF type DFNT_INT16. Setting this keyword is the same as setting either the INT keyword or the SHORT keyword.

DFNT_INT32

Set this keyword to create an attribute of HDF type DFNT_INT32. Setting this keyword is the same as setting the LONG keyword.

DFNT_UCHAR8

Set this keyword to create an attribute of HDF type DFNT_UCHAR8.

DFNT_UINT8

Set this keyword to create an attribute of HDF type DFNT_UINT8. Setting this keyword is the same as setting the BYTE keyword.

DFNT_UINT16

Set this keyword to create an attribute of HDF type DFNT_UINT16.

DFNT_UINT32

Set this keyword to create an attribute of HDF type DFNT_UINT32.

DOUBLE

Set this keyword to indicate that the attribute is composed of double-precision floating-point values. Data will be stored with the HDF type DFNT_FLOAT64. Setting this keyword is the same as setting the DFNT_FLOAT64 keyword.

FLOAT

Set this keyword to indicate that the attribute is composed of single-precision floating-point values. Data will be stored with the HDF type DFNT_FLOAT32 data type. Setting this keyword is the same as setting the DFNT_FLOAT32 keyword.

INT

Set this keyword to indicate that the attribute is composed of 16-bit integers. Data will be stored with the HDF type DFNT_INT16 data type. Setting this keyword is the same as setting either the SHORT keyword or the DFNT_INT16 keyword.

LONG

Set this keyword to indicate that the attribute is composed of longword integers. Data will be stored with the HDF type DFNT_INT32 data type. Setting this keyword is the same as setting the DFNT_INT32 keyword.

SHORT

Set this keyword to indicate that the attribute is composed of 16-bit integers. Data will be stored with the HDF type DFNT_INT16 data type. Setting this keyword is the same as setting either the INT keyword or the DFNT_INT16 keyword.

STRING

Set this keyword to indicate that the attribute is composed of strings. Data will be stored with the HDF type DFNT_CHAR8 data type. Setting this keyword is the same as setting the DFNT_CHAR8 keyword.

UINT

Set this keyword to indicate that the attribute is composed of unsigned 2-byte integers. Data will be stored with the HDF type DFNT_UINT16 data type. Setting this keyword is the same as setting the DFNT_UINT16 keyword.

ULONG

Set this keyword to indicate that the attribute is composed of unsigned longword integers. Data will be stored with the HDF type DFNT_UINT32 data type. Setting this keyword is the same as setting the DFNT_UINT32 keyword.

Example

```
; Open an HDF file.
fid = HDF_OPEN(FILEPATH('vattr_example.hdf',$
    SUBDIRECTORY = ['examples', 'data']), /RDWR)

; Locate and attach an existing vdata.
vdref = HDF_VD_FIND(fid, 'MetObs')
vdid = HDF_VD_ATTACH(fid, vdref, /WRITE)

; Attach two attributes to the vdata.
HDF_VD_ATTRSET, vdid, -1, 'vdata_contents', $
    'Ground station meteorological observations.'
HDF_VD_ATTRSET, vdid, -1, 'num_stations', 10

; Attach an attribute to one of the fields in the vdata.
HDF_VD_ATTRSET, vdid, 'TempDP', 'field_contents', $
    'Dew point temperature in degrees Celsius.'

; Get the number of attributes associated with the vdata.
num_vdattr = HDF_VD_NATTRS(vdid, -1)
PRINT, 'Number of attributes attached to vdata MetObs: ', $
    num_vdattr
```

```

; Get information for one of the vdata attributes by first finding
; the attribute's index number.
attr_index = HDF_VD_ATTRFIND(vdid, -1, 'vdata_contents')
HDF_VD_ATTRINFO, vdid, 1, attr_index, $
    NAME = attr_name, DATA = metobs_contents
HELP, attr_name, metobs_contents

; Get information for another vdata attribute using the
; attribute's name.
HDF_VD_ATTRINFO, vdid, -1, 'num_stations', DATA = num_stations, $
    HDF_TYPE = hdftype, TYPE = idltype
HELP, num_stations, hdftype, idltype
PRINT, num_stations

; Get the number of attributes attached to the vdata field
; TempDP.
num_fdatr = HDF_VD_NATTRS(vdid, 'TempDP')
PRINT, 'Number of attributes attached to field TempDP: ', $
    num_fdatr

; Get the information for the vdata field attribute.
HDF_VD_ATTRINFO, vdid, 'TempDP', 'field_contents', $
    COUNT = count, HDF_TYPE = hdftype, TYPE = idltype, $
    DATA = dptemp_attr
HELP, count, hdftype, idltype, dptemp_attr

; End access to the vdata.
HDF_VD_DETACH, vdid

; Attach a vdata which stores one of the attribute values.
vdid = HDF_VD_ATTACH(fid, 5)

; Get the vdata's name and check to see that it is indeed storing
; an attribute.
HDF_VD_GET, vdid, NAME = vdname
isattr = HDF_VD_ISATTR(vdid)
HELP, vdname, isattr

; End access to the vdata and the HDF file.
HDF_VD_DETACH, vdid
HDF_CLOSE, fid

```

IDL Output

```

Number of attributes attached to vdata MetObs:                2
ATTR_NAME      STRING      = 'vdata_contents'
METOBS_CONTENTS STRING      = 'Ground station meteorological
observations.'

```

```

NUM_STATIONS      INT          = Array[1]
HDFTYPE           STRING       = 'DFNT_INT16'
IDLTYPE           STRING       = 'INT'
10
Number of attributes attached to field TempDP:          1
COUNT           LONG          =          41
HDFTYPE           STRING       = 'DFNT_CHAR8'
IDLTYPE           STRING       = 'STRING'
DPTEMP_ATTR       STRING       = 'Dew point temperature in degrees
Celsius.'
VDNAME            STRING       = 'field_contents'
ISATTR           LONG          =          1

```

See Also

[HDF_VD_ATTRFIND](#), [HDF_VD_ATTRINFO](#), [HDF_VD_ISATTR](#),
[HDF_VD_NATTRS](#)

HDF_VD_ISATTR

The `HDF_VD_ISATTR` function returns `TRUE` (1) if the `vdata` is storing an attribute, `FALSE` (0) otherwise. HDF stores attributes as `vdatas`, so this routine provides a means to test whether or not a particular `vdata` contains an attribute.

Syntax

Result = `HDF_VD_ISATTR(VData)`

Arguments

VData

The `VData` handle returned by a previous call to `HDF_VD_ATTACH`.

Example

For an example using this routine, see the documentation for `HDF_VD_ATTRSET`.

See Also

[HDF_VD_ATTRFIND](#), [HDF_VD_ATTRINFO](#), [HDF_VD_ATTRSET](#),
[HDF_VD_NATTRS](#)

HDF_VD_NATTRS

The `HDF_VD_NATTRS` function returns the number of attributes associated with the specified `vdata` or `vdata/field` pair if successful. Otherwise, `-1` is returned.

Syntax

Result = `HDF_VD_NATTRS(VData, FieldID)`

Arguments

VData

The `VData` handle returned by a previous call to `HDF_VD_ATTACH`.

FieldID

A zero-based index specifying the field, or a string containing the name of the field, within the `VData` whose attributes are to be counted. Setting `Index` to `-1` specifies that attributes attached to the `vdata` itself are to be counted.

Example

For an example using this routine, see the documentation for `HDF_VD_ATTRSET`.

See Also

[HDF_VD_ATTRFIND](#), [HDF_VD_ATTRINFO](#), [HDF_VD_ATTRSET](#),
[HDF_VD_ISATTR](#)

HEAP_FREE

The `HEAP_FREE` procedure recursively frees all heap variables (pointers or objects) referenced by its input argument. This routine examines the input variable, including all array elements and structure fields. When a valid pointer or object reference is encountered, that heap variable is marked for removal, and then is recursively examined for additional heap variables to be freed. In this way, all heap variables that are referenced directly or indirectly by the input argument are located. Once all such heap variables are identified, `HEAP_FREE` releases them in a final pass. Pointers are released as if the `PTR_FREE` procedure was called. Objects are released as with a call to `OBJ_DESTROY`.

As with the related `HEAP_GC` procedure, there are some disadvantages to using `HEAP_FREE` such as:

- When freeing object heap variables, `HEAP_FREE` calls `OBJ_DESTROY` without supplying any plain or keyword arguments. Depending on the objects being released, this may not be sufficient. In such cases, the caller must call `OBJ_DESTROY` explicitly with the proper arguments rather than using `HEAP_FREE`.
- `HEAP_FREE` releases the referenced heap variables in an unspecified order which depends on the current state of the internal data structure used by IDL to hold them. This can be confusing for object destructor methods that expect all of their contained data to be present. If your application requires a specific order for the release of its heap variables, you must explicitly free them in the correct order. `HEAP_FREE` cannot be used in such cases.
- The algorithm used by `HEAP_FREE` to release variables requires examination of every existing heap variable (that is, it is an $O(n)$ algorithm). This may be slow if an IDL session has thousands of current heap variables.

For these reasons, Research Systems recommends that applications keep careful track of their heap variable usage, and explicitly free them at the proper time (for example, using the object destructor method) rather than resorting to simple-looking but potentially expensive expedients such as `HEAP_FREE` or `HEAP_GC`.

`HEAP_FREE` is recommended when:

- The data structures involved are highly complex, nested, or variable, and writing cleanup code is difficult and error prone.
- The data structures are opaque, and the code cleaning up does not have knowledge of the structure.

Syntax

HEAP_FREE, *Var* [, /OBJ] [, /PTR] [, /VERBOSE]

Arguments

Var

The variable whose data is used as the starting point for heap variables to be freed.

Keywords

OBJ

Set this keyword to free object heap variables only.

PTR

Set this keyword to free pointer heap variables only.

Note

Setting both the PTR and OBJ keywords is the same as setting neither.

VERBOSE

If this keyword is set, HEAP_FREE writes a one line description of each heap variable, in the format used by the HELP procedure, as the variable is released. This is a debugging aid that can be used by program developers to check for heap variable leaks that need to be located and eliminated.

Example

```
; Create a structure variable.
mySubStructure = {pointer:PTR_NEW(INDGEN(100)), $
  obj:OBJ_NEW('Idl_Container')}
myStructure = {substruct:mySubStructure, $
  ptrs:[PTR_NEW(INDGEN(10)), PTR_NEW(INDGEN(11))]}

;Look at the heap.
HELP, /HEAP_VARIABLES

; Now free the heap variables contained in myStructure.
HEAP_FREE, myStructure, /VERBOSE
HELP, /HEAP_VARIABLES
```

See Also

[HEAP_GC](#)

INTERVAL_VOLUME

The new `INTERVAL_VOLUME` procedure is used to generate a tetrahedral mesh from volumetric data. The generated mesh spans the portion of the volume where the volume data samples fall between two constant data values. This can also be thought of as a mesh constructed to fill the volume between two isosurfaces which are drawn according to the two supplied constant data values. The algorithm is very similar to the `ISOSURFACE` algorithm and expands upon the `SHADE_VOLUME` algorithm. A topologically-consistent tetrahedral mesh is returned by decomposing the volume into oriented tetrahedra. This also allows the algorithm to find the interval volume of any tetrahedral mesh.

If an auxiliary array is provided, its data is interpolated onto the output vertices and is returned. This auxiliary data array may have multiple values at each vertex. Any size-leading dimension is allowed as long as the number of values in the subsequent dimensions matches the number of elements in the input data array.

For more information on the `INTERVAL_VOLUME` algorithm, see the paper, “Interval Volume Tetrahedrization”, Nielson and Sung, *Proceedings: IEEE Visualization*, 1997.

Syntax

```
INTERVAL_VOLUME, Data, Value0, Value1, Outverts, Outconn  
[, AUXDATA_IN=array, AUXDATA_OUT=variable]  
[, GEOM_XYZ=array, TETRAHEDRA=array]
```

Arguments

Data

Input three-dimensional array of scalars that define the volume data.

Value0

Input scalar iso-value. This value specifies one of the limits for the interval volume. The generated interval volume encloses all volume samples between and including *Value0* and *Value1*. *Value0* may be greater than or less than *Value1*, but the two values may not be exactly equal. This value also cannot be a NaN, but can be +/- INF.

Value1

Input scalar iso-value. This value specifies the other limit for the interval volume. The generated interval volume encloses all volume samples between and including

Value0 and *Value1*. *Value1* may be greater than or less than *Value0*, but the two values may not be exactly equal. This value also cannot be a NaN, but can be +/- INF.

Outverts

A named variable to contain an output [3, n] array of floating point vertices making up the tetrahedral mesh.

Outconn

A named variable to contain an output array of tetrahedral mesh connectivity values. This array is one-dimensional and consists of a series of four vertex indices, where each group of four indices describes a tetrahedron. The connectivity values are indices into the vertex array returned in *Outverts*. If no tetrahedra are extracted, this argument returns the array [-1].

Keywords

AUXDATA_IN

This keyword defines the input array of auxiliary data with trailing dimensions being the number of values in *Data*.

Note

If you specify the AUXDATA_IN then you must specify AUXDATA_OUT.

AUXDATA_OUT

Set this keyword to a named variable that will contain an output array of auxiliary data sampled at the locations in *Outverts*.

Note

If you specify AUXDATA_OUT then you must specify AUXDATA_IN.

GEOM_XYZ

This keyword defines a [3, n] input array of vertex coordinates (one for each value in the *Data* array). This array is used to define the spatial location of each scalar. If this keyword is omitted, *Data* must be a three-dimensional array and the scalar locations are assumed to be on a uniform grid.

Note

If you specify GEOM_XYZ then you must specify TETRAHEDRA.

TETRAHEDRA

This keyword defines an input array of tetrahedral connectivity values. If this array is not specified, the connectivity is assumed to be a rectilinear grid over the input three-dimensional array. If this keyword is specified, the input data array need not be a three-dimensional array. Each tetrahedron is represented by four values in the connectivity array. Every four values in the array correspond to the vertices of a single tetrahedron.

Note

If you specify TETRAHEDRA then you must specify GEOM_XYZ.

Example

The following example generates an interval volume and displays the surface of the volume:

```
RESTORE, FILEPATH('clouds3d.dat', $
    SUBDIRECTORY=['examples','data'])
INTERVAL_VOLUME, rain, 0.1, 0.6, verts, conn
conn = TETRA_SURFACE(verts, conn)
oRain = OBJ_NEW('IDLgrPolygon', verts, POLYGONS=conn, $
    COLOR=[255,255,255], SHADING=1)
XOBJVIEW, oRain, BACKGROUND=[150,200,255]
```

See Also

[ISOSURFACE](#), [SHADE_VOLUME](#), [XVOLUME](#)

PATH_SEP

The `PATH_SEP` function returns the proper file path segment separator character for the current operating system. This is the character used by the host operating system for delimiting subdirectory names in a path specification. Use of this function instead of hard wiring separators makes code more portable.

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

Syntax

```
Result = PATH_SEP( [ /PARENT_DIRECTORY ] [, /SEARCH_PATH ] )
```

Arguments

None.

Keywords

PARENT_DIRECTORY

If set, `PATH_SEP` returns the standard directory notation used by the host operating system to indicate the parent of a directory.

SEARCH_PATH

If set, `PATH_SEP` returns the character used to separate entries in a search path.

See Also

[FILEPATH](#), [FILE_SEARCH](#)

QGRID3

The QGRID3 function linearly interpolates the dependent variable values to points in a regularly sampled volume. Its inputs are a triangulation of scattered data points in three dimensions, and the value of a dependent variable for each point.

Syntax

```
Result = QGRID3( XYZ, F, Tetrahedra [, DELTA=vector ] [, DIMENSION=vector ]  
[, MISSING=value ] [, START=vector ] )
```

or

```
Result = QGRID3( X, Y, Z, F, Tetrahedra [, DELTA=array ] [, DIMENSION=array ]  
[, MISSING=value ] [, START=array ] )
```

Return Value

Result is a 3-dimensional array of either single or double precision floating type, of the specified dimensions.

Arguments

XYZ

This is a 3-by-*n* array containing the scattered points.

X, Y, Z

One-dimensional vectors containing the *X*, *Y*, and *Z* point coordinates.

Tetrahedra

A longword array containing the point indices of each tetrahedron, as created by QHULL.

Keywords

Note

Any of the keywords may be set to a scalar if all elements are the same.

DELTA

A scalar or three element array specifying the grid spacing in X, Y, and Z. If this keyword is not specified, it is set to create a grid of DIMENSION cells, enclosing the volume from START to [max(x), max(y), max(z)].

DIMENSION

A three element array specifying the grid dimensions in X, Y, and Z. Default value is 25 for each dimension.

MISSING

The value to be used for grid points that lie outside the convex hull of the scattered points. The default is 0.

START

A three element array specifying the start of the grid in X, Y, and Z. Default value is [min(x), min(y), min(z)].

Example 1

This example interpolates a data set measured on an irregular grid.

```
; Create a dataset of N points.
n = 200
x = RANDOMU(seed, n)
y = RANDOMU(seed, n)
z = RANDOMU(seed, n)

; Create dependent variable.
f = x^2 - x*y + z^2 + 1

; Obtain a tetrahedra using the QHULL procedure.
QHULL, x, y, z, tet, /DELAUNAY

; Create a volume with dimensions [51, 51, 51]
; over the unit cube.
volume = QGRID3(x, y, z, f, tet, START=0, DIMENSION=51, $
    DELTA=0.02)

; Display the volume.
XVOLUME, BYTSCL(volume)
```

Example 2

This example is similar to the previous one, however in this example we use a $[3, n]$ array of points.

```
; Create a dataset of N points.
n = 200
p = RANDOMU(seed, 3, n)

; Create dependent variable.
f = p[0,*]^2 - p[0,*]*p[1,*] + p[2,*]^2 + 1

; Obtain a tetrahedra.
QHULL, p, tet, /DELAUNAY

; Create a volume with dimensions [51, 51, 51] over the unit cube.
volume = QGRID3(p, f, tet, START=0, DIMENSION=51, DELTA=0.02)

; Display the volume.
XVOLUME, BYTSCL(volume)
```

Example 3

The following example uses the data from the `irreg_grid2.txt` ASCII file. This file contains scattered three-dimensional data. This file contains bore hole data for a square mile of land. The QHULL procedure is used to triangulate the three-dimensional locations. The QGRID3 function uses the results from QHULL to grid the data into a volume. The scattered data is displayed as symbol polyline objects in the XOBJVIEW utility. The resulting gridded volume is displayed in the XVOLUME utility:

```
; Import the Data:

; Determine the path to the file. This file contains bore hole
; data for a square mile of land. The bore hole samples were
; roughly taken diagonally from the upper left corner of the
; square to the lower right corner.
file = FILEPATH('irreg_grid2.txt', $
    SUBDIRECTORY = ['examples', 'data'])

; Import the data from the file into a structure.
dataStructure = READ_ASCII(file)

; Get the imported array from the first field of
; the structure.
dataArray = TRANSPOSE(dataStructure.field1)
```

```

; Initialize the variables of this example from
; the imported array.
x = dataArray[*, 0]
y = dataArray[*, 1]
z = dataArray[*, 2]
data = dataArray[*, 3]

; Determine number of data points.
nPoints = N_ELEMENTS(data)

; Triangulate the Data with QHULL:

; Construct the convex hulls of the volume.
QHULL, x, y, z, tetrahedra, /DELAUNAY

; Grid the Data and Display the Results:

; Initialize volume parameters.
cubeSize = [51, 51, 51]
; Grid the data into a volume.
volume = QGRID3(x, y, z, data, tetrahedra, START = 0, $
    DIMENSION = cubeSize, DELTA = 0.02)
; Scale the volume to be able to view the full data value range
; with the color tables provided in the XVOLUME utility.
scaledVolume = BYTSCL(volume)

; Display the results in the XVOLUME utility.
XVOLUME, scaledVolume

; Derive the isosurface for mineral deposits with the data value
; of 2.5.
ISOSURFACE, volume, 2.5, vertices, connectivity

; Initialize a model to contain the isosurface.
oModel = OBJ_NEW('IDLgrModel')

; Initialize the polygon object of the isosurface.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, COLOR = [0, 0, 255])

; Determine the range in each direction.
xRange = [0, cubeSize[0]]
yRange = [0, cubeSize[1]]
zRange = [0, cubeSize[2]]

```

```

; Initialize an axis for each direction.
oAxes = OBJARR(3)
oAxes[0] = OBJ_NEW('IDLgrAxis', 0, RANGE = xRange, $
    LOCATION = [xRange[0], yRange[0], zRange[0]], /EXACT, $
    TICKLEN = (0.02*(yRange[1] - yRange[0])))
oAxes[1] = OBJ_NEW('IDLgrAxis', 1, RANGE = yRange, $
    LOCATION = [xRange[0], yRange[0], zRange[0]], /EXACT, $
    TICKLEN = (0.02*(xRange[1] - xRange[0])))
oAxes[2] = OBJ_NEW('IDLgrAxis', 2, RANGE = zRange, $
    LOCATION = [xRange[0], yRange[1], zRange[0]], /EXACT, $
    TICKLEN = (0.02*(xRange[1] - xRange[0])))

; Add the polygon and axes object to the model.
oModel -> Add, oPolygon
oModel -> Add, oAxes

; Rotate the model for a better perspective.
oModel -> Rotate, [0, 0, 1], 30.
oModel -> Rotate, [1, 0, 0], -45.

; Display the model, which contains the isosurface.
XOBJVIEW, oModel, /BLOCK, SCALE = 0.75, $
    TITLE = 'Isosurface at the Value of 2.5'

; Cleanup object references.
OBJ_DESTROY, [oModel]

```

See Also

[QHULL](#)

QHULL

The QHULL procedure constructs convex hulls, Delaunay triangulations, and Voronoi diagrams of a set of points of 2-dimensions or higher. It uses and is based on the program QHULL, which is described in Barber, Dobkin and Huhdanpaa, “The Quickhull Algorithm for Convex Hulls,” *ACM Transactions on Mathematical Software*, Vol. 22, No 4, December 1996, Pages 469-483.

For more information about QHULL see <http://www.geom.umn.edu/software/qhull/>.

Syntax

QHULL, *V*, *Tr*

or,

QHULL, *V0*, *V1*, [, *V2* ... [, *V6*]], *Tr* [, BOUNDS=*variable*]
 [, CONNECTIVITY=*variable*] [, /DELAUNAY] [, SPHERE=*variable*]
 [, VDIAGRAM=*array*] [, VNORMALS=*array*] [, VVERTICES=*array*]

Arguments

V

An input argument providing an *nd*-by-*np* array containing the locations of *np* points, in *nd* dimensions. The number of dimensions, *nd*, must be greater than or equal to 2.

V0, V1, V2, ..., V(N-1)

Input vectors of dimension *np*-by-1 elements each containing the *i*-th coordinate of *np* points in *nd* dimensions. A maximum of seven input vectors may be specified.

Tr

An *nd1*-by-*nt* array containing the indices of either the convex hull (*nd1* is equal to *nd*), or the Delaunay triangulation (*nd1* is equal to *nd*+1) of the input points.

Keywords

BOUNDS

If set to a variable name, return the indices of the points on the convex hull of the input points.

CONNECTIVITY

Set this keyword to a named variable in which the adjacency list for each of the np nodes is returned. The list has the following form:

Each element i , $0 \leq i < np$, contains the starting index of the connectivity list (*list*) for node i within the list array. To obtain the adjacency list for node i , extract the list elements from $list[i]$ to $list[i+1] - 1$. The adjacency list is not ordered. To obtain the connectivity list, either the DELAUNAY or SPHERE keywords must also be specified.

For example, to perform a spherical triangulation, use the following procedure call:

```
QHULL, lon, lat, CONNECTIVITY = list
```

which returns the adjacency list in the variable *list*. The subscripts of the nodes adjacent to $lon[i]$ and $lat[i]$ are contained in the array: $list[list[i] : list[i+1] - 1]$.

DELAUNAY

Performs a Delaunay triangulation and returns the vertex indices of the resulting polyhedra; otherwise, the convex hull of the data are returned.

SPHERE

Computes the Delaunay triangulation of the points which lie on the surface of a sphere. The *V0* argument contains the longitude, in degrees, and *V1* contains the latitude, in degrees, of each point.

VDIAGRAM

When specified, this keyword returns the connectivity of the Voronoi diagram in a 4-by- nv integer array. For each Voronoi ridge, i , VDIAGRAM[0:1, i] contains the index of the two input points the ridge bisects. VDIAGRAM[2:3, i] contains the indices of the Voronoi vertices.

In the case of an unbounded half-space, VDIAGRAM[2, i] is set a negative index, j , indicating that the corresponding Voronoi ridge is unbounded, and that the equation for the ridge is contained in VNORMAL[*, $-j$], and starts at Voronoi vertex [3, i].

VNORMALS

When specified, this keyword returns the normals of each Voronoi ridge that is unbounded. See the description of VDIAGRAM.

VVERTICES

When specified, this keyword returns the Voronoi vertices.

Example

For some examples using the QHULL procedure, see the [QGRID3](#) function.

See Also

[QGRID3](#)

QUERY_MRSID

The QUERY_MRSID function allows you to obtain information about a MrSID image file without having to read the file. It is a wrapper around the object interface that presents MrSID image loading in a familiar way to users of the QUERY_* image routines. However this function is not as efficient as the object interface and the object interface should be used whenever possible. See “IDLffMrSID” in Chapter 5 “New Objects” for information about the object interface.

Syntax

Result = QUERY_MRSID(*Filename* [, *Info*] [, LEVEL=*lvl*])

Return Value

Result is a long integer with the value of:

- 1 – If the query was successful (and the file type was correct).
- 0 – If the query fails.

Arguments

Filename

A scalar string containing the full path and filename of the MrSID file to query.

Info

Returns an anonymous structure containing information about the image in the file. The Info.TYPE field will return the value 'MrSID'.

The anonymous structure is detailed in the QUERY_* Routines documentation. However, the info structure filled in by QUERY_MRSID has additional members appended to the end:

- info.LEVELS – a named variable that will contain a two-element integer vector of the form [minlvl, maxlvl] that specifies the range of levels within the current image. Higher levels are lower resolution. A level of 0 equals full resolution. Negative values specify higher levels of resolution.
- Info.GEO_VALID – a long integer with a value of 1 if the file contains valid georeferencing data, or 0 if the georeferencing data is nonexistent or unsupported.

Note

Always verify that this keyword returns 1 before using the data returned by any other GEO_* keyword.

- Info.GEO_PROJTYPE – unsigned integer.
- Info.GEO_ORIGIN – 2-element double precision array.
- Info.GEO_RESOLUTION – 2-element double precision array.

See “IDLffMRSID::GetProperty” in Chapter 5 for more information on GEO_* values.

Keywords**LEVEL**

Set this keyword to an integer that specifies the level to which the DIMENSIONS field of the info structure corresponds. This can be used, for example, to determine what level is required to fit the image into a certain area. If this keyword is not specified, the dimensions at level 0 are returned.

Example

```
; Select the image file.
file = QUERY_MRSID(FILEPATH('test_gs.sid', $
    SUBDIRECTORY=['examples', 'data']), info, LEVEL = -2)

HELP, file
; IDL returns 1 indicating the correct file type
; and successful query.

; Print the range of levels of resolution available within
; the file.
PRINT, 'Range of image levels = ', info.LEVELS

; Print the image dimensions when the image level is set to -2
; as specified by LEVEL = -2 in the QUERY_MRSID statement.
PRINT, 'dimensions of image at LEVEL is -2 =', info.DIMENSIONS
; IDL returns 2048 by 2048

; Check for valid georeferencing data.
PRINT, 'Result of georeferencing query', info.GEO_VALID
; IDL returns 0 indicating that the file does not contain
; georeferencing data.
```

READ_MRSID

The new READ_MRSID function extracts and returns image data from a MrSID file at the specified level and location. It is a wrapper around the object interface that presents MrSID image loading in a familiar way to users of the READ_* image routines. However this function is not as efficient as the object interface and the object interface should be used whenever possible. See “IDLffMrSID” in Chapter 5 for information about the object interface.

Syntax

Result = READ_MRSID (*Filename* [, LEVEL=*lvl*] [, SUB_RECT=*rect*])

Return Value

ImageData returns an *n*-by-*w*-by-*h* array containing the image data where *n* is 1 for grayscale or 3 for RGB images, *w* is the width and *h* is the height.

Note

The returned image is ordered bottom-up, the first pixel returned is located at the bottom-left of the image. This differs from how data is stored in the MrSID file where the image is top-down, meaning the pixel at the start of the file is located at the top-left of the image.

Arguments

Filename

A scalar string containing the full path and filename of the MrSID file to read.

Keywords

LEVEL

Set this keyword to an integer that specifies the level at which to read the image. If this keyword is not set, the maximum level (see QUERY_MRSID) is used which returns the minimum resolution.

SUB_RECT

Set this keyword to a four-element vector [*x*, *y*, *xdim*, *ydim*] specifying the position of the lower left-hand corner and the dimensions of the sub-rectangle of the MrSID

image to return. This is useful for displaying only a portion of the high-resolution image. If this keyword is not set, the entire image will be returned. This may require significant memory if a high-resolution level is selected. If the sub-rectangle is greater than the bounds of the image at the selected level the area outside the image bounds will be set to black.

Note

The elements of SUB_RECT are measured in pixels at the current level. This means the point $x = 10$, $y = 10$ at level 1 will be located at $x = 20$, $y = 20$ at level 0 and $x = 5$, $y = 5$ at level 2.

Example

```
; Query the file.
result = QUERY_MRSID(FILEPATH('test_gs.sid', $
    SUBDIRECTORY = ['examples', 'data']), info)

; If result is not zero, read in an image from the file and
; display it.
IF (result NE 0) THEN BEGIN
    PRINT, info
    imageData = READ_MRSID(FILEPATH('test_gs.sid', $
        SUBDIRECTORY = ['examples', 'data']), SUB_RECT = $
        [0, 0, 200, 200], LEVEL = 3)
    oImage = OBJ_NEW('IDLgrImage', imageData, ORDER = 0)
    XOBJVIEW, oImage, BACKGROUND = [255,255,0]
ENDIF

; Use the file access object to query the file.
oMrSID = OBJ_NEW('IDLffMrSID', FILEPATH('test_gs.sid', $
    SUBDIRECTORY = ['examples', 'data']))
oMrSID -> GetProperty, PIXEL_TYPE=pt, $
    CHANNELS = chan, DIMENSIONS = dims, $
    TYPE = type, LEVELS = lvls
PRINT, pt, chan, dims, type, lvls

; Use the object to read in an image from the file.
lvls = -3
dimsatlvl = oMrSID -> GetDimsAtLevel(lvls)
PRINT, dimsatlvl
imageData = oMrSID -> GetImageData(LEVEL = 3)
PRINT, size(imageData)
OBJ_DESTROY, oImage
```

REAL_PART

The `REAL_PART` function returns the real part of its complex-valued argument. If the complex-valued argument is double-precision, the result will be double-precision, otherwise the result will be single-precision floating-point. If the argument is not complex, then the result will be double-precision if the argument is double-precision, otherwise the result will be single-precision.

Syntax

Result = `REAL_PART(Z)`

Arguments

Z

A scalar or array for which the real part is desired. *Z* may be of any numeric type.

Example

The following example demonstrates how you can use `REAL_PART` to obtain the real parts of an array of complex variables.

```
; Create an array of complex values:
cValues = COMPLEX([1, 2, 3],[4, 5, 6])

; Print just the real parts of each element in cValues:
PRINT, REAL_PART(cValues)
```

IDL prints:

```
1.00000    2.00000    3.00000
```

See Also

[COMPLEX](#), [DCOMPLEX](#), [IMAGINARY](#)

REGION_GROW

The `REGION_GROW` function performs region growing for a given region within an N-dimensional array by finding all pixels within the array that are connected neighbors to the region pixels and that fall within provided constraints. The constraints are specified either as a threshold range (a minimum and maximum pixel value) or as a multiple of the standard deviation of the region pixel values. If the threshold is used (this is the default), the region is grown to include all connected neighboring pixels that fall within the given threshold range. If the standard deviation multiplier is used, the region is grown to include all connected neighboring pixels that fall within the range of the mean (of the region's pixel values) plus or minus the given multiplier times the sample standard deviation. `REGION_GROW` returns the vector of array indices that represent pixels within the grown region. The grown region will not include pixels at the edges of the input array. If no pixels fall within the grown region, this function will return the value -1.

Syntax

```
Result = REGION_GROW(Array, ROIPixels [, /ALL_NEIGHBORS]  
[, STDDEV_MULTIPLIER=value | THRESHOLD=[min,max]] )
```

Arguments

Array

An N-dimensional array of data values. The region will be grown according to the data values within this array.

ROI

Pixels

A vector of indices into *Array* that represent the initial region that is to be grown.

Keywords

ALL_NEIGHBORS

Set this keyword to indicate that all adjacent neighbors to a given pixel should be considered during region growing (sometimes known as 8-neighbor searching when the array is two-dimensional). The default is to search only the neighbors that are exactly one unit in distance from the current pixel (sometimes known as 4-neighbor searching when the array is two-dimensional).

STDDEV_MULTIPLIER

Set this keyword to a scalar value that serves as the multiplier of the sample standard deviation of the original region pixel values. The expanded region includes neighboring pixels that fall within the range of the mean of the region's pixel values plus or minus the given multiplier times the sample standard deviation as follows:

$\text{Mean} \pm \text{StdDevMultiplier} * \text{StdDev}$

This keyword is mutually exclusive of THRESHOLD. If both keywords are specified, a warning message will be issued and the THRESHOLD value will be used.

THRESHOLD

Set this keyword to a two-element vector, $[\text{min}, \text{max}]$, of the inclusive range within which the pixel values of the grown region must fall. The default is the range of pixel values within the initial region. This keyword is mutually exclusive of STDDEV_MULTIPLIER. If both keywords are specified, a warning message will be issued and the THRESHOLD value will be used.

Note

If neither keyword is specified, THRESHOLD is used by default. The range of threshold values is based upon the pixel values within the original region and therefore does not have to be provided.

Example

The following example demonstrates how you can grow a pre-defined region within an image of human red blood cells.

```
; Load an image.
fname = FILEPATH('rbcells.jpg', SUBDIR=['examples','data'])
READ_JPEG, fname, img
imgDims = SIZE(img, /DIMENSIONS)

; Define original region pixels.
x = FINDGEN(16*16) MOD 16 + 276.
y = LINDGEN(16*16) / 16 + 254.
roiPixels = x + y * imgDims[0]

; Grow the region.
newROIPixels = REGION_GROW(img, roiPixels)

; Load a grayscale color table.
```

```
DEVICE, DECOMPOSED = 0
LOADCT, 0

; Set the topmost color table entry to red.
topClr = !D.TABLE_SIZE-1
TVLCT, 255, 0, 0, topClr

; Show the results.
tmpImg = BYTSCL(img, TOP=(topClr-1))
tmpImg[roiPixels] = topClr
WINDOW, 0, XSIZE=imgDims[0], YSIZE=imgDims[1], $
    TITLE='Original Region'
TV, tmpImg

tmpImg = BYTSCL(img, TOP=(topClr-1))
tmpImg[newROIPixels] = topClr
WINDOW, 2, XSIZE=imgDims[0], YSIZE=imgDims[1], $
    TITLE='Grown Region'
TV, tmpImg
```

SIMPLEX

The SIMPLEX function uses the simplex method to solve linear programming problems. Given a set of N independent variables X_i , where $i = 1, \dots, N$, the simplex method seeks to maximize the following function,

$$Z = a_1 X_1 + a_2 X_2 + \dots a_N X_N$$

with the assumption that $X_i \geq 0$. The X_i are further constrained by the following equations:

$$b_{j1} X_1 + b_{j2} X_2 + \dots b_{jN} X_N \leq c_j \quad j = 1, 2, \dots, M_1$$

$$b_{j1} X_1 + b_{j2} X_2 + \dots b_{jN} X_N \geq c_j \quad j = M_1 + 1, M_1 + 2, \dots, M_1 + M_2$$

$$b_{j1} X_1 + b_{j2} X_2 + \dots b_{jN} X_N = c_j \quad j = M_1 + M_2 + 1, M_1 + M_2 + 2, \dots, M$$

where $M = M_1 + M_2 + M_3$ is the total number of equations, and the constraint values c_j must all be positive.

To solve the above problem using the SIMPLEX function, the Z equation is rewritten as a vector:

$$Z_{\text{equation}} = [a_1 \ a_2 \ \dots a_N]$$

The constraint equations are rewritten as a matrix with $N+1$ columns and M rows, where all of the b coefficients have had their sign reversed:

$$\text{Constraints} = \begin{bmatrix} c_1 & -b_{11} & -b_{12} \dots -b_{1N} \\ c_2 & -b_{21} & -b_{22} \dots -b_{2N} \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ c_M & -b_{M1} & -b_{M2} \dots -b_{MN} \end{bmatrix}$$

Note

The constraint matrix must be organized so that the coefficients for the less-than ($<$) equations come first, followed by the coefficients of the greater-than ($>$) equations, and then the coefficients of the equal ($=$) equations.

The *Result* is a vector of $N+1$ elements containing the maximum Z value and the values of the N independent X variables (the optimal feasible vector):

$$\text{Result} = \begin{bmatrix} Z_{\max} & X_1 & X_2 & \dots & X_N \end{bmatrix}$$

The SIMPLEX function is based on the routine `simplex` described in section 10.8 of *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press, and is used by permission.

Syntax

Result = SIMPLEX(*Zequation*, *Constraints*, *M1*, *M2*, *M3*
[, *Tableau* [, *Izrov* [, *Iposv*]]] [, /DOUBLE] [, EPS = *value*] [, STATUS = *variable*])

Arguments

Zequation

A vector containing the N coefficients of the $Z_{equation}$ to be maximized.

Constraints

An array of $N+1$ columns by M rows containing the constraint values and coefficients for the constraint equations.

M1

An integer giving the number of less-than constraint equations contained in *Constraints*. *M1* may be zero, indicating that there are no less than constraints.

M2

An integer giving the number of greater-than constraint equations contained in *Constraints*. *M2* may be zero, indicating that there are no greater than constraints.

M3

An integer giving the number of equal-to constraint equations contained in *Constraints*. *M3* may be zero, indicating that there are no equal to constraints. The total of $M1 + M2 + M3$ should equal M , the number of constraint equations.

Tableau

Set this optional argument to a named variable in which to return the output array from the simplex algorithm. For more detailed discussion about this argument, see the write-up in section 10.8 of *Numerical Recipes in C*.

izrov

Set this optional argument to a named variable in which to return the output izrov variable from the simplex algorithm. For more detailed discussion about this argument, see the write-up in section 10.8 of *Numerical Recipes in C*.

iposv

Set this optional argument to a named variable in which to return the output iposv variable from the simplex algorithm. For more detailed discussion about this argument, see the write-up in section 10.8 of *Numerical Recipes in C*.

Keywords**DOUBLE**

Set this keyword to use double-precision for computations and to return a double-precision result. Set DOUBLE to 0 to use single-precision for computations and to return a single-precision result. The default is /DOUBLE if any of the inputs are double-precision, otherwise the default is 0.

EPS

Set this keyword to a number close to machine accuracy, which is used to test for convergence at each iteration. The default is 10^{-6} .

STATUS

Set this keyword to a named variable to receive the status of the operation. Possible status values are:

Value	Description
0	Successful completion.
1	The objective function is unbounded.
2	No solution satisfies the given constraints.
3	The routine did not converge.

Table 6-3: *SIMPLEX Function Status Values*

Example

The following example is taken from *Numerical Recipes in C*.

Find the Z value which maximizes the equation $Z = X_1 + X_2 + 3 X_3 - 0.5 X_4$, with the following constraints:

$$X_1 + 2X_3 \leq 740$$

$$2X_2 - 7X_4 \leq 0$$

$$X_2 - X_3 + 2X_4 \geq 0.5$$

$$X_1 + X_2 + X_3 + X_4 = 9$$

To find the solution, enter the following code at the IDL command line:

```
; Set up the Zequation with the X coefficients.
Zequation = [1,1,3,-0.5]
; Set up the Constraints matrix.
Constraints = [ $
    [740, -1, 0, -2, 0], $
    [ 0, 0, -2, 0, 7], $
    [0.5, 0, -1, 1, -2], $
    [ 9, -1, -1, -1, -1] ]
; Number of less-than constraint equations.
m1 = 2
; Number of greater-than constraint equations.
m2 = 1
; Number of equal constraint equations.
m3 = 1
;
; Call the function.
result = SIMPLEX(Zequation, Constraints, m1, m2, m3)
;
; Print out the results.
PRINT, 'Maximum Z value is: ', result[0]
PRINT, 'X coefficients are: '
PRINT, result[1:*
```

IDL prints:

```
Maximum Z value is:      17.0250
X coefficients are:
      0.000000      3.32500      4.72500      0.950000
```

Therefore, the optimal feasible vector is $X_1 = 0.0$, $X_2 = 3.325$, $X_3 = 4.725$, and $X_4 = 0.95$.

See Also

[AMOEB](#), [DFPMIN](#), [POWELL](#)

WIDGET_ACTIVEX

The `WIDGET_ACTIVEX` function is used to create an ActiveX control in IDL and also to place it into an IDL widget hierarchy. The program or class ID of the underlying IDL object that represents the ActiveX control is retrieved using the `GET_VALUE` keyword of `WIDGET_CONTROL`. This is similar to the operations used to get the window object from an IDL draw widget.

Note

IDL ActiveX functionality is only supported on the Windows NT and Windows 2000 platforms.

Note

This is the only method to create an IDL object that represents an ActiveX control. Creating an ActiveX control (an object based off the class name prefix `IDLcomActiveX$`) using `OBJ_NEW()` is not supported and the results are undefined.

Note

All ActiveX based objects created in IDL sub-class from the intrinsic IDL class `IDLcomActiveX`, which is a sub-class from `IDLcomIDispatch`.

Syntax

```
Result = WIDGET_ACTIVEX( Parent, COM_ID, [, /ALIGN_BOTTOM | ,
/ALIGN_CENTER | , /ALIGN_LEFT | , /ALIGN_RIGHT | , /ALIGN_TOP]
[, EVENT_FUNC=string] [, EVENT_PRO=string] [, FUNC_GET_VALUE=string]
[ID_TYPE=value] [, KILL_NOTIFY=string] [, /NO_COPY]
[, NOTIFY_REALIZE=string] [, PRO_SET_VALUE=string] [, SCR_XSIZE=width]
[, SCR_YSIZE=height] [, /SENSITIVE] [, UNAME=string] [, UNITS={0 | 1 | 2}]
[, UVALUE=value] [, XOFFSET=value] [, XSIZE=value] [, YOFFSET=value]
[, YSIZE=value] )
```

Arguments

Parent

The widget ID of the parent widget of the new ActiveX control.

COM_ID

The class or program ID of the COM object to create.

Note

The provided Class ID or program ID must follow the standard Microsoft naming convention. So Class IDs will contain '{}' brackets and use '-' as a separator and Program IDs will use a '.' for a separator. The use of '_' is only used with IDispatch objects in the call to OBJ_NEW() because the object name must follow standard IDL object naming syntax.

Keywords

ALIGN_BOTTOM

Set this keyword to align the new widget with the bottom of its parent base. To take effect, the parent must be a ROW base.

ALIGN_CENTER

Set this keyword to align the new widget with the center of its parent base. To take effect, the parent must be a ROW or COLUMN base. In ROW bases, the new widget will be vertically centered. In COLUMN bases, the new widget will be horizontally centered.

ALIGN_LEFT

Set this keyword to align the new widget with the left side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_RIGHT

Set this keyword to align the new widget with the right side of its parent base. To take effect, the parent must be a COLUMN base.

ALIGN_TOP

Set this keyword to align the new widget with the top of its parent base. To take effect, the parent must be a ROW base.

EVENT_FUNC

A string containing the name of a function to be called by the WIDGET_EVENT function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

EVENT_PRO

A string containing the name of a procedure to be called by the `WIDGET_EVENT` function when an event arrives from a widget in the widget hierarchy rooted at the newly-created widget.

Note

If the base is a top-level base widget that is managed by the `XMANAGER` procedure, any value specified via the `EVENT_PRO` keyword is overridden by the value of the `EVENT_HANDLER` keyword to `XMANAGER`. Note also that in this situation, if `EVENT_HANDLER` is not specified in the call to `XMANAGER`, an event-handler name will be created by appending the string “_event” to the application name specified to `XMANAGER`. This means that there is no reason to specify this keyword for a top-level base that will be managed by the `XMANAGER` procedure.

FUNC_GET_VALUE

A string containing the name of a function to be called when the `GET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to change the value that should be returned for a widget. Compound widgets use this ability to define their values transparently to the user.

ID_TYPE

The type of COM control ID passed in (class or program). If set to 0, the ID is a class ID (the default) and if set to 1, the ID is a program ID.

The following keywords are accepted by all IDL Widget types and are also accepted by `WIDGET_ACTIVEX`. The keywords are only enumerated in this specification. For details on how they operate, consult the IDL Reference Guide.

KILL_NOTIFY

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget dies. Each widget is allowed a single such callback procedure. It can be removed by setting the routine to the null string (' '). Note that the procedure specified is used only if you are not using the `XMANAGER` procedure to manage your widgets.

The callback routine is called with the widget identifier as its only argument. At that point, the widget identifier can only be used with the `WIDGET_CONTROL` procedure to get or set the user value. All other requests that require a widget ID are

disallowed for the target widget. The callback is not issued until the `WIDGET_EVENT` function is called.

If you use the `XMANAGER` procedure to manage your widgets, the value of this keyword is overwritten. Use the `CLEANUP` keyword to `XMANAGER` to specify a procedure to be called when a managed widget dies.

NO_COPY

Usually, when setting or getting widget user values, either at widget creation or using the `SET_UVALUE` and `GET_UVALUE` keywords to `WIDGET_CONTROL`, IDL makes a second copy of the data being transferred. Although this technique is fine for small data, it can have a significant memory cost when the data being copied is large.

If the `NO_COPY` keyword is set, IDL handles these operations differently. Rather than copy the source data, it takes the data away from the source and attaches it directly to the destination. This feature can be used by compound widgets to obtain state information from a `UVALUE` without all the memory copying that would otherwise occur. However, it has the side effect of causing the source variable to become undefined. On a “set” operation (using the `UVALUE` keyword to `WIDGET_BASE` or the `SET_UVALUE` keyword to `WIDGET_CONTROL`), the variable passed as value becomes undefined. On a “get” operation (`GET_UVALUE` keyword to `WIDGET_CONTROL`), the user value of the widget in question becomes undefined.

NOTIFY_REALIZE

Set this keyword to a string that contains the name of a procedure to be called automatically when the specified widget is realized. This callback occurs just once (because widgets are realized only once). Each widget is allowed a single such callback procedure. It can be removed by setting the routine to the null string (' '). The callback routine is called with the widget ID as its only argument.

PRO_SET_VALUE

A string containing the name of a procedure to be called when the `SET_VALUE` keyword to the `WIDGET_CONTROL` procedure is called for this widget. Using this technique allows you to designate a routine that sets the value for a widget. Compound widgets use this ability to define their values transparently to the user.

SCR_XSIZE

Set this keyword to the desired “screen” width of the widget, in units specified by the `UNITS` keyword (pixels are the default). In many cases, setting this keyword is the same as setting the `XSIZE` keyword.

SCR_YSIZE

Set this keyword to the desired “screen” height of the widget, in units specified by the UNITS keyword (pixels are the default). In many cases, setting this keyword is the same as setting the YSIZE keyword.

SENSITIVE

Set this keyword to control the initial sensitivity state of the widget.

If SENSITIVE is zero, the widget becomes insensitive. If nonzero, it becomes sensitive. When a widget is sensitive, it has normal appearance and can receive user input. For example, a sensitive button widget can be activated by moving the mouse cursor over it and pressing a mouse button. When a widget is insensitive, it indicates the fact by changing its appearance, looking disabled, and it ignores any input.

Sensitivity can be used to control when a user is allowed to manipulate the widget. Note that some widgets do not change their appearance when they are made insensitive, but they cease generating events.

After creating the widget hierarchy, you can change the sensitivity state using the SENSITIVE keyword with the [WIDGET_CONTROL](#) procedure.

UNAME

Set this keyword to a string that can be used to identify the widget in your code. You can associate a name with each widget in a specific hierarchy, and then use that name to query the widget hierarchy and get the correct widget ID.

To query the widget hierarchy, use the [WIDGET_INFO](#) function with the FIND_BY_UNAME keyword. The UNAME should be unique to the widget hierarchy because the FIND_BY_UNAME keyword returns the ID of the widget with the specified name.

UNITS

Set UNITS equal to 0 (zero) to specify that all measurements are in pixels (this is the default), to 1 (one) to specify that all measurements are in inches, or to 2 (two) to specify that all measurements are in centimeters.

UVALUE

The “user value” to be assigned to the widget.

Each widget can contain a user-specified value of any data type and organization. This value is not used by the widget in any way, but exists entirely for the

convenience of the IDL programmer. This keyword allows you to set this value when the widget is first created.

If UVALUE is not present, the widget's initial user value is undefined.

The user value for a widget can be accessed and modified at any time by using the GET_UVALUE and SET_UVALUE keywords to the WIDGET_CONTROL procedure.

XOFFSET

The horizontal offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

XSIZE

The width of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

YOFFSET

The vertical offset of the widget in units specified by the UNITS keyword (pixels are the default) relative to its parent. This offset is specified relative to the *upper left* corner of the parent widget.

Specifying an offset relative to a row or column major base widget does not work because those widgets enforce their own layout policies. This keyword is primarily of use relative to a plain base widget. Note that it is best to avoid using this style of widget layout.

YSIZE

The height of the widget in units specified by the UNITS keyword (pixels are the default). Most widgets attempt to size themselves to fit the situation. However, if the desired effect is not produced, use this keyword to override it. This keyword is only a “hint” to the toolkit and may be ignored in some situations.

Examples

For examples using `WIDGET_ACTIVEX`, see [Chapter 3, “Using COM Objects in IDL”](#).

WIDGET_DISPLAYCONTEXTMENU

The `WIDGET_DISPLAYCONTEXTMENU` procedure displays a shortcut menu (otherwise known as a context sensitive or pop-up menu). After buttons for the context menu have been created a context menu can be displayed using `WIDGET_DISPLAYCONTEXTMENU`. This is normally called in an event handler that has processed a context menu event. This procedure takes the ID of the widget that is the parent of the context menu, the x and y location to display the menu, and the ID of the context menu base. The ID would normally be the `event.id` value of the context menu event, and the x and y locations also come from the context event. As stated above, there may be multiple context menus for a particular widget. The last parameter of `WIDGET_DISPLAYCONTEXTMENU` allows the user to specify which menu to display. In the case of a draw widget that is the parent of a context menu, the x and y locations can be obtained from the button event structure.

When `WIDGET_DISPLAYCONTEXTMENU` is called it displays the context menu and handles the native event if the user selects a button. If a button is selected a user button event is generated and the menu is dismissed. If no button is selected (the user clicks elsewhere on the screen) then the menu is dismissed and no user event is generated. Normally no further processing would be done in the context event or draw event handler after calling `WIDGET_DISPLAYCONTEXTMENU`. The new user event is queued and will be handled in a new call to the event handler.

Syntax

`WIDGET_DISPLAYCONTEXTMENU, Parent, X, Y, ContextBase_ID`

Arguments

Parent

The widget ID of the parent of a context menu.

X

The x location, relative to the parent widget, to display the menu.

Y

The y location, relative to the parent widget, to display the menu.

ContextBase_ID

The widget ID of the context menu base that is the head of the menu to display. Use the `CONTEXT_MENU` keyword to `WIDGET_BASE` to create a context menu base. This base must be a child of the widget supplied with the `Parent` argument.

Keywords

None.

Examples

For examples using `WIDGET_DISPLAYCONTEXTMENU`, see [Chapter 4, “Using the Shortcut Menu Widget”](#).

XOBJVIEW_ROTATE

The XOBJVIEW_ROTATE procedure is used to programmatically rotate the object currently displayed in XOBJVIEW. XOBJVIEW must be called prior to calling XOBJVIEW_ROTATE. This procedure can be used to create animations of object displays.

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

Syntax

XOBJVIEW_ROTATE, *Axis*, *Angle* [, /PREMULTIPLY]

Arguments

Axis

A 3-element vector of the form $[x, y, z]$ describing the axis about which the model is to be rotated.

Angle

The amount of rotation, measured in degrees.

Keywords

PREMULTIPLY

Set this keyword to cause the rotation matrix specified by *Axis* and *Angle* to be pre-multiplied to the model's transformation matrix. By default, the rotation matrix is post-multiplied.

Example

The following example creates an animation of the test object (a surface) currently displayed in XOBJVIEW. It does this by rotating the surface through 360 degrees in increments of 10 degrees using XOBJVIEW_ROTATE, and writing the display image to a BMP file for each increment using XOBJVIEW_WRITE_IMAGE.

```
PRO RotateAndWriteObject

XOBJVIEW, /TEST
FOR i = 0, 359 DO BEGIN
    XOBJVIEW_ROTATE, [0, 1, 0], 1, /PREMULTIPLY;
    XOBJVIEW_WRITE_IMAGE, 'img' + $
        STRCOMPRESS(i, /REMOVE_ALL) + '.bmp', 'bmp'
ENDFOR

END
```

See Also

[XOBJVIEW](#), [XOBJVIEW_WRITE_IMAGE](#)

XOBJVIEW_WRITE_IMAGE

The XOBJVIEW_WRITE_IMAGE procedure is used to write the object currently displayed in XOBJVIEW to an image file with the specified name and file format. XOBJVIEW must be called prior to calling XOBJVIEW_WRITE_IMAGE.

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

Syntax

`XOBJVIEW_WRITE_IMAGE, Filename, Format [, DIMENSIONS=[x, y]]`

Arguments

Filename

A scalar string containing the name of the file to write.

Format

A scalar string containing the name of the file format to write. See `QUERY_IMAGE` for a list of supported formats.

Keywords

DIMENSIONS

Set this keyword to a 2-element vector of the form [*x*, *y*] specifying the size of the output image, in pixels. If this keyword is not specified, the image will be written using the dimensions of the current XOBJVIEW draw widget.

Example

See [XOBJVIEW_ROTATE](#).

See Also

[XOBJVIEW](#), [XOBJVIEW_ROTATE](#)

XROI

The XROI procedure is an existing procedure but has been enhanced substantially in IDL 5.5. This utility is used 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 'Translate-Scale',
'Rectangle', 'Ellipse', '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 previously defined regions of interest (see [Example 3](#)). 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** → **ROI 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 6-4: 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** → **ROI 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 6-5: 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:

- 'Translate-Scale' — Translation and scaling of ROIs. Mouse down inside the bounding box selects a region, mouse motion translates (repositions) the region. Mouse down on a scale handle of the bounding box enables scaling (stretching, enlarging and shrinking) of the region according to mouse motion. Mouse up finishes the translation or scaling.

- 'Rectangle' — Rectangular ROI drawing. Mouse down positions one corner of the rectangle, mouse motion creates the rectangle, positioning the rectangle's opposite corner, mouse up finishes the rectangular region.
- 'Ellipse' — Elliptical ROI drawing. Mouse down positions the center of the ellipse, mouse motion positions the corner of the ellipse's imaginary bounding box, mouse up finishes the elliptical region.
- '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:

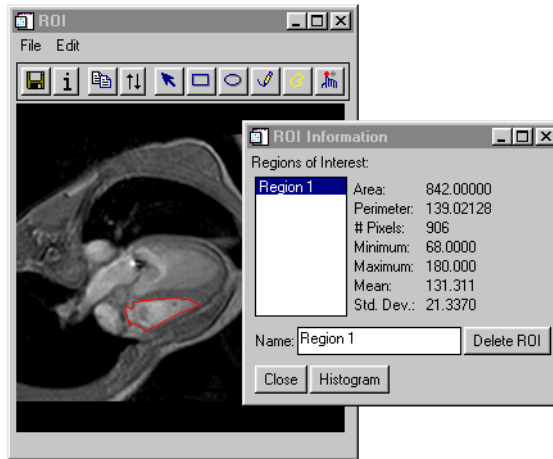






Figure 6-1: The XROI Utility

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:

	Translate/ Scale:	Click this button to translate or scale ROIs. Mouse down inside the bounding box selects a region, mouse motion translates (repositions) the region. Mouse down on a scale handle of the bounding box enables scaling (stretching, enlarging and shrinking) of the region according to mouse motion. Mouse up finishes the translation or scaling.
	Draw Rectangle:	Click this button to draw rectangular ROIs. Mouse down positions one corner of the rectangle, mouse motions creates the rectangle, positioning the rectangle's opposite corner, mouse up finishes the rectangular region.
	Draw Ellipse:	Click this button to draw elliptical ROIs. Mouse down positions the center of the ellipse, mouse motion positions the corner of the ellipse's imaginary bounding box, mouse up finishes the elliptical region.
	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** → **ROI 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, mean, and standard deviation. Values for statistical information (minimum, maximum, mean, and standard deviation) appear as “N/A” for 24-bit images.

Viewing a Histogram Plot for an ROI

To view a histogram for an ROI, use either the shortcut menu or the ROI Information dialog.

To view an ROI’s histogram plot using the shortcut menu:

1. Position the cursor on the line defining the boundary of an ROI in the drawing window and click the right mouse button. This selects the region and brings up its shortcut menu.
2. Select the **Plot Histogram** menu option from the shortcut menu.

To view an ROI’s histogram plot using the ROI Information dialog:

1. Open the ROI Information dialog by clicking the **Info** button or selecting **Edit** → **ROI Information**.
2. Select a region from the list and click the **Histogram** button on the ROI Information dialog.

Either of the previous methods opens a LIVE_PLOT dialog showing the ROI’s histogram that can be used to interactively control the plot properties.

Note

XROI’s histogram plot feature now supports RGB images.

Growing an ROI

Once a region has been created, it may be used as a source ROI for region growing. Region growing is a process of generating one or more new ROIs based upon the image pixel values that fall within the source ROI and the values of the neighboring pixels. New pixels are added to the new grown region if those image pixel values fall within a specified threshold.

Note

This option is an interactive implementation of the `REGION_GROW` function.

To create a new, grown region, do the following:

1. Within the draw area, click the right mouse button on the ROI that is to be grown. This will select the region and bring up its shortcut menu.
2. Select **Grow Region** → **By threshold** or select **Grow Region** → **By std. dev. multiple** from the shortcut menu to control how the region is grown.

The **By threshold** option grows the region to include all neighboring pixels that fall within a specified threshold range. By default, the range is defined by the minimum and maximum pixel values occurring within the original region. To specify a different threshold range, see [Using the Region Grow Properties Dialog](#) in the following section.

The **By std. dev. multiple** option grows a region to include all neighboring pixels that fall within the range of:

$$\text{Mean} \pm \text{StdDevMultiplier} * \text{StdDev}$$

where `Mean` is the mean value of the pixel values within the source ROI, `StdDevMultiplier` is a multiplier that is set using the Region Grow Properties dialog (described below), and `StdDev` is the sample standard deviation of the pixel values within the original region.

Using the Region Grow Properties Dialog

The Region Grow Properties dialog allows you to view and edit the properties associated with a region growing process. To bring up the Region Grow Properties dialog, do one of the following:

- Click the right mouse button on an ROI in the drawing window and select **Grow Region** → **Properties...** shortcut menu option.
- Select **Edit** → **Region Grow Properties...** from the XROI menu bar.

This brings up the Region Grow Properties dialog, shown in the following figure.

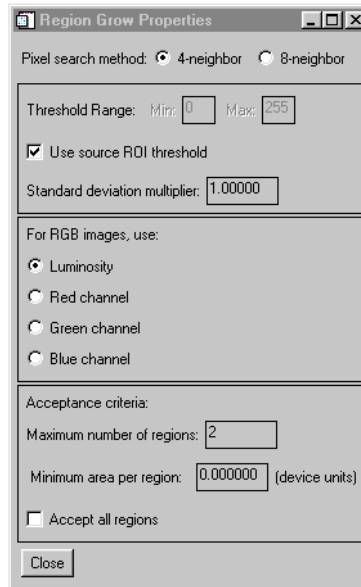


Figure 6-2: XROI's Region Grow Properties Dialog

The Region Grow Properties dialog offers the following options:

Option	Description
Pixel search method:	<p>Describes which pixels are searched when growing the original ROI. The option are:</p> <ul style="list-style-type: none"> • 4-neighbor — Searches only the four neighboring pixels that share a common edge with the current pixel. This is the default. • 8-neighbor — Searches all eight neighboring pixels, including those that are located diagonally relative to the original pixel and share a common corner.

Table 6-6: Options of the Region Grow Properties Dialog

Option	Description
Threshold range:	<p>Represents the minimum and maximum image pixel values that are to be included in the grown region when using the Grow Region → By threshold option (described in “Growing an ROI” on page 312). By default, the range of pixel values used are those occurring in the ROI to be grown.</p> <p>To change the threshold values, uncheck Use source ROI threshold and enter the minimum and maximum threshold values in the Min: and Max: fields provided.</p>
Standard deviation multiplier:	<p>Represents the factor by which the sample standard deviation of the original ROI’s pixel values is multiplied. This factor only applies when the Grow Region → By std. dev. multiple option (described in “Growing an ROI” on page 312) is used.</p> <p>Change the multiplier value by typing the value into the Standard deviation multiplier field provided.</p>
For RGB image, use:	<p>Determines the basis of region growing for an RGB (rather than indexed) image. The image data values used when growing a RGB region can be one of the following:</p> <ul style="list-style-type: none"> • Luminosity — Uses the luminosity values associated with an RGB image. This is the default method. Luminosity is computed as: $\text{Luminosity} = (0.3 * \text{Red}) + (0.59 * \text{Green}) + (0.11 * \text{Blue})$ • Red Channel, Green Channel or Blue Channel — Uses the ROI’s red, green or blue channel as a basis for region growing. Click the channel’s associated button to specify the channel to be used. <p>Note - For indexed images, the image data itself is always used for region growing.</p>

Table 6-6: Options of the Region Grow Properties Dialog (Continued)

Option	Description
Acceptance criteria:	<p>Determines which contours of the grown region are accepted as new regions, (which will also be displayed in the draw area and in the ROI Information dialog list of regions). The region growing process can result in a large number of contours, some of which may be considered insignificant. By default, no more than two regions (those with the greatest geometrical area) are accepted. Modify the acceptance criteria by altering the following values:</p> <ul style="list-style-type: none"> • Maximum number of regions: — Specifies the upper limit of the number of regions to create when growing an ROI. • Minimum area per region: — Specifies that only contours having a geometric area (computed in device coordinates) of at least the value stated are accepted and displayed. • Accept all regions: — Select this option to accept all generated contours, regardless of count or area.

Table 6-6: Options of the Region Grow Properties Dialog (Continued)

Deleting an ROI

An ROI can be deleted using either the shortcut menu or using the ROI Information dialog.

To delete an ROI using the shortcut menu:

1. Click the right mouse button on the line defining the boundary of the ROI in the drawing area that you wish to delete. This selects the region and bring up the shortcut menu.
2. Select the **Delete** menu option from the shortcut menu.

To delete an ROI using the ROI Information dialog:

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.

Example 3

XROI's **File** → **Save ROIs** option allows you to save selected regions of interest. This example shows how to restore such a save file. Suppose you have a file named `mineralRoi.sav` that contains regions of interest selected in the `mineral.png` image file. You would need to complete the following steps to restore the file:

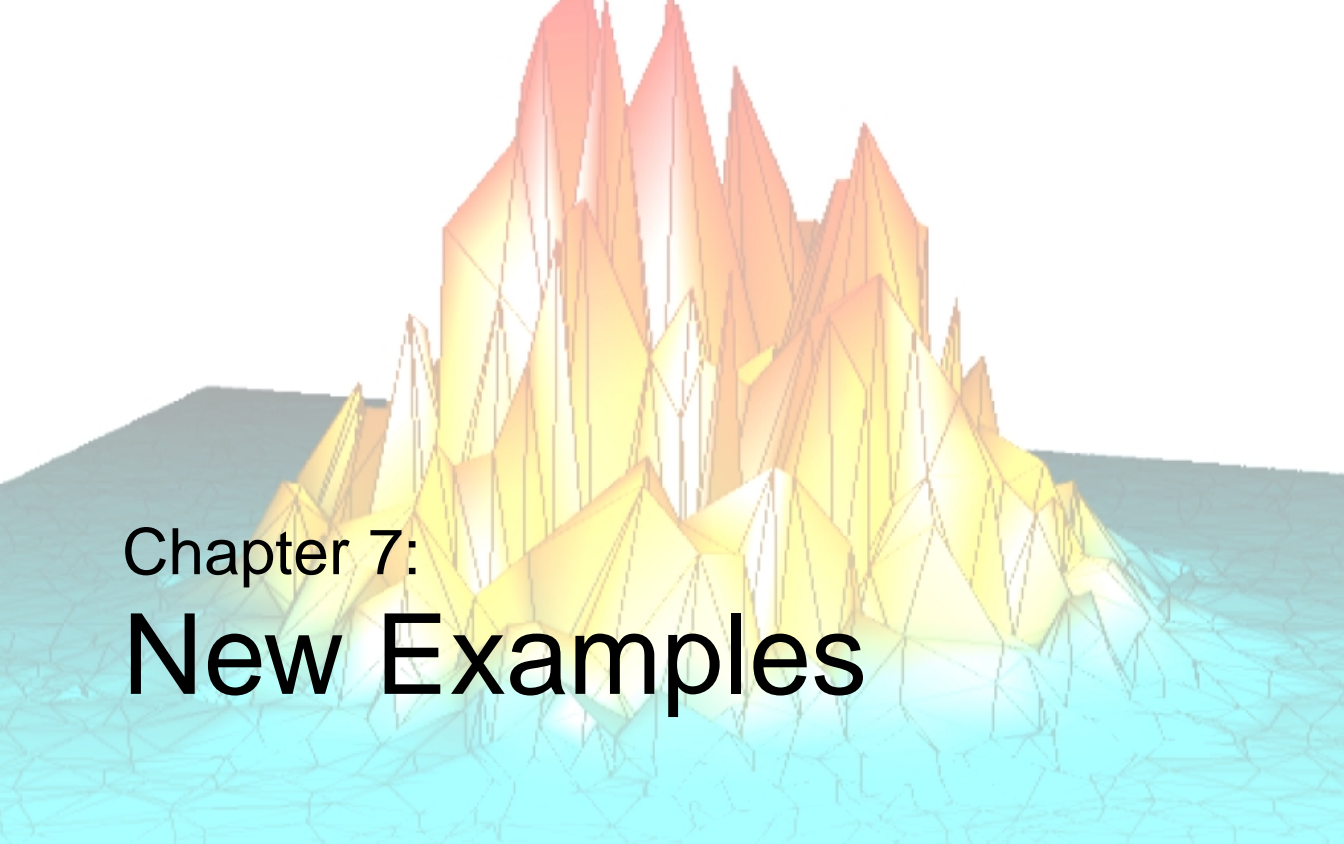
1. First, restore the file, `mineralRoi.sav`. Provide a value for the **RESTORE** procedure's **RESTORED_OBJECTS** keyword. Using the scenario stated above, you could enter the following:

```
RESTORE, 'mineralRoi.sav', RESTORED_OBJECTS = myRoi
```

2. Pass the restored object data containing your regions of interest into XROI by specifying `myRoi` as the value for **REGIONS_IN** as follows:

```
XROI, READ_PNG(FILEPATH('mineral.png', SUBDIRECTORY = $  
    ['examples', 'data'])), REGIONS_IN = myRoi
```

This opens the previously selected regions of interest in the XROI utility.



Chapter 7: New Examples

This chapter includes new documentation of some IDL examples introduced in IDL 5.5.

Overview of New Examples	320	Handling Table Widgets in GUIs	368
Mapping an Image Onto a Surface	322	Finding Straight Lines in Images	374
Centering an Image Object	325	Color Density Contrasting in an Image ..	376
Alpha Blending: Creating a Transparent Image Object	328	Removing Noise from an Image with FFT	379
Working with Mesh Objects and Routines	332	Using Double and Triple Integration	381
Copying and Printing Objects	351	Obtaining Irregular Grid Intervals	385
Capturing IDL Direct Graphics Displays .	359	Calculating Incomplete Beta and Gamma Functions	387
Creating and Restoring .sav Files	363	Determining Bessel Function Accuracy ..	390

Overview of New Examples

This chapter contains new examples highlighting a wide range of functionality in IDL. These examples provide code that can be easily followed and adapted when developing your own routines using the covered functionality.

Tip

You can copy and paste the text of each example in this chapter into the IDL Editor window and save it as a .pro file with the same name as the example routine. You can then compile and run the program to reproduce each example.

Note

If you are running IDL on UNIX, you should use only lowercase characters when naming your .pro files. For example, if you have a routine defined as `PRO MyExample`, (on UNIX) you should save this routine in a file named `myexample.pro`.

The examples are arranged into three broad categories, covering the topics described in the following table.

Category	Example Topics
Object Graphics	“Mapping an Image Onto a Surface” on page 322 describes mapping an image onto elevation data.
	“Centering an Image Object” on page 325 describes the centering of image objects using a viewplane rectangle and coordinate conversions.
	“Alpha Blending: Creating a Transparent Image Object” on page 328 describes how to create and apply an alpha channel.
	“Working with Mesh Objects and Routines” on page 332 includes clipping, decimating, merging, smoothing, and advanced, combination mesh examples.
	“Copying and Printing Objects” on page 351 includes copying and printing plot and image object displays.

Table 7-1: Topics of New Examples

Category	Example Topics
Language and Visualization	“ Capturing IDL Direct Graphics Displays ” on page 359 includes examples of capturing Direct Graphics displays on TrueColor and PseudoColor devices.
	“ Creating and Restoring .sav Files ” on page 363 describes how to create and restore binary .sav files containing variables and routines.
	“ Handling Table Widgets in GUIs ” on page 368 describes how to insert a table widget into a GUI.
Analysis	“ Finding Straight Lines in Images ” on page 374 describes using the HOUGH transform to detect straight lines.
	“ Color Density Contrasting in an Image ” on page 376 uses the RADON transform to find outlines within an image.
	“ Removing Noise from an Image with FFT ” on page 379 describes using FFT to detect and remove image noise.
	“ Using Double and Triple Integration ” on page 381 describes integrating over surfaces and volumes.
	“ Obtaining Irregular Grid Intervals ” on page 385 shows how to obtain irregular intervals from the TRIGRID routine using the XOUT and YOUT keywords.
	“ Calculating Incomplete Beta and Gamma Functions ” on page 387 describes using tolerances and iteration controls when computing the incomplete beta and gamma functions.
	“ Determining Bessel Function Accuracy ” on page 390 includes analyzing Bessel and Modified Bessel functions of the first and second kind.

Table 7-1: Topics of New Examples (Continued)

Mapping an Image Onto a Surface

The following example maps a satellite image from the Los Angeles, California vicinity onto a DEM (Digital Elevation Model) containing the area's topographical features. The realism resulting from mapping the image onto the corresponding elevation data (also known as texture mapping) provides a more informative view of the area's topography. This Object Graphics example creates an image object, containing the satellite image, and a surface object, containing the DEM data. The image is then mapped to the surface using the `IDLgrSurface::SetProperty TEXTURE_MAP` keyword.

Note

To map high resolution images onto geometric surfaces, set the `TEXTURE_HIGHRES` keyword to `IDLgrSurface::Init`. See [“High-Resolution Textures Supported by IDLgrSurface”](#) in Chapter 1 for more information.

```
PRO TextureMap

; State the path to image file.
image_file = FILEPATH('elev_t.jpg', $
    SUBDIRECTORY=['examples', 'data'])

; Import image file.
READ_JPEG, image_file, image

; State the path to DEM data file.
data_file = FILEPATH('elevbin.dat', $
    SUBDIRECTORY=['examples', 'data'])

; Import elevation data.
dem_data = BYTARR(64, 64)
OPENR, unit, data_file, /GET_LUN
READU, unit, dem_data
FREE_LUN, unit
; Increase size of data for visibility.
dem_data = CONGRID(dem_data, 128,128, /INTERP)

; Initialize the model, surface and image objects.
oModel = OBJ_NEW('IDLgrModel')
oSurface = OBJ_NEW('IDLgrSurface', dem_data, STYLE = 2)
oImage = OBJ_NEW('IDLgrImage', image, $
    INTERLEAVE = 0, /INTERPOLATE)

; Calculate normalized conversion factors and
; shift -.5 in every direction to center object
```

```

; in the window.
; Keep in mind that your view default coordinate
; system is [-1,-1], [1, 1]
oSurface -> 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
zs = NORM_COORD(zr)
zs[0] = zs[0] - 0.5
oSurface -> SETPROPERTY, XCOORD_CONV = xs, $
          YCOORD_CONV = ys, ZCOORD = zs

; Apply the image to surface (texture mapping).
oSurface->SetProperty, TEXTURE_MAP = oImage, $
          COLOR = [255, 255, 255]

; Add the surface to the model.
oModel -> Add, oSurface

; Rotate the model for better display of surface
; in the object window.
oModel -> ROTATE, [1, 0, 0], -90
oModel -> ROTATE, [0, 1, 0], 30
oModel -> ROTATE, [1, 0, 0], 30

; Display results in XOBJVIEW utility to provide
; rotation, zoom, and translation control.
XOBJVIEW, oModel, /BLOCK, SCALE = 1

; Cleanup object references.
OBJ_DESTROY, [oImage, oModel]

END

```

The result for this example is shown in the following figure.

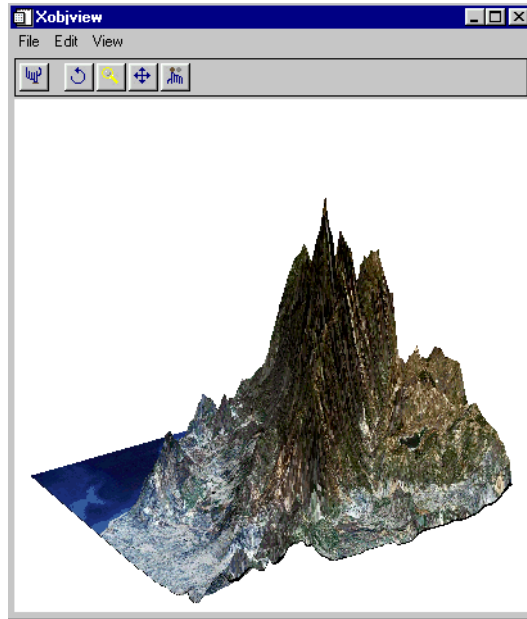


Figure 7-1: Result of Mapping an Image onto a Geometric Surface

Centering an Image Object

In many cases, Object Graphics allow you to choose from different methods to obtain the same solution. An example of this type of variety is shown when you try to center an image object in a display window. While several methods for centering an image object are available, this example shows the two most common methods for centering an image object within a display window.

The first method establishes a viewplane rectangle within a view object. The image object is added to a model object. The model object is then translated to the center of the window object.

The second method does not establish a viewplane rectangle. Instead coordinate conversions are calculated and applied to the image object to center it within the model. This method works within the normalized coordinate system of the model.

This example uses the image from the `worldelv.dat` file found in the `examples/data` directory.

```
PRO CenteringAnImage

; Determine path to file.
worldelvFile = FILEPATH('worldelv.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize image parameters.
worldelvSize = [360, 360]
worldelvImage = BYTARR(worldelvSize[0], worldelvSize[1])

; Open file, read in image, and close file.
OPENR, unit, worldelvFile, /GET_LUN
READU, unit, worldelvImage
FREE_LUN, unit

; Initialize window parameters.
windowMargin = [70, 50]
windowSize = worldelvSize + (2*windowMargin)

; First Method:  Defining the Viewplane and
;                Translating the Model.
;-----

; Initialize objects required for an Object Graphics
; display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'World Elevation:  First Method')
```

```

oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., windowSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize palette with STD GAMMA-II color table and
; use it to initialize the image object.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LOADCT, 5
oImage = OBJ_NEW('IDLgrImage', worldelvImage, $
    PALETTE = oPalette)

; Add image to model, which is added to view. Model
; is translated to center the image within the window.
; Then view is displayed in window.
oModel -> Add, oImage
oView -> Add, oModel
oModel -> Translate, windowMargin[0], windowMargin[1], 0.
oWindow -> Draw, oView

; Clean-up object references.
OBJ_DESTROY, [oView, oPalette]

; Second Method: Using Coordinate Conversions.
;-----

; Initialize objects required for an Object Graphics
; display.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'World Elevation: Second Method')
oView = OBJ_NEW('IDLgrView')
oModel = OBJ_NEW('IDLgrModel')

; Initialize palette with STD GAMMA-II color table and
; use it to initialize the image object.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LOADCT, 5
oImage = OBJ_NEW('IDLgrImage', worldelvImage, $
    PALETTE = oPalette)

; Obtain initial coordinate conversions of image object.
oImage -> GetProperty, XCOORD_CONV = xConv, $
    YCOORD_CONV = yConv, XRANGE = xRange, YRANGE = yRange

; Output initial coordinate conversions.
PRINT, 'Initial xConv: ', xConv
PRINT, 'Initial yConv: ', yConv

; Applying margins to coordinate conversions.

```

```
xTranslation = (2.*FLOAT(windowMargin[0])/windowSize[0]) - 1.
xScale = (-2.*xTranslation)/worldelvSize[0]
xConv = [xTranslation, xScale]
yTranslation = (2.*FLOAT(windowMargin[1])/windowSize[1]) - 1.
yScale = (-2.*yTranslation)/worldelvSize[1]
yConv = [yTranslation, yScale]

; Output resulting coordinate conversions.
PRINT, 'Resulting xConv: ', xConv
PRINT, 'Resulting yConv: ', yConv

; Apply resulting conversions to the image object.
oImage -> SetProperty, XCOORD_CONV = xConv, $
        YCOORD_CONV = yConv

; Add image to model, which is added to view. Display
; the view in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView, oPalette]

END
```

Alpha Blending: Creating a Transparent Image Object

In Object Graphics, image transparency is created by adding an alpha channel to an image array. The alpha channel is used to define the level of transparency in an image object. The following Object Graphics example uses the `IDLgrImage::Init` method to create an image object and employs the `BLEND_FUNCTION` keyword to specify how the transparency of the alpha channel is applied. Other methods of applying a transparent image object include using the `TEXTURE_MAP` keyword in conjunction with either the `IDLgrPolygon::Init` or the `IDLgrSurface::Init` methods.

The following example creates two image objects of MRI slices of a human head. After adding an alpha channel to the second image object, it is layered over the first image object as a transparency.

```
PRO AlphaBlend

; Determine path to file.
headFile = FILEPATH('head.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize volume array and size parameter.
headSize = [80, 100, 57]
head = BYTARR(headSize[0], headSize[1], headSize[2])
imageSize = [240, 300]

; Open file, read in volume, and close file.
OPENR, unit, headFile, /GET_LUN
READU, unit, head
FREE_LUN, unit

; Initialize window and view objects to vertically
; display two images.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = [imageSize[0], 2*imageSize[1]], $
    TITLE='MRI Slices')
oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., imageSize[0], 2*imageSize[1]])

; Initialize a model object for each image.
oModel = [OBJ_NEW('IDLgrModel'), OBJ_NEW('IDLgrModel')]

; Extract the first slice of data.
layer1 = CONGRID(head[*,*,30], imageSize[0],imageSize[1],$
    /INTERP)
```

```

; Initialize the first image layer.
oLayer1 = OBJ_NEW('IDLgrImage', layer1)

; Extract the second slice of data.
layer2 = CONGRID(head[*,*,43], imageSize[0],imageSize[1],$
    /INTERP)

; Initialize second image layer with a palette.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 12
oLayer2 = OBJ_NEW('IDLgrImage', layer2, PALETTE = oPalette)

; Add the layers to the model.
oModel[0] -> Add, oLayer1
oModel[1] -> Add, oLayer2

; Translate the first layer to the top of the
; display. Initially, the lower left corner of both
; models are at the lower left corner of the display.
; The model of the first layer must be moved above the
; second layer model to allow both to be displayed.
oModel[0] -> Translate, 0., imageSize[1], 0.

; Add the model to the view, and then display the view
; in the window.
oView -> Add, oModel
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView]

; Get the red, green and blue values of the palette.
oPalette -> GetProperty, RED_VALUES = red, $
    GREEN_VALUES = green, BLUE_VALUES = blue

; Create a four channel array for alpha blending.
alpha = BYTARR(4, imageSize[0], imageSize[1])

; Add the palette values to the first three channels.
alpha[0,*,*]= red[layer2]
alpha[1,*,*]= green[layer2]
alpha[2,*,*]= blue[layer2]

; Create a mask to remove lower pixels values from array.
mask = layer2 GT 25

; Apply the mask to the alpha (fourth) channel of the
; array. Set transparency to 80. Range is 0 (completely

```

```

; transparent)to 255 (completely opaque).
alpha[3,*,*] = mask * 80

; Initialize the alpha image object, setting blend function.
oAlpha = OBJ_NEW('IDLgrImage', alpha, $
    DIMENSIONS = imageSize, BLEND_FUNCTION = [3,4])

; Initialize the window, model and view.
oWindow = OBJ_NEW('IDLgrWindow', DIMENSIONS = imageSize, $
    LOCATION = [300,0], RETAIN = 2,$
    TITLE = 'Alpha Blending Example')
oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0,0,imageSize[0], imageSize[1]])
oModel = OBJ_NEW('IDLgrModel')

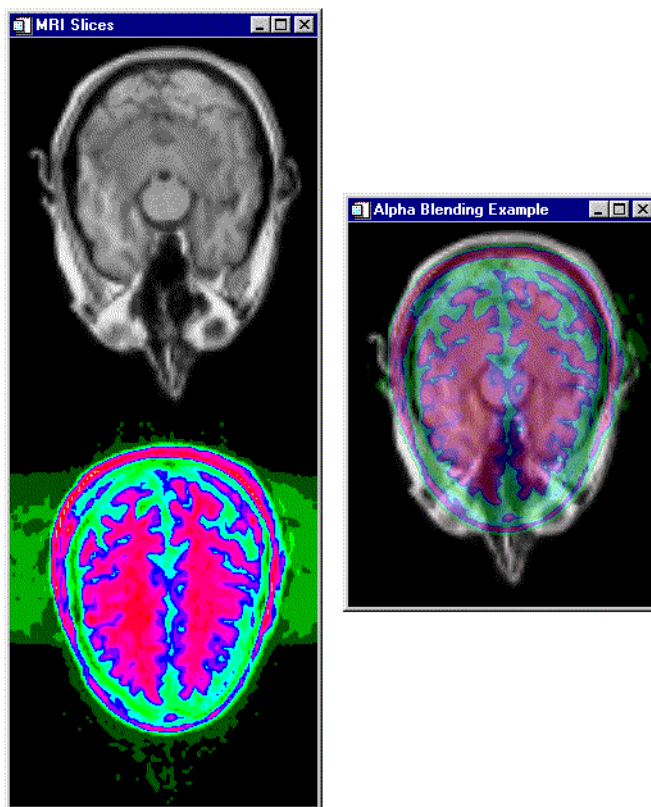
; Initialize a new image object for layer1.
oBase = OBJ_NEW('IDLgrImage', layer1)
; Add the transparent image objects AFTER adding other
; image objects to the model.
oModel -> Add, oBase
oModel -> Add, oAlpha
oView -> Add, oModel
; Display the transparent image object.
oWindow -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oView, oPalette]

END

```

The results for this example are shown in the following figure.



*Figure 7-2: Original Image Objects (left) and
Resulting Alpha Blended Image (right)*

Working with Mesh Objects and Routines

In IDL, meshes are made up of a list of vertex locations and a description of vertex connectivity. The vertex locations are usually represented by an array containing two or three columns (one column for the x values, one for the y values, and optionally one for the z values). The array of vertex locations is known as the *vertices*. The vertex relationships are represented in the *connectivity list*, which is a vector (a one-dimensional array). The connectivity list contains the information for each individual shape within the mesh. This list contains the number of vertices of each shape in the mesh followed by the index of the vertices within that shape. For example, if vertices number 0, 1, 2, and 3 make up the first shape, which is a rectangle, and vertices number 1, 2, and 4 make up the second shape, which is a triangle, then the connectivity is [4, 0, 1, 2, 3, 3, 1, 4, 2].

IDL contains many mesh-related routines. This section provides examples for clipping, decimating, merging, and smoothing meshes. This section also includes an advanced example using some of these routines together to produce an overall display. These examples use IDL polygon objects to display the meshes. The polygon object is designed for meshes. It contains a vertices input argument and a POLYGONS keyword for connectivity lists.

This section includes examples of the following:

- [“Clipping a Mesh”](#) on page 333
- [“Decimating a Mesh”](#) on page 336
- [“Merging Meshes”](#) on page 339
- [“Smoothing a Mesh”](#) on page 342
- [“Advanced Meshing: Combining Meshing Routines”](#) on page 345

Clipping a Mesh

This example clips a mesh of an octahedron (an eight-sided, three-dimensional shape similar to a cut diamond). A mesh is clipped when an imaginary plane intersects the mesh. The clipped mesh is either of the remaining sides of the original mesh after the imaginary (clipping) plane intersects.

The original octahedron mesh in this example contains one rectangle and eight triangles. The connectivity list is formed with the rectangle listed first followed by the triangles. The mesh is placed in a polygon object, which is added to a model. The model is displayed in the XOBJVIEW utility. The XOBJVIEW utility allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

When you quit out of the first XOBJVIEW display, the second XOBJVIEW display will appear. This display shows the mesh clipped with an oblique plane. The final XOBJVIEW display shows the results of using the TRIANGULATE routine to cover the clipped area. See [TRIANGULATE](#) in the *IDL Reference Guide* for more information in this routine.

```
PRO ClippingAMesh

; Create a mesh of an octahedron.
vertices = [[0, -1, 0], [1, 0, 0], [0, 1, 0], $
           [-1, 0, 0], [0, 0, 1], [0, 0, -1]]
connectivity = [4, 0, 1, 2, 3, 3, 0, 1, 4, 3, 1, 2, 4, $
               3, 2, 3, 4, 3, 3, 0, 4, 3, 1, 0, 5, 3, 2, 1, 5, $
               3, 3, 2, 5, 3, 0, 3, 5]

; Initialize model for display.
oModel = OBJ_NEW('IDLgrModel')

; Initialize polygon and polyline outline to contain
; the mesh of the octahedron.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
                  POLYGONS = connectivity, SHADING = 1, $
                  COLOR = [0, 255, 0])
oPolyline = OBJ_NEW('IDLgrPolyline', vertices, $
                   POLYLINES = connectivity, COLOR = [0, 0, 0])

; Add the polygon and the polyline to the model.
oModel -> Add, oPolygon
oModel -> Add, oPolyline

; Rotate model for better initial perspective.
oModel -> Rotate, [-1, 0, 1], 22.5
```

```

; Display model.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Original Octahedron Mesh'

; Clip mesh.
clip = MESH_CLIP([1., 1., 1., 0.], vertices, connectivity, $
    clippedVertices, clippedConnectivity, $
    CUT_VERTS = cutVerticesIndex)

; Update polygon with the resulting clipped mesh.
oPolygon -> SetProperty, DATA = clippedVertices, $
    POLYGONS = clippedConnectivity

; Display the updated model.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Clipped Octahedron Mesh'

; Determine the vertices of the clipped plane.
cutVertices = clippedVertices[*, cutVerticesIndex]

; Derive the x and y components of the clipped plane's
; vertices.
x = cutVertices[0, *]
y = cutVertices[1, *]

; Triangulate the connectivity of the clipped plane.
TRIANGULATE, x, y, triangles

; Derive the connectivity of the clipped plane from the
; results of the triangulation.
arraySize = SIZE(triangles, /DIMENSIONS)
array = FLTARR(4, arraySize[1])
array[0, *] = 3
array[1, 0] = triangles
cutConnectivity = REFORM(array, N_ELEMENTS(array))

; Initialize the clipped plane's polygon and polyline.
oCutPolygon = OBJ_NEW('IDLgrPolygon', cutVertices, $
    POLYGONS = cutConnectivity, SHADING = 1, $
    COLOR = [0, 0, 255])
oCutPolyline = OBJ_NEW('IDLgrPolyline', cutVertices, $
    POLYLINES = cutConnectivity, COLOR = [255, 0, 0], $
    THICK = 3.)

; Add polyline and polygon to model.
oModel -> Add, oCutPolyline
oModel -> Add, oCutPolygon

; Display updated model.

```

```

XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Clipped Octahedron Mesh with Clipping Plane'

; Clean-up object references.
OBJ_DESTROY, [oModel]

END

```

The results for this example are shown in the following figure.

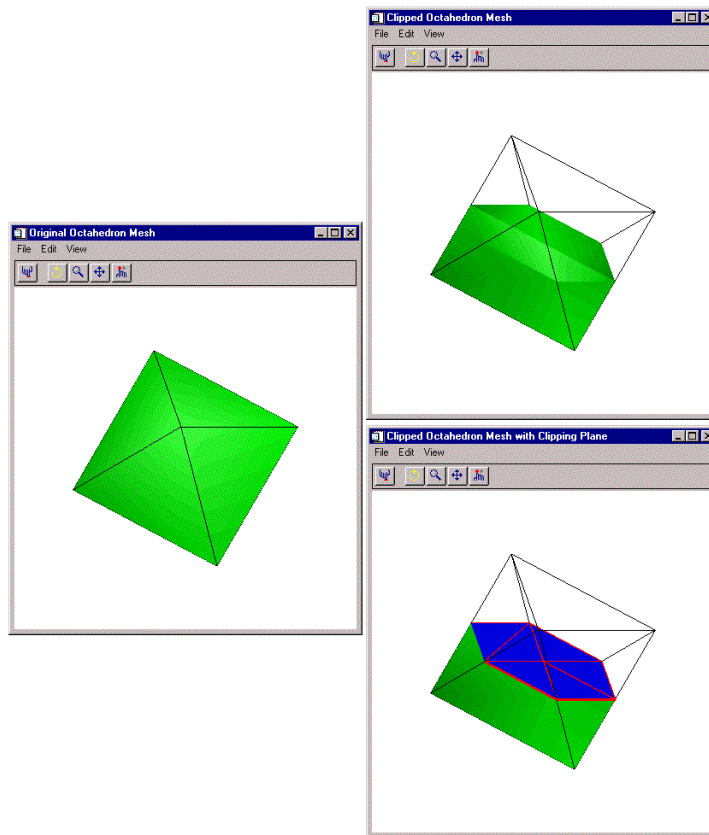


Figure 7-3: The Original Octahedron (left) and the Two Clipped Results (right)

Decimating a Mesh

This example decimates a DEM (digital elevation model) mesh. Decimation reduces either the number of vertices or the number of connections within a mesh while trying to maintain the overall shape of the mesh. Very large meshes usually contain redundant or useless information, which may slow down any interactive displays of the mesh. Decimation helps to reduce the size of these large meshes.

The DEM in this example comes from the `elevbin.dat` file found in the `examples/data` directory. The DEM is converted into a mesh with the `MESH_OBJ` procedure. The results of this routine are placed in a polygon object, which is added to a model. The models are displayed in the `XOBJVIEW` utility. The `XOBJVIEW` utility allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

The first display contains a wire outline of the DEM as a mesh. When you quit out of the first `XOBJVIEW` display, the second `XOBJVIEW` display will appear showing a filled mesh. The colors correspond to the change in elevation. The third display is the result of decimating the mesh down to 20 percent of the original number of vertices. The final display is the resulting mesh filled with the elevation colors.

```
PRO DecimatingAMesh

; Determine path to data file.
elevbinFile = FILEPATH('elevbin.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize data parameters.
elevbinSize = [64, 64]
elevbinData = BYTARR(elevbinSize[0], elevbinSize[1])

; Open file, read in data, and close file.
OPENR, unit, elevbinFile, /GET_LUN
READU, unit, elevbinData
FREE_LUN, unit

; Convert data into a mesh, which is defined by
; vertice locations and their connectivity.
MESH_OBJ, 1, vertices, connectivity, elevbinData

; Initialize a model for display.
oModel = OBJ_NEW('IDLgrModel')

; Form a polygon from the mesh.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, SHADING = 1.5, $
    COLOR = [0, 255, 0], STYLE = 1)
```

```

; Add polygon to model.
oModel -> Add, oPolygon

; Rotate model for better initial perspective.
oModel -> Scale, 1, 1, 0.25
oModel -> Rotate, [-1, 0, 1], 45.

; Display model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Original Mesh from Elevation Data'

; Derive a color table for the filled polygon.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LOADCT, 29

; Fill in the polygon mesh with the colors of the table
; (the colors correspond to the z-values of the polygon).
oPolygon -> SetProperty, STYLE = 2, $
    VERT_COLORS = BYTSCL(vertices[2, *]), $
    PALETTE = oPalette

; Display model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Filled Original Mesh'

; Decimate the mesh down to 20 percent of the original
; number of vertices.
numberVertices = MESH_DECIMATE(vertices, connectivity, $
    decimatedConnectivity, VERTICES = decimatedVertices, $
    PERCENT_VERTICES = 20)

; Update the polygon with the resulting decimated mesh.
oPolygon -> SetProperty, DATA = decimatedVertices, $
    POLYGONS = decimatedConnectivity, STYLE = 1, $
    VERT_COLORS = 0, COLOR = [0, 255, 0]

; Display updated model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
    TITLE = 'Decimation Results (by 80%)'

; Fill in the updated polygon mesh with the colors of
; the table (the colors correspond to the z-values of
; the updated polygon).
oPolygon -> SetProperty, STYLE = 2, $
    VERT_COLORS = BYTSCL(decimatedVertices[2, *]), $
    PALETTE = oPalette

; Display model in the XOBJVIEW utility.

```

```

XOBJVIEW, oModel, /BLOCK, SCALE = 1., $
  TITLE = 'Filled Decimation Results'

; Cleanup all the objects by destroying the model.
OBJ_DESTROY, [oModel, oPalette]

END

```

The results for this example are shown in the following figure.

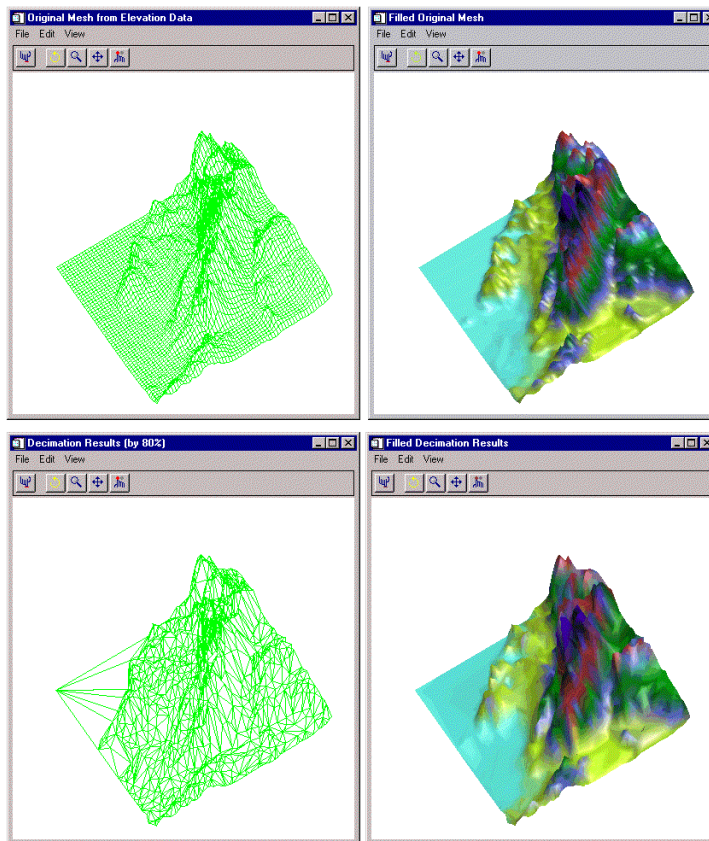


Figure 7-4: Before Decimating (top row) and After Decimating (bottom row)

Merging Meshes

This example merges two simple meshes: a single square and a single right triangle. The right side of the square is in the same location as the left side of the triangle. Each mesh is originally its own polygon object. These objects are then added to a model object. The model is displayed in the XOBJVIEW utility. The XOBJVIEW utility allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

When you quit out of the first XOBJVIEW display, the second XOBJVIEW display will appear. The meshes are merged into a single polygon object. After you quit out of the second display, the final display shows the results of decimating the merged mesh to obtain the least number connections for these vertices. Decimation can often be used to refine the results of merging.

```
PRO MergingMeshes

; Create a mesh of a single square (4 vertices
; connected counter-clockwise from the lower left
; corner of the mesh.
vertices = [[-2., -1., 0.], [0., -1., 0.], $
            [0., 1., 0.], [-2., 1., 0.]]
connectivity = [4, 0, 1, 2, 3]

; Create a separate mesh of a single triangle (3
; vertices connected counter-clockwise from the lower
; left corner of the mesh.
triangleVertices = [[0., -1., 0.], [2., -1., 0.], $
                   [0., 1., 0.]]
triangleConnectivity = [3, 0, 1, 2]

; Initialize model for display.
oModel = OBJ_NEW('IDLgrModel')

; Initialize polygon for the square mesh.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
                  POLYGONS = connectivity, COLOR = [0, 128, 0], $
                  STYLE = 1)

; Initialize polygon for the triangle mesh.
oTrianglePolygon = OBJ_NEW('IDLgrPolygon', $
                          triangleVertices, POLYGONS = triangleConnectivity, $
                          COLOR = [0, 0, 255], STYLE = 1)

; Add both polygons to the model.
oModel -> Add, oPolygon
oModel -> Add, oTrianglePolygon
```

```

; Display the model in the XOBJVIEW utility.
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Two Separate Meshes'

; Merge the square and triangle into a single mesh.
numberTriangles = MESH_MERGE(vertices, $
    connectivity, triangleVertices, $
    triangleConnectivity, /COMBINE_VERTICES)

; Output number of resulting vertices and triangles.
numberVertices = SIZE(vertices, /DIMENSIONS)
PRINT, 'numberVertices = ', numberVertices[1]
PRINT, 'numberTriangles = ', numberTriangles

; Cleanup triangle polygon object, which is no longer
; needed.
OBJ_DESTROY, [oTrianglePolygon]

; Update remaining polygon object with the results from
; merging the two meshes together.
oPolygon -> SetProperty, DATA = vertices, $
    POLYGONS = connectivity, COLOR = [0, 128, 128]

; Display results.
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Result of Merging the Meshes into One'

; Decimate polygon to 75 percent of the original
; number of vertices.
numberTriangles = MESH_DECIMATE(vertices, connectivity, $
    decimatedConnectivity, PERCENT_POLYGONS = 75)

; Output number of resulting triangles.
PRINT, 'After Decimation:  numberTriangles = ', numberTriangles

; Update polygon with results from decimating.
oPolygon -> SetProperty, DATA = vertices, $
    POLYGONS = decimatedConnectivity, COLOR = [0, 0, 0]

; Display decimation results.
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Decimation of Mesh'

; Cleanup object references.
OBJ_DESTROY, [oModel]

END

```

The results for this example are shown in the following figure.

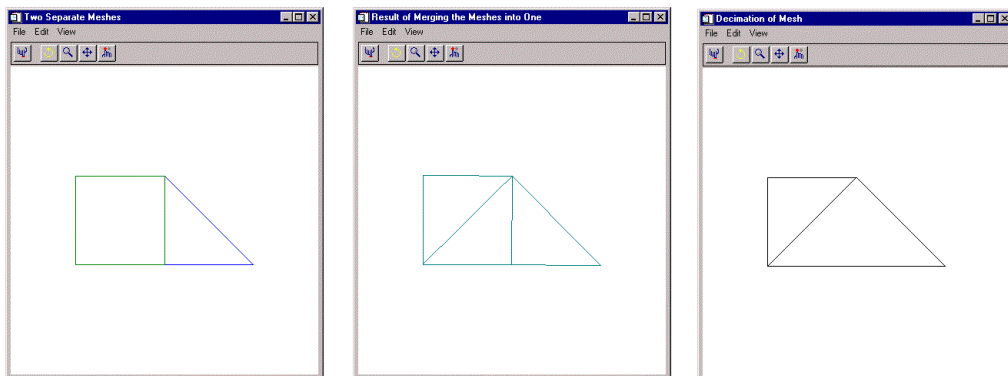


Figure 7-5: Original (left), Merged (center), and Decimated Meshes (right)

Smoothing a Mesh

This example smooths a rectangular mesh containing a spike. First, a rectangle mesh is created. This mesh is made up 10 columns and 5 rows of vertices. The vertices are connected with right triangles. The mesh is placed in a polygon object, which is added to a model object. The model is displayed in the XOBJVIEW utility. The XOBJVIEW utility allows you to click-and-drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

When you quit out of the first XOBJVIEW display, the second XOBJVIEW display will appear. The center vertex of the top row is displaced in the y-direction. This displacement causes the center of the top to spike out away from the mesh. After you quit out of the second display, the third display shows the result of smoothing the entire mesh. The final display shows the results of smoothing the spike with all the other vertices fixed.

```
PRO SmoothingMeshes

; Initialize mesh size parameters.
nX = 10
nY = 5

; Initialize the x coordinates of the mesh's vertices.
xVertices = FINDGEN(nX) # REPLICATE(1., nY)
PRINT, 'xVertices: '
PRINT, xVertices, FORMAT = '(10F6.1)'

; Initialize the y coordinates of the mesh's vertices.
yVertices = REPLICATE(1., nX) # FINDGEN(nY)
PRINT, 'yVertices: '
PRINT, yVertices, FORMAT = '(10F6.1)'

; Derive the overall vertices of the mesh.
vertices = FLTARR(3, (nX*nY))
vertices[0, *] = xVertices
vertices[1, *] = yVertices
PRINT, 'vertices: '
PRINT, vertices, FORMAT = '(3F6.1)'

; Triangulate the mesh to establish connectivity.
TRIANGULATE, xVertices, yVertices, triangles
trianglesSize = SIZE(triangles, /DIMENSIONS)
polygons = LONARR(4, trianglesSize[1])
polygons[0, *] = 3
polygons[1, 0] = triangles
PRINT, 'polygons: '
```

```

PRINT, polygons, FORMAT = '(4I6)'

; Derive connectivity from the resulting triangulation.
connectivity = REFORM(polygons, N_ELEMENTS(polygons))

; Initialize a model for the display.
oModel = OBJ_NEW('IDLgrModel')

; Initialize a polygon object to contain the mesh.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, COLOR = [0, 128, 0], $
    STYLE = 1)

; Add the polygon to the model.
oModel -> Add, oPolygon

; Display the model.
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Original Mesh'

; Introduce an irregular vertex by drastically changing
; a single y coordinate.
vertices[1, 45] = 10.

; Update polygon with new vertices.
oPolygon -> SetProperty, DATA = vertices

; Display change.
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Mesh with New Irregular Vertex'

; Smooth entire mesh to reduce the effect of the
; irregular vertex.
smoothedVertices = MESH_SMOOTH(vertices, connectivity)

; Update polygon and display results.
oPolygon -> SetProperty, DATA = smoothedVertices
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Smoothing with No Fixed Vertices'

; Determine which vertices should be fixed. Basically,
; all of the vertices should be fixed except for the
; irregular vertex.
fixed = LINDGEN((nX*nY) - 1)
fixed[45] = fixed[45:*] + 1

; Smooth mesh with resulting fixed vertices.
smoothedVertices = MESH_SMOOTH(vertices, connectivity, $
    FIXED_VERTICES = fixed)

```

```

; Update polygon and display results.
oPolygon -> SetProperty, DATA = smoothedVertices
XOBJVIEW, oModel, /BLOCK, $
    TITLE = 'Smoothing with Almost All Vertices Fixed'

; Cleanup object references.
OBJ_DESTROY, [oModel]

END

```

The results for this example are shown in the following figure.

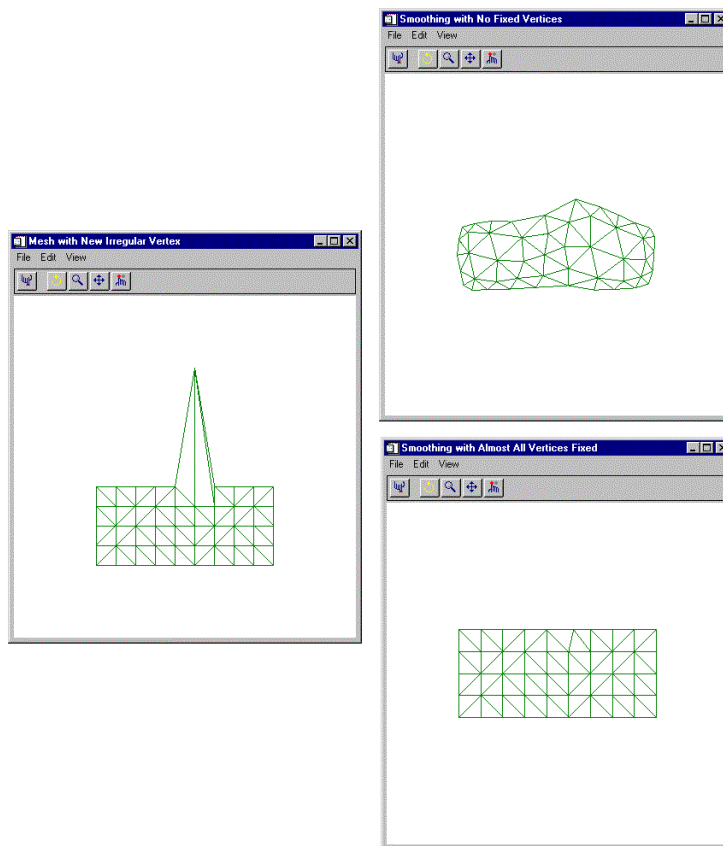


Figure 7-6: The Spiked Mesh (left) and the Two Smoothed Meshes (right)

Advanced Meshing: Combining Meshing Routines

This example uses world elevation image data (found in the `worldelv.dat` file in the `examples/data` directory) to create a spherical mesh of the earth. The `MESH_OBJ` routine is used to convert the world elevation image to a spherical mesh. The elevation is exaggerated so it can be seen on the mesh. This mesh is placed in a polygon object, which is added to a model object. The model is displayed in the `XOBJVIEW` utility. The `XOBJVIEW` utility allows you to click and drag the polygon object to rotate and translate it. See [XOBJVIEW](#) in the *IDL Reference Guide* for more information on this utility.

When you quit out of the first `XOBJVIEW` display, the second `XOBJVIEW` display will appear. This display contains the world polygon clipped at the equator. The data from the clipping process is used to define a plane polygon object. Earth mantle convection data (found in the `convec.dat` file in the `examples/data` directory) is placed on the planar polygon after making the background transparent. The convection data was measured along 0 degrees longitude so it is placed vertically at that longitude. And finally, in the third `XOBJVIEW` display, the lower hemisphere is decimated to allow quicker rotations within the `XOBJVIEW` utility.

```
PRO WorldelvMesh

; Determine path to image file.
worldelvFile = FILEPATH('worldelv.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize image parameters.
worldelvSize = [360, 360]
worldelvImage = BYTARR(worldelvSize[0], worldelvSize[1])

; Open file, read in image, and close file.
OPENR, unit, worldelvFile, /GET_LUN
READU, unit, worldelvImage
FREE_LUN, unit

; Resize image to obtain data for a 1 degree interval in
; both directions.
worldelvImage = CONGRID(worldelvImage, 360, 180, /INTERP)

; Initialize display objects.
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = [worldelvSize[0], worldelvSize[1]/2], $
    TITLE = 'Original Elevation Image')
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT = [0., 0., $
    worldelvSize[0], worldelvSize[1]/2])
oModel = OBJ_NEW('IDLgrModel')
```

```

; Initialize and set palette to the STD GAMMA-II color
; table.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LoadCT, 5

; Initialize image object.
oImage = OBJ_NEW('IDLgrImage', worldelvImage, $
    PALETTE = oPalette)

; Add the image to the model, which is added to the
; view, and then the view is displayed in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Clean-up unused object references.
OBJ_DESTROY, [oView]

; Scale image values to the earth radius. Multiple
; scaling by 50 to exaggerate elevation.
worldelvImage = 50.*1.77*(worldelvImage/255.)

; Add the earth's radius to the image. The image only
; contains elevation information from the deepest parts
; of the oceans. The earth's radius is added to obtain
; a sphere with small changes in elevation on its
; surface.
radii = worldelvImage + REPLICATE(1275.6, 360, 180)

; Derive a mesh from the exaggerated image data and the
; radius of the earth.
MESH_OBJ, 4, vertices, connectivity, radii, /CLOSED

; Initialize a model to display.
oModel = OBJ_NEW('IDLgrModel')

; Determine the radius of each vertex to provide color
; at each vertex.
sphericalCoordinates = CV_COORD(FROM_RECT = vertices, $
    /TO_SPHERE)
elevation = REFORM(sphericalCoordinates[2, *], $
    N_ELEMENTS(sphericalCoordinates[2, *]))

; Initialize polygon to contain mesh.
oPolygon = OBJ_NEW('IDLgrPolygon', vertices, $
    POLYGONS = connectivity, SHADING = 1, $
    VERT_COLORS = BYTSCL(elevation), $
    PALETTE = oPalette)

```

```

; Add polygon to model.
oModel -> Add, oPolygon

; Rotate model to place view at 0 degrees latitude.
oModel -> Rotate, [1., 0., 0.], -90.

; Display model.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Exaggerated Earth Elevation'

; Clip earth polygon along the equator.
planeCoefficients = [0., 0., 1., 0.]
numberVertices = MESH_CLIP(planeCoefficients, $
    vertices, connectivity, $
    clippedVertices, clippedConnectivity, $
    CUT_VERTS = cutVerticesIndex)

; Determine the radius of each vertex to provide color
; at each vertex.
sphericalCoordinates = CV_COORD($
    FROM_RECT = clippedVertices, /TO_SPHERE)
elevation = REFORM(sphericalCoordinates[2, *], $
    N_ELEMENTS(sphericalCoordinates[2, *]))

; Update polygon with results from clipping.
oPolygon -> SetProperty, DATA = clippedVertices, $
    POLYGONS = clippedConnectivity, $
    VERT_COLORS = BYTSCL(elevation)

; Display updated model.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
    TITLE = 'Earth Clipped at the Equator'

; Determine clipped plane's vertices.
cutVertices = clippedVertices[*, cutVerticesIndex]
x = cutVertices[0, *]
y = cutVertices[1, *]
z = cutVertices[2, *]

; Compute the center vertex of the clipped plane.
centerX = TOTAL(x)/N_ELEMENTS(x)
centerY = TOTAL(y)/N_ELEMENTS(y)
centerZ = TOTAL(z)/N_ELEMENTS(z)

; Determine the inner radius of the earth polygon.
sphericalCoordinates = CV_COORD(FROM_RECT = cutVertices, $
    /TO_SPHERE)
elevation = REFORM(sphericalCoordinates[2, *], $

```

```

        N_ELEMENTS(sphericalCoordinates[2, *]))
innerRadius = MIN(elevation)

; Derive the corner vertices of the clipping plane.
planeVertices = $
    [[centerX - innerRadius, 0, centerZ - innerRadius], $
     [centerX + innerRadius, 0, centerZ - innerRadius], $
     [centerX + innerRadius, 0, centerZ + innerRadius], $
     [centerX - innerRadius, 0, centerZ + innerRadius]]
planeConnectivity = [4, 0, 1, 2, 3]

; Determine the path to the earth's mantle convection
; data file.
convecFile = FILEPATH('convec.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize convection image and parameters.
convecSize = [248, 248]
convecImage = BYTARR(convecSize[0], convecSize[1])
convecData = BYTARR(convecSize[0], convecSize[1], 4)

; Open file, read in image, and close file.
OPENR, unit, convecFile, /GET_LUN
READU, unit, convecImage
FREE_LUN, unit

; Create mask of image. Mask out the background (zero
; values of the image, and apply mask to the alpha
; channel.
mask = BYTSCL(convecImage GT 0)
convecData[*, *, 3] = mask

; Convert indexed image to RGB image.
DEVICE, DECOMPOSED = 0
LOADCT, 27
TVLCT, red, green, blue, /GET
convecData[*, *, 0] = red[convecImage]
convecData[*, *, 1] = green[convecImage]
convecData[*, *, 2] = blue[convecImage]

; Initialize an image object of the resulting RGB image
; to be used as a texture map placed on the clipping
; plane.
oPlaneImage = OBJ_NEW('IDLgrImage', convecData, $
    INTERLEAVE = 2, BLEND_FUNCTION = [3, 4])

; Initialize polygon of clipping plane, which contains
; the texture map of the image.
oPlanePolygon = OBJ_NEW('IDLgrPolygon', $

```

```

planeVertices, POLYGONS = planeConnectivity, $
SHADING = 0, COLOR = [255, 255, 255], $
TEXTURE_MAP = oPlaneImage, $
TEXTURE_COORD = [[0, 0], [1, 0], [1, 1], [0, 1]]

; Add the clipping plane's polygon to the model.
oModel -> Add, oPlanePolygon

; Display results.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
TITLE = 'Earth Elevation and Mantle Convection'

; Decimate clipped earth polygon.
numberTriangles = MESH_DECIMATE(clippedVertices, $
    clippedConnectivity, decimatedConnectivity, $
    VERTICES = decimatedVertices, PERCENT_VERTICES = 10)

; Determine the radius of each vertex to provide color
; at each vertex.
sphericalCoordinates = CV_COORD($
    FROM_RECT = decimatedVertices, /TO_SPHERE)
elevation = REFORM(sphericalCoordinates[2, *], $
    N_ELEMENTS(sphericalCoordinates[2, *]))

; Update polygon with results from decimating.
oPolygon -> SetProperty, DATA = decimatedVertices, $
    POLYGONS = decimatedConnectivity, $
    VERT_COLORS = BYTSCL(elevation)

; Display decimation results.
XOBJVIEW, oModel, /BLOCK, SCALE = 1, $
TITLE = 'Decimated Earth and Mantle Convection'

; Cleanup the object references.
OBJ_DESTROY, [oModel, oPalette, oPlaneImage]

END

```

The results for this example are shown in the following figure.

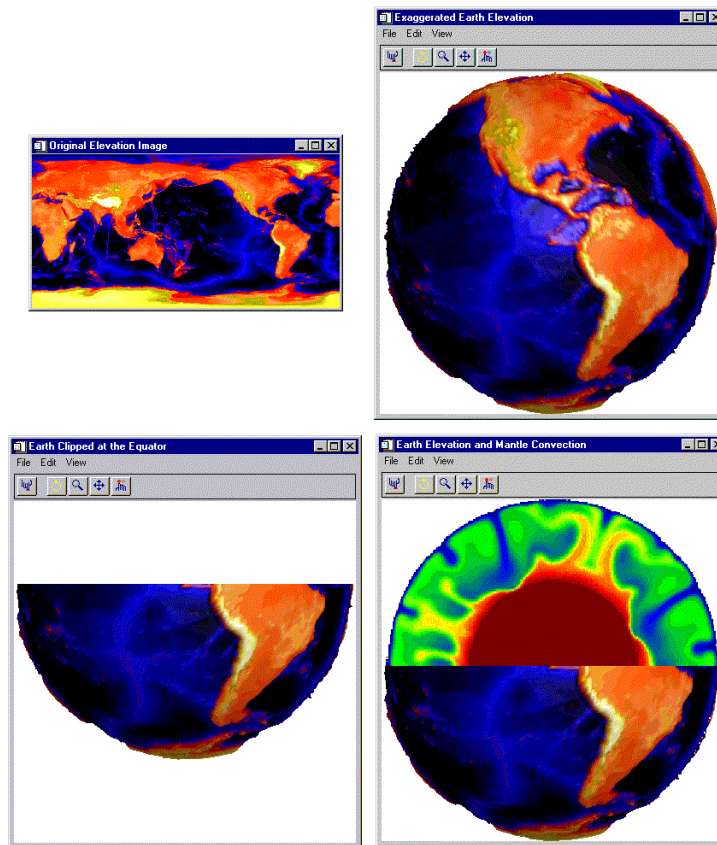


Figure 7-7: Original Image and Resulting Mesh (top row) and Clipped Mesh and Added Mantle Plane (bottom row)

Copying and Printing Objects

IDL's Object Graphics system contains five destination objects: window, buffer, VRML, clipboard, and printer. The window object is used to display to the screen. The clipboard object is used to display to the operating system's clipboard. The printer object is used to display to the system's printer. The window object is simple to use, but the use of the clipboard and printer objects depend on the type of objects to be displayed. This section covers the following topics:

- [“Copying a Plot Display to the Clipboard”](#) in the following section
- [“Printing a Plot Display”](#) on page 353
- [“Copying an Image Display to the Clipboard”](#) on page 355
- [“Printing an Image Display”](#) on page 357

Copying a Plot Display to the Clipboard

This example displays a damped sine wave plot in a window object and a clipboard object. The damped sine wave data comes from the `damp_sn2.dat` file found in the `examples/data` directory. The resolution of the clipboard is based on the resolution of the screen. The plot is displayed from the system's clipboard to a platform-related graphics file (PostScript file on UNIX, Enhanced Metafile on Windows, or a PICT file on Macintosh) and an encapsulated PostScript file on all the platforms.

```
PRO SendingPlotToClipboard

; Determine the path to the "damp_sn2.dat" file.
signalFile = FILEPATH('damp_sn2.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize the parameters of the data within the file.
signalSize = 512
signal = BYTARR(signalSize)

; Open the file, read in data, and then close the file.
OPENR, unit, signalFile, /GET_LUN
READU, unit, signal
FREE_LUN, unit

; Determine viewplane size and margins.
offsetScale = 150.
viewOffset = offsetScale*[-1., -1., 1., 1.]
signalRange = MAX(signal) - MIN(signal)

; Initialize the display objects.
```

```

windowSize = [512, 384]
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'Damped Sine Wave with Noise')
oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., signalSize, signalRange] + $
    viewOffset)
oModel = OBJ_NEW('IDLgrModel')

; Initialize the plot object.
oPlot = OBJ_NEW('IDLgrPlot', signal, COLOR = [0, 0, 255])

; Obtain plot ranges.
oPlot -> GetProperty, XRange = xPlotRange, $
    YRange = yPlotRange

; Initialize axes objects, which are based on the plot
; ranges.
oXTitle = OBJ_NEW('IDLgrText', 'Time (seconds)')
oXAxis = OBJ_NEW('IDLgrAxis', 0, RANGE = xPlotRange, $
    LOCATION = [xPlotRange[0], yPlotRange[0]], /EXACT, $
    TITLE = oXTitle, TICKDIR = 0, $
    TICKLEN = (0.02*(yPlotRange[1] - yPlotRange[0])))
oYTitle = OBJ_NEW('IDLgrText', 'Amplitude (centimeters)')
oYAxis = OBJ_NEW('IDLgrAxis', 1, RANGE = yPlotRange, $
    LOCATION = [xPlotRange[0], yPlotRange[0]], /EXACT, $
    TITLE = oYTitle, TICKDIR = 0, $
    TICKLEN = (0.02*(xPlotRange[1] - xPlotRange[0])))

; Add plot and axes to model, which is added to the
; view, and then displayed in the window.
oModel -> Add, oPlot
oModel -> Add, oXAxis
oModel -> Add, oYAxis
oView -> Add, oModel
oModel -> Translate, -50., -50., 0.
oWindow -> Draw, oView

; Determine the centimeter to pixel resolution of the
; plot on the screen.
oWindow -> GetProperty, RESOLUTION = screenResolution

; Initialize clipboard destination object.
oClipboard = OBJ_NEW('IDLgrClipboard', QUALITY = 2, $
    DIMENSIONS = windowSize, $
    RESOLUTION = screenResolution)

; Determine the type of export file, which depends on
; the screen device.

```

```

screenDevice = !D.NAME
CASE screenDevice OF
    'X': fileExtension = '.ps'
    'WIN': fileExtension = '.emf'
    'MAC': fileExtension = '.pict'
ELSE: RETURN
ENDCASE
clipboardFile = 'damp_sn2' + fileExtension

; Display the view within the clipboard destination,
; which exports to an PS, EMF, or PICT file.
oClipboard -> Draw, oView, FILENAME = clipboardFile, $
/VECTOR
oClipboard -> Draw, oView, FILENAME = 'damp_sn2.eps', $
/POSTSCRIPT, /VECTOR

; Cleanup object references.
OBJ_DESTROY, [oClipboard, oView, oXTitle, oYTitle]

END

```

Printing a Plot Display

This example sends a damped sine wave plot to a window object and a printer object. The damped sine wave data comes from the `damp_sn2.dat` file found in the `examples/data` directory. The resolution of the printed page is based on the resolution of the screen. The model object in the printer object must be scaled to maintain the same size as displayed on the screen. The location of the view must also be changed to center the display on the page.

```

PRO PrintingAPlot

; Determine the path to the "damp_sn2.dat" file.
signalFile = FILEPATH('damp_sn2.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize the parameters of the image with the file.
signalSize = 512
signal = BYTARR(signalSize)

; Open the file, read in the image, and then close the
; file.
OPENR, unit, signalFile, /GET_LUN
READU, unit, signal
FREE_LUN, unit

; Determine viewplane size and margins.
offsetScale = 150.

```

```

viewOffset = offsetScale*[-1., -1., 1., 1.]
signalRange = MAX(signal) - MIN(signal)

; Initialize the display objects.
windowSize = [512, 384]
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'Damped Sine Wave with Noise')
oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., signalSize, signalRange] + $
    viewOffset)
oModel = OBJ_NEW('IDLgrModel')

; Initialize the plot object.
oPlot = OBJ_NEW('IDLgrPlot', signal, COLOR = [0, 0, 255])

; Obtain plot ranges.
oPlot -> GetProperty, XRange = xPlotRange, $
    YRange = yPlotRange

; Initialize axes objects, which are based on the plot
; ranges.
oXTitle = OBJ_NEW('IDLgrText', 'Time (seconds)')
oXAxis = OBJ_NEW('IDLgrAxis', 0, RANGE = xPlotRange, $
    LOCATION = [xPlotRange[0], yPlotRange[0]], /EXACT, $
    TITLE = oXTitle, TICKDIR = 0, $
    TICKLEN = (0.02*(yPlotRange[1] - yPlotRange[0])))
oYTitle = OBJ_NEW('IDLgrText', 'Amplitude (centimeters)')
oYAxis = OBJ_NEW('IDLgrAxis', 1, RANGE = yPlotRange, $
    LOCATION = [xPlotRange[0], yPlotRange[0]], /EXACT, $
    TITLE = oYTitle, TICKDIR = 0, $
    TICKLEN = (0.02*(xPlotRange[1] - xPlotRange[0])))

; Add plot and axes to model, which is added to the
; view, and then displayed in the window.
oModel -> Add, oPlot
oModel -> Add, oXAxis
oModel -> Add, oYAxis
oView -> Add, oModel
oModel -> Translate, -50., -50., 0.
oWindow -> Draw, oView

; Determine the centimeter measurements of the plot
; on the screen.
screenResolution = [!D.X_PX_CM, !D.Y_PX_CM]
windowSizeCM = windowSize/screenResolution

; Initialize printer destination object.
oPrinter = OBJ_NEW('IDLgrPrinter', PRINT_QUALITY = 2, $

```

```

QUALITY = 2)

; Obtain page parameters to determine the page
; size in centimeters.
oPrinter -> GetProperty, DIMENSIONS = pageSize, $
    RESOLUTION = pageResolution
pageSizeCM = pageSize*pageResolution

; Calculate a ratio between screen size and page size.
pageScale = windowSizeCM/pageSizeCM

; Use ratio to scale the model within the printer to the
; same size as the model on the screen.
oModel -> Scale, pageScale[0], pageScale[1], 1.

; Determine the center of the page and the screen
; display in pixels.
centering = (((pageSizeCM - windowSizeCM)/4.) $
    /pageResolution) - offsetScale

; Move the view to center the page.
oView -> SetProperty, LOCATION = centering

; Display the view within the printer destination.
oPrinter -> Draw, oView, /VECTOR

; Cleanup object references.
OBJ_DESTROY, [oPrinter, oView, oXTitle, oYTitle]

END

```

Copying an Image Display to the Clipboard

This example displays an image of the Earth's mantle convection in a window object and a clipboard object. The convection image data comes from the `convec.dat` file found in the `examples/data` directory. The resolution of the clipboard is based on the resolution of the screen, which is very similar to copying a plot display. The image is displayed from the system's clipboard to a platform-related graphics file (PostScript file on UNIX, Enhanced Metafile on Windows, or a PICT file on Macintosh) and an encapsulated PostScript file on all the platforms.

```

PRO SendingImageToClipboard

; Determine the path to the "convec.dat" file.
convecFile = FILEPATH('convec.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize the parameters of the image with the file.

```

```

convecSize = [248, 248]
convecImage = BYTARR(convecSize[0], convecSize[1])

; Open the file, read in the image, and then close the
; file.
OPENR, unit, convecFile, /GET_LUN
READU, unit, convecImage
FREE_LUN, unit

; Initialize the display objects.
windowSize = convecSize
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'Earth Mantle Convection')
oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., windowSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image object with its palette.
oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LOADCT, 27
oImage = OBJ_NEW('IDLgrImage', convecImage, $
    PALETTE = oPalette)

; Add image to model, which is added to the view, and
; then the view is displayed in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Determine the centimeter to pixel resolution of the
; image on the screen.
screenResolution = [1./!D.X_PX_CM, 1./!D.Y_PX_CM]

; Initialize clipboard destination object.
oClipboard = OBJ_NEW('IDLgrClipboard', QUALITY = 2, $
    DIMENSIONS = windowSize, $
    RESOLUTION = screenResolution)

; Determine the type of export file, which depends on
; the screen device.
screenDevice = !D.NAME
CASE screenDevice OF
    'X': fileExtension = '.ps'
    'WIN': fileExtension = '.emf'
    'MAC': fileExtension = '.pict'
ELSE: RETURN
ENDCASE
clipboardFile = 'convec' + fileExtension

```

```

; Display the view within the clipboard destination,
; which exports to an PS, EMF, or PICT file.
oClipboard -> Draw, oView, FILENAME = clipboardFile, $
/VECTOR
oClipboard -> Draw, oView, FILENAME = 'convec.eps', $
/POSTSCRIPT, /VECTOR

; Cleanup object references.
OBJ_DESTROY, [oClipboard, oView, oPalette]

END

```

Printing an Image Display

This example sends an image of the Earth's mantle convection to a window object and a printer object. The convection image data comes from the `convec.dat` file found in the `examples/data` directory. The resolution of the printed page is based on the resolution of the screen. The model object in the printer object must be scaled to maintain the same size as displayed on the screen. The location of the view must also be changed to center the display on the page.

```

PRO PrintingAnImage

; Determine the path to the "convec.dat" file.
convecFile = FILEPATH('convec.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize the parameters of the image with the file.
convecSize = [248, 248]
convecImage = BYTARR(convecSize[0], convecSize[1])

; Open the file, read in the image, and then close the
; file.
OPENR, unit, convecFile, /GET_LUN
READU, unit, convecImage
FREE_LUN, unit

; Initialize the display objects.
windowSize = convecSize
oWindow = OBJ_NEW('IDLgrWindow', RETAIN = 2, $
    DIMENSIONS = windowSize, $
    TITLE = 'Earth Mantle Convection')
oView = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT = [0., 0., windowSize])
oModel = OBJ_NEW('IDLgrModel')

; Initialize the image object with its palette.

```

```

oPalette = OBJ_NEW('IDLgrPalette')
oPalette -> LOADCT, 27
oImage = OBJ_NEW('IDLgrImage', convectImage, $
    PALETTE = oPalette)

; Add image to model, which is added to the view, and
; then the view is displayed in the window.
oModel -> Add, oImage
oView -> Add, oModel
oWindow -> Draw, oView

; Determine the centimeter measurements of the image
; on the screen.
screenResolution = [!D.X_PX_CM, !D.Y_PX_CM]
windowSizeCM = windowSize/screenResolution

; Initialize printer destination object.
oPrinter = OBJ_NEW('IDLgrPrinter', PRINT_QUALITY = 2, $
    QUALITY = 2)

; Obtain page parameters to determine the page
; size in centimeters.
oPrinter -> GetProperty, DIMENSIONS = pageSize, $
    RESOLUTION = pageResolution
pageSizeCM = pageSize*pageResolution

; Calculate a ratio between screen size and page size.
pageScale = windowSizeCM/pageSizeCM

; Use ratio to scale the model within the printer to the
; same size as the model on the screen.
oModel -> Scale, pageScale[0], pageScale[1], 1.

; Determine the center of the page and the image in
; pixels.
centering = ((pageSizeCM - windowSizeCM)/2.) $
    /pageResolution

; Move the view to center the image.
oView -> SetProperty, LOCATION = centering

; Display the view within the printer destination.
oPrinter -> Draw, oView

; Cleanup object references.
OBJ_DESTROY, [oPrinter, oView, oPalette]

END

```

Capturing IDL Direct Graphics Displays

An IDL display is usually written to an image file by first capturing it into an image array and then writing the array to an image file. Successful capture of an IDL display in the Direct Graphics system depends on the visual class of your current device. If your current device has a `PseudoColor` visual class, the display should be captured as an indexed image. If your current device has a `TrueColor` visual class, the display should be captured as a RGB (red, green, and blue) image (a three-channel image). IDL's `TVRD` routine has the ability to capture either indexed or RGB images. See [TVRD](#) in the *IDL Reference Guide* for more information on this routine. This section includes the following examples:

- “[Capturing Direct Graphics Displays on PseudoColor Devices](#)” in the following section
- “[Capturing Direct Graphics Displays on TrueColor Devices](#)” on page 360

Capturing Direct Graphics Displays on PseudoColor Devices

This example changes the current device from the screen to the Z-buffer. The Z-buffer device is a `PseudoColor` device. A contour of the *elev* data (from the `marbells.dat` save file) is displayed with a color table in the Z-buffer device. The display is captured with the `TVRD` routine. `TVRD` does not require any arguments or keywords to be set when capturing a display from a `PseudoColor` device.

```
PRO CapturingADisplayinPseudoColor

; Determine path to file.
marbellsFile = FILEPATH('marbells.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Restore "elev" contained with file, which is an IDL
; save file.
RESTORE, marbellsFile

; Initialize window parameters.
windowSize = [512, 384]

; Determine name ('MAC', 'WIN', or 'X') of screen device.
screenDevice = !D.NAME

; Change display device to the Z-buffer, which is a
; pseudocolor device regardless of system settings.
SET_PLOT, 'z'

; Set size of Z-buffer device to be the same as the
```

```

; expected screen window size.
DEVICE, SET_RESOLUTION = windowSize

; Load a color table.
LOADCT, 38

; Display data. The "elev" variable is scaled to only
; show the data above 2666 feet.
CONTOUR, elev > 2666, /XSTYLE, /YSTYLE, NLEVELS = 18, $
    /FILL

; Capture display.
contourDisplay = TVRD()

; Close Z-buffer device and switch back to the
; screen device.
DEVICE, /CLOSE
SET_PLOT, screenDevice

; If the screen device is TrueColor, set the DECOMPOSED
; keyword to 0 before using any color table related
; routines.
DEVICE, DECOMPOSED = 0

; Load a color table.
LOADCT, 38

; Initialize the display window.
WINDOW, 0, XSIZE = windowSize[0], YSIZE = windowSize[1], $
    TITLE = 'Maroon Bells Elevation Data'

; Display the captured image.
TV, contourDisplay

END

```

Capturing Direct Graphics Displays on TrueColor Devices

This example requires a TrueColor display. If your screen is not a TrueColor device, you are probably running on a PseudoColor device. For capturing a display on a PseudoColor device, see [“Capturing Direct Graphics Displays on PseudoColor Devices”](#) on page 359.

In this example, a contour of the *elev* data (from the *marbells.dat* save file) is displayed with a color table. The TVRD routine is used with the TRUE keyword set to 1 to capture the display as a pixel-interleaved RGB image. TVRD requires the TRUE keyword to be set when capturing a display from a TrueColor device.

```

PRO CapturingADisplayinTrueColor

; NOTE: this example requires a TrueColor display.  If
; you do not have a TrueColor display, see the
; "capturingADisplayinPseudoColor" example routine
; for more information.

; Determine path to file.
marbellsFile = FILEPATH('marbells.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Restore "elev" contained with file, which is an IDL
; save file.
RESTORE, marbellsFile

; Initialize window parameters.
windowSize = [512, 384]

; If the screen device is TrueColor, set the DECOMPOSED
; keyword to 0 before using any color table related
; routines.
DEVICE, DECOMPOSED = 0

; Load in a color table.
LOADCT, 38

; Initialize display window.
WINDOW, 0, XSIZE = windowSize[0], YSIZE = windowSize[1], $
    TITLE = 'Maroon Bells Elevation Data'

; Display data.  The "elev" variable is scaled to only
; show the data above 2666 feet.
CONTOUR, elev > 2666, /XSTYLE, /YSTYLE, NLEVELS = 18, $
    /FILL

; Incorrect capture of display. This use of TVRD
; assumes a PseudoColor display. In other words, only
; one visual channel is being captured as an indexed
; image.
incorrectCapture = TVRD()

; Correct capture of display. Since the display is
; TrueColor, the resulting capture should contain all
; of the channels to capture all of the color
; information within the display. In other words, since
; the display is TrueColor, the resulting capture
; should be a RGB image.
correctCapture = TVRD(TRUE = 1)

```

```
; Display incorrect results.
WINDOW, 1, XSIZE = windowSize[0], YSIZE = windowSize[1], $
    TITLE = 'Incorrect Captured Display'
TV, incorrectCapture

; Set the DECOMPOSED keyword to 1 displaying a RGB image.
DEVICE, DECOMPOSED = 1

; Display correct results.
WINDOW, 2, XSIZE = windowSize[0], YSIZE = windowSize[1], $
    TITLE = 'Correct Captured Display'
TV, correctCapture, TRUE = 1

END
```

Creating and Restoring .sav Files

Using the SAVE procedure, you can easily create reusable custom templates, save variable data, or share a utility or program you have created with other IDL users by packaging routines or data into a binary .sav file. This section includes the following examples of using SAVE and RESTORE:

- “Customizing and Saving an ASCII Template” in the following section
- “Saving and Restoring the XROI Utility and Image ROI Data” on page 365

Warning

While files containing IDL variables can be restored by any version of IDL that supports the data types of the variables (in particular, by any version of IDL later than the version that created the SAVE file), files containing IDL routines can only be restored by versions of IDL that share the same internal code representation. Since the internal code representation changes regularly, you should always archive the IDL language source files (.pro files) for routines you are placing in IDL .sav files so you can recompile the code when a new version of IDL is released.

Customizing and Saving an ASCII Template

When importing an ASCII data file into IDL, you must first describe the format of the data using the interactive ASCII_TEMPLATE function. If you have a number of ASCII files that have the same format, you can create and save a customized ASCII template using the SAVE procedure. After creating a .sav file of your custom template, you can avoid having to repeatedly define the same fields and records when reading in ASCII files that have the same structure.

1. At the IDL command line, enter the following to create the variable *plotTemplate*, which will contain your custom ASCII template:

```
plotTemplate = ASCII_TEMPLATE( )
```

A dialog box appears, prompting you to select a file.

2. Select `plot.txt` located in the `examples/data` directory.

Note

Another way to import ASCII data is to use the **Import ASCII File** toolbar button on the IDLDE toolbar. To use this feature, simply click the button and select `plot.txt` from the file selection dialog.

3. After selecting the file, the **Define Data Type/Range** dialog appears. First, choose the field type. Since the data file is delimited by tabs (or whitespace) select the **Delimited** button. In the **Data Starts at Line** field, specify to begin reading the data at line 3, not line 1, since there are two comment lines at the beginning of the file. Click **Next** to continue.
4. In the **Define Delimiter/Fields** dialog box, select **Tab** as the delimiter between data elements since it is known that tabs were used in the original file. Click **Next**.
5. In the **Field Specification** dialog box, name each field as follows:
 - Click on the first row (row 1). In the **Name** field, enter `time`.
 - Select the second row and enter `temperature1`.
 - Select the third row and enter `temperature2`.
6. Click **Finish**.
7. Type the following line at the IDL command line to read in the `plot.txt` file using the custom template, `plotTemplate`:


```
PLOT_ASCII = READ_ASCII(FILEPATH('plot.txt', SUBDIRECTORY = $
    ['examples', 'data']), TEMPLATE = plotTemplate)
```
8. Enter the following line to print the `plot.txt` file data:

```
PRINT, PLOT_ASCII
```

The file contents are printed in the Output Log window. Your output will resemble the following display.

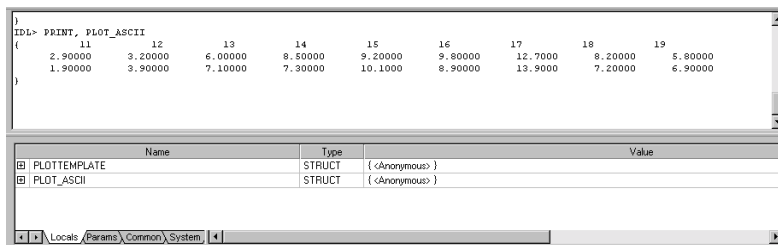


Figure 7-8: PLOT_ASCII Printout

9. Create a binary `.sav` file of your custom template by entering the following:


```
SAVE, plotTemplate, FILENAME='myPlotTemplate.sav'
```

10. To restore the template so that you can read another ASCII file, enter:

```
RESTORE, 'myPlotTemplate.sav'
```

This file contains your custom ASCII template information stored in the structure variable, `plotTemplate`.

Note

If you are attempting to restore a file that is not in your current working directory or the IDL search path, you will need to specify a path to the file. See [RESTORE](#) in the *IDL Reference Guide* for more information.

11. After restoring your custom template, you can read another ASCII file that is delimited in the same way as the original file by using the `READ_ASCII` function and specifying `plotTemplate` for the `TEMPLATE`:

```
PLOT_ASCII = READ_ASCII(FILEPATH('plot.txt', $
    SUBDIRECTORY = ['examples', 'data']), $
    TEMPLATE = plotTemplate)
```

12. Enter the following to display the contents of the file using the customized ASCII template structure previously defined using the dialog.

```
PRINT, PLOT_ASCII
```

Saving and Restoring the XROI Utility and Image ROI Data

You can easily share your own IDL routines or utilities with other IDL users by using the `SAVE` routine to create a binary file of your compiled code. The following example creates a `.sav` file of the XROI utility (a `.pro` file) and from within this file, restores a secondary `.sav` file containing selected regions of interest.

1. Type `XROI` at the command line to open the XROI utility.
2. In the file selection dialog, select `mineral.png` located in the `examples/data` directory.
3. Select the **Draw Polygon** toolbar button and roughly outline the three large, angular areas of the image.
4. Select **File** → **Save ROIs** and name the file `mineralROI.sav`. This creates a `.sav` file containing the regions of interest selected within the image.
5. In an IDL Editor or text editor, enter the following routine:

```
PRO myXRoi
```

```

; Restore ROI object data by specifying a value for the
; RESTORED_OBJECTS keyword.
RESTORE, 'mineralROI.sav', RESTORED_OBJECTS = myROI

; Open XROI, specifying the previously defined value for the
; restored object data as the value for "REGIONS_IN".
XROI, READ_PNG(FILEPATH('mineral.png', $
SUBDIRECTORY = ['examples', 'data'])), $
REGIONS_IN = myROI, /BLOCK

END

```

Save the routine as `myXRoi.pro`

6. Exit and restart IDL or enter `.FULL_RESET_SESSION` at the IDL command line before creating a `.sav` file to avoid saving unwanted session information.
7. After re-opening the `myXRoi` routine, compile the program you just created:

```
.COMPILE myXRoi.pro
```

8. Use `RESOLVE_ALL` to iteratively compile any uncompiled user-written or library procedures or functions that are called in any already-compiled procedure or function:

```
RESOLVE_ALL
```

Note

`RESOLVE_ALL` does not resolve class methods, nor procedures or functions that are called via quoted strings such as `CALL_PROCEDURE`, `CALL_FUNCTION`, or `EXECUTE`, or in keywords that can contain procedure names such as `TICKFORMAT` or `EVENT_PRO`. You must manually compile these routines.

9. Create a `.sav` file named `myXRoi.sav`, containing all of the `XROI` utility routines. When the `SAVE` procedure is called with the `ROUTINES` keyword and no arguments, it creates a `.sav` file containing all currently compiled routines. Because the routines associated with the `XROI` utility are the only ones that are currently compiled in our IDL session, we can create a `.sav` file as follows:

```
SAVE, /ROUTINES, FILENAME='myXRoi.sav'
```

10. It is not necessary to use `RESTORE` to open `myXRoi.sav`. If the main level routine is named the same as the `.sav` file, and all necessary files (in this case, `mineralROI.sav` and `myXRoi.sav`) are stored in the current working directory or the IDL search path, simply type the name of the file, minus the `.sav` extension, at the command line:

myXRoi

The following figure will appear, showing the selected regions of interest.

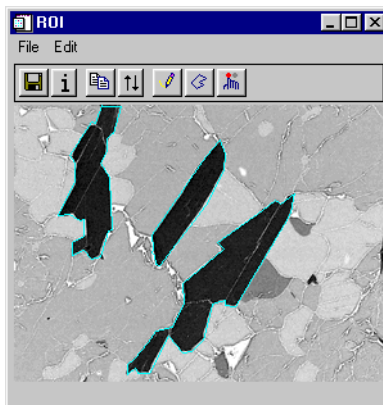


Figure 7-9: Example of Restoring the XROI Utility and ROI Image Data

Handling Table Widgets in GUIs

This example shows how to handle the events issued by a table widget within a graphical user interface (GUI) written in IDL. The example GUI presents an image from the `abnorm.dat` file in a draw widget. The `abnorm.dat` file is in the `examples/data` directory. The table widget in the GUI contains the values of the image's array. You can change the values within the table and the image display will be updated with that value. The GUI also provides labels (text) showing what events have occurred within the table. See [WIDGET_TABLE](#) in the *IDL Reference Guide* for more information about the events to IDL's table widget.

Each widget within the GUI has its own related event handler routine. Since the GUI is contained in a single program, the event handler routines appear before the GUI-creation routine. The file containing this program should be named the same as the GUI-creation (main) routine at the bottom of the program. This routine is called `WorkingWithTablesInGUIs`. The `doneEvent` routine handles the event from the **Done** button and the `TableEvent` routine handles the events from the table.

```
; NOTE: IDL GUI programs usually contain more than one
; routine; one routine creates the interface and other
; routines handle the events created by the interface.
; The "workingWithTablesInGUIs" routine is the main
; routine within this example program and is located at
; the bottom of this file. The main routine should
; always be at the end of the program file, and should
; be named the same as the program file name. You
; should look at the main routine first before trying
; to understand the event handling routines.

; A routine to handle the event issued by clicking on
; "Done" button.
PRO DoneEvent, event

; Destroy the GUI.
WIDGET_CONTROL, event.top, /DESTROY

END

; A routine to handle the events caused by the table.
PRO TableEvent, event

; Obtain the current image array from the table to
; redisplay the image when an table event occurs to
; show any updates in the table within the image.
WIDGET_CONTROL, event.id, GET_VALUE = image
```

```

; Determine the size of the image.
imageSize = SIZE(image, /DIMENSIONS)
; Redisplay image resized to fit the window.
TV, CONGRID(REVERSE(image, 2), $
    6*imageSize[0], 6*imageSize[1])

; Initialize descriptions of event types to be used
; within the type label.
CASE event.type OF
    0: description = ' (Insert Single Character)'
    1: description = ' (Insert Multiple Characters)'
    2: description = ' (Delete Text)'
    3: description = ' (Text Selection)'
    4: description = ' (Cell Selection)'
    6: description = ' (Row Height Changed)'
    7: description = ' (Column Width Changed)'
    8: description = ' (Invalid Data)'
ENDCASE

; Derive the label based on the event type that occurred.
typeIndex = 'Type: ' + STRTRIM(event.type, 2) + $
    description
; Find the reference to the type label.
typeLabel = WIDGET_INFO(event.top, $
    FIND_BY_UNAME = 'type')
; Use the reference to update the type label with the
; event type that occurred.
WIDGET_CONTROL, typeLabel, SET_VALUE = typeIndex

; If the event type is 4, a cell or cells have been
; selected. If a cell or cells have been selected, the
; selection label is updated to show a change in
; selection.
IF (event.type EQ 4) THEN BEGIN
    ; Derive the label based on the new selection.
    left = STRTRIM(event.sel_left, 2)
    top = STRTRIM(event.sel_top, 2)
    right = STRTRIM(event.sel_right, 2)
    bottom = STRTRIM(event.sel_bottom, 2)
    selectionValue = 'Left = ' + left + ', Top = ' + $
        top + ', Right = ' + right + ', and Bottom = ' + $
        bottom
    ; Find the reference to the selection label.
    selectionLabel = WIDGET_INFO(event.top, $
        FIND_BY_UNAME = 'selection')
    ; Use the reference to update the selection label
    ; with the new selection that occurred.
    WIDGET_CONTROL, selectionLabel, $
        SET_VALUE = selectionValue
ENDIF

```

```

END

; The main routine used to create the interface and
; start the event handlers.
PRO WorkingWithTablesInGUIs

; Determine path to file.
abnormFile = FILEPATH('abnorm.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize display parameters.
abnormSize = [64, 64]
abnormImage = BYTARR(abnormSize[0], abnormSize[1])

; Open file, read in image, and close file.
OPENR, unit, abnormFile, /GET_LUN
READU, unit, abnormImage
FREE_LUN, unit

; Create background base to contain the entire
; interface. This type of base is usually referred to
; as a "top level base". WIDGETs are displayed in the
; order in which they are created. Since the "top level"
; (background) is a column base, the WIDGETs in this
; program will be stacked from top to bottom:
;
;   WIDGET_DRAW (the image display)
;   WIDGET_TABLE (the table of image values)
;   WIDGET_LABELS (text describing events)
;   WIDGET_BUTTON (the done button)
topLevelBase = WIDGET_BASE(/COLUMN, $
    TITLE = 'Gated Blood Pool')

; Create a draw WIDGET to display the image.
abnormDraw = WIDGET_DRAW(topLevelBase, $
    XSIZE = 6*abnormSize[0], YSIZE = 6*abnormSize[1])

; Create a table WIDGET to view the values within the
; image's array.
abnormTable = WIDGET_TABLE(topLevelBase, $
    ; The image's rows are reversed to match the image's
    ; display.
    VALUE = REVERSE(abnormImage, 2), $
    ; The row labels are changed to match the values of
    ; the reversed-rowed image.
    ROW_LABELS = STRTRIM((abnormSize[1] - 1) - $
        INDGEN(abnormSize[1]), 2), $
    ; All events are specified to show all the possible

```

```

; events associated with the table. The cells are
; table are made editable to show how to link a table
; to an image display.
/ALL_EVENTS, /EDITABLE, $
; Allow scrolling within the table, which will be
; 4 columns by 10 rows in size.
/SCROLL, X_SCROLL_SIZE = 4, Y_SCROLL_SIZE = 10, $
; Associate an event handling routine specifically
; just for the table events to maintain structure
; within this program.
EVENT_PRO = 'TableEvent')

; Create a label to show what type of table event is
; occurring.
typeLabel = WIDGET_LABEL(topLevelBase, /ALIGN_CENTER, $
    VALUE = 'Type: ', /DYNAMIC_RESIZE, UNAME = 'type')

; Create a title for the selection label.
selectionTitle = WIDGET_LABEL(topLevelBase, $
    /ALIGN_LEFT, VALUE = 'Selection Information:')

; Create a label to show the current cell selection of
; the table.
selectionLabel = WIDGET_LABEL(topLevelBase, $
    /ALIGN_CENTER, /DYNAMIC_RESIZE, UNAME = 'selection', $
    VALUE = 'Left = 0, Top = 0, Right = 0, and Bottom = 0')

; Create a button to the user to quit out of the
; interface.
doneButton = WIDGET_BUTTON(topLevelBase, $
    ; The "VALUE" is the label displayed on the button.
    VALUE = 'Done', $
    ; Associate an event handling routine specifically
    ; just for the done event to maintain structure
    ; within this program.
    EVENT_PRO = 'DoneEvent')

; Display the interface.
WIDGET_CONTROL, topLevelBase, /REALIZE

; Determine the number reference of the window within
; the draw WIDGET. The number will be used to set the
; display to the draw WIDGET before image is shown.
WIDGET_CONTROL, abnormDraw, GET_VALUE = abnormWindow

; Set the display to the draw WIDGET's window.
WSET, abnormWindow

; If you are on a TrueColor display, set

```

```
; the DECOMPOSED keyword to 0 before using any color
; table related routines.
DEVICE, DECOMPOSED = 0

; Load a color table.
LOADCT, 5

; Display the image resized to fit the window.
TV, CONGRID(abnormImage, 6*abnormSize[0], $
    6*abnormSize[1])

; Start the event handling routines.
XMANAGER, 'WorkingWithTablesInGUIs', topLevelBase

END
```

The resulting GUI is similar to the following figure.

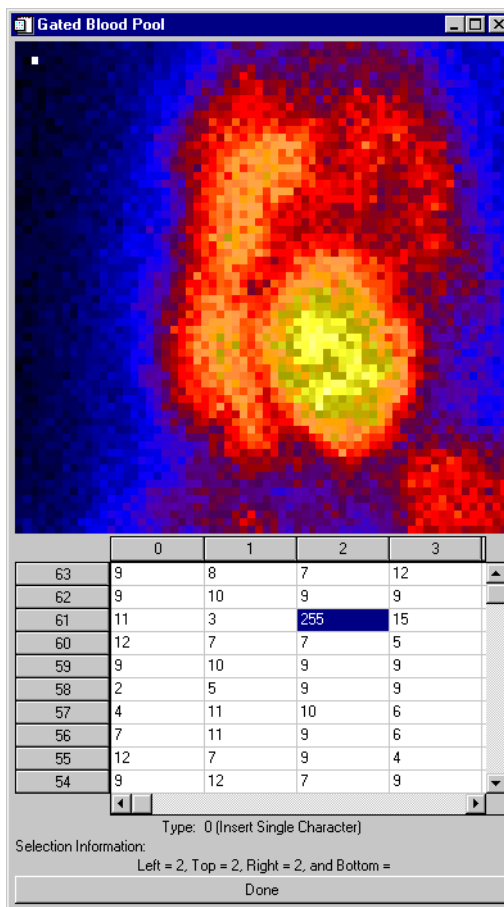


Figure 7-10: Example of a GUI Containing a Table

Finding Straight Lines in Images

This example uses the Hough transform to find straight lines within an image. The image comes from the `rockland.png` file found in the `examples/data` directory. The image is a saturation composite of a 24 hour period in Rockland, Maine. A saturation composite is normally used to highlight intensities, but the Hough transform is used in this example to extract the power lines, which are straight lines. The Hough transform is applied to the green band of the image. The results of the transform are scaled to only include lines longer than 100 pixels. The scaled results are then backprojected by the Hough transform to produce an image of only the power (straight) lines.

```
PRO FindingPowerLinesInRocklandME

; Determine path to file.
file = FILEPATH('rockland.png', $
    SUBDIRECTORY = ['examples', 'data'])

; Import image from file into IDL.
image = READ_PNG(file)

; Determine size of image.
imageSize = SIZE(image, /DIMENSIONS)

; Display cropped image
DEVICE, DECOMPOSED = 1
WINDOW, 0, XSIZE = imageSize[1], YSIZE = imageSize[2], $
    TITLE = 'Rockland, Maine'
TV, image, TRUE = 1

; Use layer from green channel as the intensity of the
; image.
intensity = REFORM(image[1, *, *])

; Determine size of intensity image.
intensitySize = SIZE(intensity, /DIMENSIONS)

; Mask intensity image to highlight power lines.
mask = intensity GT 240

; Transform mask.
transform = HOUGH(mask, RHO = rho, THETA = theta)

; Scale transform to obtain just the power lines.
transform = (TEMPORARY(transform) - 100) > 0
```

```

; Backproject to compare with original image.
backprojection = HOUGH(transform, /BACKPROJECT, $
    RHO = rho, THETA = theta, $
    NX = intensitySize[0], NY = intensitySize[1])

; Reverse color table to clarify lines. If you are on
; a TrueColor display, set the DECOMPOSED keyword to 0
; before using any color table related routines.
DEVICE, DECOMPOSED = 0
LOADCT, 0
TVLCT, red, green, blue, /GET
TVLCT, 255 - red, 255 - green, 255 - blue

; Display results.
WINDOW, 1, XSIZE = intensitySize[0], $
    YSIZE = intensitySize[1], $
    TITLE = 'Resulting Power Lines'
TVSCL, backprojection

END

```

The results for this example are shown in the following figure.

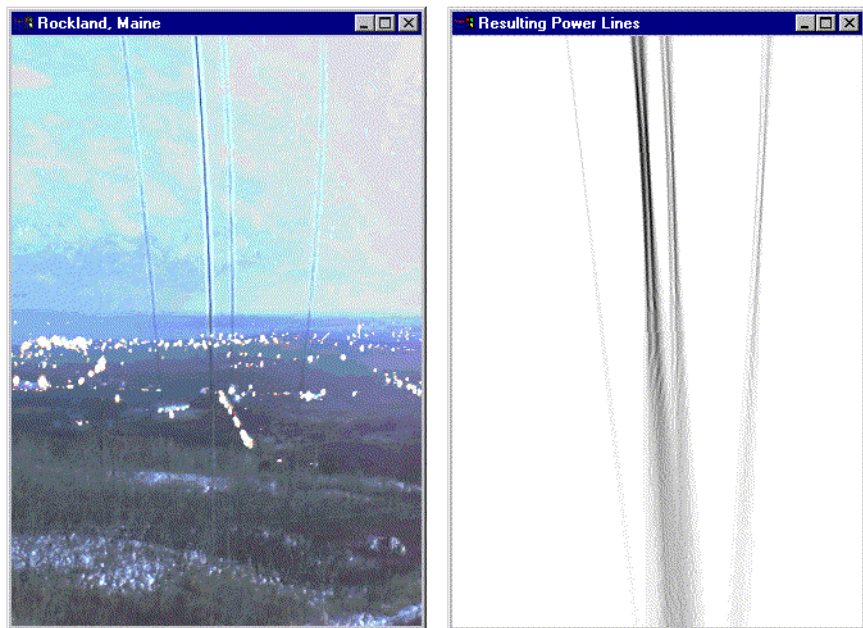


Figure 7-11: Original Image (left) and Filtered Image (right)

Color Density Contrasting in an Image

This example uses the Radon transform to provide more contrast within an image based on its color density. The image comes from the `endocell.jpg` file found in the `examples/data` directory. The image is a photomicrograph of cultured endothelial cells. The edges (outlines) within the image are defined by the Roberts filter. The Radon transform is applied to the filtered image. The high intensity values within the triangle of the center of the transform represent high color density within the filtered and original image. The transform is scaled to only include the values above the mean of the transform. The scaled results are backprojected by the Radon transform. The resulting backprojection is used as a mask on the original image. The final resulting image shows more color contrast bounded by the edges of the filtered image.

```
PRO ContrastingCells

; Determine path to file.
file = FILEPATH('endocell.jpg', $
    SUBDIRECTORY = ['examples', 'data'])

; Import image within file into IDL.
READ_JPEG, file, endocellImage

; Determine image's size, but divide it by 4 to reduce
; the image.
imageSize = SIZE(endocellImage, /DIMENSIONS)/4

; Resize image to quarter its original length and width.
endocellImage = CONGRID(endocellImage, $
    imageSize[0], imageSize[1])

; If you are on a truecolor display, set the DECOMPOSED
; keyword to the DEVICE command to zero before using
; any color table related routines.
DEVICE, DECOMPOSED = 0

; Load in the STD GAMMA-II color table.
LOADCT, 5

; Initialize the display.
WINDOW, 0, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
    TITLE = 'Original (left) and Filtered (right)'

; Display original image.
TV, endocellImage, 0
```

```

; Filter original image to clarify the edges of the
; cells.
image = ROBERTS(endocellImage)

; Display filtered image.
TVSCL, image, 1

; Transform the filtered image.
transform = RADON(image, RHO = rho, THETA = theta)

; Display transforms of the image.
transformSize = SIZE(transform, /DIMENSIONS)
WINDOW, 1, TITLE = 'Original Transform (top) and ' + $
    'Scaled Transform (bottom)', $
    XSIZE = transformSize[0], YSIZE = 2*transformSize[1]
TVSCL, transform, 0

; Scale the transform to include only the density
; values above the mean of the transform.
scaledTransform = transform > MEAN(transform)

; Display scaled transform.
TVSCL, scaledTransform, 1

; Backproject the scaled transform.
backprojection = RADON(scaledTransform, /BACKPROJECT, $
    RHO = rho, THETA=theta, NX = imageSize[0], $
    NY = imageSize[1])

; Initialize another display.
WINDOW, 2, XSIZE = 2*imageSize[0], YSIZE = imageSize[1], $
    TITLE = 'Backproject (left) and Final Result (right)'

; Display backprojection.
TVSCL, backprojection, 0

; Use the backprojection as a mask to provide
; a color density contrast of the original image.
constrastingImage = endocellImage*backprojection

; Display resulting contrast image.
TVSCL, endocellImage*backprojection, 1

END

```

The results for this example are shown in the following figure.

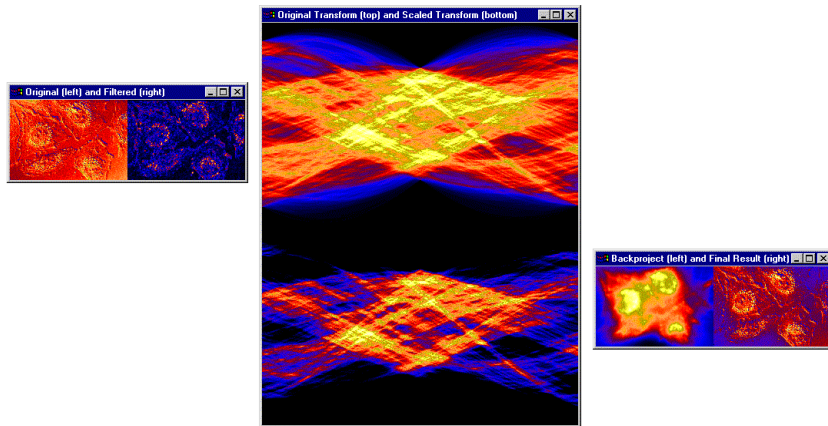


Figure 7-12: Original and Filtered Images (left), Original and Scaled Transforms (middle), and Backprojection and Final Resulting Contrast (right)

Removing Noise from an Image with FFT

This example uses the FFT transform to remove noise from an image. The image comes from the `abnorm.dat` file found in the `examples/data` directory. The first display contains the original image and its FFT transform. The noise is very evident in the image. A surface of the transform helps to determine the threshold necessary to remove the noise from the image. In the surface of the transform, the noise appears random and below a ridge containing a spike. The ridge and spike represent the actual data within the image. A mask is applied to the transform to remove the noise and the inverse transform is applied resulting in a clearer image.

```
PRO RemovingNoiseFromAnImageWithFFT

; Determine the path to the file.
file = FILEPATH('abnorm.dat', $
    SUBDIRECTORY = ['examples', 'data'])

; Initialize size parameter and image array.
imageSize = [64, 64]
image = BYTARR(imageSize[0], imageSize[1])

; Open file, read in image, and close file.
OPENR, unit, file, /GET_LUN
READU, unit, image
FREE_LUN, unit

; Initialize display parameters, including a color
; table. If you are on a TrueColor display, set
; the DECOMPOSED keyword to 0 before using any color
; table related routines.
displaySize = [128, 128]
DEVICE, DECOMPOSED = 0
LOADCT, 5
WINDOW, 0, XSIZE = 2*displaySize[0], $
    YSIZE = displaySize[1], $
    TITLE = 'Original Image : Transformation'

; Display original image.
TVSCL, CONGRID(image, displaySize[0], displaySize[1], $
    /INTERP), 0

; Transform image.
transform = ALOG(SHIFT(FFT(image), (imageSize[0]/2), $
    (imageSize[1]/2)))

; Display transformation.
TVSCL, CONGRID(transform, displaySize[0], $
```

```

displaySize[1], /INTERP), 1

; Scale transform make its minimum value equal to zero.
scaledTransform = transform - MIN(transform)

; Display results of scaling.
WINDOW, 1, TITLE = 'Transform Scaled to a Zero Minimum'
SURFACE, scaledTransform, /XSTYLE, /YSTYLE, $
    TITLE = 'Transform Scaled to a Zero Minimum'

; Filter scaled transform to only include high
; frequency data.
mask = FLOAT(scaledTransform) GT 6.
filteredTransform = (scaledTransform*mask) + $
    MIN(transform)

; Initialize display.
WINDOW, 2, XSIZE = 2*displaySize[0], $
    YSIZE = displaySize[1], $
    TITLE = 'Filtered Transformation : Results'

; Display filtered transform.
TVSCL, CONGRID(FLOAT(filteredTransform), displaySize[0], $
    displaySize[1], /INTERP), 0

; Apply inverse transformation to filtered transform.
inverseTransform = ABS(FFT(EXP(filteredTransform), $
    /INVERSE))

; Display results of inverse transformation.
TVSCL, CONGRID(inverseTransform, displaySize[0], $
    displaySize[1], /INTERP), 1

END

```

The results for this example are shown in the following figure.

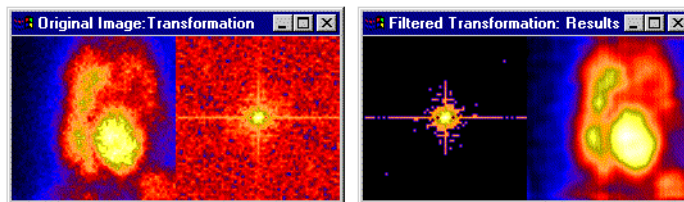


Figure 7-13: Original Image and FFT Transform (left) and Filtered FFT Transform and Resulting Image (right)

Using Double and Triple Integration

You can use the QROMB, QROMO, and QSIMP routines within the user-supplied function of these same routines. This ability allows you to perform double and triple integration. Each term of an integrand (the equation within the integral) can contain another integration method. The double and triple integrations are performed over each term of the integrand. The following two examples use double integration to determine the volume under a surface defined by a two-dimensional equation and triple integration to determine the mass of a volume with a density defined by a three-dimensional equation, respectively.

This section includes the following topics:

- [“Integrating to Determine the Volume Under a Surface \(Double Integration\)”](#) in the following section
- [“Integrating to Determine the Mass of a Volume \(Triple Integration\)”](#) on page 382

Integrating to Determine the Volume Under a Surface (Double Integration)

This example evaluates the volume under a surface by using the following double integration:

$$\text{volume} = \int_0^1 \int_0^1 (9x^2y^2 + 4xy + 1) dx dy$$

A surface is defined by a two-dimensional equation. The volume under this surface can be determined by performing a double integration over a specific region (boundary). This example performs the double integration over the range 0 to 1 in the x-direction and 0 to 1 in the y-direction. The correct solution to this integration is 3.

This example program is made up of four routines: the main routine, the integration in the y direction, the second integration of the x coefficient, and the second integration of the x^2 coefficient. The main routine is the last routine in the program. The file containing this program should be named the same as the main routine.

```
FUNCTION XSquaredCoef, x

; Integration of the x squared coefficient.
secondIntegration = 9.*x^2
RETURN, secondIntegration
```

```

END

FUNCTION XCoef, x

    ; Integration of the linear x coefficient.
    secondIntegration = x
    RETURN, secondIntegration

END

FUNCTION YDirection, y

    ; Re-write equation to consider both x coefficients.
    firstIntegration = QROMB('XSquaredCoef', 0., 1.)*y^2 $
    + 4.*(QROMB('XCoef', 0., 1.))*y + 1.
    RETURN, firstIntegration

END

PRO DoubleIntegration

    ; Determine the volume under the surface represented
    ; by 9x^2y^2 + 4xy + 1 over a specific region.
    volume = QROMB('YDirection', 0., 1. )

    ; Output results.
    PRINT, 'Resulting Volume: ', volume

END

```

Integrating to Determine the Mass of a Volume (Triple Integration)

This example evaluates the mass of a volume by using the following triple integration on a three-dimensional equation representing its density:

$$\text{mass} = \int_0^1 \int_0^1 \int_0^1 (9x^2y^2 + 8xyz + 1) dx dy dz$$

The density of a volume is defined by a three-dimensional equation. The mass of this volume can be determined by performing a triple integration over a specific region (boundary). This example performs the triple integration over the range 0 to 1 in the x-direction, 0 to 1 in the y-direction, and 0 to 1 in the z-direction. The correct solution to this integration is 3.

This example program is made up of six routines: the main routine, the integration in the z-direction, the second integration of the xy coefficient, the second integration of the second x^2y^2 coefficient, the third integration in the x coefficient, and the third integration in the x^2 coefficient. The main routine is the last routine in the program. The file containing this program should be named the same as the main routine.

```

FUNCTION XSquaredCoef, x

    ; Integration of the x squared coefficient.
    thirdIntegration = 9.*x^2
    RETURN, thirdIntegration

END

FUNCTION XCoef, x

    ; Integration of the linear x coefficient.
    thirdIntegration = x
    RETURN, thirdIntegration

END

FUNCTION XSquaredYSquaredCoef, y

    ; Integration of the y squared coefficient.
    secondIntegration = QROMB('XSquaredCoef', 0., 1.)*y^2
    RETURN, secondIntegration

END

FUNCTION XYCoef, y

    ; Integration of the linear y coefficient.
    secondIntegration = QROMB('XCoef', 0., 1.)*y
    RETURN, secondIntegration

END

FUNCTION ZDirection, z

    ; Re-write equation to consider all the x and y
    ; coefficients.
    firstIntegration = QROMB('XSquaredYSquaredCoef', 0., 1.) + $
    8.*(QROMB('XYCoef', 0., 1.))*z + 1.
    RETURN, firstIntegration

END

```

```
PRO TripleIntegration

; Determine the mass of the density represented
; by  $9x^2y^2 + 8xyz + 1$  over a specific region.
mass = QROMB('ZDirection', 0., 1. )

; Output results.
PRINT, 'Resulting Mass: ', mass

END
```

Obtaining Irregular Grid Intervals

The XOUT and YOUT keywords allow you to obtain an irregular interval from the TRIGRID routine. This example creates an irregularly-gridded dataset of a Gaussian surface. A grid is formed from these points with the TRIANGULATE and TRIGRID routines. The inputs to the XOUT and YOUT keywords are determined at random to produce an irregular interval. These inputs are sorted before setting them to XOUT and YOUT because these keywords require monotonically ascending or descending values. The lines of the resulting surface are spaced at the irregular intervals provided by the settings of the XOUT and YOUT keywords. See [TRIANGULATE](#) and [TRIGRID](#) in the *IDL Reference Guide* for more information on these routines.

```

PRO GriddingIrregularIntervals

; Make 100 normal x, y points:
x = RANDOMN(seed, 100)
y = RANDOMN(seed, 100)
PRINT, MIN(x), MAX(x)
PRINT, MIN(y), MAX(y)

; Make a Gaussian surface:
z = EXP(-(x^2 + y^2))

; Obtain triangulation:
TRIANGULATE, x, y, triangles, boundary

; Create random x values. These values will be used to
; form the x locations of the resulting grid.
gridX = RANDOMN(seed, 30)
; Sort x values. Sorted values are required for the XOUT
; keyword.
sortX = UNIQ(gridX, SORT(gridX))
gridX = gridX[sortX]
; Output sorted x values to be used with the XOUT
; keyword.
PRINT, 'gridX:'
PRINT, gridX

; Create random y values. These values will be used to
; form the y locations of the resulting grid.
gridY = RANDOMN(seed, 30)
; Sort y values. Sorted values are required for the YOUT
; keyword.
sortY = UNIQ(gridY, SORT(gridY))
gridY = gridY[sortY]
; Output sorted y values to be used with the YOUT

```

```

; keyword.
PRINT, 'gridY:'
PRINT, gridY

; Derive grid of initial values. The location of the
; resulting grid points are the inputs to the XOUT and
; YOUT keywords.
grid = TRIGRID(x, y, z, triangles, XOUT = gridX, $
              YOUT = gridY, EXTRAPOLATE = boundary)

; Display resulting grid. The grid lines are not
; at regular intervals because of the randomness of the
; inputs to the XOUT and YOUT keywords.
SURFACE, grid, gridX, gridY, /XSTYLE, /YSTYLE

END

```

A possible result for this example is shown in the following figure.

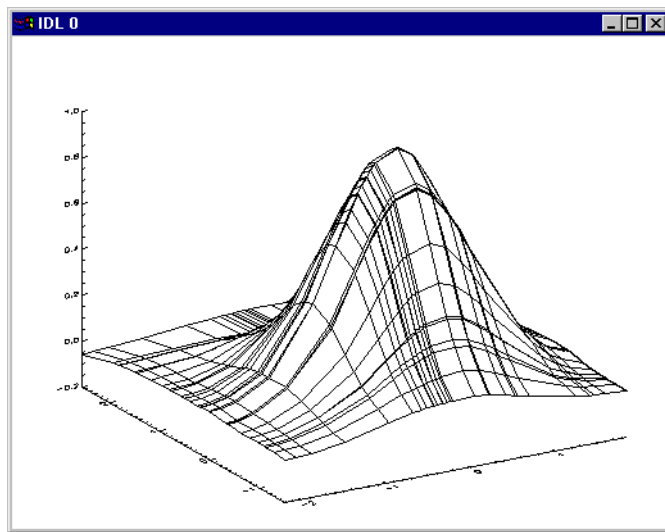


Figure 7-14: A Possible Irregular Interval Result

Calculating Incomplete Beta and Gamma Functions

Tolerance controls allow you to calculate the accuracy of the incomplete beta and gamma functions. More accuracy usually provides better results, but can cause slower computation speeds. If faster speeds are important, a less accurate calculation may be more desirable. This trade-off can be maintained through tolerances.

Iteration controls allow you to expand the computation enough to converge to a result. Calculation of these functions may not converge to a result within the default number of iterations. If the number of iterations is increased, the calculation may converge.

This section includes the following topics:

- [“Working With Tolerances in the Incomplete Beta Function”](#) in the following section
- [“Working With Iteration Controls in the Incomplete Gamma Function”](#) on page 388

Working With Tolerances in the Incomplete Beta Function

This example shows the difference in accuracy between the incomplete beta function computed with a low tolerance and the incomplete beta function computed with a high tolerance. The resulting surfaces show the relative errors of each. The relative error of the low tolerance ranges from 0 to 0.002 percent. The relative error of the high tolerance ranges from 0 to 0.0000000004 percent.

```
PRO UsingIBETAwithEPS

; Define an array of parametric exponents.
parameterA = (DINDGEN(101)/100. + 1.D) # REPLICATE(1.D, 101)
parameterB = REPLICATE(1.D, 101) # (DINDGEN(101)/10. + 1.D)

; Define the upper limits of integration.
upperLimits = REPLICATE(0.1D, 101, 101)

; Compute the incomplete beta functions.
betaFunctions = IBETA(parameterA, parameterB, $
    upperLimits)

; Compute the incomplete beta functions with a less
; accurate tolerance set.
laBetaFunctions = IBETA(parameterA, parameterB, $
```

```

        upperLimits, EPS = 3.0e-4)

; Compute relative error.
relativeError = 100.* $
        ABS((betaFunctions - laBetaFunctions)/betaFunctions)

; Display resulting relative error.
WINDOW, 0, TITLE = 'Compare IBETA with Less Accurate EPS'
SURFACE, relativeError, parameterA, parameterB, $
        /XSTYLE, /YSTYLE, TITLE = 'Relative Error', $
        XTITLE = 'Parameter A', YTITLE = 'Parameter B', $
        ZTITLE = 'Percent Error (%)', CHARSIZE = 1.5

; Compute the incomplete beta functions with a more
; accurate tolerance set..
maBetaFunctions = IBETA(parameterA, parameterB, $
        upperLimits, EPS = 3.0e-10)

; Compute relative error.
relativeError = 100.* $
        ABS((maBetaFunctions - betaFunctions)/maBetaFunctions)

; Display resulting relative error.
WINDOW, 1, TITLE = 'Compare IBETA with More Accurate EPS'
SURFACE, relativeError, parameterA, parameterB, $
        /XSTYLE, /YSTYLE, TITLE = 'Relative Error', $
        XTITLE = 'Parameter A', YTITLE = 'Parameter B', $
        ZTITLE = 'Percent Error (%)', CHARSIZE = 1.5

END

```

Working With Iteration Controls in the Incomplete Gamma Function

This example shows how increasing the maximum number of iterations can change the outcome of computing the incomplete gamma function. Normally, the calculation of the incomplete gamma function will not converge within 100 iterations (the default number of iterations) when the parametric exponent is set to 400 and the upper limit is set to 400. The ITMAX keyword to the IGAMMA routine is set to 200 to allow the calculation to converge to a value of 0.506686 within 101 iterations.

```

PRO UsingIGAMMAwithITMAX

; Define parametric exponent.
parameterA = 400.

; Define the upper limit of integration.

```

```
upperLimits = 400.  
  
; NOTE: with the above parameter and limit, IGAMMA will  
; not converge unless the number of iterations is  
; increased above the default of 100.  
  
; Compute the incomplete gamma function.  
gammaFunction = IGAMMA(parameterA, upperLimits, $  
    ITMAX = 200, ITER = numberIteration)  
  
; Output results.  
PRINT, 'Resulting Gamma Function: ', gammaFunction  
PRINT, 'Number of Iterations: ', numberIteration  
  
END
```

Determining Bessel Function Accuracy

Different orders between Bessel functions have recurrence relationships to each other. These relationships can be used to determine how accurately IDL is computing the Bessel functions. In the following examples, the recurrence relationships at each order are set to zero and the left side of the equations are plotted. These plots show how close the left side of the equations are to zero, and therefore, how accurate the Bessel functions are computed within IDL.

This section includes the following topics:

- “Analyzing the Bessel Function of the First Kind” in the following section
- “Analyzing the Bessel Function of the Second Kind” on page 392
- “Analyzing the Modified Bessel Function of the First Kind” on page 394
- “Analyzing the Modified Bessel Function of the Second Kind” on page 396

Analyzing the Bessel Function of the First Kind

This example uses the following recurrence relationship:

$$x(J_{n-1}(x) + J_{n+1}(x)) - 2nJ_n(x) = 0$$

where $J(x)$ is the Bessel function of the first kind of order $n-1$, n , or $n+1$. The resulting plots are for n equal to 1 through 6. All of these plots show that this Bessel function is calculated within machine tolerance.

```
PRO AnalyzingBESELJ

; Derive x values.
x = (DINDGEN(1000) + 1.)/100.

; Initialize display window.
WINDOW, 0, TITLE = 'Bessel Functions'

; Display the first 8 orders of the Bessel function of
; the first kind.
PLOT, x, BESELJ(x, 0), /XSTYLE, /YSTYLE, $
    XTITLE = 'x', YTITLE = 'f(x)', $
    TITLE = 'Bessel Functions of the First Kind'
OPLOT, x, BESELJ(x, 1), LINESTYLE = 1
OPLOT, x, BESELJ(x, 2), LINESTYLE = 2
OPLOT, x, BESELJ(x, 3), LINESTYLE = 3
OPLOT, x, BESELJ(x, 4), LINESTYLE = 4
OPLOT, x, BESELJ(x, 5), LINESTYLE = 5
```

```

OPLOT, x, BESELJ(x, 6), LINESTYLE = 0
OPLOT, x, BESELJ(x, 7), LINESTYLE = 1

; Initialize display window for recurrence relations.
WINDOW, 1, XSIZE = 896, YSIZE = 512, $
    TITLE = 'Testing the Recurrence Relations'
!P.MULTI = [0, 2, 3, 0, 0]

; Initialize title variable.
nString = ['0', '1', '2', '3', '4', '5', '6', '7']

; Display recurrence relationships for order 1 to 6.
; NOTE: the results of these relationships should be
; very close to zero.
FOR n = 1, 6 DO BEGIN
    equation = x*(BESELJ(x, (n - 1)) + $
        BESELJ(x, (n + 1))) - 2.*FLOAT(n)*BESELJ(x, n)
    PLOT, x, equation, /XSTYLE, /YSTYLE, CHARSIZE = 1.5, $
        TITLE = 'n = ' + nString[n] + ': Orders of ' + $
            nString[n - 1] + ', ' + nString[n] + ', and ' + $
            nString[n + 1]
    PRINT, 'n = ' + nString[n] + ': '
    PRINT, 'minimum = ', MIN(equation)
    PRINT, 'maximum = ', MAX(equation)
ENDFOR

; Return display window back to its default setting, one
; display per window.
!P.MULTI = 0

END

```

The results for this example are shown in the following figure.

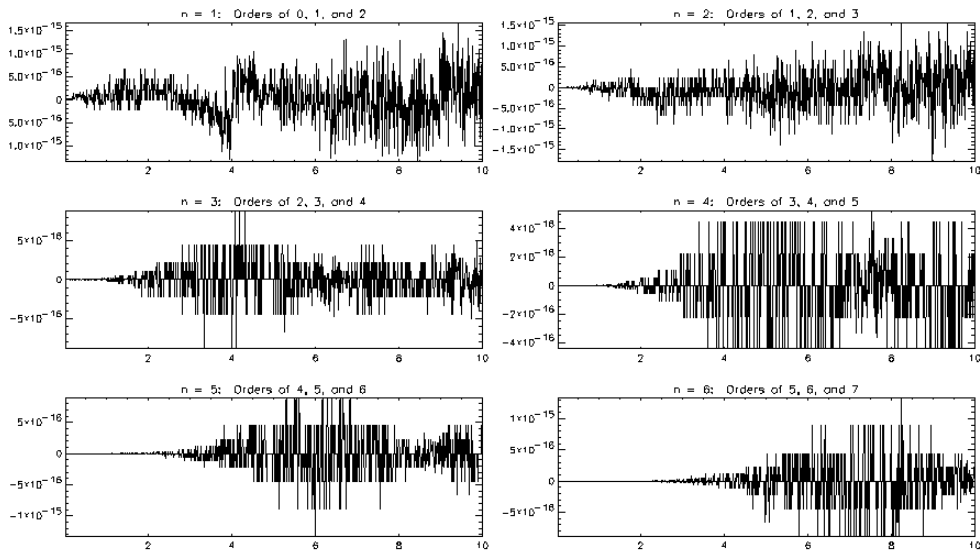


Figure 7-15: Recurrence Relationship for $J(x)$

Analyzing the Bessel Function of the Second Kind

This example uses the following recurrence relationship:

$$x(Y_{n-1}(x) + Y_{n+1}(x)) - 2nY_n(x) = 0$$

where $Y(x)$ is the Bessel function of the second kind of order $n - 1$, n , or $n + 1$. The resulting plots are for n equal to 1 through 6. All of these plots show that this Bessel function is calculated within machine tolerance.

```
PRO AnalyzingBESELY

; Derive x values.
x = (DINDGEN(1000) + 1.)/200. + 5.

; Initialize display window.
WINDOW, 0, TITLE = 'Bessel Functions'

; Display the first 8 orders of the Bessel function of
; the second kind.
PLOT, x, BESELY(x, 0), /XSTYLE, $
```

```

        /YSTYLE, YRANGE = [-1.3, 0.4], $
        XTITLE = 'x', YTITLE = 'f(x)', $
        TITLE = 'Bessel Functions of the Second Kind'
    OPLOT, x, BESELY(x, 1), LINESTYLE = 1
    OPLOT, x, BESELY(x, 2), LINESTYLE = 2
    OPLOT, x, BESELY(x, 3), LINESTYLE = 3
    OPLOT, x, BESELY(x, 4), LINESTYLE = 4
    OPLOT, x, BESELY(x, 5), LINESTYLE = 5
    OPLOT, x, BESELY(x, 6), LINESTYLE = 0
    OPLOT, x, BESELY(x, 7), LINESTYLE = 1

; Initialize display window for recurrence relations.
WINDOW, 1, XSIZE = 896, YSIZE = 512, $
    TITLE = 'Testing the Recurrence Relations'
!P.MULTI = [0, 2, 3, 0, 0] ; for multiple displays

; Initialize title variable.
nString = ['0', '1', '2', '3', '4', '5', '6', '7']

; Display recurrence relationships for order 1 to 6.
; NOTE: the results of these relationships should be
; very close to zero.
FOR n = 1, 6 DO BEGIN
    equation = x*(BESELY(x, (n - 1)) + $
        BESELY(x, (n + 1))) - 2.*FLOAT(n)*BESELY(x, n)
    PLOT, x, equation, /XSTYLE, /YSTYLE, CHARSIZE = 1.5, $
        TITLE = 'n = ' + nString[n] + ': Orders of ' + $
            nString[n - 1] + ', ' + nString[n] + ', and ' + $
            nString[n + 1]
    PRINT, 'n = ' + nString[n] + ': '
    PRINT, 'minimum = ', MIN(equation)
    PRINT, 'maximum = ', MAX(equation)
ENDFOR

; Return display window back to its default setting, one
; display per window.
!P.MULTI = 0

END

```

The results for this example are shown in the following figure.

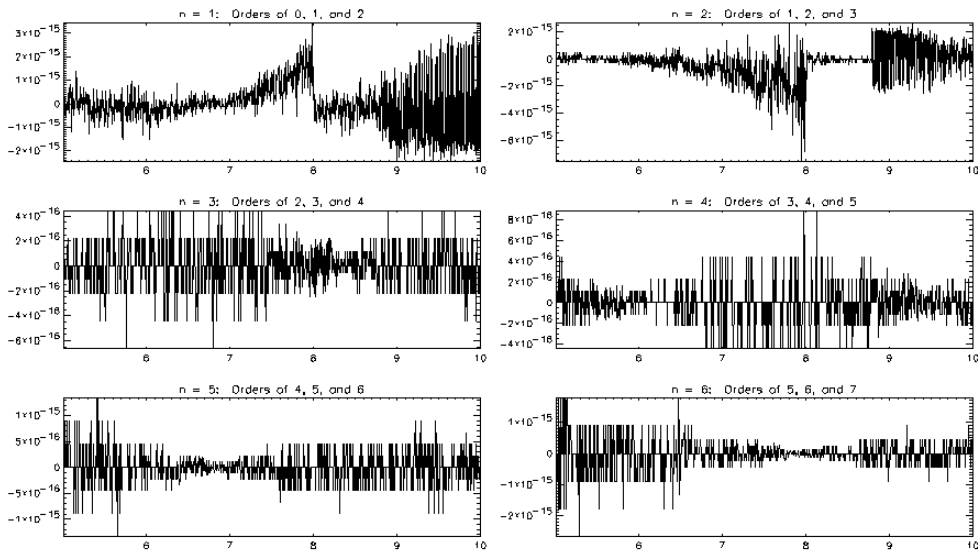


Figure 7-16: Recurrence Relationship for $Y(x)$

Analyzing the Modified Bessel Function of the First Kind

This example uses the following recurrence relationship:

$$x(I_{n-1}(x) - I_{n+1}(x)) - 2nI_n(x) = 0$$

where $I(x)$ is the modified Bessel function of the first kind of order $n - 1$, n , or $n + 1$. The resulting plots are for n equal to 1 through 6. All of these plots show that this Bessel function is calculated within machine tolerance.

```
PRO AnalyzingBESELI

; Derive x values.
x = (DINDGEN(1000) + 1.)/200.

; Initialize display window.
WINDOW, 0, TITLE = 'Modified Bessel Functions'

; Display the first 8 orders of the modified Bessel
; function of the first kind.
PLOT, x, BESELI(x, 0), /XSTYLE, /YSTYLE, $
```

```

        XTITLE = 'x', YTITLE = 'f(x)', $
        TITLE = 'Modified Bessel Functions of the First Kind'
    OPLOT, x, BESELI(x, 1), LINESTYLE = 1
    OPLOT, x, BESELI(x, 2), LINESTYLE = 2
    OPLOT, x, BESELI(x, 3), LINESTYLE = 3
    OPLOT, x, BESELI(x, 4), LINESTYLE = 4
    OPLOT, x, BESELI(x, 5), LINESTYLE = 5
    OPLOT, x, BESELI(x, 6), LINESTYLE = 0
    OPLOT, x, BESELI(x, 7), LINESTYLE = 1

; Initialize display window for recurrence relations.
WINDOW, 1, XSIZE = 896, YSIZE = 512, $
    TITLE = 'Testing the Recurrence Relations'
    !P.MULTI = [0, 2, 3, 0, 0]

; Initialize title variable.
nString = ['0', '1', '2', '3', '4', '5', '6', '7']

; Display recurrence relationships for order 1 to 6.
; NOTE: the results of these relationships should be
; very close to zero.
FOR n = 1, 6 DO BEGIN
    equation = x*(BESELI(x, (n - 1)) - $
        BESELI(x, (n + 1))) - 2.*FLOAT(n)*BESELI(x, n)
    PLOT, x, equation, /XSTYLE, /YSTYLE, CHARSIZE = 1.5, $
        TITLE = 'n = ' + nString[n] + ': Orders of ' + $
            nString[n - 1] + ', ' + nString[n] + ', and ' + $
            nString[n + 1]
    PRINT, 'n = ' + nString[n] + ': '
    PRINT, 'minimum = ', MIN(equation)
    PRINT, 'maximum = ', MAX(equation)
ENDFOR

; Return display window back to its default setting, one
; display per window.
!P.MULTI = 0

END

```

The results for this example are shown in the following figure.

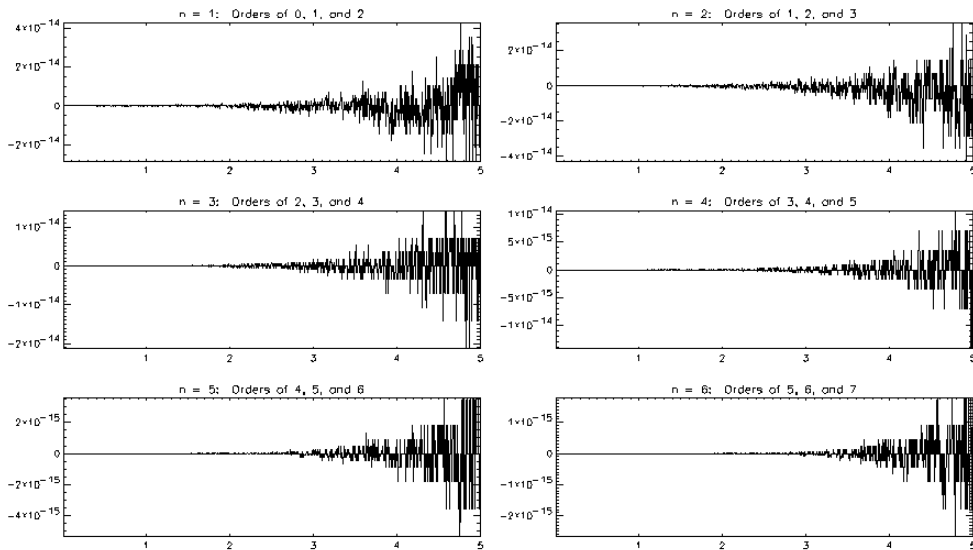


Figure 7-17: Recurrence Relationship for $I(x)$

Analyzing the Modified Bessel Function of the Second Kind

This example uses the following recurrence relationship:

$$x(K_{n-1}(x) - K_{n+1}(x)) + 2nK_n(x) = 0$$

where $K(x)$ is the modified Bessel function of the second kind of order $n - 1$, n , or $n + 1$. The resulting plots are for n equal to 1 through 6. All of these plots show that this Bessel function is calculated within machine tolerance.

```
PRO AnalyzingBESELK

; Derive x values.
x = (DINDGEN(1000) + 1.)/200. + 5.

; Initialize display window.
WINDOW, 0, TITLE = 'Modified Bessel Functions'

; Display the first 8 orders of the modified Bessel
; function of the second kind.
PLOT, x, BESELK(x, 0), /XSTYLE, /YSTYLE, $
```

```

        XTITLE = 'x', YTITLE = 'f(x)', $
        TITLE = 'Modified Bessel Functions of the Second Kind'
    OPLOT, x, BESELK(x, 1), LINESTYLE = 1
    OPLOT, x, BESELK(x, 2), LINESTYLE = 2
    OPLOT, x, BESELK(x, 3), LINESTYLE = 3
    OPLOT, x, BESELK(x, 4), LINESTYLE = 4
    OPLOT, x, BESELK(x, 5), LINESTYLE = 5
    OPLOT, x, BESELK(x, 6), LINESTYLE = 0
    OPLOT, x, BESELK(x, 7), LINESTYLE = 1

; Initialize display window for recurrence relations.
WINDOW, 1, XSIZE = 896, YSIZE = 512, $
    TITLE = 'Testing the Recurrence Relations'
    !P.MULTI = [0, 2, 3, 0, 0] ; for multiple displays

; Initialize title variable.
nString = ['0', '1', '2', '3', '4', '5', '6', '7']

; Display recurrence relationships for order 1 to 6.
; NOTE: the results of these relationships should be
; very close to zero.
FOR n = 1, 6 DO BEGIN
    equation = x*(BESELK(x, (n - 1)) - $
        BESELK(x, (n + 1))) + 2.*FLOAT(n)*BESELK(x, n)
    PLOT, x, equation, /XSTYLE, /YSTYLE, CHARSIZE = 1.5, $
        TITLE = 'n = ' + nString[n] + ': Orders of ' + $
            nString[n - 1] + ', ' + nString[n] + ', and ' + $
            nString[n + 1]
    PRINT, 'n = ' + nString[n] + ': '
    PRINT, 'minimum = ', MIN(equation)
    PRINT, 'maximum = ', MAX(equation)
ENDFOR

; Return display window back to its default setting, one
; display per window.
!P.MULTI = 0

END

```

The results for this example are shown in the following figure.

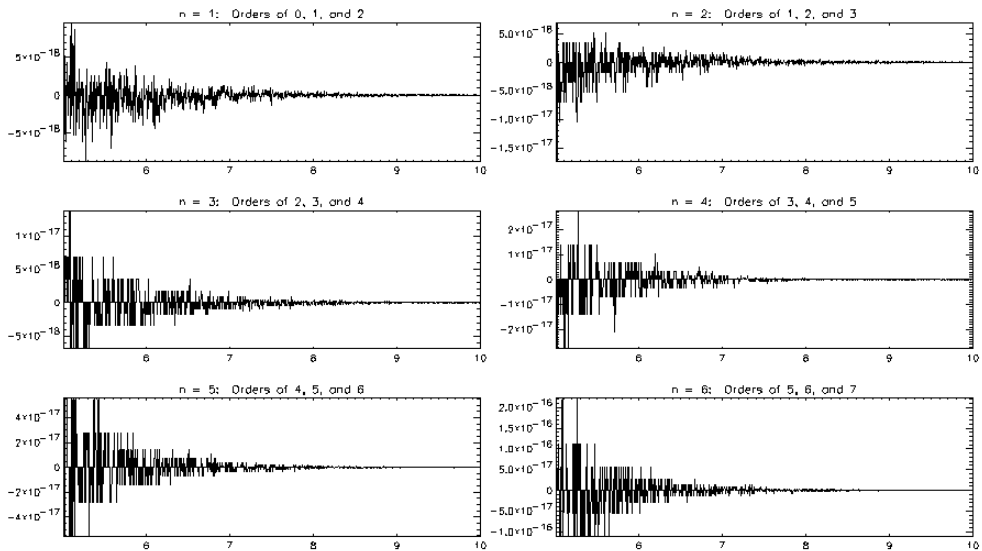


Figure 7-18: Recurrence Relationship for $K(x)$



Index

Symbols

- .sav file
 - creating, [363](#), [364–365](#)
 - restoring, [363](#), [365](#)
- .sid image files, [181](#)

A

- ActiveX, IDL hosting
 - COM uses, [139](#)
 - creating control, [150](#)
 - destroying control, [151](#)
 - dispatch, [150](#)
 - embedded control, [138](#), [152](#)
 - event propagation, [156](#)
 - instantiating, [149](#)
 - overview, [40](#)

- adding software functionality. *See* COM objects
- alpha blending, [328](#)
- alpha channel, [328](#)
- animating
 - isosurfaces, [57](#)
 - volumes, [57](#)
- array
 - creation routines, [135](#)
 - manipulation routines, [136](#)

B

- backprojection
 - Hough, [374](#)
 - Radon, [376](#)

- base widget shortcut menu
 - adding, [160](#)
 - creating, [158](#)
- BESELI function, [394](#)
- BESELJ function, [390](#)
- BESELK function, [396](#)
- BESELY function, [392](#)
- Bessel functions
 - first kind, [390](#)
 - modified first kind, [390](#), [394](#)
 - modified second kind, [390](#), [396](#)
 - recurrence relationship, [390–392](#), [394–396](#)
 - second kind, [390–392](#)
- beta function, [387](#)
- binary, unary operators, [134](#)
- building software components. *See* COM objects
- byte swapping routines, [136](#)

C

- centering image objects, [325](#)
- clipboard object, [351](#), [355](#)
- clipping meshes, [333](#)
- COM objects
 - ActiveX, IDL hosting, [139](#)
 - class and program identifiers, [140](#), [149](#)
 - creating, [176](#)
 - example, [145](#)
 - IDispatch management, [142](#)
 - IDLcomIDispatch object class, [176](#)
 - naming conventions, [142](#), [149](#)
 - naming scheme, [140](#)
 - overview, [138](#)
 - pointers, [145](#)
 - reference counting, [144](#)
 - referencing other COM objects, [145](#)
 - uses, [138](#)
- component object model. *See* COM objects
- computation speed. *See* multi-threading
- connectivity list, [332–333](#)

- context sensitive menu. *See* shortcut menus
- convex hulls, [276](#)
- coordinate conversion, [325](#)
- copying to a clipboard, [351](#)
- CPU procedure, [194](#)
- creating a .sav file, [363](#), [364–365](#)

D

- data type conversion routines, [135](#)
- decimating a mesh, [336](#)
- DEFINE_MSGBLK procedure, [197](#)
- DEFINE_MSGBLK_FROM_FILE procedure, [200](#)
- deleting a region of interest, [315](#)
- display capture in Direct Graphics
 - PseudoColor, [359](#)
 - TrueColor, [359–360](#)
- draw widget shortcut menu, [162](#)

E

- efficiency improvements. *See* multi-threading
- encapsulated PostScript file, [351](#), [355](#)
- Enhanced Metafile, [351](#), [355](#)
- ERF function, [203](#)
- ERFC function, [204](#)
- ERFCX function, [205](#)
- event handler, [368](#)

F

- FFT
 - inverse transform, [379](#)
 - transform, [379](#)
- file status, [32](#)
- FILE_INFO function, [206](#)
- FILE_SEARCH function, [210](#)
- format code
 - A, [31](#)

C(), [31](#)
 column moves, [31](#)
 D, [31](#)
 E, [31](#)
 F, [31](#)
 G, [31](#)
 I, [31](#)
 O, [31](#)
 open parenthesis, [31](#)
 T, [31](#)
 TL, [31](#)
 TR, [31](#)
 X, [31](#)
 Z, [31](#)
 freeing
 heap variables, [264](#)
 pointers, [151](#), [264](#)
 resources, [151](#)

G

gamma function, [387–388](#)
 generating tetrahedral meshes, [13](#)
 GRID_INPUT procedure, [224](#)
 GRIDDATA function, [228](#)
 gridding irregular intervals, [385](#)

H

HDF_VD_ATTRFIND function, [253](#)
 HDF_VD_ATTRINFO procedure, [254](#)
 HDF_VD_ATTRSET procedure, [256](#)
 HDF_VD_ISATTR function, [262](#)
 HDF_VD_NATTRS function, [263](#)
 heap variables
 freeing, [264](#)
 HEAP_FREE procedure, [264](#)
 high resolution textures, [12](#)
 histogram view of ROI, [311](#)

hosting ActiveX in IDL. *See* ActiveX, IDL
 hosting
 Hough
 backprojection, [374](#)
 transform, [374](#)

/

IBETA function, [387](#)
 IDispatch
 COM Class ID, [140](#)
 COM Program ID, [140](#)
 get and set properties, [144](#)
 interface, [138](#)
 managing COM objects, [142](#)
 naming conventions, [142](#)
 pointer handling, [145](#)
 IDLcomIDispatch
 GetProperty method, [179](#)
 Init method, [177](#)
 object class, [176](#)
 SetProperty method, [180](#)
 IDLffMrSID
 Cleanup method, [182](#)
 GetDimsAtLevel method, [183](#)
 GetImageData method, [185](#)
 GetProperty method, [188](#)
 Init method, [191](#)
 object class, [181](#)
 IGAMMA function, [388](#)
 image object
 centering, [325](#)
 transparent, [328](#)
 image processing routines, [135](#)
 incomplete beta function, [387](#)
 incomplete gamma function, [387–388](#)
 indexed image, [359](#)
 integration
 double, [381](#)
 triple, [381–382](#)

interpolation
 dependent variable to volume, [271](#)
 scattered data to regular, [228](#)
 INTERVAL_VOLUME procedure, [267](#)
 iteration controls, [388](#)

L

limit relaxed
 32-bit IDL, [27](#)
 A format code, [31](#)
 format code column, [31](#)
 format codes repetition count, [31](#)
 formatted I/O, [31](#)
 linear programming solutions, [287](#)
 list widget shortcut menu, [166](#)

M

mapping images onto geometry, [322](#)
 mathematical routines, [134](#)
 merging meshes, [339](#)
 MESH_OBJ procedure, [336](#), [345](#)
 meshes
 advanced example, [345](#)
 clipping, [333](#)
 decimating, [336](#)
 merging, [339](#)
 smoothing, [342](#)
 message block support, [28](#)
 modified Bessel functions
 See also Bessel functions.
 first kind, [390](#), [394](#)
 second kind, [390](#), [396](#)
 MrSID image files
 deleting, [182](#)
 dimensions, [183](#)
 extracting data, [185](#)
 loading, [181](#)
 query, [181](#)

 query properties, [188](#)
 multi-threading
 array creation routines, [135](#)
 array manipulation routines, [136](#)
 byte swapping support, [136](#)
 calculation speed, [126](#)
 controlling with CPU procedure, [129](#)
 data type conversion routines, [135](#)
 disabling with CPU procedure, [129](#)
 image processing routines, [135](#)
 math routines, [134](#)
 operators, [134](#)
 overriding default use, [126](#), [133](#)
 overview, [20](#)

N

naming conventions. *See* COM objects

O

object class enhancements, [60](#)
 obsolete routines, [122](#)
 obsoleted features, [122](#)
 OLE/COM Object Viewer, [141](#)
 optimal feasible vector, [287](#)
 overriding multi-threading, [128](#), [133](#)

P

path separation delimiters, [270](#)
 path specification, [210](#)
 PATH_SEP function, [270](#)
 performance enhancements, [20](#)
 PICT file, [351](#), [355](#)
 platforms supported, [124](#)
 pointers
 COM object use, [145](#)
 freeing, [264](#)

polygon object
 See also meshes.
 advanced example, 345
 clipping meshes, 333
 decimating meshes, 336
 displaying meshes, 332
 merging meshes, 339
 smoothing meshes, 342
 pop-up menu. *See* shortcut menus
 PostScript file, 351, 355
 printer object, 351, 353, 357
 processing speed. *See* multi-threading
 PseudoColor, 359

Q

QGRID3 function, 271
 QHULL procedure, 276
 QROMB function, 381
 QROMO function, 381
 QSIMP function, 381
 QUERY_MRSID function, 279

R

Radon
 backprojection, 376
 transform, 376
 READ_MRSID function, 281
 REAL_PART function, 283
 recursive file searching, 210
 reference counting methodology, 144
 referencing COM objects, 145
 region growing
 properties dialog, 312
 REGION_GROW overview, 14
 region of interest (ROI). *See* ROI
 REGION_GROW function, 284
 relaxed limits, 27–31
 RESOLVE_ALL procedure, 366

resources
 freeing, 151
 system, 126
 RESTORE procedure, 366
 restoring a .sav file, 363, 365
 retrieving image dimensions, 183
 RGB image, 359
 ROI
 deleting, 315
 geometric and statistical data, 303
 growing, 14, 312
 histogram view, 311
 using XROI procedure, 303
 routines enhanced, 72
 routines obsoleted, 122

S

.sav file
 creating, 363, 364–365
 restoring, 363, 365
 SAVE procedure, 365–366
 searching subdirectories, 210
 selectable list menu, 166
 shortcut menus
 creating a base, 160
 creating a draw widget, 162
 creating a text widget, 170
 deleting an ROI, 315
 displaying, 159
 events, 158
 selectable lists, 166
 .sid image files, 181
 SIMPLEX function, 287
 simplex method, 287
 smoothing meshes, 342
 string length limits, 27
 supported platforms, 124
 system variable enhancements, 121

T

table widget, [368](#)
 tetrahedral meshes, [13](#)
 text widget shortcut menu, [170](#)
 texture mapping, [322](#)
 thread pool. *See* multi-threading
 tolerance, [387](#)
 transforms
 FFT, [379](#)
 Hough, [374](#)
 inverse FFT, [379](#)
 Radon, [376](#)
 transparency
 alpha channel, [328](#)
 image object, [328](#)
 TRIANGULATE function, [333](#), [385](#)
 triangulation
 Delaunay, [276](#)
 scattered data points, [271](#)
 TRIGRID function, [385](#)
 TrueColor, [359–360](#)
 TVRD function, [359–360](#)
 type conversion routines, [135](#)

V

vertices, [332](#), [336](#), [339](#), [342](#)
 viewplane rectangle, [325](#)

Voronoi diagrams, [276](#)

W

WIDGET_ACTIVEX function, [291](#)
 WIDGET_DISPLAYCONTEXTMENU function, [298](#)
 WIDGET_TABLE, [368](#)
 widgets
 aligning keywords, [292](#)
 callbacks, [293](#), [294](#)
 sensitizing and de-sensitizing, [295](#)

X

XOBJVIEW_ROTATE procedure, [300](#)
 XOBJVIEW_WRITE_IMAGE procedure, [302](#)
 XROI
 growing a region, [312](#)
 importing images, [310](#)
 procedure, [303](#)
 using, [309](#)

Z

Z-buffer, [359](#)