



Using IDL

RSI
Research Systems Inc.

IDL Version 6.1
July, 2004 Edition
Copyright © Research Systems, Inc.
All Rights Reserved

Restricted Rights Notice

The IDL[®], ION Script[™], and ION Java[™] software programs 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 or ION software packages or their 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 and ION[™], ION Script[™], ION Java[™], are trademarks of Research Systems Inc., registered in the United States Patent and Trademark Office, for the computer program described herein.

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 1988-2001 The Board of Trustees of the University of Illinois
All rights reserved.

NCSA HDF5 (Hierarchical Data Format 5) Software Library and Utilities
Copyright 1998-2002 by the Board of Trustees of the University of Illinois. All rights reserved.

CDF Library
Copyright © 2002 National Space Science Data Center
NASA/Goddard Space Flight Center

NetCDF Library
Copyright © 1993-1999 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.

Portions of this software are copyrighted by DataDirect Technologies, 1991-2003.

Portions of this software were developed using Unisearch's Kakadu software, for which Kodak has a commercial license. Kakadu Software. Copyright © 2001. The University of New South Wales, UNSW, Sydney NSW 2052, Australia, and Unisearch Ltd, Australia.

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.

Portions of this software are copyrighted by Merge Technologies Incorporated.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>)

IDL Wavelet Toolkit Copyright © 2002 Christopher Torrence.

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



Contents

Part I: Using IDL

Chapter 1:	
Introducing IDL	17
Overview	18
Launching IDL	20
Environment Variables Used by IDL	22
Command Line Options for IDL Startup	27
Startup Files	30
Message of the Day Files	31
Using Your Mouse with IDL	32
Using Keyboard Accelerators	33
Quitting IDL	35
Helpful Resources	36

Chapter 2:	
The IDL Development Environment	41
Components of the IDLDE	42
File Menu	49
Edit Menu	53
Search Menu	55
Run Menu	57
Project Menu	62
Macros Menu	63
Window Menu	65
Help Menu	68
Keyboard Shortcuts	69
Chapter 3:	
Using the IDL Editor	73
About the IDL Editor	74
Keyboard Shortcuts	75
Searching	77
Text Selection Modes (Windows Only)	78
Chromacoded Editor (Windows Only)	80
Block Comments	81
Functions/Procedures Menu	82
Using External Editors (Motif)	83
Chapter 4:	
Using the IDL Command Input Line	85
Input to IDL	86
Executive Commands	90
Issuing Operating System Commands	92
Chapter 5:	
Setting IDL Preferences	93
Customizing IDL	94
General Preferences	97
Layout Preferences	99
Graphics Preferences	103
Editor Preferences	106

Startup Preferences	108
Font Preferences	110
Path Preferences	112
Chapter 6:	
Working With Macros	115
What are Macros?	116
Using the IDL Macro Toolbar	117
Creating UNIX Macros	118
Creating Windows Macros	121
Command Stream Substitutions	123
Building IDL Example Macros	124
Chapter 7:	
Printing in IDL	127
Overview	128
Printing Graphics	129
IDL Printer Setup in Windows	130
IDL Printer Setup in UNIX or Mac OS X	131
Chapter 8:	
Customizing IDL on Motif Systems	143
Using X Resources to Customize IDL	144
Command Line Options	147
Modifying the Control Panel	150
Action Routines	153
CDE File Manager Support	160
Chapter 9:	
Creating and Running Programs in IDL	161
Overview	162
Running IDL Programs	164
Interrupting Program Execution	168
Variable Context After Interruption	169
Aborting IDL	170
Saving Variables from an IDL Session	171
For More Information on Programming	173

Chapter 10:	
Executing Batch Jobs in IDL	175
Overview	176
Interpretation of Batch Statements	177
A Batch Example	178
Chapter 11:	
Journaling in IDL	179
Overview	180
Journaling Examples	182

Part II: Reading and Writing Data

Chapter 12:	
IDL Macros for Importing Data	185
Overview	186
Using Macros to Import Image Files	187
Using Macros to Import ASCII Files	191
Using Macros to Import Binary Files	197
Using Macros to Import HDF Files	203
Chapter 13:	
Reading and Writing Images	207
List of Commands	208
Accessing Image Files Using Dialogs	211
Accessing General Image File Formats	214
Accessing Specific Image File Formats	215
Accessing Files Using Dialogs	217
Accessing Files With Compound Widgets	219
Chapter 14:	
Reading and Writing ASCII Data	221
Overview	222
Reading an ASCII Data File	223
Advanced File Input/Output	227
Chapter 15:	
Reading and Writing Binary Data	229
Overview	230

Reading a Binary Data File	231
Advanced File Input/Output	235

Part III: Using Direct Graphics

Chapter 16: Graphics 239

Overview	240
IDL Direct Graphics	241
IDL Object Graphics	242

Chapter 17: Direct Graphics Plotting 243

Overview	244
Plotting Keyword Parameters	245
Direct Graphics Coordinate Systems	246
Annotation – The XYOUTS Procedure	255
Plotting Symbols	257
Polygon Filling	261
Tick Marks	265
Logarithmic Scaling	269
Multiple Plots on a Page	271
Specifying the Location of the Plot	273
Plotting Missing Data	275
Date/Time Plotting	276
Using the AXIS Procedure	287
Using the CURSOR Procedure	291

Chapter 18: Plotting Multi-Dimensional Arrays 293

Overview	294
Contour Plots	295
Overlaying Images and Contour Plots	301
Additional Contour Options	306
The SURFACE Procedure	310
Three-Dimensional Graphics	313
Three-Dimensional Transformations	323
Shaded Surfaces	329

Volume Visualization	332
References	336
Chapter 19:	
Map Projections	337
Overview	338
The MAP_SET Procedure	340
The MAP_GRID Procedure	343
The MAP_CONTINENTS Procedure	344
Graphics Techniques for Mapping	345
Map Projections Described	347
Azimuthal Projections	348
Cylindrical Projections	358
Pseudocylindrical Projections	363
Putting Data on Maps	366
High-Resolution Continent Outlines	368
References	370
Chapter 20:	
Image Display Routines	371
Overview	372
Images	373
Imaging Routines	374
Image Display	375
Reading from the Display Device	379
Color Tables	381
TrueColor Displays	389
Controlling the Device Cursor	393
References	394
Chapter 21:	
Signal Processing	395
Overview	396
Digital Signals	397
Signal Analysis Transforms	400
The Fourier Transform	401
Interpreting FFT Results	402

Displaying FFT Results	403
Using Windows	409
Aliasing	413
FFT Algorithm Details	414
The Hilbert Transform	415
The Wavelet Transform	417
Convolution	418
Correlation and Covariance	419
Digital Filtering	420
Finite Impulse Response (FIR) Filters	421
FIR Filter Implementation	425
Infinite Impulse Response Filters	427
Routines for Signal Processing	431
References	433
Chapter 22:	
Mathematics	435
IDL's Numerical Recipes Functions	437
Accuracy & Floating-Point Operations	438
Arrays and Matrices	440
Correlation Analysis	446
Curve and Surface Fitting	450
Eigenvalues and Eigenvectors	453
Gridding and Interpolation	459
Hypothesis Testing	462
Integration	465
Linear Systems	470
Nonlinear Equations	478
Optimization	480
Sparse Arrays	482
Time-Series Analysis	485
Multivariate Analysis	488
References	494

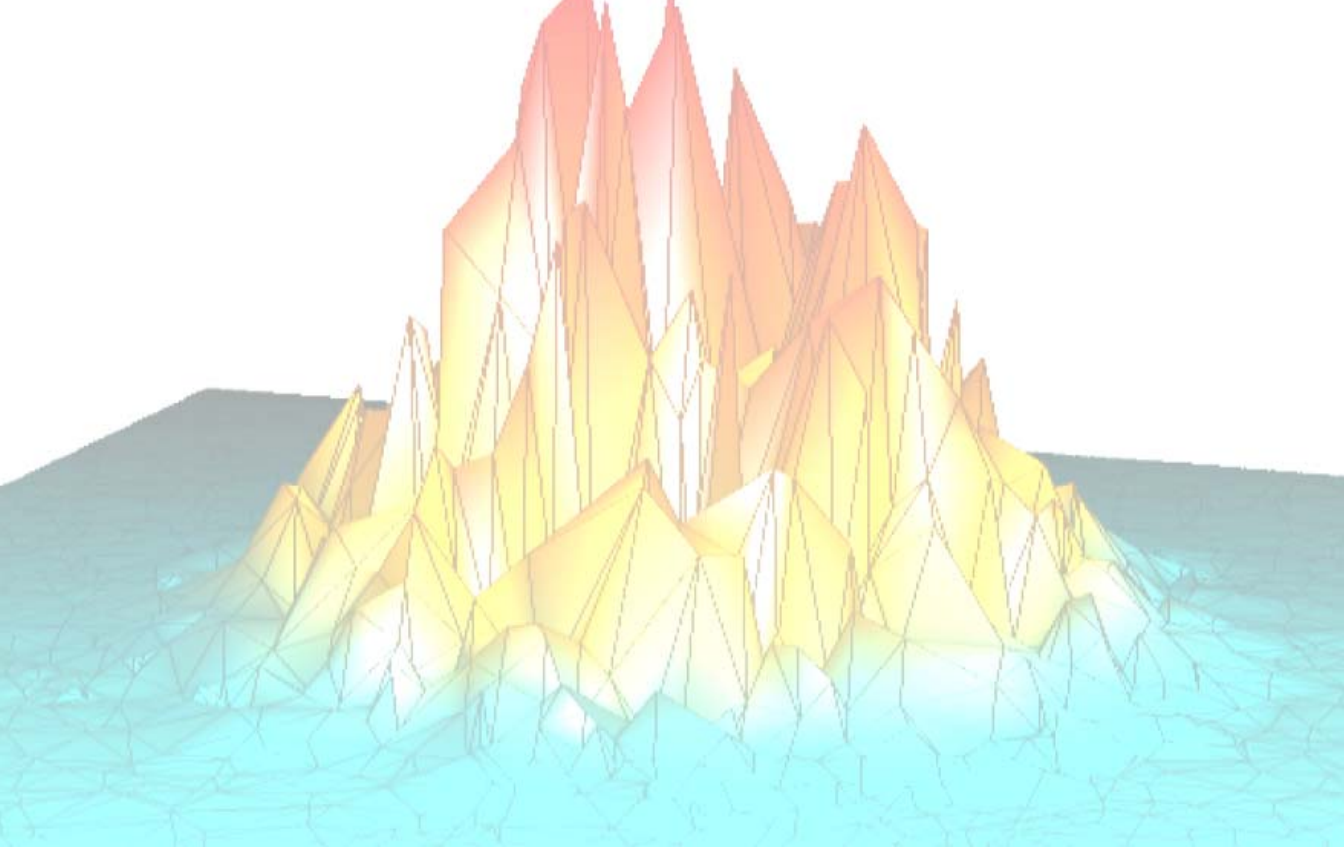
Part IV: Object Graphics

Chapter 23:	
Object Graphics	499
Overview	500
Direct versus Object Graphics	502
How to Use Object Graphics	504
Overview of Object Graphics Classes	506
Container Objects	508
Structure Objects	509
Atomic Graphic Objects	510
Composite Objects	513
Attribute Objects	514
Helper Objects	515
Destination Objects	516
File Format Objects	517
Properties of Objects	519
Alpha Channel and Objects	521
Undocumented Graphic Object Classes	527
Chapter 24:	
The Graphics Object Hierarchy	529
Overview	530
Scenes	531
Viewgroups	532
Views	533
Models	534
Atomic Graphic Objects	535
Attribute and Helper Objects	536
The Rendering Process	537
Simple Plot Example	539
Chapter 25:	
Transformations	541
Overview	542
Viewport	543
Projection	545

Eye Position	547
View Volume	549
Model Transformations	552
Coordinate Conversion	555
Example: Centering an Image	557
Example: Displaying a Surface	560
Virtual Trackball and 3D Transformations	563
Chapter 26:	
Working with Color	565
Overview	566
Color and Digital Data	567
Indexed Color Model	568
RGB Color Model	569
Color and Destination Objects	570
Palettes	571
Using Color	572
How IDL Interprets Color Values	574
Chapter 27:	
Using Attributes and Helpers	577
Overview	578
Font Objects	579
Palette Objects	582
Pattern Objects	583
Symbol Objects	585
Tessellator Objects	589
Chapter 28:	
Working with Axes and Text	591
Overview	592
Axis Objects	593
Text Objects	611
Chapter 29:	
Working with Plots and Graphs	615
Contour Objects	616
Polygon Objects	619

Polyline Objects	625
Plot Objects	626
Legend Objects	630
A Plotting Routine	633
Chapter 30:	
Working with Surfaces	637
Surface Objects	638
Light Objects	643
An Interactive Surface Example	646
Chapter 31:	
Working with Images	649
Image Objects	650
Colorbar Objects	654
Saving an Image to a File	656
Chapter 32:	
Working with Volumes	659
Volume Objects	660
Volume Object Attributes	662
Chapter 33:	
Selecting Objects	667
Selection and Data Picking	668
Selection	669
A Selection Example	671
Data Picking	672
A Data Picking Example	673
Chapter 34:	
Using Destination Objects	675
Overview	676
Window Objects	677
Using Window Objects	680
Instancing	682
Buffer Objects	684
Clipboard Objects	685

Printer Objects	687
VRML Objects	694
Bitmap and Vector Graphic Output	695
Chapter 35:	
Subclassing from Object Graphics	705
Creating Composite Classes or Subclasses	706
Chapter 36:	
Performance Tuning Object Graphics	709
Overview	710
Polygon Mesh Optimization	711
Normal Computations	714
Retained Graphics and Expose Events	715
Improving Redraw Performance	716
Back-face Culling	717
Lighting	718
Index	719



Part I: Using IDL



Chapter 1: Introducing IDL

This chapter includes information about IDL, the IDL documentation set, and how to contact RSI Technical Support. The following topics are covered in this chapter:

Overview	18	Message of the Day Files	31
Launching IDL	20	Using Your Mouse with IDL	32
Environment Variables Used by IDL	22	Using Keyboard Accelerators	33
Command Line Options for IDL Startup ...	27	Quitting IDL	35
Startup Files	30	Helpful Resources	36

Overview

IDL (the *Interactive Data Language*) is a complete computing environment for the interactive analysis and visualization of data. IDL integrates a powerful, array-oriented language with numerous mathematical analysis and graphical display techniques. Programming in IDL is a time-saving alternative to programming in FORTRAN or C. Using IDL, tasks which require days or weeks of programming with traditional languages can be accomplished in hours. You can explore data interactively using IDL commands and then create complete applications by writing IDL programs.

The advantages of IDL include:

- IDL is a complete, structured language that can be used interactively and to create sophisticated functions, procedures, and applications.
- Operators and functions work on entire arrays (without using loops), simplifying interactive analysis and reducing programming time.
- Compilation and execution of IDL commands provides instant feedback and hands-on interaction.
- Rapid 2D plotting, multi-dimensional plotting, volume visualization, image display, and animation allow immediate observation of your computation's results.
- Support for OpenGL-based hardware accelerated graphics.
- Many numerical and statistical analysis routines—including Numerical Recipes routines—are provided for analysis and simulation of data.
- IDL's flexible input/output facilities allow you to read any type of custom data format. Support is provided for:
 - common image standards: BMP, GEO TIFF, Interfile, JPEG, JPEG 2000, PICT, PNG, PPM, SRF, TIFF, X11 Bitmap, and XWD.
 - scientific data formats: CDF, HDF, and NetCDF.
 - other data formats: ASCII, Binary, DICOM, DXF, WAV, and XDR.
- IDL widgets can be used to quickly create multi-platform graphical user interfaces to your IDL programs.

- IDL programs run across all supported platforms (UNIX, Macintosh and Microsoft Windows) with little or no modification. This application portability allows you to easily support a variety of computers.
- Existing FORTRAN and C routines can be dynamically-linked into IDL to add specialized functionality. Alternatively, C and FORTRAN programs can call IDL routines as a subroutine library or display engine.

Launching IDL

To launch the IDL program, do one of the following:

On Windows platforms — Launching IDL means starting the IDL Development Environment application (no command-line mode is available under Windows). The IDL Development Environment is described in detail in [Chapter 2, “The IDL Development Environment”](#). To start IDL, double-click on the IDL icon or select IDL from the **Start** menu.

On UNIX platforms — IDL offers two interfaces:

- In *command-line mode*, IDL uses a text-only interface and sends output to your terminal screen or shell window. (Graphics are displayed in IDL graphics windows.) To start IDL in command-line mode, enter `idl` at the shell prompt.
- In *graphical mode*, IDL displays the IDL Development Environment, an X-windows application that allows you to select options from menus, work with a built-in editor, and more. The IDL Development Environment is described in detail in [Chapter 2, “The IDL Development Environment”](#). To start IDL in graphical mode, enter `idlde` at the shell prompt.

On the Macintosh MacOS X platform — IDL is launched much the same as on UNIX platforms:

Start OrobOSX by double clicking the OrobOSX icon. OrobOSX launches XDarwin and displays a UNIX X-windows command line in a MacOS X window.

- In *command-line mode*, IDL uses a text-only interface and sends output to your terminal screen or shell window. (Graphics are displayed in IDL graphics windows.) To start IDL in command-line mode, enter `idl` at the UNIX prompt.
- In *graphical mode*, IDL displays the IDL Development Environment, an X-windows application that allows you to select options from menus, work with a built-in editor, and more. The IDL Development Environment is described in detail in [Chapter 2, “The IDL Development Environment”](#). To start IDL in graphical mode, enter `idlde` at the UNIX prompt.

Launching the iTools

The IDL Intelligent Tools (iTools) are a set of interactive utilities that combine data analysis and visualization with the task of producing presentation quality graphics. The iTools are based on the IDL Object Graphics system. IDL provides several pre-

built iTools for immediate interactive use. Each of these tools is designed around a specific data or visualization type, these are:

- **iPlot** — for two and three dimensional plots (line, scatter, polar, and histogram style)
- **iSurface** — for surface representations
- **iContour** — for contour maps
- **iImage** — for image displays
- **iMap** — for map displays
- **iVolume** — for volume visualizations

To launch each iTool, simply type the name of the iTool listed previously at the Command Line of the IDL Development Editor. For more information on using the iTools, see the *iTool User's Guide*.

The iTools are built upon an object-oriented framework, or set of object classes, that serve as the building blocks for the interface and functionality of the Intelligent Tools. IDL programmers can easily use this framework to create custom iTools. For more information on creating custom iTools, see the *iTool Developer's Guide*.

Startup Options

You can specify options to the command that starts IDL. On UNIX platforms, simply append the option flag after the `idl` or `idlde` command at the shell prompt. On Windows platforms, modify the **Target** field of the properties dialog for the IDL icon to include the option flag. See “[Command Line Options for IDL Startup](#)” on page 27 for a listing of the available startup options.

Troubleshooting

When IDL is ready to accept a command, it displays the `IDL>` prompt. If IDL does not start, take the following action depending upon the operating system you are running:

- Windows: Be sure that the path listed in the **Properties** dialog for the IDL icon accurately reflects the location of the IDL executable file `idlde.exe`.
- UNIX: Be sure that your `PATH` environment variable includes the directory that contains IDL.

Environment Variables Used by IDL

When IDL starts, it sets the values of a variety of *system variables*. System variables are a special class of predefined variables that are available to all IDL program units; they are described in detail in [Appendix D, “System Variables”](#) in the *IDL Reference Guide* manual. The values of some system variables can be specified by the user when IDL starts, either via operating system *environment variables* or via *preferences* specified within the IDL Development Environment. In order to set these system variables, IDL does the following things when it starts up:

1. It checks for the presence of environment variables that correspond to the system variables. If an environment variable exists, IDL uses the value specified by the environment variable to set the value of the corresponding system variable.

Note

Note that some environment variables used by IDL do not have corresponding system variables; see below for details.

2. If the environment variable does not exist, and a preference for the system variable’s value has been set via the IDL Development Environment’s Preferences dialog, IDL uses the preference value to set the value of the corresponding system variable.
3. If neither the environment variable nor a preference for the system variable’s value exist, IDL uses a default value.

Note

In some cases, the aspects of IDL’s behavior that can be controlled by environment variables can also be controlled by setting the values of the corresponding system variables in a *startup script*. See [“Startup Files”](#) on page 30 for details.

Setting Environment Variables

The process used to set environment variables varies depending on the operating system you are using.

UNIX and MacOS X Systems

On UNIX systems, environment variables are generally specified in a file read by your shell program at startup. Syntax for setting environment variables varies depending on the shell you are using, as does the file you use to specify the variables.

If you are unsure how to set environment variables on your system, consult the system documentation or a system administrator.

For example, to set the environment variable `IDL_PATH` to the value `/usr/local/idl` when using a C shell (`csh`), you would add the following line to your `.cshrc` file:

```
setenv IDL_PATH /usr/local/idl
```

Similarly, to set the same variable when using a Bourne shell (`sh`), you would add the following line to your `.profile` file:

```
IDL_PATH="/usr/local/idl" ; export IDL_PATH
```

Microsoft Windows Systems

On Microsoft Windows systems, environment variables are set in the Environment Variables dialog, which is accessible from the System Control panel. Some Windows versions allow you to set environment variables either only for the user you logged in as (“user variables”) or for all users (“system variables”) — setting IDL environment variables as user variables means that other users who log on to the computer will not have access to your environment variable values.

Environment Variables — All Platforms

The following environment variables are checked on all platforms.

HOME

IDL uses the value of the `$HOME` environment variable when storing user-specific information in the local file system.

Note

Under Microsoft Windows, the `HOME` environment variable may not be set in all cases. If it is not set, IDL first attempts to substitute the `USERPROFILE` environment variable (which usually looks something like `C:\Documents and Settings\username` where `username` is the login name of the current user). If `USERPROFILE` is not set, IDL uses the value of the first of the following it finds: the `TEMP` environment variable, the `TMP` environment variable, the Windows system directory.

IDL_CPU_TPOOL_NTHREADS

Set this environment variable to a value greater than 0 to specify the number of threads IDL should use in thread pool computations instead of defaulting to the number of CPUs present in the underlying hardware. This defines the number of threads used by IDL when thread pool usage is not otherwise specified. Setting the CPU procedure TPOOL_NTHREADS keyword, or routine-specific thread pool keywords at the time of execution overrides this environment variable setting.

“!CPU” in Appendix D of the *IDL Reference Guide* manual provides details on the state of the system processor and of IDL’s use of it. [Chapter 15, “Multithreading in IDL”](#) in the *Building IDL Applications* manual provides information on situations when limiting the number of threads used by IDL may be beneficial.

IDL_DEVICE

Set this environment variable equal to the name of the default IDL graphics device. Setting this value is the same as setting the value of the IDL system variable !D.NAME. Note that the concept of a graphics device applies only to IDL Direct Graphics; IDL Object Graphics do not use the current graphics device. See “!D System Variable” in Appendix D of the *IDL Reference Guide* manual for details.

IDL_DIR

Set this environment variable equal to the path to the main IDL directory. Setting this value is the same as setting the value of the IDL system variable !DIR. See “!DIR” in Appendix D of the *IDL Reference Guide* manual for details.

IDL_DLM_PATH

Set this environment variable equal to the path to the directory or directories containing IDL dynamically loadable modules. At startup, IDL uses the value of this environment variable, if it exists, to initialize the IDL system variable !DLM_PATH. Due to the nature of DLMs, the value of !DLM_PATH cannot be changed after IDL has started. See “!DLM_PATH” in Appendix D of the *IDL Reference Guide* manual for details. For information on how IDL_DLM_PATH is interpreted by IDL at startup, along with syntax options for specifying the path string, see “The Path Definition String” under “EXPAND_PATH” in the *IDL Reference Guide* manual.

Note

On Windows, using the IDL_DLM_PATH environment variable is the only way to specify the path to DLMs.

IDL_HELP_PATH

Set this environment variable equal to the path to the directory or directories containing IDL help files. At startup, IDL uses the value of this environment variable, if it exists, to initialize the IDL system variable `!HELP_PATH`. See “[!HELP_PATH](#)” in Appendix D of the *IDL Reference Guide* manual for details. For information on how `IDL_HELP_PATH` is interpreted by IDL at startup, along with syntax options for specifying the path string, see “[The Path Definition String](#)” under “[EXPAND_PATH](#)” in the *IDL Reference Guide* manual.

IDL_PATH

Set this environment variable equal to the path to the directory or directories containing IDL library (`.pro` and `.sav`) files. At startup, IDL uses the value of this environment variable, if it exists, to initialize the IDL system variable `!PATH`. See “[!PATH](#)” in Appendix D of the *IDL Reference Guide* manual for details. For information on how `IDL_PATH` is interpreted by IDL at startup, along with syntax options for specifying the path string, see “[The Path Definition String](#)” under “[EXPAND_PATH](#)” in the *IDL Reference Guide* manual.

Note

If you set the `IDL_PATH` environment variable, you *must* include the token `<IDL_DEFAULT>` in the path you specify if you want IDL’s default libraries to be included in the `!PATH` system variable.

IDL_PATH_CACHE_DISABLE

Create this environment variable to disable IDL’s path caching mechanism. The existence of this variable is sufficient to disable path caching; the specific value of the variable is unimportant.

Note

It is very rarely desirable to completely disable the path caching mechanism. See “[PATH_CACHE](#)” in the *IDL Reference Guide* manual for complete details.

IDL_STARTUP

Set this environment variable equal to the path to an IDL batch file that contains a series of IDL statements which are executed each time IDL is run. See “[Startup Files](#)” on page 30 for further details.

IDL_TMPDIR

IDL, and code written in the IDL language, sometimes need to create temporary files. The location where these files should be created is highly system-dependent, and local user conventions are often different from standard practice. By default, IDL selects a reasonable location based on operating system and vendor conventions. Set the `IDL_TMPDIR` environment variable to override this choice and explicitly specify the location for temporary files.

The `GETENV` system function handles `IDL_TMPDIR` as a special case, and can be used by code written in IDL to obtain the temporary file location. See “[GETENV](#)” in the *IDL Reference Guide* manual for more information.

Environment Variables — UNIX and MacOS X

The following environment variables are used by IDL for UNIX or MacOS X.

DISPLAY

IDL uses the `DISPLAY` environment variable to choose which X display is used to display graphics.

TERM

As with any X Windows program, IDL uses the standard UNIX environment variable `TERM` to determine the type of terminal in use when IDL is in command-line mode.

LM_LICENSE_FILE

IDL’s FlexLM-based license manager uses the value of this environment variable to determine where to search for valid license files. Consult the license manager documentation for details.

Command Line Options for IDL Startup

You can alter some IDL behaviors by supplying command line switches along with the IDL command. Different switches are available on different platforms. IDL can also be started in non-interactive “batch” mode by specifying the name of a batch file at startup time. See [Chapter 10, “Executing Batch Jobs in IDL”](#) for details.

IDL for UNIX Switches

Switch	Description
<code>-vm=file</code>	Start the IDL Virtual Machine. The file argument should be an IDL <code>.sav</code> file. If no file is specified, IDL displays a file selection dialog. See Chapter 22, “Distributing IDL Applications” in the <i>Building IDL Applications</i> manual for details on creating applications that run in the IDL Virtual Machine.
<code>-rt=file</code>	Start IDL with a runtime license. The file argument should be an IDL <code>.sav</code> file. If no file is specified, IDL attempts to run a file named <code>runtime.sav</code> . See Chapter 22, “Distributing IDL Applications” in the <i>Building IDL Applications</i> manual for details on creating runtime applications.
<code>-em=file</code>	Start IDL with an embedded license. The file argument should be an IDL <code>.sav</code> file that contains an embedded (“unlimited right to distribute”) IDL license. See Chapter 22, “Distributing IDL Applications” in the <i>Building IDL Applications</i> manual for details on creating applications with an embedded IDL license.
<code>-queue</code>	For users of counted floating licenses, setting this command switch causes IDL to wait for an available license before beginning an IDL task such as batch processing.
<code>-w</code>	Start IDL with the graphical user interface. This is the same as entering <code>idlide</code> at the command prompt.
<code>-autow</code>	Start IDL with the graphical user interface if possible, otherwise start IDL in command-line mode.

Table 1-1: UNIX Command Line Switches

Switch	Description
-32	Start IDL in 32-bit mode, otherwise IDL starts in 64-bit mode by default for those platforms that support 64-bit. If you have not installed the 64-bit version, the 32-bit version will automatically be started. If you have not installed the 32-bit version, this flag will not work.
-nw	Run IDL in command-line mode no matter what. Note that specifying <code>idlde -nw</code> at the shell prompt will start IDL in command line mode.

Table 1-1: UNIX Command Line Switches

Note

In addition to the above command-line switches that control IDL's behavior, there are numerous command-line switches that control the appearance of the IDL Development Environment on Motif systems. See [“Command Line Options”](#) in Chapter 8 for details.

IDL for Windows Switches

The following switches are used with the `idlde.exe` executable. For example, use the command:

```
C:\RSI-Directory\bin\bin.x86\idlde.exe -queue
```

where *RSI-Directory* is the directory where you have installed IDL.

Switch	Description
-queue	For users of counted floating licenses, setting this command switch causes IDL to wait for an available license before beginning an IDL task such as batch processing. To set this switch, change the shortcut properties of the IDL 6.1 desktop icon so that the target line reads: <code>C:\RSI-Directory\bin\bin.x86\idlde.exe -queue</code> where <i>RSI-Directory</i> is the directory where you have installed IDL.

Table 1-2: Windows Command Line Switches

The following switches are used with the `idlrt.exe` executable. For example, use the command:

```
C:\RSI-Directory\bin\bin.x86\idlrt.exe -vm=file.sav
```

where *RSI-Directory* is the directory where you have installed IDL.

Switch	Description
<code>-vm=file</code>	Start the IDL Virtual Machine. The file argument should be an IDL <code>.sav</code> file. If no file is specified, IDL displays a file selection dialog. See Chapter 22, “Distributing IDL Applications” in the <i>Building IDL Applications</i> manual for details on creating applications that run in the IDL Virtual Machine.
<code>-em=file</code>	Start IDL with an embedded license. The file argument should be an IDL <code>.sav</code> file that contains an embedded (“unlimited right to distribute”) IDL license. See Chapter 22, “Distributing IDL Applications” in the <i>Building IDL Applications</i> manual for details on creating applications with an embedded IDL license.

Table 1-3: Windows Command Line Switches

Startup Files

A *startup file* is a batch file that is executed automatically each time the IDL is started. You can specify a startup file in either of two ways:

- By specifying the name of the startup file in the **Startup File** field of the **Startup** tab of the IDL Development Environment Preferences dialog.
- By specifying the name of the startup file as the value of the `IDL_STARTUP` environment variable. Note that this method works whether you use IDL in command-line mode or via the IDL Development Environment. If you use the IDL Development Environment and you set both the environment variable *and* the **Startup File** preference, IDL will use the value of the environment variable.

Common uses for startup files include the following:

- Restoring variable data contained in a `.sav` file or reading in commonly used data
- Setting common keywords to the `DEVICE` procedure
- Setting up system graphic preferences
- Specifying shared or private color maps for PseudoColor devices

Startup files are executed one statement at a time. It is not possible to define program modules (procedures, functions, or main-level programs) in the startup file. For more information on creating batch files, see [Chapter 10, “Executing Batch Jobs in IDL”](#).

Message of the Day Files

When IDL starts, it displays the contents of the `motd.txt` file, located in the `help/motd` subdirectory of the IDL distribution, in the Output Log. You can use this *Message of the Day* file to provide information to IDL users every time IDL starts.

In addition, IDL will display the contents a file with the name `platform.txt` located in the `help/motd` subdirectory of the IDL distribution, where *platform* is a string corresponding to the current operating system platform. For example, on Microsoft Windows systems, IDL displays a file named `win32.txt`.

You can determine the correct name for this file on a given platform by using the following IDL command:

```
PRINT, !VERSION.OS
```

and appending the “.txt” extension to the operating system name.

If you do not want to see either the `motd.txt` file or the platform-specific file each time IDL starts, remove them from the `help/motd` subdirectory of the IDL distribution.

Note

The `motd.txt` and platform-specific files are simply an ASCII text files—not IDL programs or batch files. To execute a series of IDL commands, select a startup file as described in “[Startup Files](#)” on page 30.

Using Your Mouse with IDL

IDL supports the use of mice with up to three buttons. Because some systems use mice with one or two buttons, IDL provides mechanisms for simulating a three-button mouse.

Using a Two-Button Mouse

Many mice used with Microsoft Windows systems have only two buttons. To simulate a middle-button press, hold down the `CONTROL` key and press the left mouse button.

Using a Macintosh (One-Button) Mouse

Many mice used with Macintosh systems have only one button. The X Window System software provided with MacOS X provides multi-button mouse emulation, allowing you to configure the system to generate middle- and right-button press events. See your MacOS X system documentation for details.

Using Keyboard Accelerators

IDL supports the use of keyboard *accelerators* or *shortcuts* in two different contexts: in the IDL Development Environment and in IDL widget applications.

Keyboard shortcuts that are active in the IDL Development Environment are described in “[Keyboard Shortcuts](#)” on page 69.

Keyboard shortcuts can also be defined for individual buttons and menu items in an IDL widget application. Defining shortcut key combinations is the responsibility of the IDL programmer who creates the widget application; if you are using a widget application and are unsure about whether keyboard shortcuts have been defined, contact the author of the widget application.


For information on adding keyboard accelerators to your own widget applications, see “[Enhancing Widget Application Usability](#)” in Chapter 30 of the *Building IDL Applications* manual.

Enabling Alt Key Accelerators on Macintosh

If you are using IDL on a Macintosh and wish to use keyboard accelerators that use the **Alt** key, you will need to perform the following steps to make the **Apple** (Command) key to function as the **Alt** key:

1. Create a `.xmodmap` file in your home folder and add the following three lines to it:

```
clear mod1
clear mod2
add mod1 = Meta_L
```

When Apple’s X11 program starts, this file will automatically be read, and the Apple key will be mapped to the left meta key , which for IDL’s purposes is the **Alt** key. (Windows **Alt** key accelerators are mapped to the Macintosh **Apple** key, not the **Option (alt)** key.)

2. Run Apple’s X11 program and change its preferences. Under **Input** in the X11 Preferences dialog, make sure that the following two items are *unchecked*:
 - Follow system keyboard layout
 - Enable key equivalents under X11

Note

You must relaunch Apple’s X11 program for these changes to take effect.

Once you have performed these steps, keyboard shortcuts will operate in the normal Macintosh fashion — namely, pressing the **Apple** key in conjunction with X, C, and V will perform cut, copy and paste. The IDLDE's other shortcuts and any widget accelerators defined to use the **Alt** key will also work.

Quitting IDL

To quit IDL, do one of the following:

- Enter the `EXIT` command at the IDL command prompt.
- If you are running the IDL Development Environment (IDLDE), select the **Exit** option from the **File** menu.
- Under Microsoft Windows, press **Alt+F4**.
- Under UNIX or MacOS X, if you use IDL's command-line mode, press **Ctrl+D** as the first character in command-line mode causes IDL to exit back to the operating system. The `EXIT` procedure has the same function. If **Ctrl+D** is not the first character, it simply ends the input line as if a return had been entered.

Note

When using IDL's command-line mode under UNIX or MacOS X, you can normally press **Ctrl+Z** to suspend IDL and return you to the shell process without exiting IDL. After completing any shell commands, type `fg` to return IDL to the foreground. Although the UNIX suspend character can be changed by the user outside of IDL, this is rarely done. For the purposes of this manual, we assume the default convention.

Helpful Resources

There are many tools along the way which will help you learn and use IDL. You will find answers to questions here whether you are simply looking-up the use for a common routine in the IDL Help System or understanding this manual's use of conventions.

The IDL Help System

On each platform, the IDL Help System can be accessed directly through IDL to give you information. To access help in IDL, do one of the following:

- Enter the ? command at the IDL command prompt.
- If you are running the IDL Development Environment (IDLDE), select the **Help** option from the Menu Bar.
- In IDL for Windows, press **Ctrl+F1**

Reporting Problems

We strive to make IDL as reliable and bug free as possible. However, no program with the size and complexity of IDL is perfect, and problems do surface. When you encounter a problem with IDL, the manner in which you report it has a large bearing on how well and quickly we can fix it.

The relnotes.txt file accompanying each release includes information about new features in that release, bug fixes, and known problems which may be of help.

This section is intended to help you report problems in a way which helps us to address the problem rapidly.

Background Information

Sometimes, a problem only occurs when running on a certain machine, operating system, or graphics device. For these reasons, we need to know the following facts when you report a problem:

- Your IDL installation number.
- The version of IDL you are running.
- The type of machine on which it is running.
- The operating system version it is running under.

- The type and version of your windowing system if you are on UNIX.
- The graphics device, if the problem involves graphics and you know what graphics device is on your system.

The installation number is assigned by us when you purchase IDL and is included in the license information that we sent you. The IDL version, site number, and type of machine are printed when IDL is started.

For example, the following startup announcement appears indicating you are running IDL version 5.6 under Sun Solaris using installation number xxxxx-x, under a floating license located on a particular license manager.

```
IDL Version 5.6, Solaris (sunos sparc m64).  
(c) 2002, Research Systems, Inc.  
Installation number: xxxxx-x.  
Licensed for use by: RSI IDL floating licenses
```

Under UNIX, the version of the operating system can usually be found in the file `/etc/motd`. It is also printed when the machine boots. In any event, your system administrator should know this information.

Under Windows, select **About** from the **Help** menu in the Windows Explorer.

Double Check

Before reporting a problem double check with the manual or a local expert if one is available. Sometimes, it is a simple matter of misinterpreting what is supposed to happen.

If you cannot determine what should happen in a given situation by consulting the reference manual, the manual needs to be improved on that topic. Please let us know if you feel that the manual was vague or unclear on a subject.

Another question to ask is whether the problem lies within IDL, or with the system running IDL. Is your system properly configured with enough virtual memory and sufficient operating system quotas? Does the system seem stable and is everything else working normally?

Describing The Problem

When describing the problem, it is important to use precise language. Terms like crashes, blows up, and fails are vague and open to interpretation. Does it really crash IDL and leave you looking at an operating system prompt? This is how RSI technical support personnel interpret a problem report of a crash. If the behavior being reported refers to an unexpected error message being issued before returning another prompt,

then describing it as a crash becomes misleading. What is really meant by a term like "fails?"

It is also important to separate concrete facts from conjecture about underlying causes. For example, a statement such as "IDL dumps core when allocating dynamic memory" is not nearly as useful as this statement, "IDL dumps core when I execute the following statements... "

Reproducibility

Intermittent problems are by far the hardest kind to fix. In general, if we can't make it happen on our machine, we can't fix it. It is far more likely that we can help you if you can tell us a sequence of IDL statements that cause the problem to happen. Naturally, there are degrees of reproducibility. Situations where a certain sequence of statements causes the problem 1 time in 3 tries are fairly likely to be fixable. Situations where the problem happens once every few months and no one is sure what triggered it are nearly impossible to identify and correct.

Simplify the Problem

In accordance with RSI Technical Support policy, when reporting a problem, it is important to give us the shortest possible series of IDL statements that cause it. Here are some suggestions for simplifying your problem:

Copy the procedure and function files that are involved to a scratch second copy. Never modify your only copy!

Eliminate everything not involved in demonstrating the problem. Don't do this all at once. Instead, do it in a series of slow careful steps. Between each step, stop and run IDL on the result to ensure that the problem still appears.

If a simplification causes the problem to disappear, then slowly restore the statements involved until you can identify the source of the problem. The end result of such simplification should be a small number of IDL statements that demonstrate the problem.

If the problem does not involve file Input/Output, strive to eliminate all file I/O statements. Use IDL routines to generate a dummy data set, rather than including your own data if at all possible. If your problem report does not involve your data, it will be much easier for us to reproduce.

On the other hand, if the problem involves file Input/Output, and the problem only happens with a certain data file or type of data, we will need to look at your data or a sample of your data.

If it is necessary to send us your data, use one of the following methods:

- If the data set is small, please send it as an attachment in your email to us: support@RSInc.com.
- If the data set is large, please place it on our ftp site at: <ftp.RSInc.com/incoming>.

Be sure to include the commands that reproduce your problem in your message to use. If you have placed your data on the ftp site, include the name of the data set and when it was uploaded.

Problems with Dynamic Loading

Under some operating systems, the `CALL_EXTERNAL` and `LINKIMAGE` system routines allow you to dynamically load routines written in other languages into IDL. This is a very powerful technique for extending IDL, but it is considerably more difficult than simply writing IDL statements. At this level, the programmer is outside the user level shell of IDL and is not protected from programming errors. These errors could give incorrect results or crash IDL. In such situations, the burden of proving that a problem is within IDL and not the dynamically loaded code is entirely the programmer's.

Although it is certainly true that a problem in this situation can be within IDL, it is very important that you exhaust all other possibilities before reporting the problem. If you decide that you need to report the problem, the comments above on simplifying things are even more important than usual. If you send us a small example that exhibits the problem, we may be able to respond with a correction or advice.

Contact Us

To report a problem, contact us at the following addresses:

Electronic Mail

support@RSInc.com

Telephone

(303) 786-9900

(303) 786-9909 (Fax)

(303) 413-3920 (IDL technical support direct line)

Mail

Research Systems, Inc.

4990 Pearl East Circle

Boulder, CO 80301

Web Site

<http://www.RSInc.com>

Typographical Conventions

The following typographical conventions are used throughout this manual and the entire IDL documentation set:

- **UPPER CASE type**
IDL functions and procedures, and their keywords are displayed in UPPER CASE type. For example, the calling sequence for an IDL procedure looks like this:

```
CONTOUR, Z [, X, Y]
```

- **Mixed Case type**
IDL object class and method names are displayed in Mixed Case type. For example, the calling sequence to create an object and call a method looks like this:

```
object = OBJ_NEW('IDLgrPlot')
object -> GetProperty, ALL=properties
```

- ***Italic type***
Arguments to IDL procedures and functions — data or variables you must provide — are displayed in italic type. In the above example, *Z*, *X*, and *Y* are all arguments.
- **Square brackets ([])**
Square brackets used in calling sequences indicate that the enclosed arguments are optional. Do not type the brackets. In the above CONTOUR example, *X* and *Y* are optional arguments. Square brackets are also used to specify array elements.
- **Courier type**
In examples or program listings, things that you must enter at the command line or in a file are displayed in courier type. Results or data that IDL displays on your computer screen are shown in **courier bold** type. An example might direct you to enter the following at the IDL command prompt:

```
array = INDGEN(5)
PRINT, array
```

In this case, the results are shown like this:

```
0      1      2      3      4
```




Chapter 2:

The IDL Development Environment

This chapter describes the IDL Development Environment.

Components of the IDLDE	42	Project Menu	62
File Menu	49	Macros Menu	63
Edit Menu	53	Window Menu	65
Search Menu	55	Help Menu	68
Run Menu	57	Keyboard Shortcuts	69

Components of the IDLDE

The IDL Development Environment (IDLDE) is a convenient multiple-document graphical user interface that includes built-in editing and debugging tools. This section describes briefly the components of the IDLDE. The Windows version is shown on the left and the UNIX version is shown on the right within the following figure.

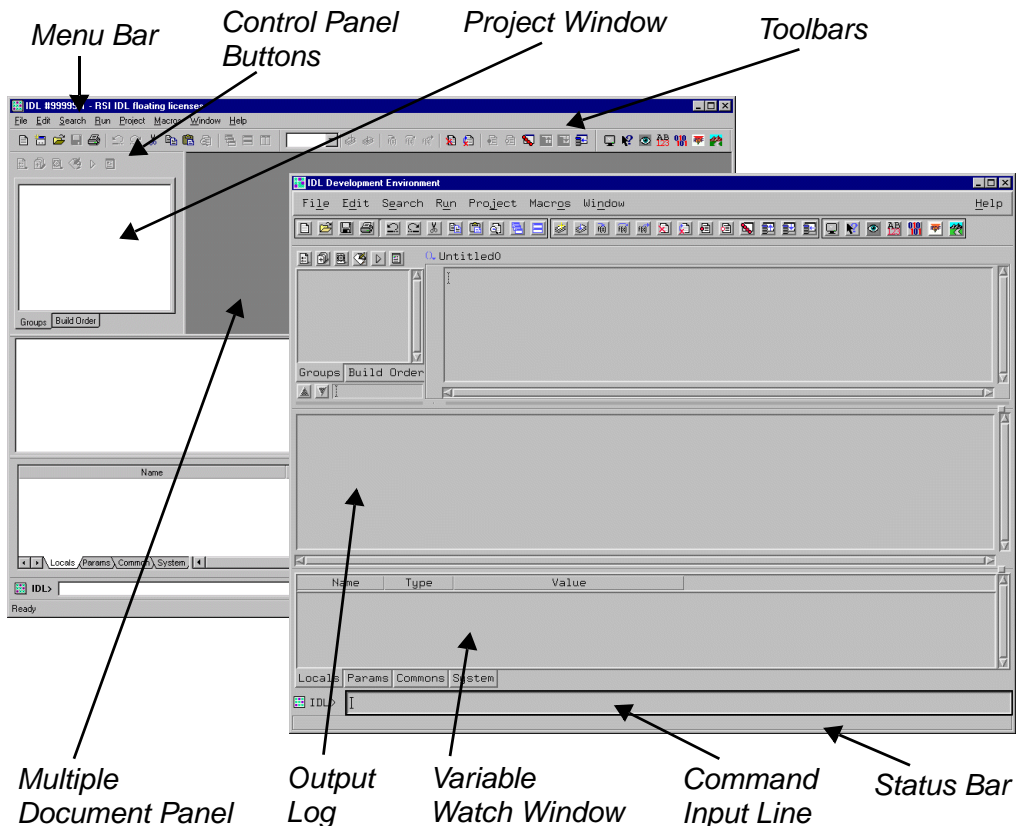


Figure 2-1: The IDL Development Environment for Windows (left) and UNIX (right).

Note

Individual components are similar across the two platforms.

Menu Bar

The menu bar, located at the top of the main IDLDE window, allows you to control various IDLDE features. When you select an option from a menu item in the IDLDE, the Status Bar displays a brief description.

The menu bar consists of the following menu items:

Menu Item	Description of Functions
File Menu	The File Menu gives you options such as opening, closing and creating new Editor windows and Projects and other options such as printing, printer setup, preferences and exiting IDL.
Edit Menu	The Edit Menu provides edit-related options such as undo, redo, cut, copy, paste, delete, select all, clear all and clear log.
Search Menu	The Search Menu allows you to find text in currently active Editor windows as well as other options such as find again, find selection, enter selection, replace, replace & find, go to line and go to definition.
Run Menu	Run Menu items are enabled when an IDL program is loaded into an IDL Editor window. The run menu allows you program related functionality such as compiling, resolving dependencies, resetting, and editing programs among other things. For more information on running programs in IDL, see Chapter 9, “Creating and Running Programs in IDL” .
Project Menu	The Project Menu provides project-related functionality such as adding/removing files, grouping and moving files, building, running and exporting projects and so on. For more information on working with IDL projects, see Chapter 21, “Creating IDL Projects” in the <i>Building IDL Applications</i> manual.
Macros Menu	The Macro Menu provides functionality for creating new macros and using existing macros in IDL. For more about working with macros in IDL, see Chapter 6, “Working With Macros” .

Table 2-1: The IDL Menus

Menu Item	Description of Functions
Window Menu	The Window Menu gives functionality related to Multiple Document Panel windows.
Help Menu	The Help Menu allows you to call IDL Online Help. You can call the entire Online Help system in the IDL Online Help Viewer or find help by topic. For more information on the IDL Help System, see “ Helpful Resources ” on page 36.

Table 2-1: The IDL Menus (Continued)

You can display menu commands for each menu using the following methods:

- Clicking the menu on the Menu bar.
- Pressing the **ALT** key plus the underlined letter in the menu’s title. For example, to display the File menu, press **ALT+F**.

You can select or execute a menu command using the following methods:

- Clicking the item in the menu.
- Pressing the **ALT** key plus the underlined letter in the menu’s title, and then pressing the letter underlined in the menu item. For example, to select the menu item **File** → **Open**, press **ALT+F+O**.
- Using the cursor and the arrow keys to highlight a menu item, and then pressing the **Enter** key.

Note

Many items (on each platform) have keyboard shortcuts displayed to the right of the corresponding menu option.

Toolbars

You can choose any combination of three toolbars: **Standard**, **Run & Debug**, and **Macros**. To change the toolbars displayed, use the **Windows** menu to access the **Toolbar** pulldown menu and select or de-select any combination of the three toolbars. In addition in IDL for Windows, when you open a GUIBuilder window, its associated toolbar is displayed.

When you position the mouse pointer over a toolbar button, the Status Bar displays a brief description. If you click on a toolbar button which represents an IDL command, the IDL command issued is displayed in the Output Log.

Project Window

On all platforms, the IDL Project Window allows you to manage, compile, run, and create distributions of all the files needed to develop an IDL application. All of your application files can be organized for ease of access, and to be easier to export to other developers, colleagues, or users. For further information on the Projects Window, refer to [Chapter 21, “Creating IDL Projects”](#) in the *Building IDL Applications* manual in *Building IDL Applications*.

Multiple Document Panel

On all platforms, the section of the main IDL window where IDL Editor windows are displayed is known as the multiple document panel.

GUIBuilder Windows

Under Microsoft Windows, IDL GUIBuilder windows allow you to interactively create user interfaces. Then, you can generate the IDL code that defines the interface and the code to contain the event-handling routines. You can modify the code, compile, and run the application in the IDLDE.

You can have any number of GUIBuilder windows open simultaneously.

To open a GUIBuilder window, you can select **New** then **GUI** from the **File** menu, or you can select **Open** from the **File** menu. You can also open GUIBuilder windows using the toolbar buttons.

When you minimize a GUIBuilder window, a Windows title bar with the name of the file appears in the Multiple Document Panel.

For information about the GUIBuilder, see [Chapter 27, “Using the IDL GUIBuilder”](#) in the *Building IDL Applications* manual.

Editor Windows

IDL Editor windows allow you to write and edit IDL programs (and other text files) from within IDL. Any number of Editor windows can exist simultaneously. No Editor windows are open when IDL is first started. Editor windows can be created by selecting **File** → **New** or **File** → **Open**.

You can access different files from the **Windows** menu by clicking on the appropriate numbered file. See [“Creating a Simple Program”](#) in Chapter 10 of the *Building IDL Applications* manual for more information on the IDL Editor.

If you click the right mouse button while positioned over an editor window, a popup menu appears allowing you to quickly access several of the most convenient commands. The popup menu changes to display common debugging commands if IDL is running a program.

If a program error or breakpoint is encountered, IDLDE displays the relevant file, opening it if necessary. The line at which the breakpoint or error occurred is marked. See [Chapter 18, “Debugging an IDL Program”](#) in the *Building IDL Applications* manual for more on IDL’s debugging commands.

Graphics Windows

IDL Graphics windows appear when you use IDL to plot or display data.

You can copy the contents of a Graphics window—Direct or Object—directly to the operating system clipboard in a bitmap format using **CTRL+C**.

When an IDL Graphics window is minimized (iconized), the icon displays the name of the IDL window. This icon appears on the desktop, not in the Multiple Document Panel, as with an iconized Editor window.

Warning

If the backing store is not set when a window is iconized, it will not be refreshed upon return. For more information about setting the backing store for graphics windows, see [“Graphics Preferences”](#) on page 103.

Output Log

On all platforms, output from IDL is displayed in the Output Log window, which appears by default when IDLDE is first started. Only one Output Log window can exist at a time.

In IDL for Windows, if you click the right mouse button while positioned over the Output Log, a popup menu appears allowing you to move to a specified error. Clear the contents of the Output Log, or copy selected contents.

In IDL for UNIX, if you click the right mouse button while positioned over the Output Log, a popup menu appears allowing you to move to a specified error or clear the contents of the Output Log.

Variable Watch Window

The Variable Watch Window appears by default when you start the IDLDE. It keeps track of variables as they appear and change during program execution. For more information about the Variable Watch Window, see “[The Variable Watch Window](#)” in Chapter 18 of the *Building IDL Applications* manual.

Command Input Line

The Command Input Line is an IDL prompt where you can enter IDL commands. The text output by IDL commands is displayed in the Output Log window.

If you click the right mouse button while positioned over the Command Input Line, a popup menu appears displaying up to 20 commands from the command recall buffer.

Select an entry in the popup menu to repeat the command. See “[Command Recall and Line Editing](#)” in Chapter 4 for additional information about the command recall buffer.

Status Bar

When you position the mouse pointer over a **Control Panel** button (**Toolbar** button in IDL for Windows) or select an option from a menu in IDLDE, the Status Bar displays a brief description.

Docking/Undocking

In IDL for Windows, four sections of the IDLDE can be moved within and unanchored from the main IDLDE window: the Toolbars, Output Log, Variable Watch Window, and Command Input Line. Click on the border and drag the left mouse button. You will notice the outline of the chosen section moving with your mouse. When a location is chosen, release the mouse button to dock the window. If you move this outline so that it overlaps an edge of the window space being used by the IDLDE, the section will be docked to the nearest available side of the main IDLDE window. The Toolbars, Output Log, Variable Watch Window, and Command Input Line will remain between the Menu Bar and the Status Bar when docked. They can be docked in any order to an edge. If the outline doesn't overlap an edge, the section will float on the desktop. If you hold down the [**Ctrl**] key, the sections will float instead of docking to the nearest available side of the IDLDE.

Control Panel Buttons

In IDL for UNIX, the Control Panel buttons issue IDL commands for the currently-selected Editor window when pressed. The IDL command issued is displayed in the Output Log. By default, there are three different toolbars and the buttons displayed as well as the commands they issue are completely configurable (see [Chapter 5, “Setting IDL Preferences”](#) for more on these toolbars). When you position the mouse pointer over a Control Panel Button, the Status Bar displays a brief description.

File Menu

New

Select from the following sub-menu items:

- **Editor [CTRL+N]**: Opens a new IDL Editor window.
- **GUI (Microsoft Windows Only)**: opens a new IDL GUIBuilder file. For information about the IDL GUIBuilder, see [Chapter 27, “Using the IDL GUIBuilder”](#) in the *Building IDL Applications* manual.
- **Project...**: opens the **New Project** dialog.
- **Visualization**: Launches an iTool. See [“Introducing the iTools”](#) in Chapter 1 of the *iTool User’s Guide* manual for more on iTools.

Each window is titled `Untitledn` or `UntitledPrncn` (where n is the numerical index of the new editor window or GUIBuilder file) until saved with another name.

Open... [CTRL+O]

Select this option to open a text file for editing. (On Microsoft Windows platforms, you can also select an IDL GUIBuilder `*.prc` portable resource file.) The **Open** dialog appears. Select the file you want to open or type the file name and click **Open** (Windows) or **OK** (Motif). You can select a continuous range of files by holding down the **Shift** key after selecting the first file, or select multiple separated files by selecting each file while holding down the **Control** key. A new IDL Editor window is created to contain each text file.

Note

On Motif platforms, if the **Multiple Windows** option is selected, a new IDL Editor window is created outside the main window to contain each text file. [“Layout Preferences”](#) on page 99 for details.

You can also open text files from the Command Input Line. To open text files, enter the following at the IDL prompt:

```
.EDIT file1 [file2 ... filen]
```

where `file` is the name of the text file you want to open. If the path is not specified in the Path Preferences from the File menu, you must enter the full path for file. See [“.EDIT”](#) in the *IDL Reference Guide* manual for more information.

Close

Select this option to close the currently-selected IDL Editor window. If you have made changes in an IDL Editor window, you are asked if you want to save the changes before closing the window.

Open Project...

Select this option to open a new IDL Project. The **Open** dialog appears. Select the project you want to open and click **Open**.

Save Project

Select this option to save the current IDL Project. If the Project has not yet been saved, you are prompted for a filename with the **Save As** dialog.

Save Project As...

Select this option to save the current IDL Project to a specified filename. The **Save As** dialog appears.

Close Project

Select this option to close the current IDL Project. If you have made changes in to the project, you are asked if you want to save the changes before closing the window.

Save [CTRL+S]

Select this option to save the contents of an IDL Editor window. If the file has not yet been saved, you are prompted for a filename with the **Save As** dialog.

Note

Changes made to a previously-compiled routine are not available to IDL until that routine is re-compiled. Executing the routine without first saving and re-compiling simply re-runs the previously-compiled version, without incorporating recent changes.

Select the **Compile** option in the **Run** menu to return to the main program level and re-compile the routine. Select **Compile from Memory** in the **Run** menu to save and compile recent changes to a temporary file.

Save As...

Select this option to save the contents of an IDL Editor window to a specified filename. The **Save As** dialog appears. (On Motif platforms, you can select this

option with the keyboard shortcut **Ctrl+W**.) On Windows, when the **File** → **Save As...** option is selected, the default file name is the name of the last procedure or function in the file. On UNIX, the default file name is `*.pro`. For portability between platforms, the filename will be all lowercase letters.

Revert to Saved

Select this option to reload the last saved version of the document.

Warning

Unsaved changes are lost without warning.

Generate .pro (Microsoft Windows Only)

On a Microsoft Windows system, select this option to generate source code files from GUIBuilder interface definitions. When you generate code for the first time, all options open the **Save As** dialog so that you can select a location and specify a filename. The following are generated:

- The widget definition code to a `*.pro` file.
- The event-handler callback code to a `*_eventcb.pro` file.

For information about the IDL GUIBuilder generated code, see “[Generating Files](#)” in Chapter 27 of the *Building IDL Applications* manual.

Print... [CTRL+P]

On Microsoft Windows systems, select this option to print the contents of the currently-selected window to the default printer immediately. On Motif systems, the Print dialog appears.

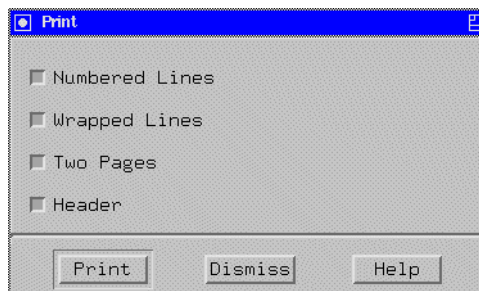


Figure 2-2: The Motif Print Dialog.

Select **Numbered Lines** to include line numbers in the printout. Select **Wrapped Lines** to cause lines longer than the width of the printed page to wrap to a new line. Select **Two Pages** to print two pages per sheet of paper (each logical page is printed at half normal size). Select **Header** to include file information at the top of each page.

Print Setup...

Select this option to change the printer and printing options. The Print (Windows) or Printer Setup (Motif) dialog appears. For further information on setting up a printer, see [Chapter 7, “Printing in IDL”](#).

Recent Files

Select this option to open recently opened or created files. This menu item lists the last ten opened or created files. (On Microsoft Windows systems, it includes both text and GUIBuilder files.) To open a file on this list, select it.

On Motif systems, to change the maximum number of files displayed from ten to another number, modify the `idlde.numRecentFiles` resource in your `.idlde` resource file. See [Chapter 8, “Customizing IDL on Motif Systems”](#), for details.

Recent Projects

Select this option to open recently opened project files.

Preferences...

Select this option to display the tabbed Preferences dialog, which allows you to customize your interaction with the IDLDE environment. The options available via the Preferences dialog are described in detail in [Chapter 5, “Setting IDL Preferences”](#).

Exit [CTRL+Q]

Select this option to exit IDL.

Edit Menu

Undo [CTRL+Z (Windows), ALT+Z (Motif)]

Select this option to undo previous editing actions. Multiple undo operations are supported; the first reverses the most recent operation, the next reverses the second most recent operation, etc. If the most recent action is irreversible, this option will not be accessible.

Redo [CTRL+Y (Windows), ALT+Y (Motif)]

Select this option to redo previously undone editing actions. Multiple redo operations are supported; the first redo reverses the most recent undo, etc.

Cut [CTRL+X (Windows), ALT+X (Motif)]

Select this option to remove currently-selected text from an IDL Editor window or the Command Input Line to the Windows clipboard.

Copy [CTRL+C (Windows), ALT+C (Motif)]

Select this option to copy the currently-selected text in an IDL Editor window, Output Log window, or Command Input Line to the clipboard. **Copy** also allows you to copy graphics from an IDL graphics window or draw widget to the clipboard.

Paste [CTRL+V (Windows), ALT+V (Motif)]

Select this option to paste the contents of the Windows clipboard at the current insertion point. The insertion point can only be placed in an IDL Editor window.

Delete [DEL]

Select this option to delete the currently-selected text. The deleted text is not placed on the clipboard.

Select All

Use this option to highlight the entire contents of an IDL Editor window.

Clear All [CTRL+DEL (Windows)]

Use this option to clear the entire contents of the current IDL Editor window.

Clear Log [CTRL+Y (Motif)]

Use this option to clear the entire contents of the Output Log.

Properties (Microsoft Windows Only)

Select this option to open the GUIBuilder Properties dialog, which you can use to set the attribute and event properties for a widget.

For information on the Properties dialog, see [“Using the Properties Dialog”](#) in Chapter 27 of the *Building IDL Applications* manual.

Menu (Microsoft Windows Only)

Select this option to open the GUIBuilder Menu Editor, which you can use to define menus for top-level base widgets and button widgets.

For information on the Menu Editor, see [“Using the Menu Editor”](#) in Chapter 27 of the *Building IDL Applications* manual.

Search Menu

Find... [CTRL+F (Windows), ALT+F (Motif)]

Select this option to find text in an IDL Editor window or windows. The Search or Find/Replace dialog appears.

Enter the text to find in the field marked **Search for** or **Find**; click **Find next** to highlight the search text in the currently active file.

Platform Differences

- On Windows platforms, you can also choose an entry from the pulldown list of recent search terms rather than entering a new term in the **Search for** field.
- On Windows platforms, you can specify replacement text by checking the **Replace with** checkbox and entering a replacement term. Click **Replace** to replace the selected text.

Check the **Case sensitive** checkbox to match the case of the text you enter. Check **Whole words only** to match only entire words (the default is to match sub-strings). To replace all instances of the search text, check the **Replace all** checkbox and click **Replace**. Select **Forward from cursor** or **Backward from cursor** to specify the direction in which you would like to begin the search, or **Entire file** to search from the beginning of the file.

By default, the search will take place in the currently-selected window. Choose a different file or **All Windows** from the pulldown list marked **Search in file** to search other windows.

Find Again [F3 (Windows), ALT+G (Motif)]

Select this option to repeat the previous Find operation.

Find Selection [CTRL+E (Windows), ALT+I (Motif)]

Select this option to find the next occurrence of the selected text in an IDL Editor window.

Enter Selection [ALT+T (Motif)]

Select this option to enter selected text in the **Find** field of the Find/Replace dialog.

Replace... [CTRL+H (Windows), ALT+R (Motif)]

Select this option to find text in an IDL Editor window and replace it with new text. The Replace dialog box appears. The Replace dialog has the same controls as the Search dialog, described above in the Find item. By default, the **Replace with** checkbox is checked.

Replace & Find [ALT+P (Motif)]

Select this option to repeat the most recent search-and-replace operation.

Replace Again [SHIFT+F3]

Select this option to repeat the previous Replace operation.

Go To Line... [CTRL+G]

Select this option to jump directly to the specified line number in an IDL Editor window. The Go To Line dialog appears.

Go To Definition [CTRL+D (Windows), CTRL+T (Motif)]

Use this option to go to and mark with a current line indicator (blue arrow) the procedure or function call of the item next to which the cursor is positioned. The item must be either user-defined or a procedure or function written in IDL, and must have been compiled during the current IDLDE session.

Run Menu

Run Menu items are enabled when an IDL program is loaded into an IDL Editor window and compiled. If you click the right mouse button while positioned over an editor window, a popup menu appears allowing you to quickly access several of the most convenient commands. The popup menu changes to display common debugging commands if IDL is running a program. See [Chapter 18, “Debugging an IDL Program”](#) in the *Building IDL Applications* manual for more information.

Compile *filename.pro* [CTRL+F5]

Select this option to compile a `.pro` file. The currently-selected file is only recognized as an IDL procedure or function if suffixed with `.pro`. Selecting this option is the same as entering `.COMPILE` at the Command Input Line, with the appropriate Editor window selected in the Multiple Document Panel.

You can also compile files from the Command Input Line. Enter the following at the IDL prompt:

```
.COMPILE file1 [file2 ... fileN]
```

where `file` is the name of the file you want to open. IDL opens your files in editor windows and compiles the procedures and functions contained therein. If the path is not specified in the **Path Preferences** from the **File** menu, you must enter the full path for file.

See [“.COMPILE”](#) in the *IDL Reference Guide* manual for a more detailed explanation.

Compile *filename.pro* from Memory [CTRL+F6]

Select this option to save and compile changes to the current editor window without affecting the last-saved version of the file. The temporary file created allows you to experiment without committing changes to the permanent file. Selecting this option is the same as entering `.COMPILE -f` at the Command Input Line. See [“.COMPILE”](#) in the *IDL Reference Guide* manual for a more detailed explanation.

Compile All

Select this option to compile all currently open `*.pro` files.

Run *filename* [F5]

Select this option to execute the file called `filename` contained in the currently-active editor window. Selecting this option is the same as entering the procedure

name at the Command Input Line or using the `.GO` executive command at the Command Input Line. If the file is interrupted while running, selecting this option resumes execution; it is the same as entering `.CONTINUE` at the Command Input Line. For more information, see `.CONTINUE` and `.GO` in the *IDL Reference Guide*.

Warning

In order for the **Run** option to reflect the correct procedure name in the **Run** menu, the `.pro` filename must be the same as the main procedure name. For example, the file named `squish.pro` must include:

```
pro squish
```

Resolve Dependencies [ALT+F5 (Motif)]

Select this option to iteratively compile all un-compiled IDL routines that are referenced in any open and compiled files. Selecting this option is the same as entering `RESOLVE_ALL, /QUIET` at the Command Input Line. The `QUIET` keyword suppresses informational messages. See “[RESOLVE_ALL](#)” in the *IDL Reference Guide* manual for a more detailed explanation.

Profile

Select this option to access the Profile dialog. The IDL Code Profiler allows you to analyze the performance of your applications. You can identify which modules are used most frequently, and which modules take up the greatest amount of time. For more information about the IDL Code Profiler, see “[The IDL Code Profiler](#)” in Chapter 14 of the *Building IDL Applications* manual.

Test GUI [CTRL+T (Microsoft Windows Only)]

Select this option to test the GUI interface in a GUIBuilder window. This option allows you to see how the interface you have designed will look when it is running.

To exit test mode:

Press the **Esc** key.

or

Click the **X** in the upper-right corner of the application window of the running test application.

Note

This option is not available if a blocking widget is currently active.

Break [CTRL+BREAK (Windows), CTRL+C (Motif)]

Select this option to interrupt program execution. IDL inserts a marker to the left of the line at which program execution was interrupted.

Stop [CTRL+R]

Select this option to stop program execution and return to the main program level. Selecting this item is the same as entering the following at the Command Input Line:

```
RETALL
WIDGET_CONTROL, /RESET
CLOSE, /ALL
HEAP_GC, /VERBOSE
```

See [RETALL](#), [WIDGET_CONTROL](#), [CLOSE](#), or [HEAP_GC](#) in the *IDL Reference Guide* for more detailed explanations.

Reset

Select this option to completely reset the IDL environment. This option executes `.RESET_SESSION`. See `“.RESET_SESSION”` in the *IDL Reference Guide* manual for more information.

Step Into [F8]

Select this option to execute a single statement in the current program. The current-line indicator advances one statement. If the statement being stepped into calls another IDL procedure or function, statements from that procedure or function are executed in order by successive Step commands. Selecting this item is the same as entering `.STEP` at the IDL Command Input Line. See `“.STEP”` in the *IDL Reference Guide* manual for a more detailed explanation.

Step Over [F10]

Select this option to execute a single statement in the current program. The current-line indicator advances one statement. If the statement which is stepped over calls another IDL procedure or function, statements from that procedure or function are executed to the end without interactive capability. Selecting this item is the same as entering `.STEPOVER` at the IDL Command Input Line. See `“.STEPOVER”` in the *IDL Reference Guide* manual for a more detailed explanation.

Step Out [CTRL+F8]

Select this option to continue processing until the current program returns. Selecting this item is the same as entering `.OUT` at the IDL Command Input Line. See “.OUT” in the *IDL Reference Guide* manual for a more detailed explanation.

Trace...

Select this option to access the Trace Execution dialog. You can modify the interval between successive `.STEP` or `.STEPOVER` commands, depending on whether **Step into routines** or **Step over routines** is checked. The current-line indicator points to program lines as they are executed. Selecting this item at full speed is the same as entering `.TRACE` at the IDL command prompt. See “.TRACE” in the *IDL Reference Guide* manual for a more detailed explanation.

Run to Cursor [F7]

Select this option to execute statements in the current program up to the line where the cursor is positioned. Selecting this item is the same as setting a one-time breakpoint at a specific line. See “BREAKPOINT” in the *IDL Reference Guide* manual for a more detailed explanation.

Run to Return [CTRL+F7]

Select this option to execute statements in the current procedure or function up to the line where the return is positioned. Selecting this item is the same as setting a one-time breakpoint at a specific line. See “.RETURN” in the *IDL Reference Guide* manual for a more detailed explanation.

Set Breakpoint [F9]

Select this option to set a breakpoint on the current line.

See Chapter 18, “Debugging an IDL Program” in the *Building IDL Applications* manual for a more detailed explanation.

Disable Breakpoint [CTRL+F12 (Motif)]

Select this option to access disable a breakpoint in the current line.

See Chapter 18, “Debugging an IDL Program” in the *Building IDL Applications* manual for a more detailed explanation.

Edit Breakpoint...

Select this option to access the **Edit Breakpoint** dialog.

See [Chapter 18, “Debugging an IDL Program”](#) in the *Building IDL Applications* manual for a more detailed explanation.

Up Stack [CTRL+Up]

Select this option to move up the call stack by one.

Down Stack [CTRL+Down]

Select this option to move down the call stack by one.

List Call Stack

Select this option to display the current nesting of procedures and functions. Selecting this item is the same as entering `HELP, /TRACEBACK` at the IDL Command Input Line. See [“HELP”](#) in the *IDL Reference Guide* manual for a more detailed explanation.

Project Menu

For more information on the following menu items, see [Chapter 21, “Creating IDL Projects”](#) in the *Building IDL Applications* manual.

Add/Remove Files...

Select this option to add or remove files from the current project.

Remove Selected [CTRL+H (Motif Only)]

Select this option to remove the currently selected file from your IDL Project.

Move To (Motif Only)

Select this option to remove the currently selected file from your IDL Project.

Groups...

Selecting this option displays the **Project Groups** dialog from which you can create a new group or rename, remove, move up or down, or set to filter specific file types for the default groups within an IDL Project.

Options...

Select this option to change the options for a project. The **Project Options** dialog is displayed.

Compile

Select this option to compile files in a project. You can choose either **All Files** to compile all the source files in a project or **Modified Files** to compile only the files that have been modified since the last compile.

Build

Select this option to build your project.

Run

Select this option to run the application defined by your project.

Export

Select this option to export your project.

Macros Menu

Edit...

Select this item to access the **Edit Macros** dialog. Macros which have already been defined are listed in the **Macros:** field. To edit a macro, click on the macro to access its characteristics and click **OK** when your adjustments are complete.

To add a macro, click **Add...**, which will access the **Add Macro** dialog. Enter the name of the new macro in the given field and click **OK**. Enter the IDL command to be executed by the new macro in the **IDL Command:** field. Enter the menu item name, the full path to the toolbar bitmap file, the tooltip text, and the status bar text in the appropriate fields. Select the accelerator by specifying the key in the **Key:** field and then optionally clicking on any combination of **CTRL**, **ALT** and **Shift**.

Note

Bitmap files for toolbar buttons must be 16 pixels by 16 pixels, and must contain 256 colors or fewer.

To remove a macro, click **Remove**. To change the position of a macro in the **Macro** menu and on the **Macro Toolbar**, click on the macro to highlight it and click on either **Move Up** or **Move Down**.

Import... (Microsoft Windows Only)

Use this menu selection to display the **Import Macros** dialog box. Use this dialog to select the previous IDL installation from which you want macros to be imported.

Print Var (Windows), Print Variable (Motif)

Select this option to print the selected variable. Selecting this item is the same as entering `PRINT, x` at the IDL Command Input Line, where `x` is the selected variable.

Help On Var (Windows), Help On Variable (Motif)

Select this option to list attributes of the selected variable. Selecting this item is the same as entering `HELP, x, /STRUCTURE` at the IDL Command Input Line, where `x` is the selected variable.

Import Image

Select this option to import an image file into IDL. For more information, see [“Using Macros to Import Image Files”](#) on page 187.

Import ASCII

Select this option to import an ASCII file into IDL. For more information, see [“Using Macros to Import ASCII Files”](#) on page 191.

Import Binary

Select this option to import a binary file into IDL. For more information, see [“Using Macros to Import Binary Files”](#) on page 197.

Import HDF

Select this option to import an HDF file into IDL. For more information, see [“Using Macros to Import HDF Files”](#) on page 203.

Demo

Select this option to access IDL’s Demo application.

Window Menu

Read Only (Motif Only)

Select this option to enable or disable editing of the currently selected window. A filled square next to the item indicates Read-Only status.

Next [F6 (Windows), F11 (Motif)]

Select this option to shift IDL's focus to the next numbered editor window.

Previous [SHIFT+F6 (Windows), ALT+F11 (Motif)]

Select this option to shift IDL's focus to the previous numbered editor window.

Cascade

Select this option to cascade all the IDL **Editor** windows within the main window.

Tile Horizontally (Microsoft Windows Only)

Select this option to tile all the IDL **Editor** windows on top of one another within the main window.

Tile Vertically (Microsoft Windows Only)

Select this option to tile all the IDL **Editor** windows side-by-side within the main window.

Tile (Motif Only)

Select this option to arrange all open windows in a non-overlapping fashion.

Close All

Select this option to close all open files. If a file has not yet been saved, you are prompted to save the changes.

Arrange Icons

Select this option to arrange all minimized **Editor** or **Graphics** windows.

Close All

Select this option to close all IDL **Editor** windows. If the text within an IDL **Editor** window has changed, you are asked if you want to save the file before closing.

Configure (Motif Only)

Select this option to access a pulldown menu which alters the appearance of the IDLDE. Select each toggle option to hide or show each component. For more information about each component, see [“Components of the IDLDE”](#) on page 42.

- **Hide Control** (Show Control)
- **Hide View** (Show View)
- **Hide Log** (Show Log)
- **Hide Variable Watch** (Show Variable Watch)
- **Hide Command** (Show Command)
- **Hide Status** (Show Status)
- **Hide Project** (Show Project)

Command Input [CTRL+I] (Microsoft Windows Only)

If this menu item has a check mark by it, the IDL **Command Input Line** is visible in the main IDL window. If this item does not have a check mark next to it, the IDL command input line is not visible. Click on this menu item to toggle between the two states.

Output Log [CTRL+L] (Microsoft Windows Only)

If this menu item has a check mark by it, the **Output Log** is visible in the main IDL window. If this item does not have a check mark next to it, the **Multiple Document Panel** is maximized in the main IDL window. Click on this menu item to toggle between the two states.

Variable Watch [CTRL+A] (Microsoft Windows Only)

If this menu item has a check mark by it, the **Variable Watch Window** is visible in the main IDL window. If this item does not have a check mark next to it, the **Variable Watch Window** is not visible. Click on this menu item to toggle between the two states.

Project (Microsoft Windows Only)

If this menu item has a check mark by it, the **Project Window** is visible in the main IDL window. If this item does not have a check mark next to it, the **Project Window** is not visible. Click on this menu item to toggle between the two states.

Toolbars

Select this option to access a pulldown menu with the three Windows toolbars: **Standard**, **Run & Debug**, and **Macros**. If a toolbar has a check mark by it, it is visible below the Menu bar items.

Status Bar (Microsoft Windows Only)

If this menu item has a check mark by it, the **Status** bar is visible at the very bottom of the Main IDL window.

Numbered Windows

The numbered menu items at the bottom of the **Window** menu display open files. Select any of these menu items to make that window the current window.

Help Menu

Contents...[CTRL+F1]

Select this menu item to display the IDL **Online Help** Viewer.

Find Topic... [F1]

Select this menu item to display the **Search** dialog for IDL **Online Help**.

About IDL...

Select this option to display information on the IDL version in use.

Keyboard Shortcuts

Most of the IDL Development Environment menu options can be accessed from the keyboard instead of clicking on the menus.

Note

See “[Using Keyboard Accelerators](#)” on page 33 for information about using IDL’s keyboard shortcuts on a Macintosh.

The following table lists all of the available keyboard equivalents. Note that these equivalents are also shown to the right of each menu item in the menus themselves.

Windows Keyboard Shortcut	UNIX Keyboard Shortcut	Function
CTRL+A	n/a	Toggle Variable Watch Window
CTRL+C	ALT+C	Copy selection to clipboard
CTRL+D	CTRL+T	Go to definition
CTRL+E	ALT+I	Find highlighted selection
CTRL+F	ALT+F	Start Find dialog
F3	ALT+G	Find Again
CTRL+G	CTRL+G	Start Go To Line dialog
CTRL+H	ALT+R	Start Replace dialog
n/a	ALT+P	Replace and Find Again
n/a	ALT+T	Enter Selection in the Find dialog
CTRL+I	n/a	Toggle Command Input Line
CTRL+L	n/a	Toggle Output Log
CTRL+N	CTRL+N	Open new (empty) Editor Window
CTRL+O	CTRL+O	Open file
CTRL+P	CTRL+P	Print currently-active file

Table 2-2: IDLDE Keyboard Shortcuts

Windows Keyboard Shortcut	UNIX Keyboard Shortcut	Function
CTRL+Q	CTRL+Q	Exit IDL
CTRL+R	n/a	Stop the IDL environment
CTRL+S	CTRL+S	Save currently-active file
n/a	CTRL+W	Save to a new file (Save As...)
n/a	CTRL+Y	Erase contents of Output Log.
CTRL+V	ALT+V	Paste selection from clipboard at insertion point
CTRL+X	ALT+X	Cut selection to clipboard
CTRL+Y	ALT+Y	Redo last undo
CTRL+Z	ALT+Z	Undo previous editing action
CTRL+Break	CTRL+C	Interrupt program execution /Break
CTRL+Del	n/a	Clear current Editor window
CTRL+F1	n/a	Online Help Contents Page
n/a	ALT+F5	Resolve Dependencies
CTRL+F5	CTRL+F5	Compile currently-selected file
CTRL+F6	CTRL+F6	Compile program from memory
CTRL+F7	CTRL+F7	Execute file to return
CTRL+F8	CTRL+F8	Continue processing until program returns: .OUT
CTRL+↑	CTRL+↑	Move up call stack
CTRL+↓	CTRL+↓	Move down call stack
Delete	Delete	Delete selection
F1	n/a	Start Find Topic in Online Help
F5	F5	Run

Table 2-2: IDLDE Keyboard Shortcuts (Continued)

Windows Keyboard Shortcut	UNIX Keyboard Shortcut	Function
F5	F6	Continue stopped program: .CONTINUE
F6	F11	Display next-numbered Editor window
F7	F7	Execute file to cursor
F8	F8	Execute a single statement: .STEP
F9	F9	Set / Clear breakpoint
F10	F10	Execute a single statement: .STEPOVER
Shift+F6	ALT+F11	Display previously-numbered Editor window

Table 2-2: IDLDE Keyboard Shortcuts (Continued)



Chapter 3: Using the IDL Editor

This chapter describes the use of IDL's built-in editor.

About the IDL Editor	74	Chromacoded Editor (Windows Only)	80
Keyboard Shortcuts	75	Functions/Procedures Menu	82
Searching	77	Using External Editors (Motif)	83
Text Selection Modes (Windows Only)	78		

About the IDL Editor

Although any text editor can be used to create an IDL program file, the IDL Editor included in the IDL Development Environment contains features that simplify the process of writing IDL code. For example, if you indent a line using the Tab key, the following lines will be indented as well.

If you use the IDL Development Environment, files are opened in the IDL Editor by default. On UNIX platforms, you can simplify the process of using another editor; see [“Using External Editors \(Motif\)”](#) on page 83 for details.

If you have a file open in the IDL Editor and you modify the file using another editor (on any platform), IDL will warn you that the file in the IDL Editor window has changed, and give you a chance to reload the file.

Keyboard Shortcuts

The following keyboard shortcuts are available in IDL Editor windows:

Key (Windows)	Key (Motif)	Action
← →↑↓	← →↑↓	Move cursor left or right one character, up or down one line.
Ctrl+←	Ctrl+B	Move left one word.
Ctrl+→	Ctrl+F	Move right one word.
End	Ctrl+E	Move to end of current line.
Home	Ctrl+A or Home	Move to beginning of current line.
Page Down	Page Down	Move to next screen.
Page Up	Page Up	Move to previous screen.
Shift+Tab		Move cursor one tab-stop left.
Ctrl+Home	Ctrl+Home	Move to beginning of file.
Ctrl+End	Ctrl+End	Move to end of file.
	Ctrl+V	Delete word to the left of the cursor.
Ctrl+K		Delete word to the right of the cursor.
	Ctrl+K	Delete everything in the current line to the right of the cursor.
	Ctrl+U	Delete everything in the current line to the left of the cursor.
Delete	Ctrl+D	Delete the next character.
Ctrl+U		Make selected text (or the character to the right of the cursor) lower-case.
Ctrl+Shift+U		Make selected text (or the character to the right of the cursor) upper-case.

Table 3-1: IDL Editor Window Key Definitions

Key (Windows)	Key (Motif)	Action
Ctrl+Z	Alt+Z	Undo last action.
Ctrl+Y	Alt+Y	Redo last undone action.
Ctrl+X	Alt+X	Cut selection to clipboard.
Ctrl+Shift+Y		Cut line containing cursor to clipboard.
Ctrl+C	Alt+C	Copy selection to clipboard.
Ctrl+V	Alt+V	Paste contents of clipboard at current cursor location.
Ctrl+]		Find matching (, {, or [character.
Tab		Indent highlighted lines one tab-stop right.

Table 3-1: IDL Editor Window Key Definitions (Continued)

Searching

The IDL Editor window provides a comprehensive search-and-replace mechanism, allowing you to search for occurrences of a text string in one or more open files. See [“Search Menu”](#) in Chapter 2 for details.

Text Selection Modes (Windows Only)

Under Microsoft Windows, the IDL Editor provides three ways of selecting text: stream mode, line mode, and column mode.

- Stream mode selects text in a stream, beginning with the first character selected and ending with the last character, just as if you were reading the text.

```
IDL is a complete computing environment for the
interactive analysis and visualization of data.
```

Figure 3-1: A selected stream of text.

- Line mode selects full lines of text.

```
IDL is a complete computing environment for the
interactive analysis and visualization of data.
IDL integrates a powerful, array-oriented language
with numerous mathematical analysis and graphical
display techniques.
```

Figure 3-2: Text selection using Line Mode.

- Box mode selects text from one screen column to the next. Selecting text in column mode is similar to drawing a rectangle around the text you wish to select.

```
IDL is a complete computing environment for the
interactive analysis and visualization of data.
IDL integrates a powerful, array-oriented language
with numerous mathematical analysis and graphical
display techniques.
```

Figure 3-3: Column Mode text selection.

Switch between the three modes by clicking the right mouse button while positioned over an Editor window. Select the “Selection Mode” option to access a pull-down

menu with the three text selection modes. The option with a check mark by it is the currently selected text selection mode. If you have text already selected, the selected area will change to reflect the new mode.

Chromacoded Editor (Windows Only)

The IDL Editor in IDL for Windows supports chromacoding—different types of IDL statements appear in different colors. To change the default colors used for different types of IDL statements, select **File** → **Preferences**, and select the **Editor** tab.

Turning Chromacoding Off

By default, the Windows IDL Editor uses chromacoding. To turn off chromacoding, select **File** → **Preferences**, select the **Editor** tab, and uncheck the **Enable colored syntax** checkbox.

Block Comments

The IDLDE provides methods to quickly comment and uncomment blocks of code lines. To comment or uncomment lines of code, you may either select the lines to be commented/uncommented or you may simply place the cursor somewhere on the desired line. Commenting and uncommenting can be performed using:

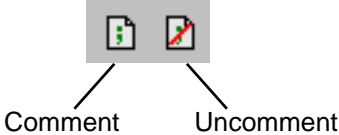
Method	Description
Toolbar	Click the Comment or Uncomment toolbar items. <div style="text-align: center; margin-top: 10px;">  </div>
Menu	Use the Edit → Comment or Edit → Uncomment menu items.
Context menu	Right click over a line (or block of selected lines) to display the context menu. Select Comment or Uncomment from the menu.

Table 3-2: Block Comment Methods

Functions/Procedures Menu

When you open a file in the IDL Editor, all functions and procedures defined in that file are listed in the Functions/Procedures Menu. On Windows, this feature appears as a pull-down menu located on the IDLDE toolbar. On Motif, this menu is accessed through the () button in the upper left corner of the Editor window.

Select a procedure or function from the drop-down list to move the cursor to the beginning of that procedure or function. This is especially useful for navigating large program files containing multiple procedures and functions.

Using External Editors (Motif)

If you wish to use more sophisticated editing features on Motif platforms, you can create an IDLDE macro to open a file that is currently open in the IDL Editor in another editor such as emacs or vi. Use the following procedure to create a macro:

1. Select **Macros** → **Edit** menu to bring up the Edit Macros dialog box. You can use this dialog to create, edit, or remove macros.
2. Complete the fields in the Edit Macros dialog:
 - **Name:** The name that you wish to appear in the **Macros** list in the Edit Macros dialog. For example, enter `Edit` in `emacs`.
 - **Label:** The name that you wish to appear on the **Macros** menu. For example, enter `emacs`.
 - **Bitmap:** The bitmap to use as the toolbar button label. Use the file paths and file name extensions discussed in “[Bitmaps for Control Panel Buttons](#)” in Chapter 6.
 - **Status bar text:** The text that appears in the status bar when the mouse is help over the menu item or toolbar button.
 - **Tip text:** The text for the tool tip that appears when the mouse is held over the toolbar button.
 - **IDL command:** The IDL command to execute when the macro is selected. To create a macro for editing in Emacs, enter the following:

```
SPAWN, 'emacs +%L %P &'
```
 - Select the **Menu** and/or **Toolbar** checkbox to specify whether the macro will appear in the **Macros** menu and/or the toolbar.
3. Create the new macro by pressing the **Add** button. If you entered `emacs` in the **Label** field, a new “`emacs`” macro is added to the Macros list.
4. To add a macro for editing in vi, repeat the above steps, but enter the following in the “IDL command” field:

```
SPAWN, 'xterm -e vi +%L %P &'
```

To use the new macros, open the desired file in the IDL editor, then select the desired **Macros** menu item or toolbar button.

The IDLDE always checks if the current file has been externally modified before using it. If a file was modified with an external editor, IDLDE notifies you, and asks you to reload the file before using it (you can also use the **Revert to Saved** option from the **File** menu to reload the file).



Chapter 4: Using the IDL Command Input Line

The following topics are covered in this chapter:

Input to IDL	86	Issuing Operating System Commands	92
Executive Commands	90		

Input to IDL

Commands entered at the IDL prompt are usually interpreted as IDL statements to be executed. Other interpretations include executive commands that control execution and compilation of programs, shell commands, and so on. Input to the IDL prompt is interpreted according to the first character of the line, as shown in the following table.

Note

The information in this chapter applies equally to IDL used in command-line mode or via the IDL Development Environment.

First Character	Action
.	Executive command.
?	Help inquiry.
\$	Command to be sent to operating system.
@	Batch file initiation.
↑ or ↓ key	Recall/edit previous commands.
CTRL+D	In UNIX command-line mode, exits IDL, closes all files, and returns to operating system.
CTRL+Z	In UNIX command-line mode, suspends IDL.
All others	IDL statement.

Table 4-1: Interpretation of the First Character in an IDL Command

Command Recall and Line Editing

By default, IDL saves the last 20 commands entered in a *recall buffer*. These command lines can be recalled, edited, and re-entered. The up-arrow key (↑) on the keypad recalls the previous command you entered to IDL, moving backward through the command history list. Pressing it again recalls the previous line, and so on. The down-arrow key (↓) on the keypad moves forward through the command history. When a command is recalled, it is displayed at the IDL prompt and can be edited and/or entered.

You can view the contents of the recall buffer in the following ways:

- Use the arrow keys to view the entries in the buffer one at a time.
- Use the HELP procedure with the RECALL_COMMANDS keyword to display the entire contents of the recall buffer in the IDL Output Log.
- In the IDL Development Environment, by right-clicking on the **Command Input Line**. The 20 most recent commands in the command recall buffer are displayed, and can be selected and re-executed.

The line-editing abilities and the keys that activate them differ somewhat between the different operating systems.

Note

The behavior can also differ within the same operating system, between the Command prompt for IDL and the Command line on the IDLDE.

The table below lists the edit functions and the corresponding keys.

Function	UNIX	Windows
Move cursor to start of line	CTRL+A or Home	Home
Move cursor to end of line	CTRL+E or End	End
Move cursor left one character	Left arrow	Left arrow
Move cursor right one character	Right arrow	Right arrow
Move cursor left one word	CTRL+B, (R13 on Sun Keyboard)	CTRL+left arrow
Move cursor right one word	CTRL+F, (R15 on Sun Keyboard)	CTRL+right arrow
Delete from current to start of line	CTRL+U	
Delete from current to end of line	CTRL+K	
Delete entire line		

Table 4-2: Command Recall and Line Editing Keys

Function	UNIX	Windows
Delete current character	CTRL+X or CTRL+D	Delete
Delete previous character	CTRL+H, or Backspace, or Delete	Backspace
Delete previous word	CTRL+W, or ESC-Delete	
Generate IDL keyboard interrupt	CTRL+C	CTRL+break
Move back one line in recall buffer	CTRL+N, Up arrow	Up arrow
Move forward one line in recall buffer	Down arrow	Down arrow
Redraw current line	CTRL+R	
Overstrike/Insert	ESC-I	
EOF if current line is empty, else EOL	CTRL+D	
Search recall buffer for text	Available only in command-line mode. Enter ^, then input <i>search string</i> at prompt.	
Insert the character at the current Executive Commands position	any character	any character

Table 4-2: Command Recall and Line Editing Keys (Continued)

The command recall feature is enabled by setting the system variable `!EDIT_INPUT` to a non-zero value (the default is 1) and is disabled by setting it to 0. See “`!EDIT_INPUT`” in Appendix D of the *IDL Reference Guide* manual for details.

Copying and Pasting Multiple IDL Code Lines

You can paste multiple lines of text from the clipboard to the command line. You simply need to place some text in the clipboard and paste it into the command line. Any source of text is valid, with emphasis on the requirement that the text be

convertible to ASCII. When copying text from an IDE editor, the selection mode can be stream, line, or box.

Note

Line and box modes automatically put a trailing carriage return at the end of the text. When pasted, the last line is executed.

Be sure when you paste multiple lines that they only contain a single IDL command or are lines which include statements that utilize line continuation characters (\$). Multi-line statements will produce unintended IDL interpreter behavior or errors.

Lines are transferred to the command line as is. Namely, leading white space is not removed and comment lines are sent to the IDL interpreter without distinction.

Note

Tabs are converted to white space based on the tab size indicated by the IDE editor preferences.

Changing the Number of Lines Saved

You can change the number of command lines saved in the recall buffer by setting `!EDIT_INPUT` equal to a number other than one (in the IDL Development Environment, you can set this value via the **General** tab of the IDLDE Preferences dialog as well.) In order for the change to take effect, IDL must be able to process the assignment statement before providing a command prompt. This means that you must put the assignment statement in the IDL startup file. (See “[Startup Files](#)” in Chapter 1 for more information on startup files.)

For example, placing the line

```
!EDIT_INPUT = 50
```

in your IDL startup file changes the number of lines saved in the command recall buffer to 50.

See “`!EDIT_INPUT`” in Appendix D of the *IDL Reference Guide* manual and “[General Preferences](#)” in Chapter 5 for additional details.

Executive Commands

IDL executive commands compile programs, continue stopped programs, and start previously compiled programs. All of these commands begin with a period and must be entered in response to the IDL prompt. Commands can be entered in either uppercase or lowercase and can be abbreviated. Under UNIX, filenames are case sensitive; under Microsoft Windows, filenames can be specified in any case.

Note

Comments (prefaced by the semicolon character in IDL code) are not allowed within executive commands.

Executive commands are summarized in the table below. See the *IDL Reference Guide* for in-depth descriptions of these commands.

Command	Action
<code>.COMPILE</code>	Compiles text from files or keyboard without executing.
<code>.CONTINUE</code>	Continues execution of a stopped program.
<code>.EDIT</code>	Opens files in editor windows of the IDLDE.
<code>.FULL_RESET_SESSION</code>	Does everything <code>.RESET_SESSION</code> does, plus additional reset tasks such as unloading sharable libraries.
<code>.GO</code>	Executes previously compiled main program from beginning.
<code>.OUT</code>	Continues program execution until the current routine returns.
<code>.RESET_SESSION</code>	Resets much of the state of an IDL session without requiring the user to exit and restart the IDL session.
<code>.RETURN</code>	Continues execution until encountering a <code>RETURN</code> statement.

Table 4-3: Executive Commands

Command	Action
<code>.RNEW</code>	Erases main program variables and then executes <code>.RUN</code>
<code>.RUN</code>	Compiles and possibly executes text from files or keyboard
<code>.SKIP</code>	Skips over the next statement and then single steps
<code>.STEP</code>	Executes a single statement (abbreviated as <code>.S</code>)
<code>.STEPOVER</code>	Executes a single statement if the statement does not call a routine (abbreviated as <code>.SO</code>)
<code>.TRACE</code>	Similar to <code>.CONTINUE</code> , but displays each line of code before execution

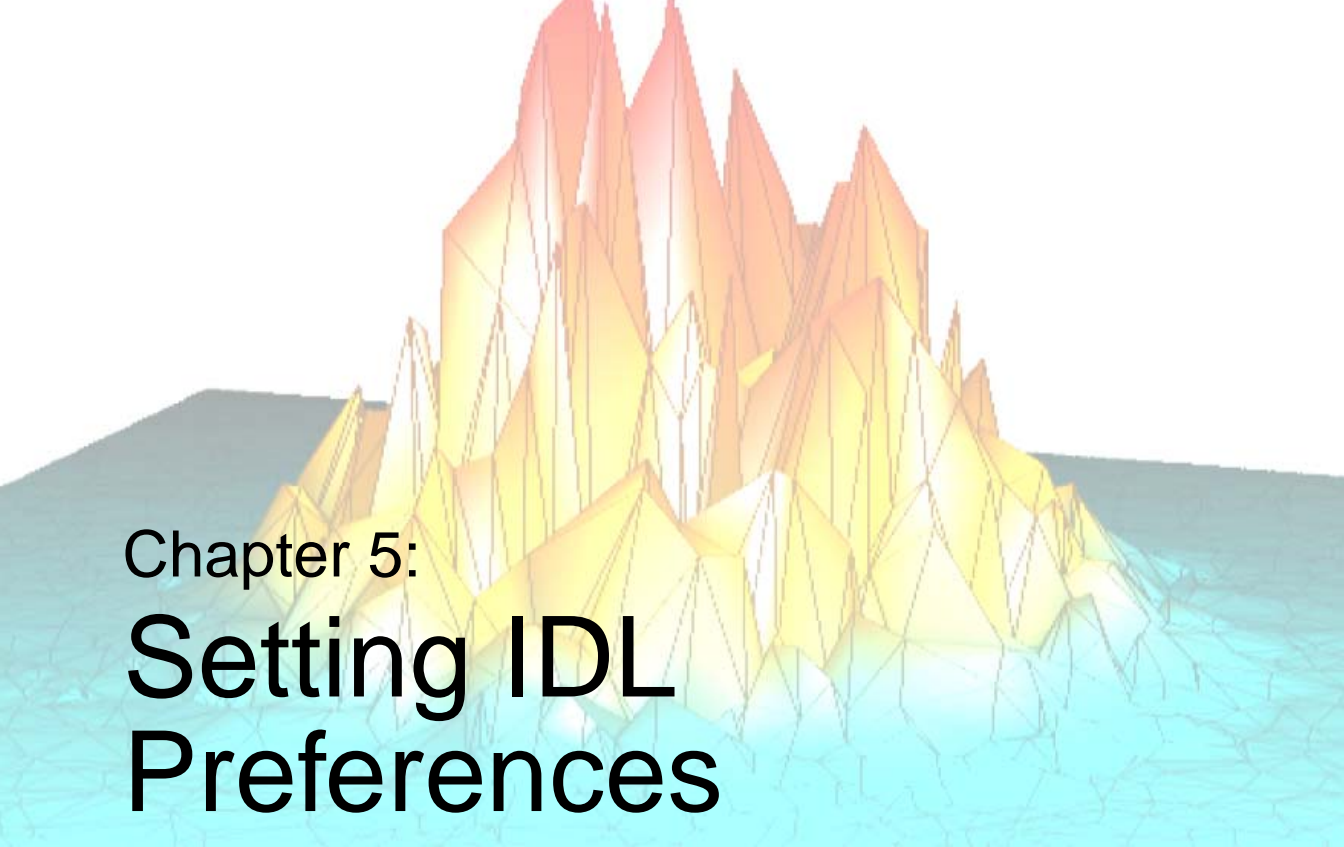
Table 4-3: Executive Commands (Continued)

Issuing Operating System Commands

Operating system commands can be sent to a subprocess for execution by entering the \$ character followed by the command at the IDL prompt.

The SPAWN procedure is a more flexible way of accomplishing the same thing because it can be used within an IDL program while \$ can only be entered interactively. In addition, the standard output of the command can be saved in an IDL string array by SPAWN. Hence, \$ can be thought of as an interactive-only abbreviation for SPAWN.

For more information on the SPAWN procedure, see “[SPAWN](#)” in the *IDL Reference Guide* manual.



Chapter 5: Setting IDL Preferences

The IDL Development Environment can be customized by setting *preferences*. This chapter describes the sections of the Preferences dialog:

Customizing IDL	94	Editor Preferences	106
General Preferences	97	Startup Preferences	108
Layout Preferences	99	Font Preferences	110
Graphics Preferences	103	Path Preferences	112

Customizing IDL

Various settings for the IDL Development Environment can be customized using the Preferences dialog. To open the **Preferences** dialog, select **Preferences** from the IDL Development Environment **File** menu.

Note

On UNIX platforms, including Macintosh OS X, some settings can also be customized by editing IDL's resource files. For further information about editing resource files on UNIX and Macintosh OS X, see [Chapter 8, "Customizing IDL on Motif Systems"](#).

The **Preferences** dialog contains tabbed sections that allow you to customize your interaction with the IDLDE environment. The tabs and their uses are described below.

Note

The terminology used on the Preferences dialogs differs between Microsoft Windows and Motif systems. In this documentation, if the wording is significantly different between the two platforms, the wording used in the Windows dialogs is listed first, followed by the wording used in the Motif dialogs.

Tab	Description
General Preferences	This tab allows you to specify how the IDLDE session begins and ends, to control the number of lines in the recall buffer and the Output Log, and to designate how the files should be opened and read.
Layout Preferences	This tab allows you to specify the location and size of the main IDLDE window on the screen. You can also designate which components of the IDLDE will be visible.
Graphics Preferences	This tab allows you to set the layout of windows that contain IDL graphics, and to specify the backing store.
Editor Preferences	This tab allows you to customize the IDL's built-in editor and also offers several compiling options.

Table 5-1: Preference Dialog Tabs

Tab	Description
Startup Preferences	This tab allows you to specify the main IDL directory, the working directory, and a startup file.
Font Preferences	This tab allows you to specify different fonts, styles, and sizes for the Editor, Command Input Line and Output Log.
Path Preferences	This tab allows you to specify the IDL Files Search Path. Entries in this tab are used to set the system variable !PATH if the IDL_PATH environment variable has not be set.

Table 5-1: Preference Dialog Tabs (Continued)

Platform Differences

Microsoft Windows and UNIX platforms (including Macintosh OS X) implement the **Preferences** dialog using different dialog application buttons. The following table lists the buttons, the platforms on which they are found in the **Preferences** dialog, and the action performed when the button is used.

Platform	Button	Result
Windows, UNIX	OK	Changes are applied to the current session and the Preferences dialog is dismissed.
	Apply	Changes are applied to the current session but not saved. The Preferences dialog remains visible.
Windows only	Reset	Restores the Preferences on the dialog to the preference values from the start of the current IDL session.
	Cancel	Any changes which were not applied are ignored and the Preferences dialog is dismissed.

Table 5-2: Preferences Dialog Button Descriptions

Platform	Button	Result
UNIX only	Save	Changes are applied to the current session and saved. If the option has an asterisk next to it, you must save and restart the IDLDE for the change to take effect.
	Dismiss	Dismisses the Preferences dialog box without applying any changes, unless previously applied.
	Help	Displays IDL Online Help.

Table 5-2: Preferences Dialog Button Descriptions (Continued)

General Preferences

The **General** tab of the Preferences dialog has three sections: **Program**, **Log and Command windows**, and **Files**.

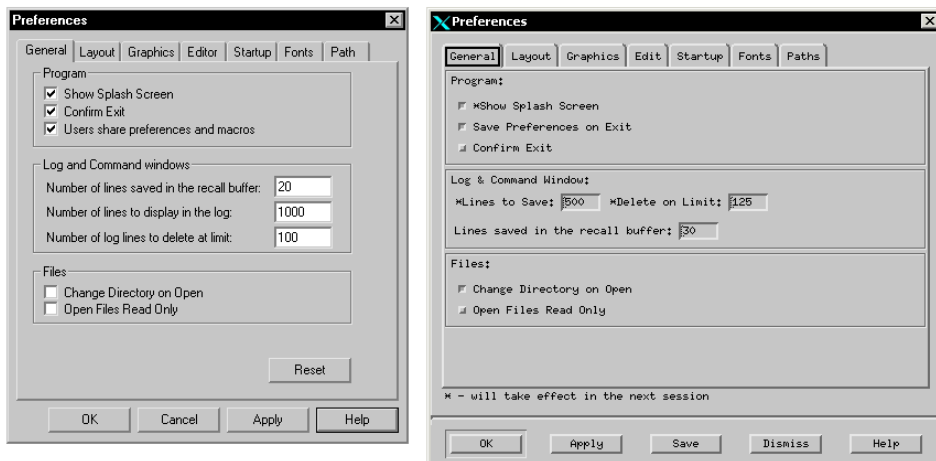


Figure 5-1: General Preferences Dialog

Program Section

You can specify how IDL handles starting up and exiting. Click on the following check boxes to apply or disable the options:

- **Show Splash Screen** — Select this option to show the IDL splash screen on startup. This selection takes effect the next time an IDL session is started.
- **Save Preferences on Exit (Motif only)** — Select this option to save all the settings specified in the Preference tabs when the current IDL session exits. If you deselect this option, changes made to the current IDL session and not explicitly saved (by clicking either the **Save** or **OK** button) will be discarded when the current session ends.
- **Confirm Exit** — Select this option to display a warning dialog when you exit IDL.
- **Users share preferences and macros (Windows only)** — If this check box is selected, all users on the Windows computer share the same set of preferences and macros (which means that any user can change preferences and macros for

all users on the computer). By default, each user on the Windows computer has a unique set of preferences and macros; changes affect only the current user.

Note

You must have Administrator rights on the Windows computer to change the “share preferences” setting. The checkbox will be greyed out if you are logged on as a user without administrative rights.

Log and Command Windows Section

The number of lines saved in the recall buffer for the Command Line has an impact on the performance of IDL. The amount of memory required for greater numbers of saved lines in the buffer affects the speed at which IDL runs. Click in the field next to each description and enter your adjusted value to change the settings.

- **Number of lines saved in the recall buffer** — This field controls the maximum number of lines saved in the recall buffer. (See “[Command Recall and Line Editing](#)” in Chapter 4 for information on using the recall buffer.) The default is 20 lines.
- **Number of lines to display in the log / Lines to Save** — This field controls the maximum number of lines retained by the **Output Log** window. The default is 1000 lines for Microsoft Windows systems and 500 for UNIX systems.
- **Number of log lines to delete at limit / Delete on Limit** — This field controls the number of lines that will be deleted from the Output Log window when the maximum number of lines is reached. The earliest lines in the log are deleted. The default is 100 for Microsoft Windows systems and 125 for UNIX systems.

Files Section

You can change the way in which IDL handles opening files. Select or clear the following check boxes to apply or disable the options:

- **Change Directory on Open** — Select this option to cause IDL to change the current working directory when you open a file. The new current working directory will be the directory that contains the opened file.
- **Open Files Read Only** — Select this option to open files so that they can be viewed, but not changed.

Layout Preferences

This tab allows you to control the appearance and placement of the IDLDE.

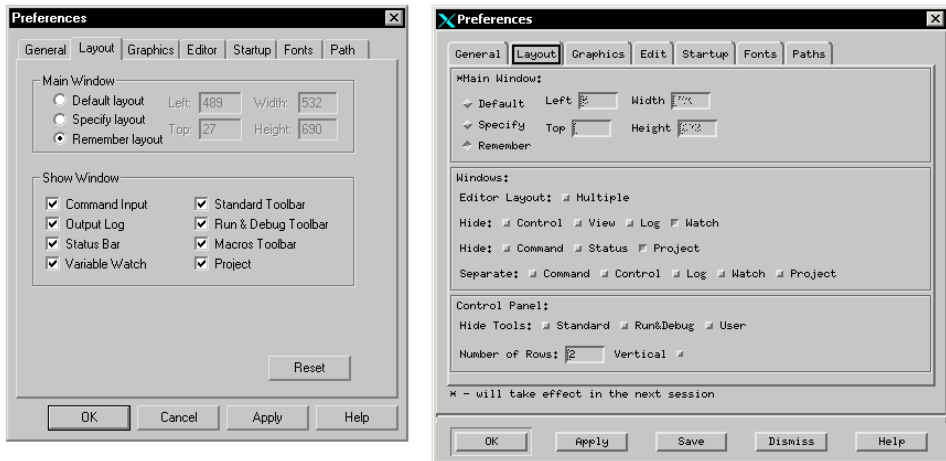


Figure 5-2: Layout Preferences Dialog

Main Window Section

Use the fields in this section to specify the default size and placement of the IDL Development Environment’s main window. (See “[Components of the IDLDE](#)” in Chapter 2 for descriptions of the components of the IDLDE.)

- Select the **Default Layout** radio button to use the IDLDE’s default layout, which depends on the size and resolution of your computer screen. If you select this radio button, all of the IDLDE’s windows and toolbars will be displayed in their standard locations.
- Select the **Specify Layout** radio button to manually specify the layout of the IDLDE:
 - Enter the number of pixels from the left-hand edge of the screen the IDLDE window should be displayed in the **Left** field.
 - Enter the number of pixels from the top edge of the screen the IDLDE window should be displayed in the **Top** field.
 - Enter the width of the IDLDE window in pixels in the **Width** field.

- Enter the height of the IDLDE window in pixels in the **Height** field.

Note that if you select the **Default Layout** radio button after specifying values in these fields, your values will be replaced with “-1” to indicate that the default values will be used the next time IDL starts.

- Select the windows and toolbars to be displayed from the **Show Window** section (Windows) or **Windows** and **Control Panel** sections (Motif).

Click **Apply** to apply your changes to the current IDLDE window without saving the values. (This allows you to use the **Layout** tab to control the appearance of the IDLDE for the current session without making your changes permanent.) Click **OK** to apply your changes and save the values; they will be used the next time IDL starts.

- Select the **Remember Layout** radio button and click **OK** to save the current layout of the IDLDE windows for use the next time IDL starts. This options is useful if you have configured the windows manually and wish to save your changes.

Undocking IDLDE windows

Some of the elements of the IDLDE can be “undocked” from the interface and appear as separate, free-floating windows. On Microsoft Windows systems, use the mouse to select an element and drag it away from the main IDLDE window to undock the element. On Motif systems, you can use the checkboxes in the **Windows** section to undock elements.

The following elements can be undocked:

- Command Input Line
- Toolbars
- Output Log
- Variable Watch Window
- Project Window

Show Window Section (Microsoft Windows Only)

By default, all the listed options are checked, signifying that they are all visible in the IDLDE main window. Click on the check boxes to show or hide the following windows.

- Command Input line;
- Output Log window;

- Status Bar;
- Variable Watch window;
- Standard Toolbar;
- Run & Debug Toolbar;
- Macros Toolbar;
- Project window

Click **Apply** to apply your changes to the current IDL session. (This is the same as selecting the corresponding options in the **Window** menu.) Select the **Specify Layout** radio button and click **OK** to make the current layout the default.

Windows Section (Motif Only)

Use the options in this section to control the appearance of the window elements of the IDLDE.

- **Editor Layout** — Click **Multiple** to display open Editor and Project windows separately from the main IDLDE window. Note that if the **Multiple Windows** option is enabled, the choice to hide or view the Editor windows is not available.
- **Hide** — Select the check box for elements of the IDLDE you wish to hide from view. By default, none of the sections are hidden.
 - **Control** hides the toolbars;
 - **View** hides the Project window and the Editor window;
 - **Log** hides the Output Log window;
 - **Watch** hides the Variable Watch window;
 - **Command** hides the Input Command Line;
 - **Status** hides the fly over status line at the base of the Main IDL window;
 - **Project** hides the Project window and extends the Editor window to the full width of the IDLDE.
- **Separate** — Select the check box for the constituent window you want to separate from the IDLDE Main Window. When the **Separate** action is applied, the element is “undocked” from the interface and appears as separate, free-floating window.

Click **Apply** to apply your changes to the current IDL session. (This is the same as selecting the corresponding options in the Window menu.) Select the **Specify Layout** radio button and click **OK** to make the current layout the default.

Control Panel Section (Motif Only)

You can specify how you would like to display the various toolbars on the Control Panel.

- **Hide Tools** — Select the check box for any of the available toolbars (**Standard**, **Run & Debug**, and **User**) to hide that toolbar.
- **Number of Rows** — Enter the number of rows to use in displaying any visible toolbars. You can select from 1 to 3 rows.
- **Vertical** — Select this check box to cause the toolbars to be stacked vertically one on top of the other rather than horizontally next to each other.

Graphics Preferences

This tab allows you to control the layout and default size of IDL Graphics windows. You can also control IDL's default use of backing store and the size of the TrueType font cache. Note that the values set here are defaults; the values can be overridden when a graphics window is created.

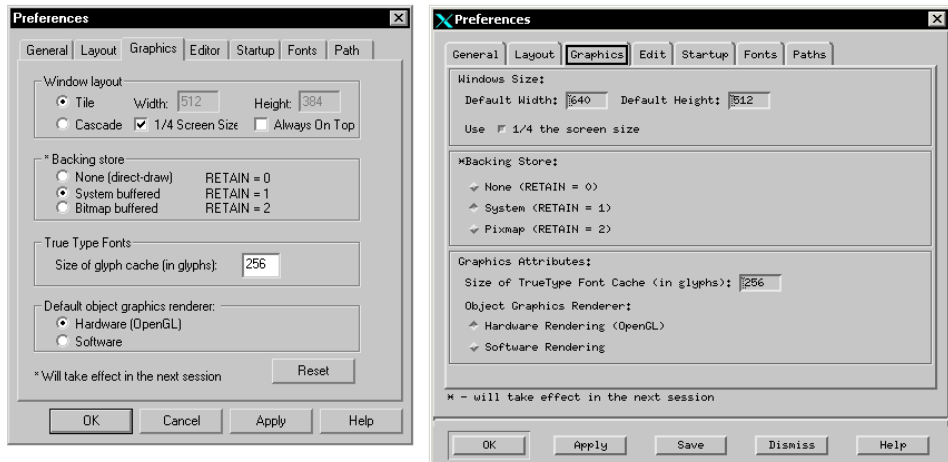


Figure 5-3: Graphics Preferences Dialog

Window layout / Windows Size Section

Specify the default width and height of IDL graphics windows in the **Width** and **Height** fields. Alternatively, you can specify that graphics windows have a default width and height of half the screen width and height by checking the **1/4 Screen Size** checkbox.

Platform Differences

On Windows systems, you can specify that graphics windows should be created size-by-side, with no overlap by selecting the **Tile** radio button, or that they should be created overlapping by selecting the **Cascade** radio button. Select the **Always On Top** checkbox to ensure that graphics windows float above all other IDL windows.

Backing Store Section

When backing store is enabled, a copy of each Graphics window is kept in memory; the copy is used to refresh the window when it has been covered and uncovered.

IDL's performance may increase when no backing store is used, since the amount of memory required to save files can affect the speed at which IDL will run. Settings in this section correspond to settings of the **RETAIN** keyword to the **DEVICE** procedure; see "[Backing Store](#)" in Appendix A of the *IDL Reference Guide* manual for more information.

- **None** (RETAIN = 0): Select this option to refrain from keeping a copy of the window. In some situations, disabling backing store may lead to an increase in IDL's performance.
- **System** (RETAIN = 1): Select this option to request backing store from the windowing system. This is the default.
- **Bitmap / Pixmap** (RETAIN = 2): Select this option to specify that IDL should maintain the backing store using its own memory.

Note

Backing Store preference changes do not take effect until the next IDL session.

True Type Fonts Section

IDL saves TrueType fonts as a set of *glyphs*; each glyph represents the triangulation data for drawing one character. The **Size of TrueType Font Cache** (in glyphs) field allows you to set the number of glyphs to keep in cache memory; keeping glyphs in memory speeds drawing of fonts in IDL graphics windows. The default number of glyphs in cache memory is 256, roughly two TrueType font sets.

Enter the number of TrueType characters for which to save triangulation information. Saving the triangulation information for TrueType characters means that IDL will not have to calculate the polygons to draw the next time a character of the same font and size is rendered. Larger values will use more memory but can increase drawing speed if multiple fonts are used. The default is 256.

Default object graphics renderer / Graphics Attributes Section

IDL supports two methods of rendering object graphics: via a hardware graphics accelerator or via a software rendering package. Select **Hardware** rendering if your system has OpenGL graphics accelerator hardware. Select **Software** rendering otherwise.

See “[Hardware vs. Software Rendering](#)” in Chapter 34 for information about the differences between the two rendering systems.

Editor Preferences

This tab allows you to specify settings for the built-in IDL Editor and control the way IDL compiles files loaded in editor windows. On Microsoft Windows systems, this tab also allows you to specify syntax-highlighting and other editor features.

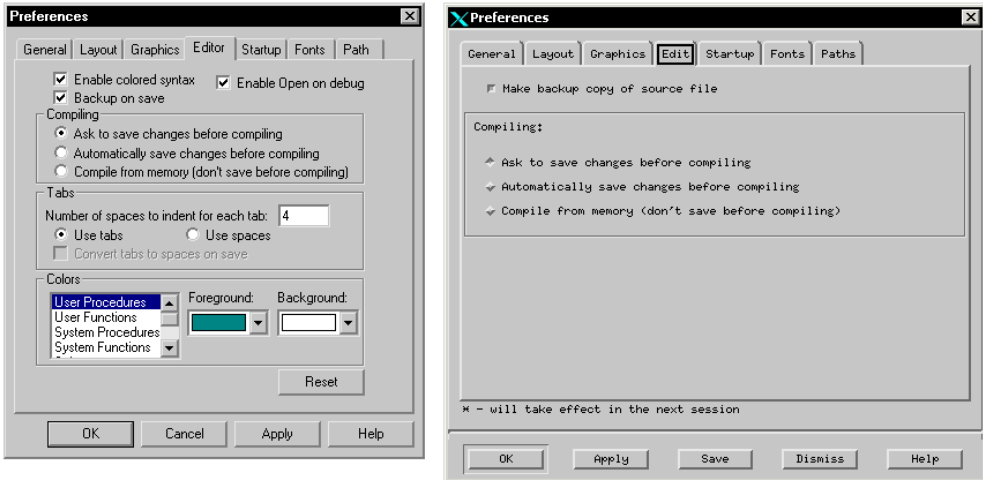


Figure 5-4: Editor Preferences Dialog

Backup on Save

Select the **Backup on Save / Make backup copy of source file** check box to cause IDL to create a backup of the original file when saving a file in an IDL editor window.

Compiling Section

Select the **Ask to save changes before compiling** radio button if you would like to save changes when you compile a program in an IDL editor window. This is the default.

Select the **Automatically save changes before compiling** radio button if you do not want to be prompted each time you compile, but do want to save the changes.

Select the **Compile from memory (don't save before compile)** radio button if you do not want to save files before compiling them.

Note

You can override your default selection by selecting the appropriate menu item from the **Run** menu.

Microsoft Windows Editor Configuration

On Microsoft Windows systems, you can choose to use *syntax highlighting* in IDL editor windows. If syntax highlighting is turned on, IDL statements are displayed in different colors. Select the **Enable colored syntax** checkbox to enable syntax highlighting.

If you want IDL to open the source file for a program that generates an error in an IDL editor window, select the **Enable Open on debug** checkbox.

Tabs Section

You can specify the width of the white space to be used when you press the TAB key in an IDL editor window. Enter a number in the **Number of spaces to indent for each tab** field to specify the width of the indent to be used.

If you want the IDL editor to insert a tab character (ASCII 9) when you press the TAB key, select the **Use tabs** radio button. If you want IDL to insert the specified number of space characters (ASCII 32) when you press the TAB key, select the **Use spaces** radio button.

If you have selected the **Use spaces** radio button, you have the option to convert tab characters to spaces when the file is saved by selecting the **Convert tabs to spaces on save** checkbox.

Colors Section

Use this section to select the colors that will be used in the IDL editor when syntax highlighting is enabled. To set colors, select a type of IDL statement from the scrolling listbox at left, then select the foreground and background colors for that type of statement.

Startup Preferences

This tab allows you to specify the location of the main IDL directory, the default working directory, and any startup file to be run.

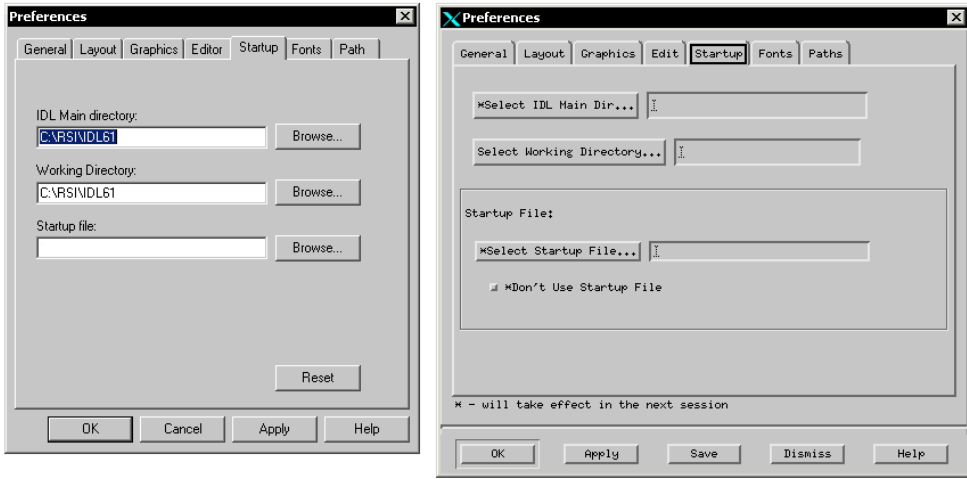


Figure 5-5: Startup Preferences Dialog

IDL Main Directory

Use this field to select the location of the main IDL directory. The default is the location you specified when you installed IDL. There is no reason to change this entry. The location of the home IDL directory is shown primarily for informational purposes.

Working Directory

This field allows you to set the initial working directory for future IDL sessions. The [General Preferences](#) tab contains a “Change Directory on Open” option, which also affects the working directory.

Startup file

Use this field to specify the name of an IDL batch file to be executed automatically each time IDL is run. See [“Startup Files”](#) on page 30 for additional details.

On Motif platforms, you can disable the use of the startup file by selecting the **Don’t Use Startup File** checkbox.

Font Preferences

This tab allows you to specify fonts to be used in various sections of the IDLDE interface.

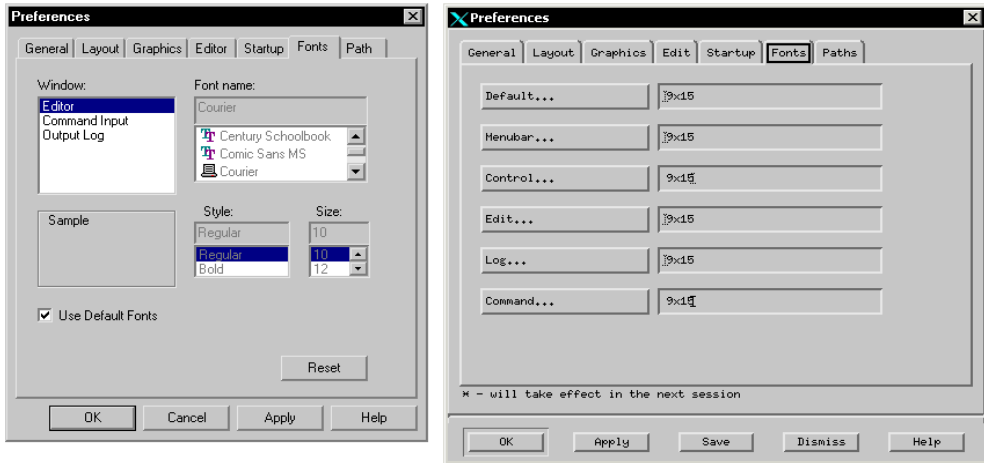


Figure 5-6: Font Preferences Dialog

Microsoft Windows

Under Microsoft Windows, IDL uses a standard Windows font-selection dialog. You can select different fonts for IDL Editor windows, the Command Input Line, and the Output Log. Click on one of these areas in the **Window** list, then select the font, style, and size using the appropriate lists. Click **Use Default Fonts** to change to the IDL default font selections for all three areas.

UNIX

This tab allows you to control which fonts are to be used for the main IDL window. Click on any of the following buttons to specify the relevant font:

- **Default** — dialog boxes
- **Menubar** — menu items
- **Control** — the Control Panel
- **Edit** — editor windows

- **Log** — the Output Log
- **Command** — the Command Input Line

Selecting a Font

Clicking any of the buttons on the **Fonts** tab of the Preferences dialog brings up the **Select Font** dialog. This dialog allows you to select fonts from the X Windows Server font database, based on the attributes *Foundry*, *Family*, *Weight*, *Slant*, *SetWidth*, and *Size*. Using this dialog is similar to using the `xfontsel` X Window utility. See your X Window system font documentation for additional details. Once you have selected a font, click **OK** to accept your selection or **Cancel** to abandon it.

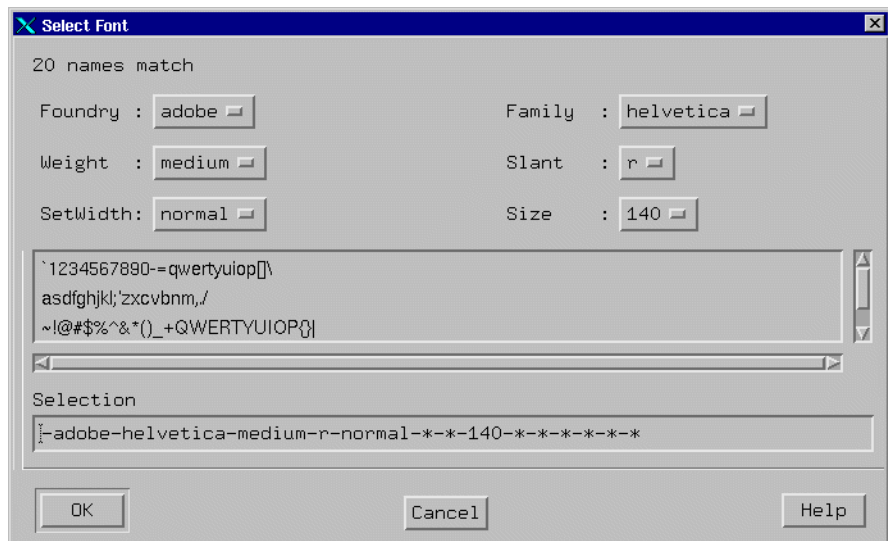


Figure 5-7: Motif Select Font Dialog.

Path Preferences

This tab allows you to control where IDL looks for procedures and functions. The path elements specified in the **Search Path / IDL Files Search Path** are used to set the !PATH system variable when the IDLDE starts.

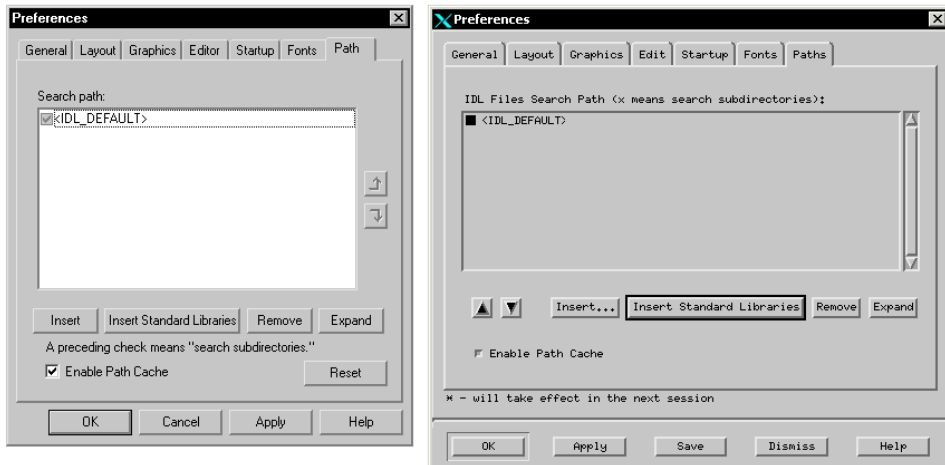


Figure 5-8: Path Preferences Dialog

Note

If you have set the environment variable `IDL_PATH`, IDL will set the `!PATH` system variable based on the contents of the `IDL_PATH` environment variable at startup, overriding any settings made in this dialog. After IDL has started, however, you can modify the current value of the `!PATH` system variable using this dialog. See “`!PATH`” in the *IDL Reference Guide* manual for additional details on how `!PATH` is set.

Search Path / IDL Files Search Path

The IDLDE Path Preferences dialog uses the same mechanism to expand the elements of the **Search Path** field as is used by the `EXPAND_PATH` function. By default, this field is populated with a single entry: `<IDL_DEFAULT>`, indicating that the default IDL path will be used. If the `IDL_PATH` environment variable is *not* set, when the IDLDE starts up, it will expand this token into the default value of the `!PATH` system variable. See “[The Path Definition String](#)” under “[EXPAND_PATH](#)”

in the *IDL Reference Guide* manual for complete details on how this token is expanded.

If the box to the left of a path element is checked, all directories below the listed directory that contain at least one `.pro` or `.sav` file will be included in `!PATH`. (This mechanism is analogous to the use of a “+” symbol in an `EXPAND_PATH` path definition string.)

Note

If the `<IDL_DEFAULT>` entry is present, the box to its left is both checked and greyed out (Windows) or completely blacked out (Motif), indicating that the token will always be expanded.

You can modify the value of the `!PATH` system variable in the following ways using this dialog:

- **Change the order of the path elements** — using the up- and down-arrows, you can reorder the path elements. When searching the directories in the `!PATH` system variable for files, IDL will use the first matching file it finds. If you have multiple files with the same name in different directories within `!PATH`, you may need to adjust the order in which the directories are scanned.
- **Insert...** — To add a path to the **Search Path** list, click **Insert...** to display the Select Directory dialog. The new path is inserted before the first selected path. If none of the paths are selected, the new path is appended to the end of the list.
- **Insert Standard Libraries** — Click **Insert Standard Libraries** to insert the `<IDL_DEFAULT>` path element into the list.
- **Remove** — Click on **Remove** to delete the selected path.
- **Expand** — Click on **Expand** to include the individual subdirectories of the selected path element in the **Search Path** list. When you click **Expand**, the checkmark is removed from the original path element, since the subdirectories are now explicitly included in the path search list.

See “[Running IDL Programs](#)” in Chapter 9 for more information on how `!PATH` is used by IDL when compiling and running programs.

Enable Path Cache

Select **Enable Path Cache** to enable IDL’s path caching mechanism. Path caching is enabled by default, and in almost all cases should be left enabled. See “[PATH_CACHE](#)” in the *IDL Reference Guide* manual for more information about IDL’s path cache.



Chapter 6: Working With Macros

This chapter discusses the following topics:

What are Macros?	116	Creating Windows Macros	121
Using the IDL Macro Toolbar	117	Command Stream Substitutions	123
Creating UNIX Macros	118	Building IDL Example Macros	124

What are Macros?

A macro allows you to execute commonly-used IDL tasks with the press of a mouse button or through a single keystroke ("hot key") combination. In IDL you can create your very own macros using:

- routines
- procedures
- statements
- command stream substitutions

For example you may customize and extend the functionality of the IDL Development Environment (such as writing a procedural macro to change IDL's working directory, which we will see later in this section).

Using the IDL Macro Toolbar

IDL offers several existing macro options on its Macro Toolbar. These macros allow you quick access to commonly used IDL functionality such as printing a variable, importing various file types, and running the IDL Demos.

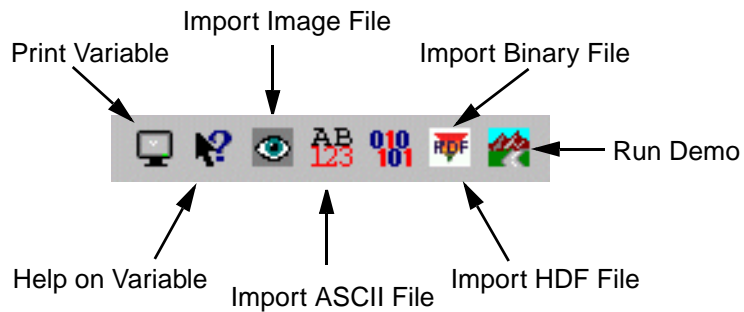


Figure 6-1: The IDLDE's Macro Toolbar

Creating UNIX Macros

You can modify the contents of the **Macros** menu and macros toolbar, either using the **Edit Macros** dialog (displayed by selecting **Edit...** from the **Macros** menu) or by manually editing the user resource (`.idlde`) file.

Using the Edit Macros Dialog

The Edit Macros dialog allows you to add, remove, or modify macros that appear either in the **Macros** menu or the **Macros** toolbar.

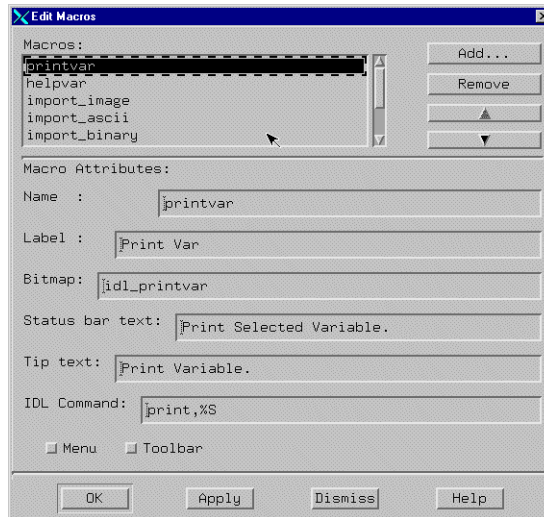


Figure 6-2: The Edit Macros Dialog.

To add a new macro, do the following:

1. Enter a name for your macro in the **Name** field. The Name appears only in the Edit Macros dialog.
2. Enter a label for your macro in the **Label** field. The label will be used in the **Macros** menu (if selected).
3. Enter the name of the bitmap (`.xbm` or `.xpm`) file associated with the macro in the **Bitmap** field. The bitmap will be used on the **Macros** toolbar (if selected). See [“Bitmaps for Control Panel Buttons”](#) on page 119 for details.

4. Enter text to be displayed on the IDLDE status bar in the **Status bar text** field.
5. Enter text to be displayed as a tooltip when the mouse cursor is positioned over the toolbar button in the **Tip text** field.
6. Enter the IDL command to be executed in the **IDL Command** field. See “[Command Stream Substitutions](#)” on page 123 for information on the types of dynamic information that can be included in the command.

In addition to IDL-language commands, you can attach IDL Motif Action Routines to a macro. See “[Action Routines](#)” on page 153 for details.

7. Select the **Menu** checkbox if you want the macro to appear on the **Macros** menu.
8. Select the **Toolbar** checkbox if you want the macro to appear on the Macros toolbar.
9. Click **Add** to add the new macro, then click **OK**.

To Remove an existing macro, select it from the list and click **Remove**. To rearrange macros in the list, use the up- and down-arrow buttons.

Bitmaps for Control Panel Buttons

It is recommended that bitmaps for control panel buttons:

1. Be in either XBM (X11 bitmap file) or XPM (X11 system pixmap file) format, with the file extension `.xbm` or `.xpm`.
2. Supply the full path name to the bitmap file. Alternatively, if the bitmap is located in one of the following directories, you can supply only the base file name:
 - `$IDL_DIR/resource/X11/lib/app_defaults`
 - `$IDL_DIR/resource/X11/lib/app_defaults/bitmaps`
 - `$HOME`
 - `$HOME/bitmaps`

Note

The above directories show the default search path for a bitmap file if nothing other than the root file name is specified in the `.idlde` file.

Manually Editing the Resource File

Although there is little advantage in doing so, you can also modify the **Macros** menu or toolbar by manually editing either your own local IDL resource file or the system-wide resource file. For details, see [“Modifying the Control Panel”](#) on page 150.

Creating Windows Macros

You can modify the contents of the **Macros** menu and macros toolbar using the **Edit Macros** dialog (displayed by selecting **Edit...** from the **Macros** menu). The **Edit Macros** dialog allows you to add, remove, or modify macros that appear either in the **Macros** menu or the **Macros** toolbar.

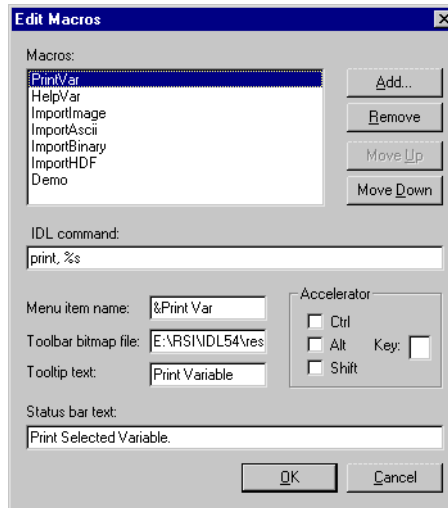


Figure 6-3: The Edit Macros Dialog.

To add a new macro, do the following:

1. Click **Add** and enter a name for your new macro. The name you specify appears only in the Edit Macros dialog.
2. Enter the IDL command to be executed in the **IDL Command** field. See [“Command Stream Substitutions”](#) on page 123 for information on the types of dynamic information that can be included in the command.
3. If you want your macro to be included in the **Macros** menu, enter a label for your macro in the **Menu Item Name** field.
4. If you want your macro to be included in the Macros toolbar, enter the full path name of the bitmap button file in the **Toolbar bitmap file** field. Bitmaps used as macro buttons in IDL must be 16 by 16 pixel `.bmp` files. IDL’s default

bitmaps are stored in the `resources/bitmaps` subdirectory of the IDL distribution.

5. Enter text to be displayed as a tooltip when the mouse cursor is positioned over the toolbar button in the **Tooltip text** field. This value is ignored if no bitmap file is specified.
6. Enter text to be displayed on the IDLDE status bar in the **Status bar text** field.
7. Optionally, in the **Accelerator** field, enter a keystroke shortcut combination for your new macro. Note that you can create a macro that is available only by pressing the keystroke combination if you supply neither a label for the **Macros** menu nor a bitmap for the Macros toolbar.

To Remove an existing macro, select it from the list and click **Remove**. To rearrange macros in the list, use the up- and down-arrow buttons.

Click **OK** to accept your changes or **Cancel** to abandon them.

Command Stream Substitutions

You can use command stream (%) substitutions as shortcuts to incorporate certain types of information into the IDL command for your macro.

Command Stream Substitution	Result
%F	The filename associated with the currently active editor window.
%P	The full path filename associated with the currently active editor window.
%N	The base name of the filename without its path or suffix.
%B	The base name of the filename without its path, but with its suffix.
%S	The currently selected text.
%L	The line number with the current insertion point.
%%	Inserts the “%” character.

Table 6-1: Listing of Useful Command Stream Substitutions

Note

When creating a new macro, you may store the macro in the folder (directory) which IDL has already provided for the existing IDLDE macros. This folder exists in the `lib\macros` directory of your installation directory. If you wish to create a unique folder for the storage of only macros which you have created you may do so.

Building IDL Example Macros

Below are two examples that illustrate how a macro is created in IDL. The first example below is a UNIX-only example; the second example will work on either Microsoft Windows or UNIX.

Creating a Macro to Call a Text Editor in IDL for UNIX

On UNIX platforms, you can create a macro to open a file that is currently open in the IDL Editor in another editor, such as `emacs` or `vi`. Use the following procedure to create the macro:

1. Select **Macros** → **Edit** menu to bring up the **Edit Macros** dialog box.
2. Complete the following fields in the **Edit Macros** dialog:
 - Enter “Edit in emacs” in the **Name** field.
 - Enter “emacs” in the **Label** field.
 - Leave the **Bitmap** field blank. This macro will appear only in the **Macros** menu.
 - Leave the **Status bar text** field blank. This value is used only when a toolbar button is present.
 - Leave the **Tip text** field blank. This value is used only when a toolbar button is present.
 - Enter the following in the **IDL command** field (notice that we are using the `%L` and `%P` command stream substitutions):


```
SPAWN, 'emacs +%L %P &'
```
 - Select the **Menu** checkbox to specify whether the macro will appear in the **Macros** menu.
3. Create the new macro by clicking the **Add** button. The new **emacs** macro is added to the **Macros** menu.

To use this new macro, open the desired file in the IDL editor, then select **emacs** from the **Macros** menu.

To add a macro for editing in `vi`, repeat the above steps, but enter the following in the **IDL command** field:

```
SPAWN, 'xterm -e vi +%L %P &'
```

Note

The IDLDE always checks to determine whether the current file has been externally modified before using it. If a file was modified with an external editor, IDLDE notifies you, and asks you to reload the file before using it. You can also use the **Revert to Saved** option from the **File** menu to reload the file.

Change Working Directory Macro

The following macro will select and change your current working directory. The steps below describe the fields of the Macros dialog on a Microsoft Windows system, but the macro will work equally well on a UNIX system.

First we will create a `.pro` file in IDL which will display a platform-specific directory-selection dialog.

1. From the IDLDE, open a new IDL Editor window by selecting **File** → **New** → **Editor**.
2. Type (or copy) the following lines of code into the new Editor window to form a program:

```
PRO cd_test
dir = DIALOG_PICKFILE(/DIRECTORY)
IF (dir) THEN BEGIN
    PRINT, 'Changing to: ', dir
    CD, dir
ENDIF
END
```

3. Save the file as `cd_test.pro` in a directory included in IDL's path. (The file must be in IDL's path so that IDL will find it automatically when the command `cd_test` is executed by the macro we will create.)
4. Select **Macros** → **Edit** menu to bring up the Edit Macros dialog box.
5. Click **Add** to create a new macro. Enter "Change Directories" as the macro name.

6. Complete the following fields in the Edit Macros dialog:
 - Enter “cd_test” in the **IDL command** field.
 - Enter “Change Directories” in the **Menu item name** field.
 - Leave the **Toolbar bitmap file** field blank. This macro will appear only in the **Macros** menu.
 - Leave the **Tooltip text** field blank. This value is used only when a toolbar button is present.
 - Leave the **Status bar text** field blank. This value is used only when a toolbar button is present.

To use the new macro, select “Change Directories” from the **Macros** menu.



Chapter 7: Printing in IDL

The following topics are covered in this chapter:

Overview	128	IDL Printer Setup in Windows	130
Printing Graphics	129	IDL Printer Setup in UNIX or Mac OS X	131

Overview

IDL allows you two ways to print:

- Printing graphics from the IDL language.
- Printing IDL source code from the **File** menu of the IDLDE.

While these sources are fundamentally different, the methods used to specify and configure a print device according to your operating system are the same. These topics are covered in the following sections.

Printing Graphics

Beginning with IDL version 5.0, IDL interacts with a system-level printer manager to allow printing of both IDL Direct Graphics and IDL Object Graphics. On Windows platforms, IDL uses the operating system's built-in printing facilities; on UNIX platforms, IDL uses the Xprinter print manager from Bristol Technology.

Use the `DIALOG_PRINTERSETUP` and `DIALOG_PRINTJOB` functions to configure your system printer and control individual print jobs from within IDL.

Printing IDL Direct Graphics

To print IDL Direct Graphics, you must first use the `SET_PLOT` procedure to make `PRINTER` your current device. Issue IDL commands as normal to create the graphics you wish to print, then use the `CLOSE_DOCUMENT` keyword to `DEVICE` to actually initiate the print job and print something from your printer. You can also create multiple pages before closing the document as well as being able to use tile graphics with the `!P.MULTI` system command. See “[IDL Graphics Devices](#)” in Appendix A of the *IDL Reference Guide* manual for details.

Printing IDL Object Graphics

To print IDL Object Graphics, you must create a printer object to use as a destination for your Draw operations. You can also print multiple documents with the `IDLgrPrinter` object. See “[Printer Objects](#)” on page 687 for information about printer objects and their use. Also see “[Bitmap and Vector Graphic Output](#)” on page 695 for a complete discussion of when to output to bitmap or vector graphics based on picture content.

IDL Printer Setup in Windows

Setting up a printer in IDL for Windows uses the common Windows Printer Setup dialog. For more information on setting up a Printer on Windows, see your Windows operating system documentation or support.

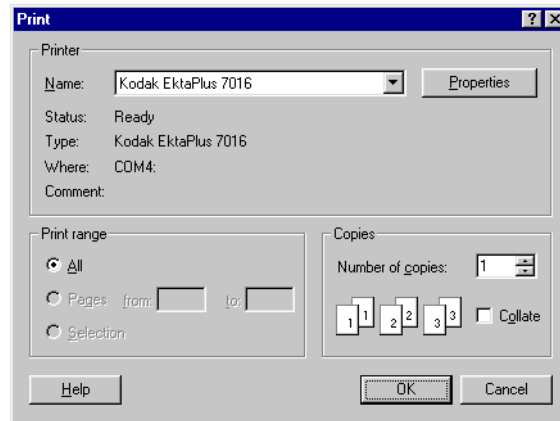


Figure 7-1: Common Printer Setup Dialog in Windows

IDL Printer Setup in UNIX or Mac OS X

IDL for UNIX uses the Xprinter print technology from Bristol Technology to create and output information to a wide variety of printers. This section describes the Xprinter setup dialogs.

The Xprinter Setup Dialog

The Xprinter Setup dialog allows you to select model-specific printer options such as paper trays, paper size, page orientation, and the UNIX print spooler command. Printer options are saved in the `$HOME/.XprinterDefaults` file. Once configured, the desired information is saved to the file system and used in future IDL sessions.

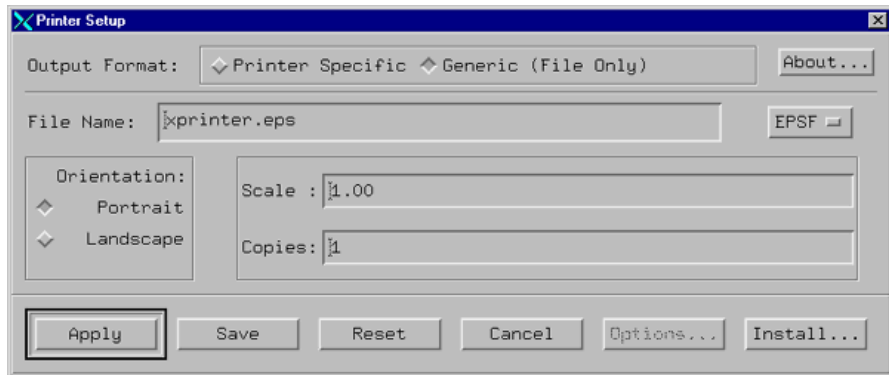


Figure 7-2: The Printer Setup Dialog

Printer Setup Dialog Buttons

The action area of the Printer Setup dialog contains six buttons:

Button	Description
OK	Writes current configuration information to your default printer information file <code>\$HOME/.XprinterDefaults</code> . This button also dismisses the dialog.

Table 7-1: Printer Setup Dialog Buttons

Button	Description
Save	Writes current configuration information to your default printer information file <code>\$HOME/.XprinterDefaults</code> .
Reset	Reloads default configuration from <code>\$HOME/.XprinterDefaults</code> .
Cancel	Closes dialog and cancels all configuration changes.
Options	Displays the options dialog box that lets you select an alternate printer setup. This button is disabled if output is configured to be sent to a file instead of a printer.
Install	Displays the installation dialog box that allows you to add or remove printer devices and printer ports from the <code>\$HOME/.XprinterDefaults</code> file.

Table 7-1: Printer Setup Dialog Buttons (Continued)

Configuring Printer Setup Options

Specify the following options on the initial Printer Setup dialog:

Option	Description
Output Format:	Specify whether to send output to a file or a printer. If you choose Printer Specific, you can send output to any printer type/port combination configured in your <code>\$HOME/.XprinterDefaults</code> file. If the port is FILE:, Xprinter creates an output file for the specified printer type. If you choose Generic (File Only), print output is sent to an Encapsulated PostScript or generic PCL file.
Printer:	This field appears only if you select Output Format: Printer Specific. It specifies the name of the default printer type/port to which to send print output. Click the Options button to specify a different printer type/port combination.

Table 7-2: Specifying Printer Setup Options

Option	Description
File Name:	This field appears only if you choose Output Format: Generic (File Only). Type the name of the print file you wish to create. To pipe print output to a command, enter a ! character as the first character and then specify the command to which to send output. For example, to send output to the lp command, enter the following: !lp
EPSF PCL4 PCL5	This field only appears if you select Output Format: File. Click this button to display a list of output file types and select the desired type. Available types are EPSF (Encapsulated PostScript), PCL4, and PCL5.
Orientation	Specify portrait or landscape.
Scale	To increase the size of the output, specify a value greater than 1.00. To reduce the size, specify a value less than 1.00. For example, a value of 2.00 would double the size of the output; a value of 0.50 would reduce it by half.
Copies	Specify the number of copies to print.

Table 7-2: Specifying Printer Setup Options (Continued)

To set additional options, such as selecting a different printer or changing the page size, click the **Options** button. The Options dialog appears.

The Options dialog is only available when sending output to a printer.

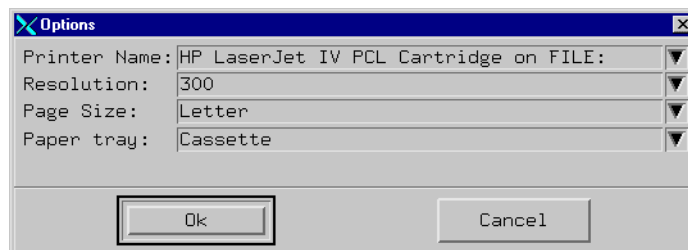


Figure 7-3: The Options Dialog

Use this dialog to set the Printer Setup options:

Option	Description
Printer Name	Use this field to select the current printer. Click the down arrow to display a list of configured printers.
Resolution	Specify printer resolution with this field. Values vary depending on printer.
Page Size	Specify paper size with this field. Values vary depending on printer.
Paper tray	Specify paper tray with this field. Values vary depending on printer.
Duplex	Specify duplex options (if the selected printer supports duplex printing). Valid values include None (no duplex printing), Duplex Tumble (flips over the short edge), and Duplex No Tumble (flips over the long edge). If the selected printer does not support duplexing, this field is disabled.

Table 7-3: The Printer Setup Options

Adding a New Printer to the List of Printer Choices

To add a new printer to your list of available printers:

- Define a port, which is an alias for the print command.
- Associate the port with the printer's PPD file.

Defining a New Port

To define a new port using the Printer Setup dialog:

- Display the Ports dialog. From the Printer Setup dialog, select **Install**, **Add Printer**, and **Define New Port**.

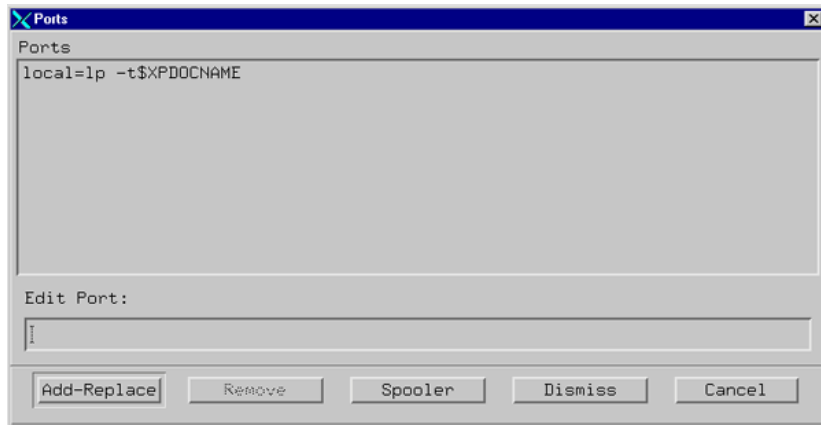


Figure 7-4: Defining a New Port

Type the port definition in the **Edit Port** edit box. Port definitions have the following format:

```
port=print_command
```

The `print_command` is the command for sending output to the printer port. If you were to have two printers named ORION and SIRIUS for example, the definitions would appear as follows:

```
ORION=rsh bandit "lp -d ps"
SIRIUS=rsh bandit "lp -d ps -T pcl5"
```

Both printers here are connected to the system `bandit`, so the print command is a remote shell command executed on `bandit`. ORION is a PostScript printer, so the command `lp -d ps` is executed on `bandit` to print to ORION. SIRIUS though is a PCL5 printer, so the print command executed on `bandit` to print to SIRIUS is `lp -d ps -T pcl5`.

- Click **Add/Replace** and the new port is now included in the list of current port definitions.
- Repeat the above step for each printer to which you wish to send output.

Note

To create a printer port for each available queue on hp700 systems, click the Spooler button on the Ports dialog. This command creates a default printer port for each available printer queue returned by the `lpstat -a` command.

Modifying an Existing Port

In order to modify an existing port using the Printer Setup dialog:

- Display the Ports dialog. From the Printer Setup dialog, click **Install, Add Printer**, and **Define New Port**.
- Select the port you wish to modify and edit the port information in the **Edit Port** edit box.
- Click **Add/Replace**. The modified port is now included in the list of current port definitions.

Matching a Printer Device to a Port

In order to match a printer device to a port using the Printer Setup dialog:

- Display the Add Printer dialog. From the Printer Setup dialog, click **Install** and **Add Printer**.
- In the **Printer Devices** field, select the description that matches the printer you are to install. If no description matches this printer, contact your printer vendor for a printer description (PPD) file.

- Select the desired port in the Current Port Definitions list box and click Add Selected. The new printer is now included in the list of currently installed printers.

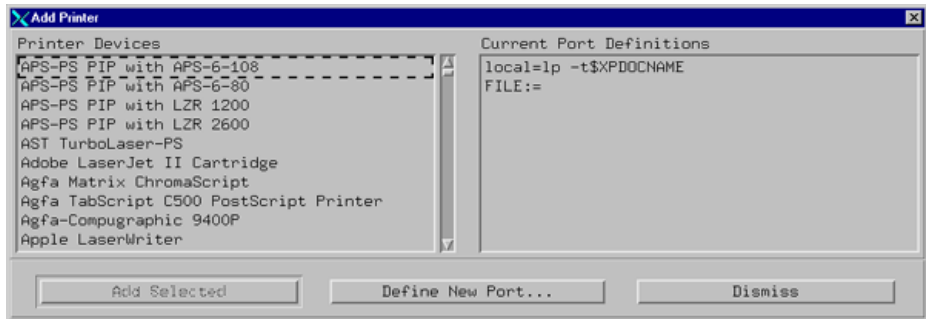


Figure 7-5: Adding a Printer

Removing an Installed Printer

In order to remove a printer device/port combination using the Printer Setup dialog:

- Display the Printer Installation dialog. From the Printer Setup dialog, click **Install**.
- In the **Currently Installed Printers** list box, select the printer you wish to remove and click on **Remove Selected**.

Manually Modifying Default Printer Setup Values

Xprinter retrieves default printer setup information from the file `.XprinterDefaults` in your home directory. If this file does not exist, Xprinter reads the information from the file `$XPHOME/Xprinter/XprinterDefaults` or `$XPPATH/XprinterDefaults`.

Note

For IDL, `$XPATH` is set to `$IDL_DIR/resource/xprinter`.

The Xprinter Printer Setup dialog writes modifications to the default information in `$HOME/.XprinterDefaults`. However, it never modifies the default information in the file `$XPHOME/XprinterDefaults` or `$XPPATH/XprinterDefaults`. If the file `$HOME/.XprinterDefaults` does not already exist, the Xprinter Printer Setup dialog creates it.

Although the most common way to modify the default Printer Setup is using the Printer Setup dialog, which updates `$HOME/.XprinterDefaults` automatically, you may also edit this file with any text editor and make changes directly.

You may also set up the `$HOME/.XprinterDefaults` file to do the following:

- Define printer ports.
- Match printer types to defined ports.
- Specify the default printer.
- Specify printer-specific options.

Defining a Port

A printer port is an alias for the print command. It is defined in the [ports] section of `$HOME/.Xpdfaults` and appears as part of the Printer Name in the Printer Setup dialog. For instance, the following is the first Printer Name in the Printer Setup dialog before you make any changes to `$HOME/.XprinterDefaults`:

```
AppleLaserWriter v23.0 PostScript on FILE:
```

For this Printer Name, `FILE:` is the port name. To send output to a printer instead of a file, you first must define a port for each printer to which you wish to direct output. Port entries in the [ports] section have this format:

```
port=print_command
```

The `print_command` is the command for sending output to the printer port. For instance, if you have two printers (ORION and SIRIUS), your [ports] section may appear as follows:

```
[ports]
ORION=rsh bandit "lp -d ps"
SIRIUS=rsh bandit "lp -d ps -T pcl5"
```

In the above, both printers are connected to the system `bandit`, so the print command is a remote shell command executed on `bandit`. `ORION` is a PostScript printer, so the command `lp -d ps` is executed on `bandit` to print to `ORION`. `SIRIUS`, though, is a PCL5 printer, and thus the print command executed on `bandit` to print to `SIRIUS` is `lp -d ps -T pcl5`.

If a printer is connected to your local system, you will need to add an entry for that printer as well. For the local printer, your entry should be like the following:

```
[ports]
ORION=rsh bandit "lp -d ps"
SIRIUS=rsh bandit "lp -d ps -T pcl15"
LOCAL=lp -d ps
```

Your printer port can be any name you choose except FILE:, which is the only reserved port name. It causes Xprinter to create a print file formatted specifically for the specified printer type.

An entry must be created in the [ports] section for every printer to which you wish to be able to print.

Matching a Printer Type to a Defined Port

After you have defined a port for each printer, you must tell Xprinter what type of printer is associated with each port. List device types in the [devices] section of the .XprinterDefaults file. Each entry in the [devices] section has the following format:

```
alias=PPD_file driver,port
```

Note

There must be a space between the PPD_file and driver and a comma between the driver and the port. The following table describes each part of this entry.

Field	Description
alias	The alias is a descriptive name used to identify the printer. It can be anything you choose. The alias is the name which appears in the Printer Setup dialog (such as HP LaserJet III SI PostScript).
PPD_file	The PPD_file is the name of the printer description (PPD) file used by the printer, without a .PPD extension. Search in the directory \$XPHOME/xprinter/ppds/ to find the PPD file for your printer.
driver	The driver is the type of driver your printer uses. Value values are PostScript, PCL4, and PCL5.
port	The port is the printer port as listed in the [ports] section of the .XprinterDefaults file (ORION, SIRIUS, and LOCAL in the example [ports] section).

Table 7-4: Associating a Printer with a Port

Here's an example configuring three printers:

Port	Printer Type	Output Type
ORION	HP LaserJet IIISi PostScript v52.3	PostScript
SIRIUS	HP LaserJet 4M PCL Cartridge	PCL
LOCAL	QMS-PS 2200 v52.3	PostScript

Table 7-5: Example Configuration

First, be sure to choose an alias for each printer. In order to make it simpler to identify the printer from the Printer Setup dialog you wish to use, you may use the following aliases:

```
HP LaserJet PS
HP LaserJet PCL
QMS PS
```

It is important to note that if you utilize the Printer Setup dialog to associate ports and PPD files, you cannot specify a printer alias. You must instead choose an alias from the predefined listing that appears in the Printer Devices list box in the Add Printer dialog. The corresponding PPD file is already associated with the printer aliases in this list box.

Now, identify the PPD file associated with each of these printers.

Thus the [devices] section of the `.XprinterDefaults` file would be as follows:

```
[devices]
HP LaserJet PS=HP3SI523 PostScript,ORION
HP LaserJet PCL=HP4M PCL,SIRIUS
QMS PS=Q2200523 PostScript,LOCAL
```

After these entries have been added to your `.XprinterDefaults` file, the following printer choices are available from the Printer Setup dialog:

```
HP LaserJet PS on ORION
HP LaserJet PCL on SIRIUS
QMS PS on LOCAL
```

Specifying a Default Printer

After you have configured all available printers, you may select one of them as the default printer. To make a specific printer the default printer on the Printer Setup dialog, add an entry (in the following format) to the [windows] section of the `.XprinterDefaults` file:

```
[windows]
device=PPD_file,driver,port
```

Simply provide the same information that you used in the [devices] section. Only the format of the entry is different; there is a comma between the PPD_file and the driver instead of a space.

For example, suppose you wish the default printer to be the printer at port ORION. The [windows] section would appear as follows:

```
[windows]
device=HP3SI523,PostScript,ORION
[ports]
ORION=rsh bandit "lp -d ps"
SIRIUS=rsh bandit "lp -d ps -T pcl5"
LOCAL=lp -d ps
[devices]
HP LaserJet PS=HP3SI523 PostScript,ORION
HP LaserJet PCL=HP4M PCL,SIRIUS
QMS PS=Q2200523 PostScript,LOCAL
```

In your default .XprinterDefaults file, the [windows] entry appears:

```
[windows]
device=NULL,PostScript,FILE:
```

Since no PPD file is listed (NULL), the default on the Printer Setup dialog is to print generic PostScript to a file. You may specify the filename and change the type of output to PCL on the Printer Setup dialog.

Specifying Printer-Specific Options

You may include a section that lists the default printer-specific options for each printer defined in the devices section. The options available vary between differing printers, but typical options include number of copies, page size, paper tray, and orientation. An example follows of a printer-specific section for a default printer in the example .XprinterDefaults file:

```
[HP3SI523,PostScript]
Scale=0.80
Copies=1
PaperTray=Lower
PageSize=Letter
Orientation=Portrait
DPI=300
```




Chapter 8: Customizing IDL on Motif Systems

This chapter describes techniques for customizing versions of IDL running under the X Window System (Motif) graphical user interface.

Using X Resources to Customize IDL . . .	144	Action Routines	153
Command Line Options	147	CDE File Manager Support	160
Modifying the Control Panel	150		

Using X Resources to Customize IDL

X Resources in Brief

The component widgets of an X Window application each have two names, a class name that identifies its type (e.g., XmText for the Motif text widget) and an instance name (e.g., command, the name of the IDLDE command input text widget). The class name can be used to set resources for an entire class of widgets (e.g., to make all text widgets have a black background) while the instance name is used for control of individual widgets (e.g., set the IDLDE command input window font without affecting other widgets).

Applications consist of a tree of widgets, each having a class name and an instance name. To specify a resource for a given widget, list the names of the widgets lying between the top widget and the target widget from left to right, separated by periods. In a moderately complicated widget hierarchy, only some of the widgets are of interest; there are intervening widgets that serve uninteresting purposes (such as a base that holds other widgets). A star (*) character can be used as a wildcard to skip such widgets. Another fact to keep in mind is that a given resource specification is interpreted as broadly as possible to apply to any widget matching that description. This allows a very small set of resource specifications to affect a large number of widgets.

Resource Files

There are two resource files used to customize the IDL Development Environment. An installation-wide resource file called `idl` is located in

```
$IDL_DIR/resource/X11/lib/app-defaults
```

and a user resource file called `.idlde` is located in your home directory.

Modifying the global `idl` resource file effects an installation-wide customization. Changes to the `idl` file will be lost with a new installation.

The user resource file, `.idlde`, customizes individual versions of IDLDE and is divided into two sections. The first section contains user-defined customization resources. You can place comments starting with “!” or “!!!” in the first section of `.idlde`. When newer versions of `.idlde` are written, system comments are prefixed with “!!!”. The second section of `.idlde` is used to store preferences; it is modified when preferences are saved and shouldn’t be modified manually.

If you use IDL in command-line mode rather than via the IDL Development Environment, you can include resources in the `.Xresources` file located in your home directory.

Format of IDL Resources

All IDL resource strings begin with the characters “`Idl`”. For example, the resource

```
Idl.colors
```

controls the number of entries of the system shared colormap IDL will use.

Resource strings that apply only to the IDL Development Environment begin with the characters “`Idl*idlde`”. For example, the resource

```
Idl*idlde*hideCommand
```

controls whether the IDLDE Command Input Line is visible when IDL starts up.

Resources that include the string “`idlde`” must be included either in the system-wide `Idl` resource file, or in a `.idlde` file in your home directory. Resources that apply to IDL whether it is running in command-line mode or via the IDLDE can be included in either the system-wide `Idl` resource file or in a `.Xresources` file in your home directory.

To specify a value for an X resource, append a colon character and the value after the resource string. Whitespace is ignored. For example:

```
Idl.colors:-10
```

is the same as

```
Idl.colors:           -10
```

X Resources Used by IDL

IDL uses a large number of resources to control the behavior and appearance of the IDL Development Environment and any graphical application written in IDL. To learn more about the specific resources used, or to modify individual values, inspect the installation-wide resource file `Idl`, located in

```
$IDL_DIR/resource/X11/lib/app-defaults
```

Note

RSI suggests that you make any changes to the IDL resources in a user-specific `.idlde` file or `.Xresources` file.

Reserving Colors

When IDL starts, it attempts to secure entries in the shared system color map for use when drawing graphics. If the entry `Idl.colors` exists in the `Idl` resource file, IDL will attempt to allocate the number of colors specified from the shared colormap. If for some reason it cannot allocate the requested number of colors from the shared colormap, IDL will create a private colormap. Using a private colormap ensures that IDL has the number of colormap entries necessary, but can lead to colormap flashing when the cursor or window focus moves between IDL and other applications.

One way to avoid creating a private colormap for IDL is to set the `Idl.colors` resource equal to a negative number. This causes IDL to try to use the shared colormap, allocating all but the specified number of colors. For example:

```
Idl.colors: -10
```

instructs IDL to allocate all but 10 of the currently available colors for its use. Thus, if there are a total of 220 colors not yet reserved by other applications (such as the windowing system), IDL will allocate 210 colors from the shared colormap.

The IDLDE application itself uses between 10-15 colors. On startup, the IDLDE will attempt to use colors in the shared colormap, but will reserve colors for itself if appropriate matching colors in the shared colormap are not found. As a result, running IDL with the IDLDE may use more colors than running IDL with the `tty` (plain command line) interface.

If you experience colormap flashing when using the IDLDE, but not when you use the plain `tty` interface, try adjusting the number of colors used by the IDLDE interactively, using the `-colors` startup flag. For example,

```
idlde -colors -15
```

starts the IDLDE and allocates all but 15 of the currently available colors. When you find an appropriate number of colors to reserve, you can set the `idlde.colors` resource in the `Idl` resource file or in your personal `.idlde` file accordingly.

Command Line Options

The IDLDE can also be customized from the command line using the command line flags described below. Command line flags are given precedence over global resource files (`Idl`) and user resource files (`.idlde`). For more information about resources, see [“Using X Resources to Customize IDL”](#) on page 144.

Example

Type the following at the operating system command line to start the UNIX IDLDE using separate main-level windows to display files:

```
idlde -multi
```

The available command line flags follow:

```
-e file [-e file1 -e file2...]
```

Opens specified files at startup.

-nocommand

Hides the Output Log window and Command Input Line at startup. The related resource is `Idl*idlde*hideCommand: True`.

-command

Displays Log window and Command Input window at startup. The related resource is `Idl*idlde*hideCommand: False`.

-nocontrol

Hides the Control panel buttons at startup. The related resource is `Idl*idlde*hideControl: True`.

-control

Displays the Control Panel buttons at startup. The related resource is `Idl*idlde*hideControl: False`.

-nolog

Hides the Output Log at startup. The related resource is `Idl*idlde*hideLog: True`.

-log

Displays the Output Log at startup. The related resource is `Idl*idlde*hideLog: False`.

-nostartup

Does not execute startup file on startup (including `IDL_STARTUP`). The related resource is `Idl*idlde.noStartupFile: True`.

-startup

Executes startup file on startup (including `IDL_STARTUP`). The related resource is `Idl*idlde.noStartupFile: False`.

-startupfile "file"

Executes **file** at startup (overrides `IDL_STARTUP` environment variable). If **startupfile** is not specified, the environment variable `IDL_STARTUP` is used as the startup file (if defined). The related resource is `Idl*idlde.startupFile: file` where `file` is the full path name of the startup file.

-nostatus

Hides the Status Bar at startup. The related resource is `Idl*idlde*hideStatus: True`.

-status

Displays the Status Bar at startup. The related resource is `Idl*idlde*hideStatus: False`.

-path "path"

Append **path** to the IDL path (defined using `IDL_PATH` environment variable). The related resource is `Idl*idlde.path: path` where `path` is the full path to be appended.

-quiet

Inhibits display of the IDL startup announcement and message of the day (motd) file.

-readonly

Opens files as read-only. The related resource is `Idl*idlde.readOnly: True`.

-readwrite

Open files as read-writeable. The related resource is `Idl*idlde.readOnly: False`.

-single

Displays files in a single window, which is a child of the main IDLDE window. The related resource is `Idl*idlde*multiWindowEdit: False`.

-multi

Displays files in multiple windows, each one in a separate main level window. The related resource is `Idl*idlde*multiWindowEdit: True`.

-view

Displays the Multiple Document Panel in single window mode at startup. The related resource is `Idl*idlde*hideView: False`.

-noview

Hides the Multiple Document Panel at startup. The related resource is `Idl*idlde*hideView: True`.

-title "Title"

Use **Title** as the title of the main IDLDE window. The related resource is `idlde.title`.

Modifying the Control Panel

The **Control Panel**, with the resource name `control`, is located below the IDL Development Environment **Menu** bar. The Control Panel bar is a `RowColumn` widget containing buttons which serve as shortcuts for common commands.

You can modify the existing Control Panel settings by editing the `idlde*control` values in the system-wide `idl` resource file or overriding those settings in your local `.idlde` file. In addition, you can add buttons to the **Macros** toolbar or menu by adding resources to your `.idlde` file.

Note

If you wish to add, modify, or remove the buttons on the **Macros** toolbar or menu, you can do so via the IDLDE interface using the **Edit Macros** dialog. See [“Creating UNIX Macros”](#) on page 118 for details. Whether you modify your macros using the dialog or by editing a resource file manually, the results are the same. There is little advantage to adding macros to the `.idlde` file manually.

Adding Macros Toolbar Buttons

The `idlButtonsUser` resource defines the *resource name* for each button on the **Macros** toolbar in the Control Panel. The resource name details button attributes, such as its label or pixmap, its associated IDL command, and its status bar message.

To add a button to the **Macros** toolbar, make the following modifications to the `.idlde` file:

- Add a new name to the `idlde*control*idlButtonsUser` list. The buttons are created in the order specified.
- Add `idlde*control*<new button>*labelString` or `labelPixmap` resources (or both). These resources define the button text or image. If you choose to use a pixmap label, be sure the file you specify abides by the restrictions described in [“Bitmaps for Control Panel Buttons”](#) on page 119.
- Add an `idlde*control*<new button>*idlCommand` resource. This is the text of the IDL command to execute. You can also include command stream substitutions; see [“Command Stream Substitutions”](#) on page 123 for details.

Alternatively, you can add an `idlAction` resource. See [“Action Routines”](#) on page 153 for details.

- Add an `idlde*control*<new button>*hint` resource. This is the text that appears in the Status Bar when the cursor is positioned over the new button.
- Add an `idlde*control*<new button>*tip` resource. This is the text that appears as a “tooltip” when the cursor is positioned over the new button.

If you want your changes to be available to all users on the system, you can also modify the system-wide `Idl` resource file, located in the following directory:

```
$IDL_DIR/resource/X11/lib/app-defaults
```

Adding Macros Menu Entries

To add entries into the **Macros** menu, follow the same steps outlined above, modifying the `idlde*menubar*macrosMenu*macrosListUser` resource and substituting `idlde*menubar*macrosMenu*<new menu item>` for `idlde*control*<new button>` in the above steps.

Examples

To add a button called **Reset All** to the **Control Panel** with a color pixmap stored in the file `resetall.xpm` located in your home directory, add the following resources to the `.idlde` file in your `$HOME` directory:

```
idlde*control*idlButtonsUser: <exiting buttons> resetall
idlde*control*resetall*labelPixmap: resetall.xpm
idlde*control*resetall*labelString: Reset All
idlde*control*resetall*idlCommand:\
RETALL & WIDGET_CONTROL,/RESET
idlde*control*resetall*statusString:\
Stop execution of the current code and return to\
the main programming level
```

Note that in this example the new button is added at the end of the list of existing buttons. You can locate the new button anywhere in the list.

To specify a pixmap located in particular directory, specify the full file path of the pixmap file, for example:

```
idlde*control*resetall*labelPixmap:\
/home/user/bitmaps/resetall.xpm
```

To create two rows of the Control Panel from the default of one row, set the `numColumns` resource to 2:

```
idlde*control*numColumns: 2
```

To use label (text) buttons in the Control Panel set `labelType` to `XmSTRING`. To use icon (graphics) buttons set `labelType` to `XmPIXMAP`.

```
idlde*control*labelType: XmSTRING  
or  
idlde*control*labelType: XmPIXMAP
```


Action Routines

Most Motif widgets supply action routines which can be bound to events (such as keypress events). Action routines provided by IDL can be used to define commands for Control Panel buttons or menu items by using the `idlAction` resource.

The following action routines can be used in the same manner as the IDL commands specified in an `idlCommand` resource. The syntax to add an action routine to a control panel button is:

```
Idl*idlde*control*buttonName*idlAction: Action
```

or

```
Idl*idlde*control*buttonName*idlAction: Action(Arguments)
```

where *buttonName* is the name of the button and *Action* is the name of the action routine. Arguments to the action routine, if require, are enclosed in parentheses.

IdlBreakpoint

Use `IdlBreakpoint` to control the placement of breakpoints. If no parameter is specified, the breakpoint is set on the current line. At least one of the arguments from the following table must be set:

Argument	Action
SET	Set a breakpoint on the current line.
CLEAR	Clear the breakpoint on the current line.
TOGGLE	Toggle (SET or CLEAR) the state of the breakpoint on the current line.
COMPLEX	Display breakpoint dialog to set a complex breakpoint.
LIST	List all currently set breakpoints

Table 8-1: Breakpoint Arguments

For example, to use this action routine to clear a breakpoint, the *Action* specified would be:

```
IdlBreakpoint(CLEAR)
```

IdlClearLog

Use `IdlClearLog` to erase the contents of the Output Log.

IdlClearView

Use `IdlClearView` to clear the contents of the currently-active file in the Multiple Document Panel.

IdlCommandHide

Use `IdlCommandHide` to hide or expose the Command Area, which includes the Command Input Line and the Output Log. One of the following arguments must be set: **Show**, **Hide**, or **Toggle**.

IdlCompile

Use `IdlCompile` to compile the file in the currently-active editor window. One of the arguments from the following table must be set:

Argument	Action
FILE	Compiles the currently-active file.
TEMPORARY	Compiles the currently-active file into a temporary file
RESOLVE	Resolves all referenced and uncompiled IDL routines

Table 8-2: Compiling Arguments

IdlControlHide

Use `IdlControlHide` to hide or expose the Control Panel. One of the following arguments must be set: **Show**, **Hide**, or **Toggle**.

IdlEdit

Use `IdlEdit` to manipulate the contents of the currently-selected editor window. One of the arguments from the following table must be set:

Argument	Action
UNDO	Undo previous editing action.
REDO	Redo previously undone action.
CUT	Remove currently-selected text to UNIX clipboard.
COPY	Copy currently-selected text to UNIX clipboard.
PASTE	Paste contents of UNIX clipboard at current insertion point.
SELECTALL	Select all of the text in the currently-selected editor window.
GOTODEF	Display the definition of the currently-selected procedure or function.
GOTOLINE	Move directly to the specified line number.

Table 8-3: Editor Window Editing Arguments

IdlEditMacros

Use `IdlEditMacros` to display the **Edit Macros** dialog.

IdlExit

Use `IdlExit` to cause IDLDE to act as though the EXIT command has been entered. Note that this is usually tied to a menu accelerator (**Ctrl-Q** in this case), so this routine is rarely called directly.

IdlFile

Use `IdlFile` to manipulate the currently-selected editor window. One of the arguments in the following table must be set:

Argument	Action
NEW	Creates a new editor window.
OPEN	Opens an existing file.
SAVE	Saves the contents of the currently-selected editor window.
PRINT	Prints the contents of the currently-selected editor window.

Table 8-4: Editor Window Arguments

IdlFileReadOnly

Use `IdlFileReadOnly` to specify the read/write status of the currently-active editor window. One of the arguments from the following table must be set:

Argument	Action
READONLY	Disable editing of the currently-selected editor window.
READWRITE	Enables editing of the currently-selected window.

Table 8-5: Read/Write Arguments

IdlFunctionKey

Use `IdlFunctionKey` to allow entry of an IDL command into the input command stream. It is typically used to tie IDL commands to function keys. For example:

```
<Key>F5:IdlFunctionKey("print, 'F5 pressed')\n
```

IdlInterrupt

Use `IdlInterrupt` to cause IDLDE to receive an interrupt. Note that this is usually tied to Ctrl-C as a menu accelerator.

IdlListStack

Use `IdlListStack` to display the current nesting of procedures and functions (calling stack).

IdlLogHide

Use `IdlLogHide` to hide or expose the Output Log. One of the following arguments must be set: **Show**, **Hide**, or **Toggle**.

IdlRecallCommand

Use `IdlRecallCommand` to recalls previously entered commands into the command widget. Either the **BACK** or the **FORWARD** argument must be specified to indicate the direction of the recall. For example:

```
<Key>osfUp:IdlRecallCommand(BACK)\n
```

IdlReset

Use `IdlReset` to reset the IDL environment.

IdlRun

Use `IdlRun` to execute the currently-active file.

IdlSearch

Use `IdlSearch` to call the **Find** dialog for a search of the current **Multiple Document Panel**. One of the optional arguments from the following table may be used:

Argument	Action
FIND	Displays a search dialog (default).
FINDAGAIN	Finds the next occurrence of the specified string.

Table 8-6: Find Dialog Arguments

Argument	Action
FINDSELECTION	Finds next occurrence of the current selection.
ENTERSELECTION	Enters the current selection as the search string in the Find dialog.
REPLACE	Replaces the search string, with a specified replacement string.
REPLACEFIND	Finds the next occurrence of the search string, and replaces it with the specified replacement string.

Table 8-6: Find Dialog Arguments

IdlStatusHide

Use `IdlStatusHide` to hide or expose the Status Bar. One of the following arguments must be set: **Show**, **Hide**, or **Toggle**.

IdlStep

Use `IdlStep` to control statement execution for debugging. At least one of the arguments from the following table must be set.

Argument	Action
INTO	Executes a single statement in the current program. If nested procedures or functions are encountered, they are also executed in single-statement mode.
OVER	Executes a single statement in the current program. If nested procedures or functions are encountered, they are run until completion, whereupon interactive control returns.
OUT	Continues execution until current routine returns.

Table 8-7: Debugging Arguments

Argument	Action
SKIP	Skips one statement and executes following statement.
CONTINUE	Continues execution of an interrupted program.
TOCURSOR	Executes file until encountering the cursor.
TORETURN	Executes file until encountering the return.

Table 8-7: Debugging Arguments

IdlTrace

Use `IdlTrace` to display a dialog box to control program tracing.

IdlViewHide

Use `IdlViewHide` to hide or expose the **Multiple Document Panel**. One of the following arguments must be set: **Show**, **Hide**, or **Toggle**.

IdlWindows

Use `IdlWindows` to manipulate the state of the Editor windows. One of the arguments from the following table must be set:

Argument	Action
CASCADE	Arrange open windows in a staggered, overlapping fashion.
TILE	Arrange all windows in a non-overlapping fashion.
MULTI	Open windows outside the IDLDE interface.
SINGLE	Display the most recent window on the Multiple Document Panel.

Table 8-8: Editor Window Display Arguments

CDE File Manager Support

This section provides information on setting up the IDL Development Environment to support the Common Desktop Environment (CDE) File Manager for HP-UX, Solaris, and Tru64 UNIX. Setting up the IDLDE to provide CDE File Manager support allows you to do the following:

- IDL `.pro` files can be dragged and dropped from the CDE File Manager onto the IDLDE.
- Double-clicking on an IDL `.pro` file opens the file in the IDLDE.

During installation, the CDE Action File, `Idl.dt`, is installed in `/etc/dt/appconfig/types/C/` on the local machine, if this directory has write permissions. This file provides the necessary support for the CDE File Manager on that machine. However, if IDL is installed on a server and is run on various client machines, the CDE Action File must be manually copied to each client machine on which you want to provide support for the CDE File Manager. This file must also be copied to the server if the `/etc/dt/appconfig/types/C/` directory did not have write permissions at installation time. If the local machine does not contain the `Idl.dt` file in the `/etc/dt/appconfig/types/C/` directory, do the following:

1. Locate the `Idl.dt` file in the directory `$IDL_DIR/bin`.
2. Copy `Idl.dt` to the `/etc/dt/appconfig/types/C/` directory on the local machine. If the `/etc/dt/appconfig/types/C/` directory does not exist, copy the file to the `/usr/dt/appconfig/types/C/` directory.
3. In the `Idl.dt` file, verify that the path to the `RSI_Dir/bin` directory in the `ACTION Idl` section is correct for the local machine. For example, if IDL is located in the `/usr/local/rsi/` directory, the `ACTION Idl` section should look like this:

```
ACTION Idl
{
    LABEL                Idl
    TYPE                 COMMAND
    EXEC_STRING          /bin/sh -c `
/usr/local/rsi/idl_setup.ksh:idl -c -e %Arg_1%`
    ICON                 Dtactn
    WINDOW_TYPE         NO_STDIO
    DESCRIPTION         Start Idl Development Environment
}
```

4. Logout and log back in to the CDE.



Chapter 9: Creating and Running Programs in IDL

The following topics are covered in this chapter:

Overview	162	Aborting IDL	170
Running IDL Programs	164	Saving Variables from an IDL Session ..	171
Interrupting Program Execution	168	For More Information on Programming ..	173
Variable Context After Interruption	169		

Overview

In addition to being a useful interactive data analysis tool, IDL is a powerful programming language. Many of IDL's programming language features and constructs can be used either interactively at the IDL command line or as part of a larger program — which can itself be invoked at the IDL command line or by other programs.

IDL programs can take any of the following forms:

Batch Files

A *batch file* contains one or more IDL statements or commands. Each line of the file is read and executed before proceeding to the next line. This makes batch files different from main-level programs, which are compiled as a unit before being executed, and program files, in which all modules contained in the file are compiled as an unit before being executed. A file created by the **JOURNAL** routine is an example of a batch file. See [Chapter 10, “Executing Batch Jobs in IDL”](#) for additional information on batch files.

Note

Batch files are sometimes referred to as *include files*, since they can be used to “include” the multiple IDL statements contained in the file in another file.

Main-Level Programs

Short programs or procedures are often be entered at the IDL command line. These *main-level programs* consist of a series of program statements that are compiled and executed once an END statement is encountered. Main-level programs are not explicitly named, and cannot be called from other programs. For more information on main-level programs, see [“Main-Level Programs”](#) in Chapter 10 of the *Building IDL Applications* manual.

Named Programs

Longer routines and programs, consisting of more than a few lines, are typically given their own explicit names, allowing them to be called from other programs as well as executed at the IDL command line. Named programs are stored in disk files created using a text editor. The graphical IDL Development Environment includes a built-in text editor, but any text editor can be used to create named IDL programs.

Files containing IDL programs, procedures, and functions are assumed to have the filename extension `.pro`.

See [“Running IDL Programs”](#) on page 164 for the methods used to execute program files. For more detailed information on IDL program files, see [“What is an IDL Program?”](#) in Chapter 10 of the *Building IDL Applications* manual.

SAVE Files

You can create binary files containing data variables, system variables, functions, procedures, or objects using the SAVE procedure. Named IDL programs can be stored in SAVE files and distributed to other IDL users, who will be able to execute the program but who will not have access to the IDL code that creates it. For information on running IDL routines stored in a SAVE file, see [“Executing Programs Stored in a SAVE File”](#) on page 166. For information on storing a program in a SAVE file, see [“Saving Compiled IDL Programs and Data”](#) in Chapter 10 of the *Building IDL Applications* manual.

Other common uses for SAVE files include:

- To store the system variables and common blocks associated with an IDL session. For more information on this application of SAVE files see [“Saving Variables from an IDL Session”](#) on page 171.
- To store variable data such as an image with its associated user-defined color properties and size definitions. See [“Saving Variables from an IDL Session”](#) on page 171.

Running IDL Programs

IDL programs can be executed in the following ways:

Executing a Batch File

To execute a batch file, enter the name of the file, prefaced with the “@” character, at the IDL prompt:

```
@batchfile
```

where *batchfile* is the name of the file containing IDL statements.

Note

This syntax can also be used within an IDL program file.

For more on batch files, see [Chapter 10, “Executing Batch Jobs in IDL”](#).

Executing a Main-Level Program

To execute a Main-level program, use the `.RUN` or `.RNEW` executive command to enter the program at the IDL command line. The program will be executed automatically after you enter the `END` statement. To run the Main-level program again, use the `.GO` executive command.

For more on Main-level programs, see “[Main-Level Programs](#)” in Chapter 10 of the *Building IDL Applications* manual.

Executing Programs Stored in a Program File

IDL program files, identified with a `.pro` extension, can be compiled and executed using the following methods:

- Using the IDLDE interface
- Typing the routine name at the command line
- Running the program using an executive command

Using the IDLDE Interface

To run an IDL program using the IDLDE interface, do the following:

1. Open the file in the IDLDE editor. For example, select:
File → Open → RSI\IDL61\examples\demo\demosrc\d_uscensus.pro

2. Compile the file by selecting **Run** → **Compile filename**
where *filename* is the name of the file opened in the IDLDE editor (`d_uscensus.pro`, in this example).
3. Execute the file by selecting **Run** → **run filename**
where *filename* is the name of the file opened in the IDLDE editor (`d_uscensus.pro`, in this example).

From the IDL Command Line

When a file is specified by typing only the filename at the IDL prompt, IDL searches the current directory for *filename.pro* (where *filename* is the file specified) and then for *filename.sav*. If no file is found in the current directory, IDL searches in the same way in each directory specified by `!PATH`. If a file is found, IDL automatically compiles the contents and executes any functions or procedures that have the same name as the file specified (excluding the extension). See “[Automatic Compilation](#)” in Chapter 10 of the *Building IDL Applications* manual for additional details.

Using the previous example, run the US Census Data demo by entering the following at the command line:

```
d_uscensus
```

Using Executive Commands

When a file is specified using either the `.RUN`, `.RNEW`, `.COMPILE`, or `@` command followed by the filename, IDL searches the current directory for *filename.pro* (where *filename* is the file specified) and then for *filename.sav*. If no file is found in the current directory, IDL searches in the same way in each directory specified by `!PATH`. If a file is found, IDL compiles or runs the file as specified by the executive command used. Executive commands can be entered only at the IDL command prompt and are often used when executing main-level program files. See “[Executive Commands](#)” on page 90 for more information.

Warning

If the current directory contains a subdirectory with the same name as *filename*, IDL will consider the file to have been found and stop searching. To avoid this problem, specify the extension (`.pro` or `.sav`, usually) when entering the run, compile, or batch file executive command.

The details of how `!PATH` is initialized and used differ between the various operating systems, although the overall concept is the same. See “[!PATH](#)” in Appendix D of the *IDL Reference Guide* manual for more information.

Executing Programs Stored in a SAVE File

IDL SAVE files (created using the [SAVE](#) procedure) can contain one or more routines that have been packaged into a single binary file. (Variables, including system variables, can also be packaged into a SAVE file: see [“Saving Variables from an IDL Session”](#) on page 171 for a more complete discussion.)

Note

While IDL routines or data can be saved in a file with any extension, it is common to use the extension `.sav` for SAVE files. Using the `.sav` extension has two benefits: it makes it clear to another IDL user that the file contains IDL routines or data, and it allows IDL to locate routines with the same base name as the file in SAVE files located in IDL's path.

For an example of creating and restoring a SAVE file, see [“Restoring Compiled IDL Programs and Data”](#) in Chapter 10 of the *Building IDL Applications* manual.

Routines stored in SAVE files can be restored and executed in the following ways:

From the IDL Command Line

If the program you wish to run is stored in a SAVE file with the same base name as the program and the extension `.sav`, and the SAVE file is stored in a directory included in the IDL `!PATH` system variable, simply entering the name of the routine at the IDL command prompt will restore the program and execute it immediately. For example, if an IDL program named `myroutine` is stored in `myroutine.sav`, which is located in a directory included in `!PATH`, entering the following at the IDL command line will restore the routine and execute it:

```
myroutine
```

See [“Automatic Compilation”](#) in Chapter 10 of the *Building IDL Applications* manual for additional details.

Using the RESTORE Procedure

You can use the [RESTORE](#) procedure to restore all of the routines contained in a SAVE file without executing them. Once a routine has been restored, you can execute it simply by typing its name at the IDL command prompt. For example, if an IDL program named `myroutine` is stored in `myroutine.sav`, which is located in a directory that is *not* in `!PATH`, entering the following at the IDL command line will restore the routine and execute it:

```
RESTORE, 'path/myroutine.sav'  
myroutine
```

where *path* is the full path to the `myroutine.sav` file. See “[The RESTORE Procedure](#)” in Chapter 10 of the *Building IDL Applications* manual for additional details.

Using the IDL_Savefile Object

You can use the `IDL_Savefile` object class to gain information about the contents of a SAVE file, and to selectively restore items from the save file. Once a routine has been restored via calls the `IDL_Savefile` object, you can execute it simply by typing its name at the IDL command prompt. For example, if an IDL program named `myroutine` is stored in `myroutine.sav`, which is located in a directory that is *not* in `!PATH`, entering the following at the IDL command line will restore the routine and execute it:

```
obj = OBJ_NEW('IDL_Savefile', 'path/myroutine.sav')
obj->Restore, 'myroutine'
myroutine
```

where *path* is the full path to the `myroutine.sav` file. See “[The IDL_Savefile Object](#)” in Chapter 10 of the *Building IDL Applications* manual for additional details.

Interrupting Program Execution

To manually stop programs that are running, issue a *keyboard interrupt* by typing **Ctrl+C** (UNIX) or **Ctrl+Break** (Windows). A message indicating the statement number and program unit being executed is issued on the terminal or IDL Command Log acknowledging the interrupt. The values of variables can be examined, statements can be entered from the keyboard, and variables can be changed. The program can be resumed by issuing the **.CONTINUE** executive command to resume or the **.STEP** executive command to execute the next statement and stop.

Variable Context After Interruption

The variable context after a keyboard interrupt is that of the program unit in which the interrupt occurred. By typing the statement **RETURN**, the program context will revert to the next higher calling level. The **RETALL** command returns control to the main program level. If any doubt arises as to which program unit in which the interrupt occurred, the **HELP** procedure can be used to determine the program context. IDL checks after each statement to see if an interrupt has been typed. Execution does not stop until the statement that was active finishes; thus, considerable time can elapse from the time the interrupt is typed to the time the program interrupts.

Aborting IDL

If you find it necessary to abort IDL rather than exiting cleanly using the [EXIT](#) command, do one of the following:

- **UNIX:** As with any UNIX process, IDL can be aborted by typing **Ctrl+**. This is a very abrupt exit—all variables are lost, and the state of open files will be uncertain. Thus, although it can be used to exit of IDL in an emergency, its use should be avoided.

Note

After aborting IDL by using **Ctrl+**, you may find that your terminal is left in the wrong state. You can restore your terminal to the correct state by issuing one of the following UNIX commands:

```
% reset    or % stty echo -cbreak
```

- **Windows:** There is no abort character for IDL for Windows.

Saving Variables from an IDL Session

The [SAVE](#) and [RESTORE](#) procedures and the [IDL_Savefile](#) object combine to provide the ability to save the state of variables and system variables to restore them at a later time. Variables that are used from session to session can be saved as and recovered from a SAVE file. A startup file can be set up to execute the RESTORE command every time IDL is started. (See “[Startup Files](#)” on page 30 for information on specifying a startup files.) For an overview of how to restore and run SAVE files, see “[Executing Programs Stored in a SAVE File](#)” on page 166.

Tip

You can also distribute IDL code in binary format, sharing programs and applications with other IDL users. For an example of creating SAVE files of compiled programs, see “[Saving Compiled IDL Programs and Data](#)” in Chapter 10 of the *Building IDL Applications* manual for more information.

The state of variables in an IDL session can be saved quickly and easily, and can be restored to the same point. This feature allows you to stop work, and later resume at a convenient time. Variables that you may wish to create a SAVE file of include frequently used data files or system variable definitions.

Data can be conveniently stored in SAVE files, relieving the user of the need to remember the dimensions of arrays and other details. It is very convenient to store images this way. For instance, if the three variables *R*, *G*, and *B* hold the color table vectors, and the variable *I* holds the image variable, the IDL statement,

```
SAVE, FILENAME = 'image.sav', R, G, B, I
```

will save everything required to display the image properly in a file named `image.sav`. At a later date, the simple command,

```
RESTORE, 'image.sav'
```

will recover the four variables from the file.

System variables can also be saved and later applied to another session of IDL. For instance, you may choose to customize `!PATH`, the system variable defining the directories IDL will search for libraries, batch/include files, and executive commands or `!P`, the system variable that controls the definition of graphic elements associated with plot procedures. You can save these definitions in a SAVE file and later automatically restore or selectively restore the variables to apply the settings to other IDL sessions.

To save and restore the state of all current and system variables within an IDL session, you could use the following statement:

```
SAVE, /ALL, FILENAME = 'myIDLsession.sav'
```

The ALL keyword saves all common blocks, system variables and local variables from the current IDL session. See [Chapter 3, “Constants and Variables”](#) in the *Building IDL Applications* manual for information on these elements of an IDL session.

Note

Routines and variables cannot be saved in the same file. Setting the ALL keyword does not save routines.

To restore the session information, enter:

```
RESTORE, 'myIDLsession.sav'
```

Note

If the file is not located in your current working directory, you will need to define the path to the file.

Long iterative jobs can save their partial results in a SAVE format to guard against losing data if some unexpected event such as a machine crash should occur.

Note

A SAVE file containing data will always be restorable. However, SAVE files created prior to IDL version 5.5 that contain IDL procedures, functions, and programs are not always portable between different versions of IDL. If you created your SAVE file with a version of IDL earlier than 5.5, you will need to recompile your original .pro files and re-create the SAVE file using the current version of IDL.

For More Information on Programming

Here we have just touched on the possibilities that IDL offers for programmers. For more information on how to prepare and run programs, see the *Building IDL Applications* manual.



Chapter 10: Executing Batch Jobs in IDL

The following topics are covered in this chapter:

Overview	176	A Batch Example	178
Interpretation of Batch Statements	177		

Overview

You can run IDL in non-interactive mode (batch mode) by entering the character @ followed by the name of a file containing IDL executive commands and statements. Commands and statements are executed in the order they are contained in the file, as if they had been entered at the IDL command prompt.

Tip

For information on how to specify a batch file as a startup file that is automatically executed when IDL is started, see “[Startup Files](#)” on page 30.

Batch execution can be terminated before the end of the file, with control returning to interactive mode without exiting IDL, by calling the STOP procedure from the batch file. Calling the EXIT procedure from the batch procedure has the usual effect of terminating IDL.

To enter batch mode from the interactive mode, enter:

```
@filename
```

at the IDL prompt, where *filename* is the name of the batch file. (Note that the @ symbol must be the first character on the line in order for it to be interpreted properly.) IDL reads commands from the specified file until the end of the file is reached. Batch files can be nested by placing a call to one batch file within another.

If *filename* does not include a file extension, IDL searches the current working directory and the directories specified by the !PATH system variable for a file with *filename* as its base, with the file extension .pro. If *filename*.pro is not found in a given directory, IDL searches for *filename* with no extension in that directory. If *filename* is found (with or without the .pro extension), the file is executed and the search ends. If *filename* includes a full path specification, IDL does not search the directories in !PATH.

Interpretation of Batch Statements

Each line of a batch file is interpreted exactly as if it was entered from the keyboard. In batch mode, IDL compiles and executes each statement before reading the next statement. This differs from the interpretation of programs compiled using `.RNEW` or `.RUN`, in which all statements in a program are compiled as a single unit and then executed.

Labels are illegal in the batch mode because each statement is compiled and executed independently.

Multiline statements must be continued on the next line using the `$` continuation character, because IDL terminates every interactive mode statement not ending with `$` by an `END` statement. A common mistake is to include a multiple-line block statement in a batch file as shown below.

```
; This will not work in batch mode.  
FOR I = 1, 10 DO BEGIN  
    A = X[I]  
    ...  
    ...  
ENDFOR
```

In batch mode, IDL compiles and executes each line separately, causing syntax errors in the above example because no matching `ENDFOR` is found on the line containing the `BEGIN` statement when the line is compiled. The above example could be made to work by writing the block of statements as a single line using the `$` (continuation) and `&` (multiple commands on a single line) characters.

A Batch Example

An example of an IDL executive command line that initiates batch execution:

```
@myfile
```

This command causes the file `myfile.pro` to be used for statement and command input. If this file is not in the current directory, the directories specified by `!PATH` are also searched.

An example of the contents of a batch file follows:

```
; Run program A:  
. RUN proga  
; Run program B:  
. RUN progb  
; Print results:  
PRINT, AVALUE, BVALUE  
; Close unit 3:  
CLOSE, 3  
<eof>
```

The batch file should not contain complete program units. Complete program units should be compiled and run by using the `.RUN` and `.RNEW` commands in the batch files, as illustrated above.



Chapter 11: Journaling in IDL

The following topics are covered in this chapter:

Overview	180	Journaling Examples	182
--------------------------------	-----	---	-----

Overview

Journaling provides a record of an interactive session by saving in a file all text entered from the terminal in response to a prompt. In journaling, all text entered to the IDL prompt is entered directly into the file, and any text entered from the terminal in response to any other input request (such as with the READ procedure) is entered as a comment. The result is a file that contains a complete description of the IDL session.

JOURNAL has the form:

```
JOURNAL[ , Argument ]
```

where *Argument* is either a filename (if journaling is not currently in progress) or an expression to be written to the file (if journaling is active).

The first call to JOURNAL starts the logging process. If no argument is supplied, a journal file named `idlsave.pro` is started.

Warning

Under all operating systems, creating a new journal file will cause any existing file with the same name to be lost. Supply a filename argument to JOURNAL to avoid destroying desired files.

When journaling is not in progress, the value of the system variable `!JOURNAL` is zero. When the journal file is opened, the value of this system variable is set to the number of the logical file unit on which the file is opened. This allows IDL routines to check if journaling is active. You can send any arbitrary data to this file using the normal IDL output routines. In addition, calling JOURNAL with an argument while journaling is in progress results in the argument being written to the journal file as if the PRINT procedure had been used. In other words, the statement,

```
JOURNAL,
```

is equivalent to

```
PRINTF, !JOURNAL, Argument
```

with one significant difference—the JOURNAL statement is not logged to the file, only its output; while the PRINTF statement will be logged to the file in addition to its output.

Journaling ends when the JOURNAL procedure is called again without an argument or when IDL is exited. The resulting file serves as a record of the interactive session that went on while journaling was active. It can be used later as an IDL batch input

file to repeat the session, and it can be edited with any text editor if changes are necessary.

Journaling Examples

As an example of Journaling in IDL, consider the following IDL statements:

```

;Start journaling to file demo.pro:
JOURNAL, 'demo.pro'
;Prompt for input:
PRINT, 'Enter a number:'
;Read the user response into variable Z:
READ, Z
;Send an IDL comment to the journal file using JOURNAL:
JOURNAL, '; This was inserted with JOURNAL.'
;Send another comment using PRINTF:
PRINTF, !JOURNAL, '; This was inserted with PRINTF.'
;End journaling:
JOURNAL

```

If these statements are executed by a user named Doug on a Sun workstation named quixote, the resulting journal file `demo.pro` will look like the following:

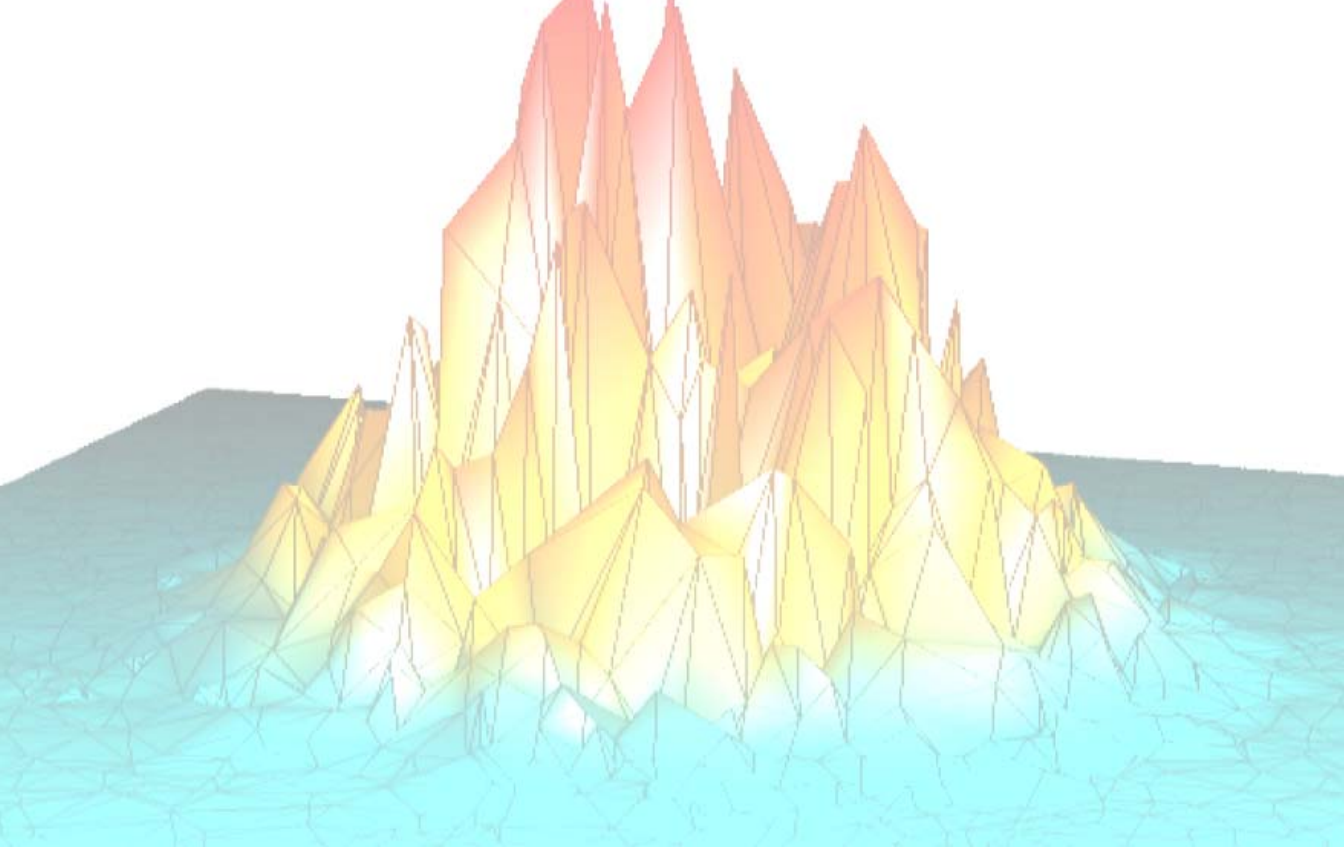
```

; IDL Version 5.3 (sunos sparc)
; Journal File for doug@quixote
; Working directory: /home/doug/IDL
; Date: Mon Sept 9 14:38:24 1999

PRINT, 'Enter a number:'
;Enter a number:
READ, Z
; 87
; This was inserted with JOURNAL.
; This was inserted with PRINTF.
PRINTF, !JOURNAL, '; This was inserted with PRINTF.'

```

Note that the input data to the `READ` statement is shown as a comment. In addition, the statement to insert the text using `JOURNAL` does not appear, while the statement using `PRINTF` does appear.



Part II: Reading and Writing Data



Chapter 12: IDL Macros for Importing Data

This chapter describes the following topics.

Overview	186	Using Macros to Import Binary Files	197
Using Macros to Import Image Files	187	Using Macros to Import HDF Files	203
Using Macros to Import ASCII Files	191		

Overview

IDL contains macros to ease the importing of data into IDL. This chapter introduces these macros and describes how to import image, ASCII, binary, and Scientific Data Format (SDF) files. These macros are available through the **Macros** menu and also through IDL **Tool Bar** buttons.

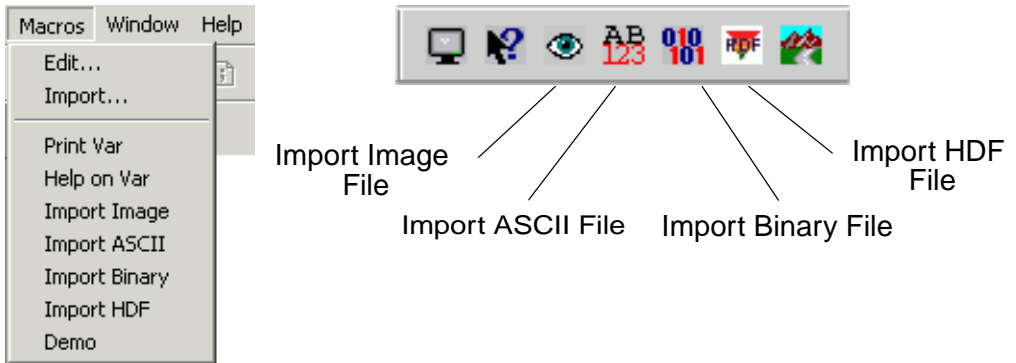


Figure 12-1: Importing Data Macros Menu (Left) and Tool Bar Buttons (Right)

Using Macros to Import Image Files

To import an image file into IDL, complete the following steps:

1. Select the **Import Image File** tool bar button. The **Select Image File** dialog is displayed.

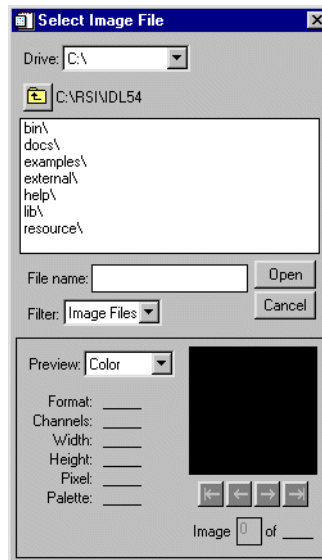


Figure 12-2: Select Image File Dialog

2. Select a file to import. For example, select the `rsi-directory/examples/data/muscle.jpg` file where `rsi-directory` is the installation directory for IDL.

You can now see a preview of this image as well as other information about the file in the lower section of the Select Image File dialog. You can change the preview to **Color**, **Grayscale**, or **No Preview**. If the image file had more than one actual image, you can see them using the arrow buttons to scroll through the images. You can only read in one image of a multi-image file. The image in the preview is the image that will be read.

3. Click **Open**.
4. The file has been opened into a structure variable named `MUSCLE_IMAGE`.

Images opened with the **Import Image File** macro are stored in structure variables which are named *filename_IMAGE* where *filename* is the name of the file you opened without the extension.

Note

IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix “var” is added to the variable name. Any spaces within *filename* are converted to underscores. Any other illegal characters within *filename* are removed.

So, the file we just opened (*muscle.jpg*) is now in the structure variable named **MUSCLE_IMAGE**. The file is a structure with the following fields:

- **IMAGE** — The actual image array.
- **R** — The red color table vectors.
- **G** — The green color table vectors.
- **B** — The blue color table vectors.
- **QUERY** — Contains information about the image.
 - **CHANNELS** — The number of channels in the image.
 - **HAS_PALETTE** — Specifies if the palette is present. 1 if the palette is present, else 0. If your image is *n*-by-*m* the palette is usually present and the R, G, and B color table vectors mentioned above will contain values. If your image is 3-by-*n*-by-*m*, the palette will not be present and the R,G, and B color table vectors will not contain any values.
 - **IMAGE_INDEX** — The index of the image of the file. The default is 0, the first image in the file. If there are multiple images in the file that you read, this will be the number (or index) of the image.
 - **NUM_IMAGES** — The number of images in the original file.
 - **PIXEL_TYPE** — The IDL Type Code of the image pixel format. Valid types are:

PIXEL_TYPE returned	Data Types
1	Byte

Table 12-1: Values for PIXEL_TYPE in the Structure

PIXEL_TYPE returned	Data Types
2	Integer
3	Longword Integer
4	Floating Point
5	Double-precision Floating Point
12	Unsigned Integer
13	Unsigned Longword Integer
14	64-bit Integer
15	Unsigned 64-bit Integer

Table 12-1: Values for PIXEL_TYPE in the Structure

- TYPE — The image type. Valid return values are:
BMP, JPEG, PNG, PPM, SRF, TIFF, DICOM

The structure can be viewed in the Variable Watch Window.

Name	Type	Value
MUSCLE_IMAGE	STRUCT	{ <Anonymous> }
IMAGE	BYTE	Array[652, 444]
R	BYTE	Array[256]
G	BYTE	Array[256]
B	BYTE	Array[256]
QUERY	STRUCT	{ <Anonymous> }
CHANNELS	LONG	1
DIMENSIONS	LONG	Array[2]
HAS_PALETTE	INT	0
IMAGE_INDEX	LONG	0
NUM_IMAGES	LONG	1
PIXEL_TYPE	INT	1
TYPE	STRING	JPEG

Figure 12-3: Variable Watch Window Showing MUSCLE_IMAGE Structure

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.element_name[.element_name]

For example, if you want to view the image, enter the following:

```
TV, MUSCLE_IMAGE.IMAGE
```

This displays the following image:

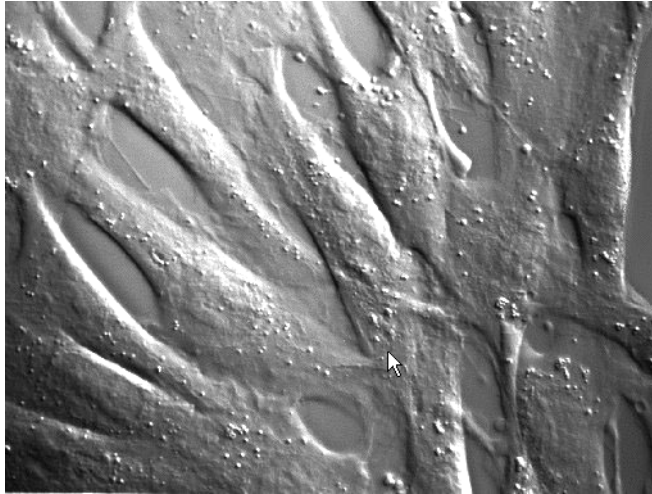


Figure 12-4: MUSCLE_IMAGE.IMAGE

If you want to know the file type, enter the following:

```
PRINT, MUSCLE_IMAGE.QUERY.TYPE
```

IDL prints:

```
JPEG
```

Using Macros to Import ASCII Files

To import an ASCII file into IDL, complete the following steps:

1. Select the **Import ASCII File** tool bar button. The **Select an ASCII file to read** dialog displays.

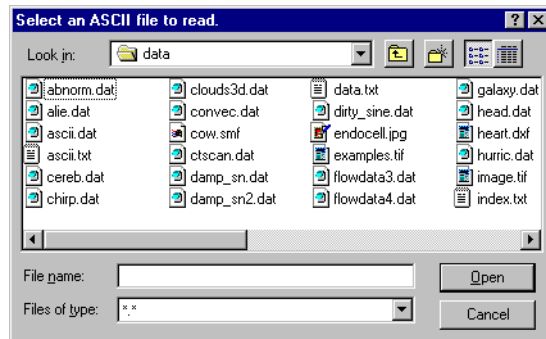


Figure 12-5: Select an ASCII file to read Dialog

2. Select a file to import. For example, select the `rsi-directory/examples/data/ascii.txt` file where `rsi-directory` is the installation directory for IDL. Click **Open**.
3. In the **Define Data Type/Range** dialog, you specify information about your file. The first few lines of the file are displayed to help you find the information you need to specify.

First, select the type of field which best describes your data. You can either choose **Fixed Width** which specifies that the data is aligned in columns, or **Delimited** which specifies that the data is separated by commas, whitespace, etc. In this example, the data is delimited by commas so we'll select the **Delimited** radio button.

Next, enter a character or string that is used to comment lines within the file in the **Comment String to Ignore:** field. In this example, if we read the first few lines of this file, it defines the % character as the comment character. Enter the % sign in the **Comment String to Ignore:** field.

Next, enter the line number in which the data starts in the **Data Starts at Line:** field. In this example, the data starts on line 6 so we'll enter that value in the field.

Click **Next**.

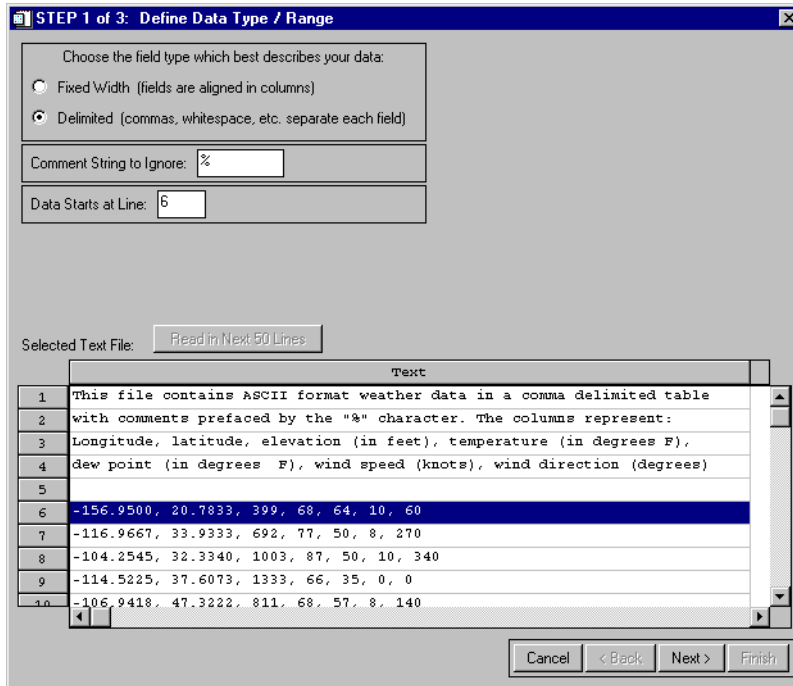


Figure 12-6: Define Data Type/Range Dialog

4. In the **Define Delimiter/Fields** dialog, we'll specify the information about the actual data in the file.

First, we'll enter the number of columns or fields in the **Number of Fields Per Line:** field. In this example, there are 7 fields.

Next, we'll enter how the data is delimited. You can choose **White Space**, **Comma**, **Colon**, **Semicolon**, **Tab**, or **Other**. If you specify **Other**, you must then enter the characters in the field. In this example, we'll select **Comma** since the data is delimited by commas.

Click **Next**.

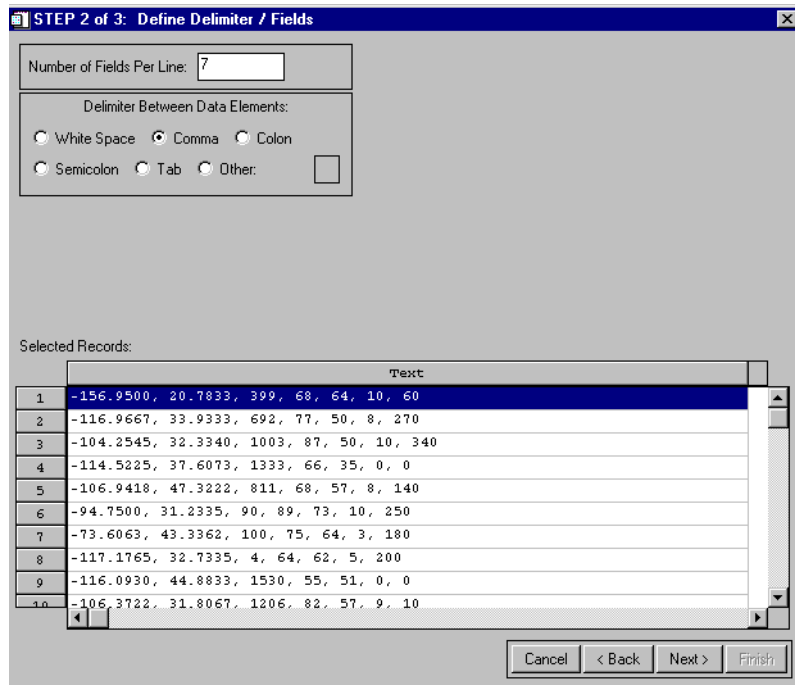


Figure 12-7: Define Delimiter/Fields Dialog

5. In the **Field Specification** dialog, we'll enter information about the contents of each column or field in the data.

First, select the first field in the data in the box in the upper left of the dialog. Enter the name of the field in the **Name** field and the type of data represented in the **Type** field. In this example we'll specify **Longitude** and **Floating** for the fields. Continue naming all the fields in the data using this procedure. In this example, we'll use Latitude – Floating; Elevation – Long; Temperature – Long; DewPoint – Long; WindSpeed – Long; WindDir – Long for the other field pairs.

You can also group some or all of the fields into one field by using the **Group** or **Group All** buttons. In this example, there is no need to group any of the fields.

Next, select the value to assign missing data. You can select the IEEE standard for NaN or a custom value. In this example, we'll choose **IEEE NaN**.

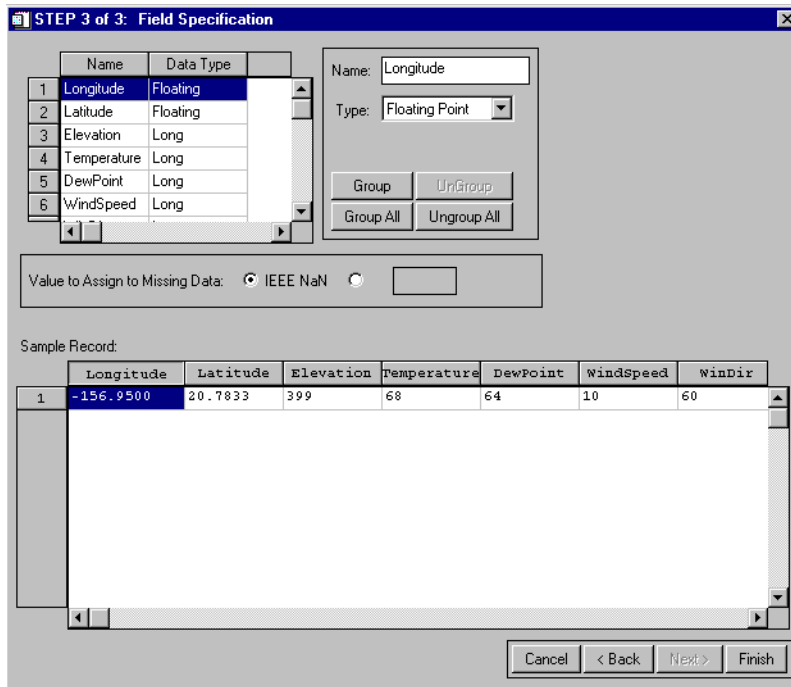


Figure 12-8: Field Specification Dialog

6. Click **Finish**.

ASCII files opened with the **Import ASCII File** macro are stored in structure variables which are named *filename_ASCII* where *filename* is the name of the file you opened without the extension.

Note

IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix "var" is added to the variable name. Any spaces within *filename* are converted to underscores. Any other illegal characters within *filename* are removed.

So, the file we just opened (ascii.txt) is now in the structure variable named ASCII_ASCII. The variable is a structure with each field name being an element of the structure.

The structure can be viewed in the Variable Watch Window.

Name	Type	Value
ASCII_ASCII	STRUCT	{ <Anonymous> }
LONGITUDE	FLOAT	Array[15]
LATITUDE	FLOAT	Array[15]
ELEVATION	LONG	Array[15]
TEMPERATURE	LONG	Array[15]
DEWPOINT	LONG	Array[15]
WINDSPEED	LONG	Array[15]
WINDIR	LONG	Array[15]

Figure 12-9: Variable Watch Window Showing ASCII_ASCII Structure

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.element_name

For example, if you want to view the Longitude field, enter the following:

```
Print, ASCII_ASCII.LONGITUDE
```

IDL prints:

```
-156.950    -116.967    -104.255    -114.522    -106.942
-94.7500    -73.6063    -117.176    -116.093    -106.372
-93.2237    -109.635    -76.0225    -93.1535    -118.721
```

If you want to plot Temperature, enter the following:

```
PLOT, ASCII_ASCII.TEMPERATURE
```

The following figure results.

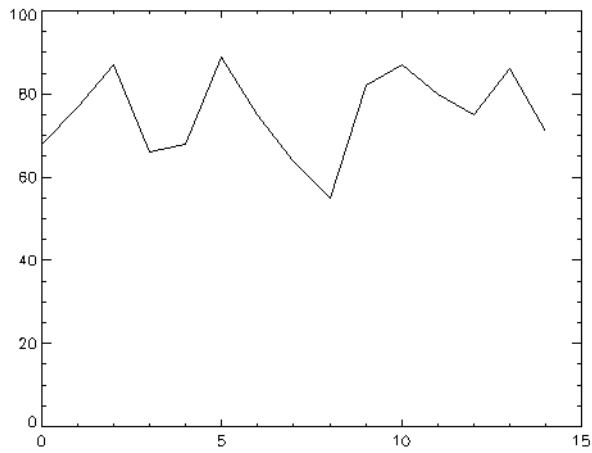


Figure 12-10: Plot of ASCII_ASCII.TEMPERATURE

Using Macros to Import Binary Files

Sometimes, data is stored in files as arrays of bytes instead of a known format like JPEG or TIFF. These files are referred to as binary files.

Note

The **Import Binary File** macro is intended for use in loading raw binary data from files into IDL. Such data is comprised of bits that are meaningful — as integers or floating-point numbers for example — with no special processing (except possibly byte-order swapping) required. Commercial spreadsheet or word processing files, for example, are binary but they are not raw in the above sense, and thus are not good candidates for use with this macro.

Also note that the **Import Binary File** macro is intended for use in loading data from files the contents of which you have some knowledge about. To effectively read data with this macro, you must be able to supply literal values or expressions that specify the type and location of the data in the file you wish to read.

To import a binary file into IDL, complete the following steps:

1. Select the **Import Binary File** tool bar button. The **Select a binary file to read** dialog is displayed.

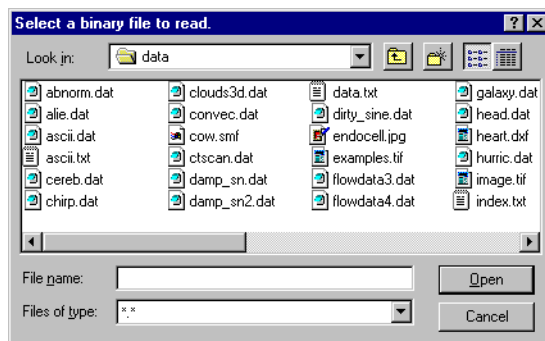


Figure 12-11: Select a binary file to read Dialog

2. Select a file to import. For example, select the `rsi-directory/examples/data/surface.dat` file where `rsi-directory` is the installation directory for IDL. Click **Open**.

3. In the **Binary Template** dialog box, specify information about your file.

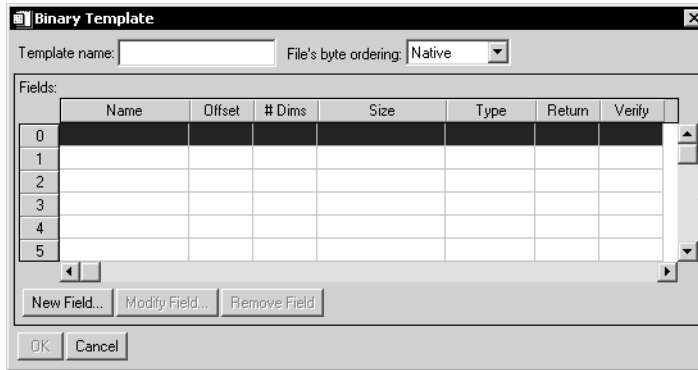


Figure 12-12: The Binary Template dialog

First, enter the name of the template you are going to create in the **Template name:** field. For this example, “marbellstemplate” is used.

Next, select the byte order in the file in the **File’s byte ordering:** pull-down menu. The choices are:

- **Native** — The type of storage method that is native to the machine you are currently running. Little Endian for Intel microprocessor-based machines and Big Endian for Motorola microprocessor-based machines. No byte swapping will be performed.
- **Little Endian** — A method of storing numbers so that the least significant byte appears first in the number. For example, given the hexadecimal number A02B, the little endian method specifies the number to be stored as 2BA0. Specify this if the original file was created on a machine that uses an Intel microprocessor.
- **Big Endian** — A method of storing numbers so that the most significant byte appears first in the number. For example, given the hexadecimal number A02B, the big endian method specifies the number to be stored as A02B. Specify this if the original file was created on a machine that uses a Motorola microprocessor.

The file `surface.dat` was created on a machine that uses an Intel microprocessor. For this example, select **Little Endian** for the byte order.

4. Now we are ready to enter the field values for the file. You can have multiple fields within a binary file. Click the **New Field...** button in the lower-left corner of the **Binary Template** dialog box.

In the **New Field** dialog (shown at the end of these example steps), enter the name of the field in the **Field name:** text box. In this example, enter “A” as the field name.

Next, you need to specify where in the file to start reading. The options are:

- **Offset** — Specifies the byte offset or where to begin reading the file. The > symbol specifies to offset forward from a byte position, the < symbol specifies to offset backward from a byte position.
- **From beginning of file** — Specifies to start reading this field starting with the first byte of the file plus any **Offset** specified.
- **From initial position in file/From end of previous field** — This field changes depending upon if this is the first field or any other field besides the first. If this is the first field you are defining, this option specifies to read from the beginning of the file plus any **Offset** specified. If this is not the first field, this option changes to **From end of previous field** and specifies to begin reading the field where the previous field ended plus any **Offset** specified.

In this example, since this is the first field in the file and we don’t have any header information in the file, specify **From the beginning of file** without any offset.

Next, select whether or not you want this field to be returned to IDL when a file is read. For example, you may have a section of your binary file that contains header information. If you create a field for this section, you do not want it returned to IDL. In this case, you would not select **Returned in the result**. You must specify at least one field to be returned to IDL. In this example, we want to return the field we’re creating so we’ll check the box in the upper-right corner marked **Returned in the result**.

Next, you need to specify whether or not you want to verify any of the data you are returning in the **Verified equal to** field. This field is only available if the field is a scalar. This can be any valid IDL expression that evaluates to a scalar. For this example, we won’t verify any of the data.

Next, you need to specify the type of data that is in this field. In this example, the data is integer type data so select the Integer (16 bits) at the **Type** pull-down menu. The valid values for **Type** are:

- Byte (unsigned 8-bits)
- Integer (16-bits)
- Long (32-bits)
- Long64 (64 bits)
- Float (32 bits)
- Double-Precision (64-bits)
- Unsigned Integer (16 bits)
- Unsigned Long (32-bits)
- Unsigned Long64 (64-bits)
- Complex (real-imaginary pair of floats)
- Double-Precision Complex (pair of doubles)

Next, specify the number of dimensions contained in the data in the **Number of dimensions:** pull-down menu. This will activate a corresponding number of boxes in the dimensions section of the dialog. In this example, the data is two-dimensional.

Finally, enter the size of each dimension in the field. If you select the **Allow expressions for dimension sizes** check box, you enter any valid IDL expression that returns the size of the dimension. You can also choose to reverse the order of the data by selecting the **Reverse** check box for each dimension. This can be useful when image data is returned in the reverse order and appears upside down. In this example, the data is contained in a 350-by-450 array, so enter 350 for the size of the **1st dimension** and 450 for the size of the **2nd dimension** in the text fields marked **Size:**.

Click **OK**.

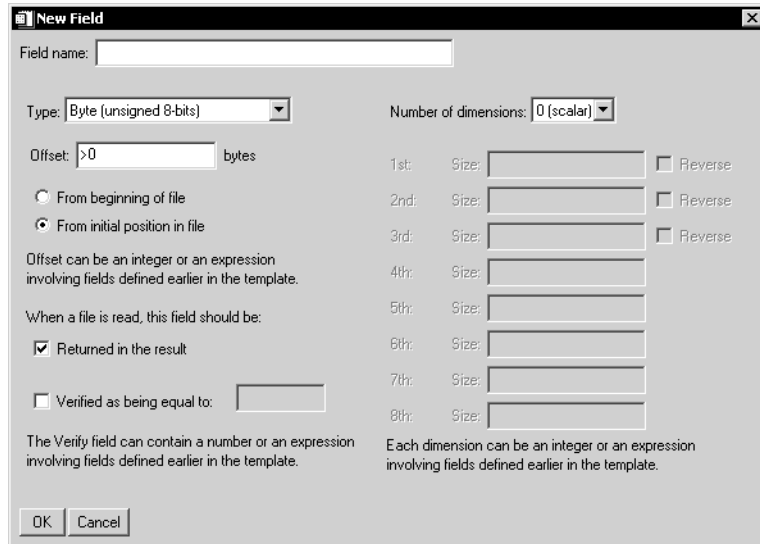


Figure 12-13: Modifying fields in Binary Template

5. You can now see the information that you entered in the **Binary Template** dialog. If you need to enter more fields, select the **New Field** button. Repeat the steps until you have entered all the fields in the binary file.

In this example, there is only one field. Click **OK**.

Binary files opened with the **Import Binary File** macro are stored in structure variables which are named *filename_BINARY* where *filename* is the name of the file you opened without the extension.

Note

IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix “var” is added to the variable name. Any spaces within *filename* are converted to underscores. Any other illegal characters within *filename* are removed.

So, the file we just opened (*surface.dat*) is now in the structure variable named *SURFACE_BINARY*. The variable is a structure with each field name being an element of the structure.

The structure can be viewed in the Variable Watch Window.

Name	Type	Value
<input type="checkbox"/> SURFACE_BINARY	STRUCT	{ <Anonymous> }
<input type="checkbox"/> A	INT	Array[350, 450]
<input type="checkbox"/> [0,0]	INT	3198

Locals Params Common System

Figure 12-14: Variable Watch Window Showing `MARBELLS_BINARY` Structure

You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.element_name

For example, display the image by entering:

```
TVSCL, SURFACE_BINARY.A
```

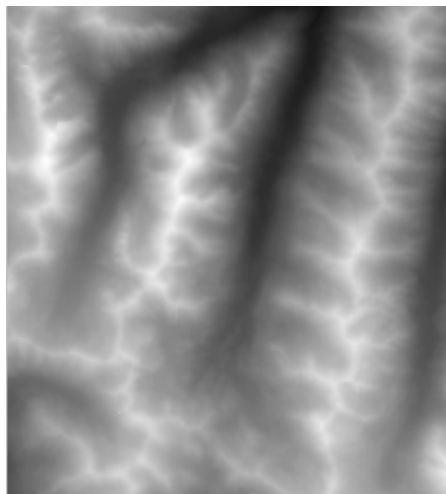


Figure 12-15: `Surface.dat` displayed using TVSCL

Using Macros to Import HDF Files

To import a Hierarchical Data Format (HDF), HDF-EOS, or NETCDF file into IDL, complete the following steps:

1. Select the **Import HDF File** tool bar button. The **Select a valid HDF, NETCDF or HDF-EOS file** dialog is displayed.

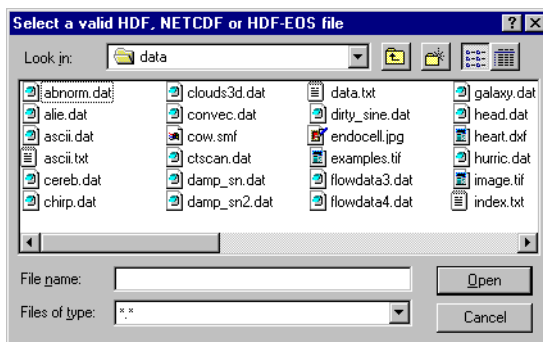


Figure 12-16: Select a valid HDF, NETCDF or HDF-EOS file Dialog

2. Select a file to import. Click **Open**.
3. The **HDF Browser** window is displayed (shown at the end of these example steps). In the **HDF Browser** window, select the data in the file you want to import into IDL.

In the **Display** pull-down menu, select the type of file you are reading. The two options are:

- HDF/NETCDF
- HDF-EOS

Next, select the type of data you want to import. The following tables describe the options available for the two display choices from the pull-down menu.

Menu Selection	Description
HDF/NetCDF Summary	

Table 12-2: Menu Options for HDF/NetCDF Data Types

Menu Selection	Description
DF24 (24-bit Images)	24-bit images and their attributes
DFR8 (8-bit Images)	8-bit images and their attributes
DFP (Palettes)	Image palettes
SD (Variables/Attributes)	Scientific Datasets and attributes
AN (Annotations)	Annotations
GR (Generic Raster)	Images
GR Global (File) Attributes	Image attributes
VGroups	Generic data groups
VData	Generic data and attributes

Table 12-2: Menu Options for HDF/NetCDF Data Types

Menu Selection	Description
HDF-EOS Summary	
Point	EOS point data and attributes
Swath	EOS swath data and attributes
Grid	EOS grid data and attributes

Table 12-3: Menu Options for HDF-EOS Data Types

Once you have selected the type of data, information is displayed that shows the different elements of data available in the file you are opening. For example, if it is an image file, you will see the names of the images displayed. Select the item to import.

If you have selected an image, 2D data set, or 3-by-*n*-by-*m* data set from the pull-down menu, you can click on the **Preview** button to view the image. If you have selected a data item that can be plotted in two dimensions, click on the **Preview** button to view a 2D plot of the data (the default); or click on the **Preview Surface** radio button to display a surface plot; click on the **Preview Contour** radio button to display a contour plot; or click on the **Preview Show3**

radio button for an image, surface, and contour display. You can also select the **Fit to Window** check box to fit the image to the window.

Next, if you want the data or metadata item you are previewing to be imported into IDL, select the **Read** check box to extract the current data or metadata item from the HDF file.

Next, specify a name for the extracted data or metadata item.

Note

The **Read** check box must be selected for the item to be extracted. Default names are generated for all data items, but may be changed at any time by the user.

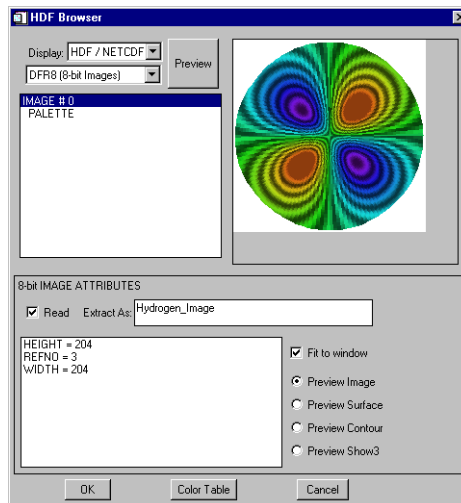


Figure 12-17: HDF Browser Window

4. Continue selecting to read and name the data or metadata items you want to import into IDL.
5. Click **OK**.

HDF, NETCDF, or HDF-EOS files read with the **Import Binary File** macro are stored in structure variables which are named *filename_DF* where *filename* is the name of the file you opened without the extension.

Note

IDL variables must begin with a letter, and may contain only letters, digits, the underscore character, or the dollar sign. If the first character of *filename* is not a letter, the prefix “var” is added to the variable name. Any spaces within *filename* are converted to underscores. Any other illegal characters within *filename* are removed.

The variable is a structure with each data or metadata name being an element of the structure. You can specify which part of the structure variable you want to access by using the following syntax:

variable_name.data_name

For example, if you imported two data elements out of a file named hydrogen.hdf and you named the elements IMAGE1 and IMAGE2, you could access each individual data element using the following:

HYDROGEN_DF.IMAGE1

HYDROGEN_DF.IMAGE2

If you wanted to view IMAGE1, you would enter:

```
TV, HYDROGEN_DF . IMAGE1
```

For more information on IDL support of HDF and other scientific data formats, see the *Scientific Data Formats* manual.



Chapter 13: Reading and Writing Images

This chapter provides an introduction to reading and writing image data using the latest commands and user interfaces found in IDL.

List of Commands	208	Accessing Specific Image File Formats . .	215
Accessing Image Files Using Dialogs	211	Accessing Files Using Dialogs	217
Accessing General Image File Formats . . .	214	Accessing Files With Compound Widgets	219

List of Commands

The following routines are used for reading and writing image data. These IDL routines provide access to specialized functionality in the case of more specific applications.

Compound Widgets and Dialogs

CW_FILESEL	A compound widget for file selection.
DIALOG_PICKFILE	Allows the user to interactively pick a file, or multiple files, using the platform's own native graphical file-selection dialog.
DIALOG_READ_IMAGE	A graphical user interface used for reading image files.
DIALOG_WRITE_IMAGE	A graphical user interface used for writing image files.

Images (Generalized)

QUERY_IMAGE	Reads the header of a file and determines if it is recognized as an image file.
READ_IMAGE	Reads the image contents of a file and returns the image in an IDL variable.
WRITE_IMAGE	Writes an image and its color table vectors, if any, to a file of a specified type.

Images (Specific Formats)

QUERY_BMP	Obtains information about a BMP image file without having to read the image.
QUERY_DICOM	Tests a file for compatibility with <code>READ_DICOM</code> and returns an optional structure containing information about images in the DICOM file.
QUERY_JPEG	Obtains information about a JPEG image file without having to read the image.

<code>QUERY_JPEG2000</code>	Obtains information about a JPEG 2000 image file without having to read the image.
<code>QUERY_PICT</code>	Obtains information about a PICT image file without having to read the image.
<code>QUERY_PNG</code>	Obtains information about a PNG image file without having to read the image.
<code>QUERY_PPM</code>	Obtains information about a PPM image file without having to read the image.
<code>QUERY_SRF</code>	Obtains information about a SRF image file without having to read the image.
<code>QUERY_TIFF</code>	Obtains information about a TIFF image file without having to read the image.
<code>READ_BMP</code>	Reads a Microsoft Windows Version 3 device independent bitmap image (.bmp) and returns a byte array containing the image.
<code>READ_DICOM</code>	Reads an image from a DICOM file along with any associated color table.
<code>READ_INTERFILE</code>	Reads image data stored in Interfile (v3.3) format and returns a 3D array.
<code>READ_JPEG</code>	Reads JPEG (Joint Photographic Experts Group) format compressed images from files or memory.
<code>READ_JPEG2000</code>	Reads JPEG 2000 images from files or memory.
<code>READ_PICT</code>	Reads the contents of a PICT (version 2) format image file and returns the image and color table vectors (if present).
<code>READ_PNG</code>	Reads the image contents of a Portable Network Graphics (PNG) image file.
<code>READ_PPM</code>	Reads the contents of a PGM (gray scale) or PPM (portable pixmap for color) format image file and returns the image in the form of a 2D byte array (for gray scale images) or a $(3, n, m)$ byte array (for TrueColor images).
<code>READ_SRF</code>	Reads the contents of a Sun rasterfile and returns the image and color table vectors (if present).

<code>READ_TIFF</code>	Reads multi channel image TIFF format files and returns the image and color table vectors.
<code>READ_X11_BITMAP</code>	Reads bitmaps stored in the X Windows X11 bitmap format.
<code>READ_XWD</code>	Reads the contents of a file created by the <code>xwd</code> (X Windows Dump) command and returns the image and color table vectors.
<code>WRITE_BMP</code>	Writes an image and its color table vectors to a Microsoft Windows Version 3 device independent bitmap file (.bmp).
<code>WRITE_JPEG</code>	Writes compressed images to a JPEG (Joint Photographic Experts Group) file which is a standardized compression method for full-color and gray-scale images.
<code>WRITE_JPEG2000</code>	Writes compressed images to a JPEG 2000 file.
<code>WRITE_NRIF</code>	Writes an image and its color table vectors to an NCAR Raster Interchange Format (NRIF) rasterfile.
<code>WRITE_PICT</code>	Writes an image and its color table vectors to a PICT (version 2) format image file.
<code>WRITE_PNG</code>	Writes a 2D or 3D IDL variable into a Portable Network Graphics (PNG) image file.
<code>WRITE_PPM</code>	Writes an image to a PPM (TrueColor) or PGM (gray-scale) image file.
<code>WRITE_SRF</code>	Writes an image and its color table vectors to a Sun Raster File (SRF) image file.
<code>WRITE_TIFF</code>	Writes an image and its color table vectors to a Tagged Image Format (TIFF) image file.

Accessing Image Files Using Dialogs

Selecting an Image File

The `DIALOG_READ_IMAGE` function is a graphical user interface which is used for reading image files. This interface simplifies the use of IDL image file I/O. Users are able to preview images with a quick and simple browsing mechanism which will also report important information about the image file. The user has the option to view the image in color, grayscale, or no preview.

```
Result = DIALOG_READ_IMAGE( )
```

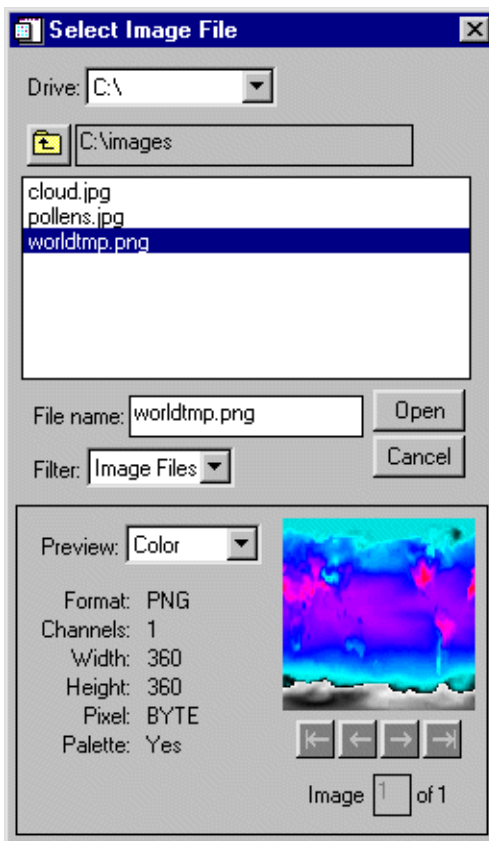


Figure 13-1: The `DIALOG_READ_IMAGE` dialog

Button	Function
Open	Opens the selected image file.
Cancel	Cancel the current image selection.
Arrow Keys	Pages through multiple images in the file.

Table 13-1: Save Image File Buttons

Saving an Image File

The `DIALOG_WRITE_IMAGE` function is a graphical user interface which is used for writing/saving image files. This interface simplifies the use of IDL image file I/O.

```
myimage = DIST(100)
result = DIALOG_WRITE_IMAGE(myimage, FILENAME='myimage.tif')
```

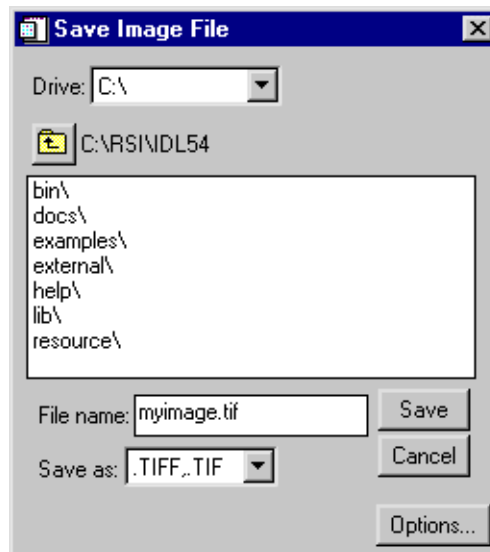


Figure 13-2: The `DIALOG_WRITE_IMAGE` dialog

Button	Function
Save	Saves the image file.
Cancel	Cancels the save function.
Options	Brings up a dialog box of image format save options.

Table 13-2: Save Image File Dialog Buttons

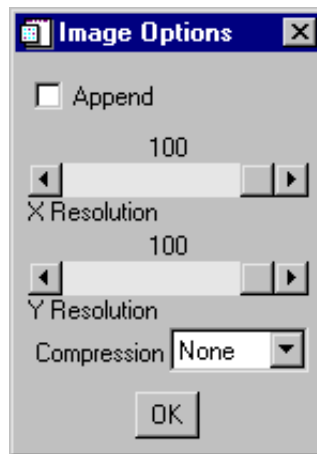


Figure 13-3: Image Options

Accessing General Image File Formats

Querying an Image File

The `QUERY_IMAGE` function reads the header of an image file and determines if it is recognized as an image file. If it is an image file, an optional structure containing the information about the image is returned. The Info structure for all image types has the following fields:

Tag	Definition
CHANNELS	Long
DIMENSIONS	One-dimensional long array
FILENAME	Scalar string
HAS_PALETTE	Integer
IMAGE_INDEX	Long
NUM_IMAGES	Long
PIXEL_TYPE	Integer
TYPE	Scalar string

Table 13-3: Info Structure for Images

Reading an Image File

The `READ_IMAGE` function reads the image contents of a file and returns the image in an IDL variable.

Writing an Image File

The `WRITE_IMAGE` function writes an image and its color table vectors to a file of a specified type. The `WRITE_IMAGE` function can write most types of image files.

Accessing Specific Image File Formats

QUERY_* Routines

IDL has added a consistent set of query routines to the existing IDL image file format API to allow users to obtain information about files without having to read them into memory.

All of the QUERY_ routines return a status, which determines if the file is appropriate to use the corresponding READ_ routine. In addition, these routines return an anonymous structure containing all of the available information for that image format, such as the image dimensions, number of samples per pixel, pixel type, palette info, and the number of images in the file. The following is a list of the current QUERY_ routines:

QUERY_BMP	QUERY_PNG
QUERY_DICOM	QUERY_PPM
QUERY_JPEG	QUERY_SRF
QUERY_JPEG2000	QUERY_TIFF
QUERY_PICT	

READ_* Routines

IDL includes a number of routines for reading standard graphics file formats. These routines read the image file format and returns the image and color table vectors (if present). The following is a list of the current READ_ routines:

READ_BMP	READ_PNG
READ_DICOM	READ_PPM
READ_JPEG	READ_SRF
READ_JPEG2000	READ_TIFF
READ_PICT	

WRITE_* Routines

IDL has added a consistent set of write routines to the existing IDL image file format functions to allow users to write an image and its color table vectors to a file of a specified type. The following is a list of the current WRITE_ routines:

WRITE_BMP	WRITE_PNG
WRITE_DICOM	WRITE_PPM
WRITE_JPEG	WRITE_SRF
WRITE_JPEG2000	WRITE_TIFF
WRITE_PICT	

Accessing Files Using Dialogs

File Selection

The `DIALOG_PICKFILE` function allows the user to interactively pick a file using the platform's own native graphical file selection dialog. The user can also enter the name of the file.

Directory Selection

The `DIRECTORY` keyword allows the user to select a directory rather than a file name with the `DIALOG_PICKFILE` function. See [DIALOG_PICKFILE](#) in the *IDL Reference Guide* for details.

```
Result = DIALOG_PICKFILE( )
```

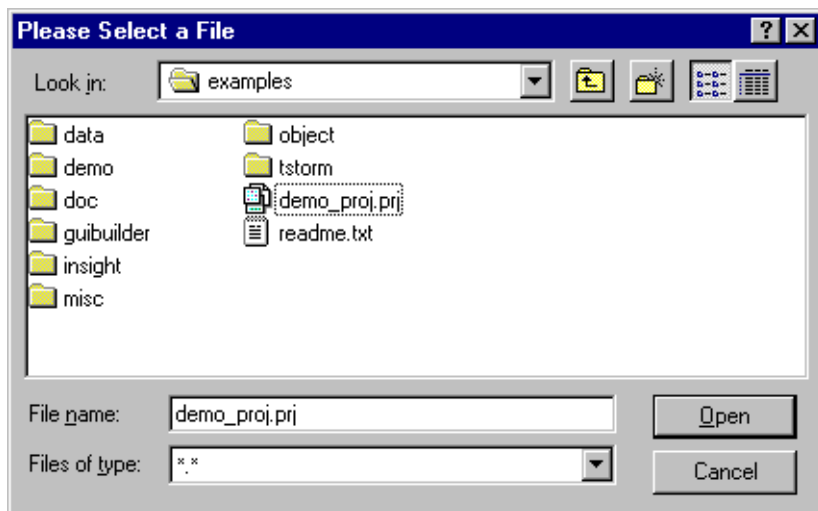


Figure 13-4: `DIALOG_PICKFILE`

Multiple File Selection

The `MULTIPLE_FILES` keyword allows multiple file selection in the dialog. When this keyword is set, the user can select multiple files using the platform-specific selection method and `DIALOG_PICKFILE` can return a string or an array of strings that contains the full path name of the selected file or files.

```
Result = DIALOG_PICKFILE(/MULTIPLE_FILES)
```

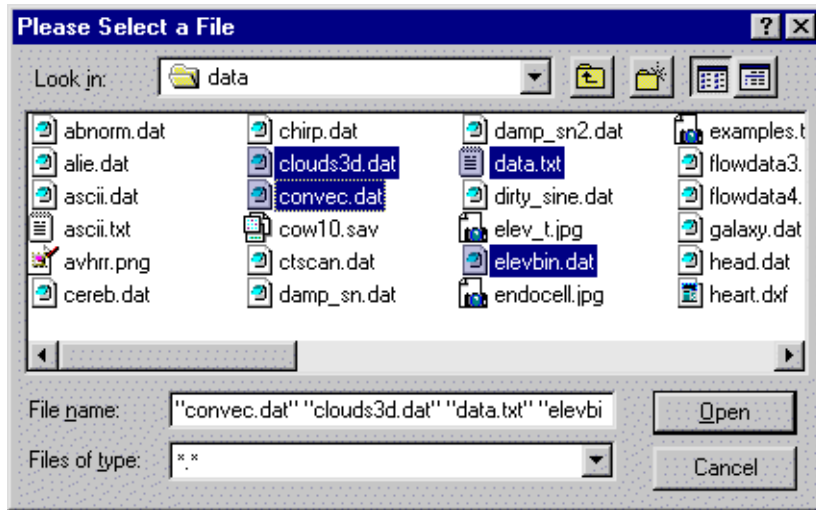


Figure 13-5: MULTIPLE_FILES Selection

Accessing Files With Compound Widgets

Selecting a File

The `CW_FILESEL` is a compound widget which can be used in a component fashion as well as adding multiple file filter selection. The following example illustrates how `CW_FILESEL` could be used to create a widget for opening image files:

```

PRO image_opener_event, event

    WIDGET_CONTROL, event.top, GET_UVALUE=state, /NO_COPY

    CASE event.DONE OF
        0: BEGIN
            state.file = event.VALUE
            WIDGET_CONTROL, event.top, SET_UVALUE=state, /NO_COPY
        END
        1: BEGIN
            IF (state.file NE '') THEN BEGIN
                img = READ_IMAGE(state.file)
                TV, img
            ENDIF
            WIDGET_CONTROL, event.top, SET_UVALUE=state, /NO_COPY
        END
        2: WIDGET_CONTROL, event.top, /DESTROY
    ENDCASE

END

PRO image_opener

    DEVICE, DECOMPOSED=0, RETAIN=2

    base = WIDGET_BASE(TITLE = 'Open Image', /COLUMN)
    filesel = CW_FILESEL(base, /IMAGE_FILTER, FILTER='All Files')
    file=''
    state = {file:file}

    WIDGET_CONTROL, base, /REALIZE
    WIDGET_CONTROL, base, SET_UVALUE=state, /NO_COPY

    XMANAGER, 'image_opener', base

END

```

This code opens the following dialog:



Figure 13-6: CW_FILESEL



Chapter 14: Reading and Writing ASCII Data

This chapter provides an introduction to reading and writing ASCII data using the commands and user interfaces found in IDL.

Overview	222	Advanced File Input/Output	227
Reading an ASCII Data File	223		

Overview

IDL recognizes two types of ASCII data files: free format files, and explicit format files. A free format file uses commas or tabs and spaces to distinguish each element in the file. An explicit format file distinguishes elements according to the commands specified in a format statement.

Reading an ASCII Data File

Most ASCII files are free format files. IDL uses three commands for reading ASCII data files: `READ`, `READF`, and `READS`. The `READ` procedure reads free format data from standard input, `READF` reads free format data from a file, and `READS` reads free format data from a string variable.

Using the `ASCII_TEMPLATE` Function

The `ASCII_TEMPLATE` function generates a template defining an ASCII file format. In this example, two routines are used to input an ASCII data file into IDL. The first routine, `ASCII_TEMPLATE`, is a widget program which allows the user to describe the data organization of the file. This routine creates a template which is used to read the data file, according to the template specifications, by the second routine called `READ_ASCII`. The template is an IDL variable that can be reused by other files with the same organization. The following example creates a template for an ASCII file using the `ASCII_TEMPLATE` function.

```
template = ascii_template( )
```

This command assigns the description of the data to a variable named `template`. IDL will display a dialog box which prompts the user to select a file.

Note

If a filename is specified in the parentheses after the `ASCII_TEMPLATE` function, this screen will not appear.

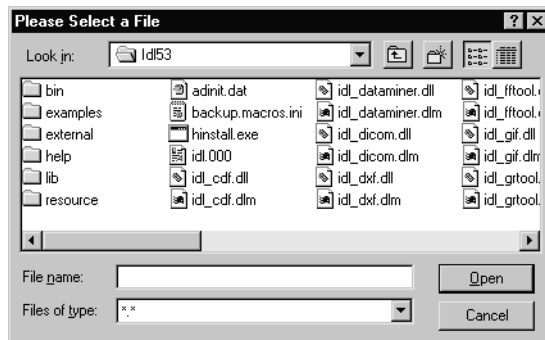


Figure 14-1: File Selection Dialog Box

Once a file has been selected, IDL displays the first of three pages of the ASCII_TEMPLATE dialog.

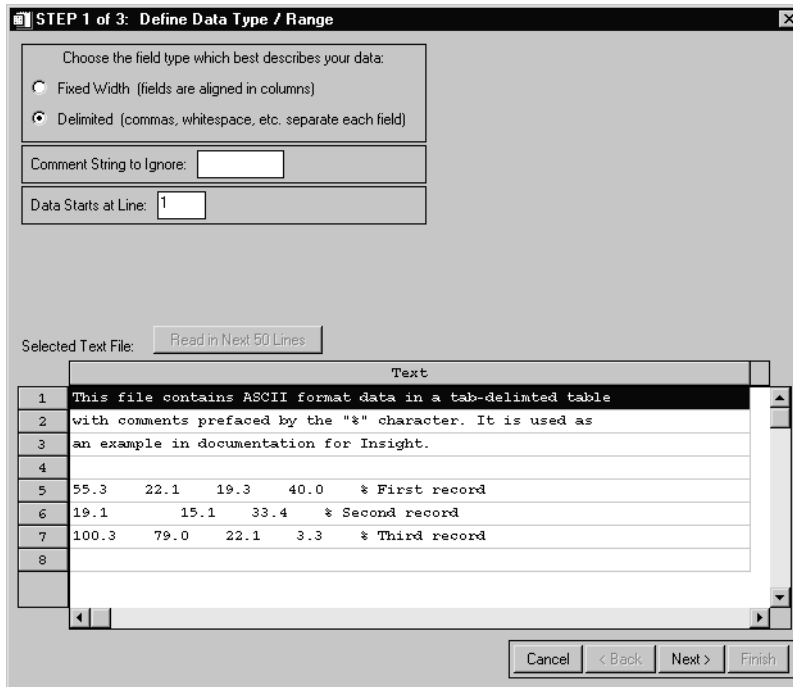


Figure 14-2: ASCII Template - Define Data Type / Range

The first page displays a representative sample of lines from the data file with their numbers on the left. Select the field type that best describes the data. Click the **Next** button on the bottom-right corner of the screen to move to the next page.

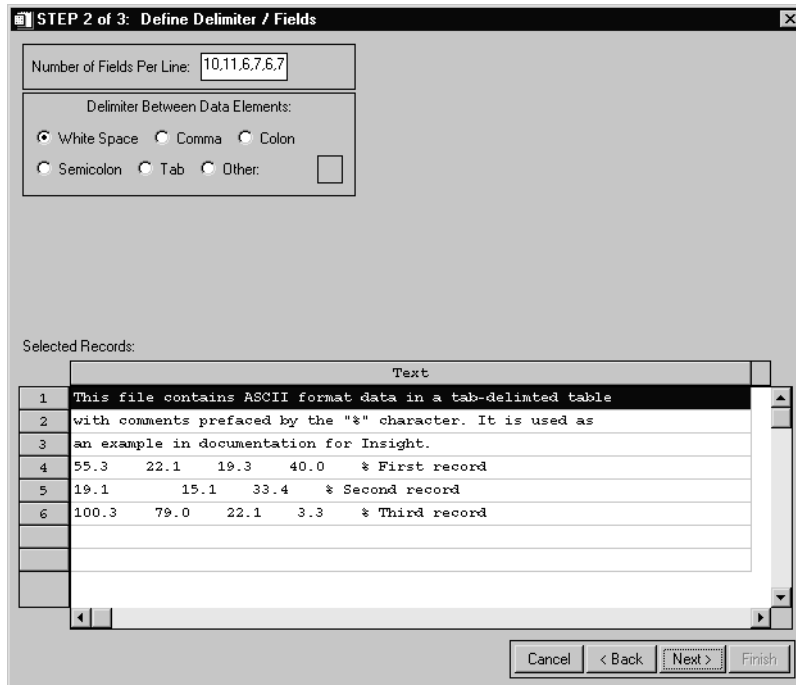


Figure 14-3: ASCII Template - Define Delimiter / Fields

The second page displays the number of fields per line which is listed as three and the white space is selected for the data delimiter. Click the **Next** button on the bottom right corner of the screen to move to the next page.

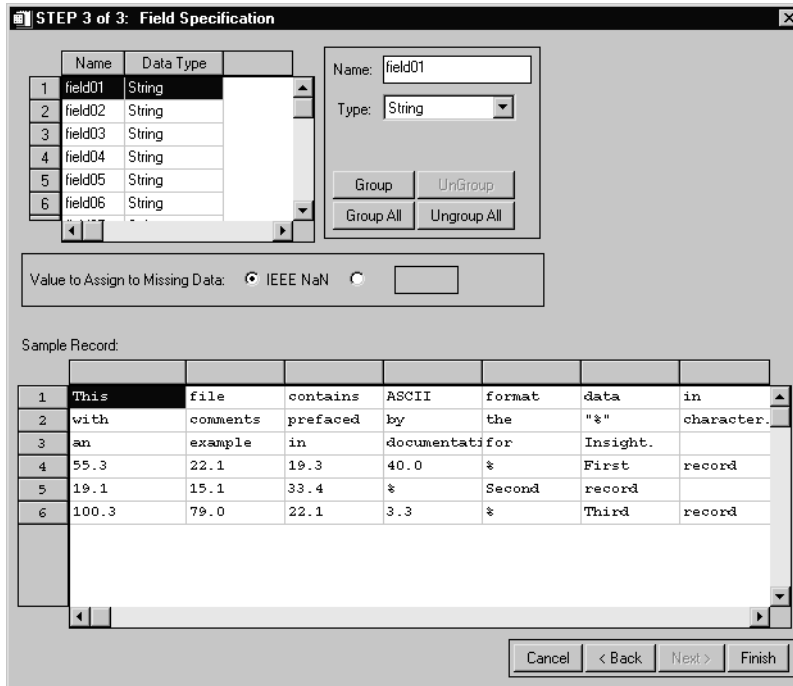


Figure 14-4: ASCII Template - Field Specification

The third page displays the columns in the data set which can be named and their data type specified. Name the fields by typing in the name text at the upper right of the form. Click the **Finish** button on the bottom-right corner of the screen.

The result is an IDL structure variable that describes the data in the file and can be used as input to the `READ_ASCII` command.

Advanced File Input/Output

For information on more advanced file I/O capabilities, see [Chapter 11, “Files and Input/Output”](#) in the *Building IDL Applications* manual.



Chapter 15: Reading and Writing Binary Data

This chapter provides an introduction to reading and writing binary data using the commands and user interfaces found in IDL.

Overview	230	Advanced File Input/Output	235
Reading a Binary Data File	231		

Overview

Binary data or binary data files are more compact than ASCII data files and are frequently used for large data files. Binary data files are stored as one long stream of bytes in a file.

Reading a Binary Data File

To read binary data files, define the variables, open the file for reading, and read the bytes into those variables with the `READU` command. Each variable reads as many bytes out of the file as required by the specified data type and organizational structure.

It is also possible (and often more convenient) to read data from a binary file using the `BINARY_TEMPLATE` and the `READ_BINARY` functions, as described in the following section.

Using the `BINARY_TEMPLATE` Function

A binary template serves as a description of the format of the data in a binary file. A single template can be re-used for all binary files that are organized in the same way. The template specifies user defined fields and file byte order. Templates are created using the `BINARY_TEMPLATE` function and used as the value of the `TEMPLATE` keyword to the `READ_BINARY` function.

When the `BINARY_TEMPLATE` function is invoked, the following dialog is displayed:

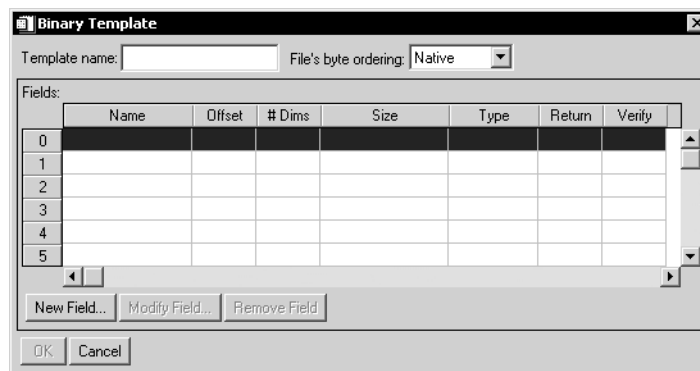


Figure 15-1: Binary Template

The **Template Name** is optional, and can be any string.

The byte order in the file is selected using the using the **File's byte ordering:** pull-down menu. The choices are:

- **Native** — The type of storage method that is native to the machine you are currently running. Little Endian for Intel microprocessor-based machines and Big Endian for Motorola microprocessor-based machines. No byte swapping will be performed.
- **Little Endian** — A method of storing numbers so that the least significant byte appears first in the number. For example, given the hexadecimal number A02B, the little endian method specifies the number to be stored as 2BA0. Specify this if the original file was created on a machine that uses an Intel microprocessor.
- **Big Endian** — A method of storing numbers so that the most significant byte appears first in the number. For example, given the hexadecimal number A02B, the big endian method specifies the number to be stored as A02B. Specify this if the original file was created on a machine that uses a Motorola microprocessor.

Fields are read in the order in which they are listed in the main dialog for `BINARY_TEMPLATE`, with offsets being added to the current file position pointer before each field is read. If a field has already been defined, clicking in the **Return** column will toggle the value of the field between Yes and No. Fields that are not marked for return can be used for calculations by other fields in the template. At least one field must be marked Yes for return in order for the `BINARY_TEMPLATE` function to return a template. Click **New Field...** to enter the description of a new template field. The New Field dialog appears:

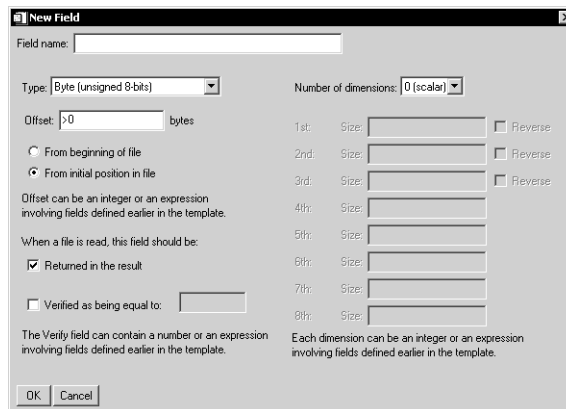


Figure 15-2: Binary Template - New Field

The **Field Name** can be any string.

The **Type** of each Template-specified field is selected from a droplist that offers the following IDL types: byte, integer, long, float, double, complex, dcomplex, uint, ulong, long64 and ulong64. Strings are read as an array of bytes for later conversion to type STRING.

Offsets can be specified using integer values, field names, or any valid IDL expression.

- An absolute integer offset specifies a fixed location (in bytes) from the beginning of the file (or the initial file position for an externally opened file).
- A relative integer offset specifies a position relative to the current file position pointer after the previous field (if any) is read. Relative offsets are shown in the BINARY_TEMPLATE user interface with a preceding > or < character, to indicate a positive (>) or negative (<) byte offset.
- Expressions can include the names of fields that will be read *before* the current field — that is, the field number of the referenced field must be lower than the field number of the field being defined.

The **Verify** field can contain an integer, field name, or any valid IDL expression. Only scalar fields can be verified. READ_BINARY reports an error if a verification fails.

The **Number of Dimensions** of a field can be set via a droplist of values 0 (scalar) to 8 (which is the maximum number of dimensions that an IDL variable can have.) The size of each dimension can be an integer, field name, or any valid IDL expression. Any of the first three dimensions of array data can also be specified to be reversed in order.

Note

If BINARY_TEMPLATE is called by a program that is running in the IDL Virtual Machine, the **Offsets**, **Verify**, and **Size** fields can contain integers or field names, but *not* an IDL expression.

Click **OK** to create the new field definition, and repeat to define all necessary fields.

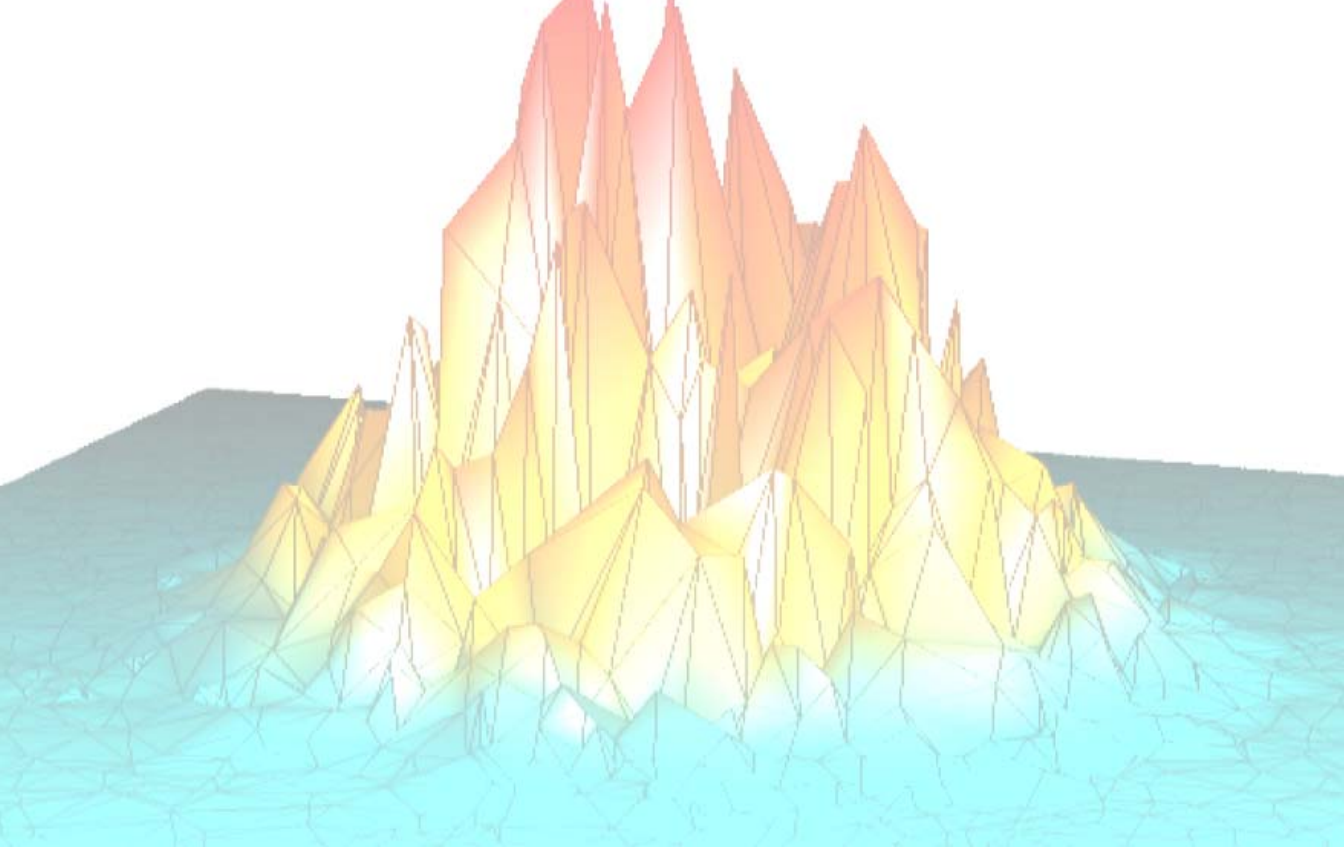
The BINARY_TEMPLATE function returns a structure variable containing the template. The template variable can be saved and used as the value of the TEMPLATE keyword to the READ_BINARY function:

```
template = BINARY_TEMPLATE(file.dat)
Result = READ_BINARY('file.dat', TEMPLATE=template)
```

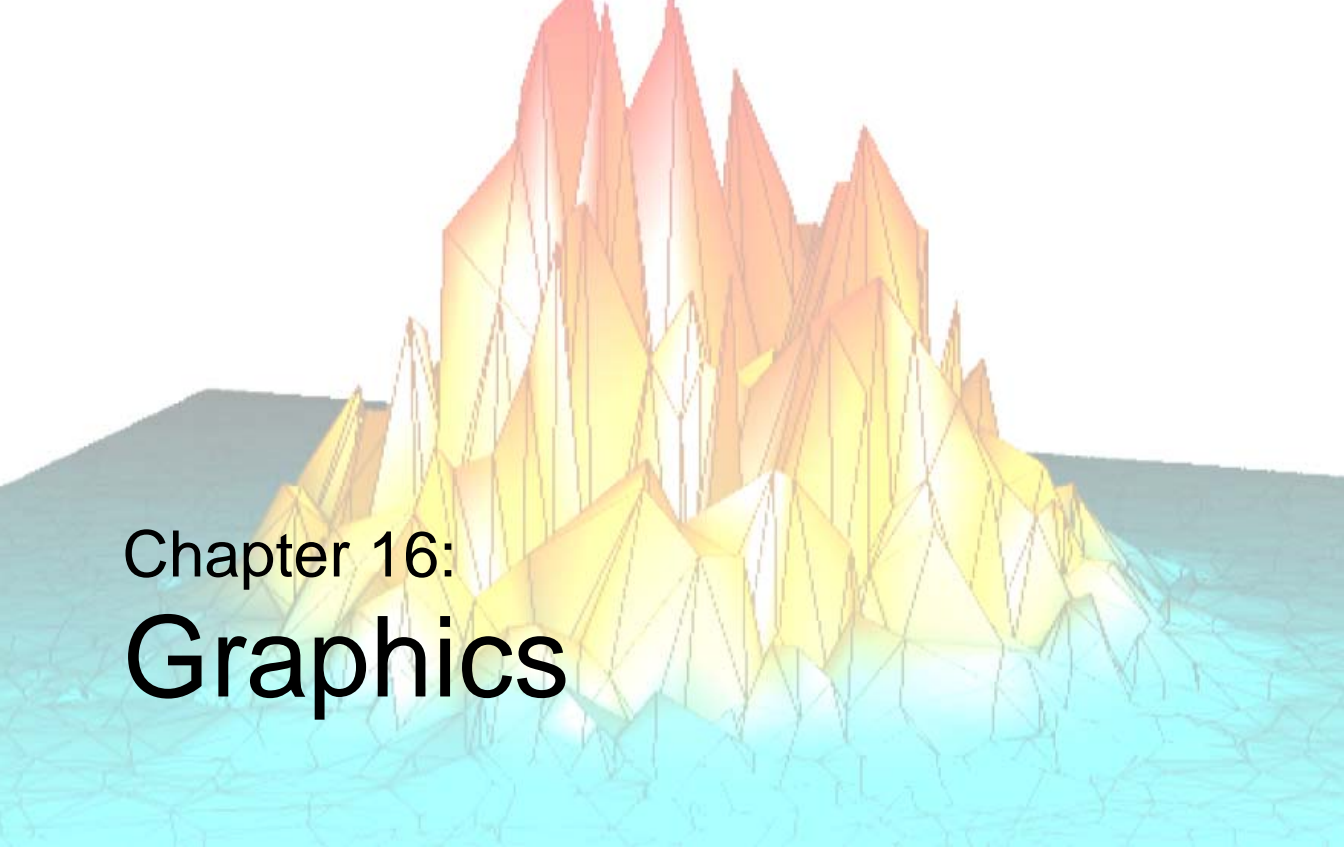
where `file.dat` is a binary data file to be read. The template variable can also be reused as the value of the `TEMPLATE` keyword to `BINARY_TEMPLATE`.

Advanced File Input/Output

For information on more advanced file I/O capabilities, see [Chapter 11, “Files and Input/Output”](#) in the *Building IDL Applications* manual.



Part III: Using Direct Graphics



Chapter 16: Graphics

The following topics are covered in this chapter:

Overview	240	IDL Object Graphics	242
IDL Direct Graphics	241		

Overview

Beginning with IDL version 5.0, IDL supports two distinct graphics modes: Direct Graphics and Object Graphics. Direct Graphics rely on the concept of a current graphics device; IDL commands like `PLOT` or `SURFACE` create images directly on the current graphics device. Object Graphics use an object-oriented programmers' interface to create graphic objects, which must then be drawn, explicitly, to a destination of the programmer's choosing.

IDL Direct Graphics

If you have used versions of IDL prior to version 5.0, you are already familiar with IDL Direct Graphics. The salient features of Direct Graphics are:

- Direct Graphics use a graphics device (**X** for X-windows systems displays, **WIN** for Microsoft Windows displays, **PS** for PostScript files, etc.). You switch between graphics devices using the **SET_PLOT** command, and control the features of the current graphics device using the **DEVICE** command.
- IDL commands that existed in IDL 4.0 use Direct Graphics. Commands like **PLOT**, **SURFACE**, **XYOUTS**, **MAP_SET**, etc. all draw their output directly on the current graphics device.
- Once a direct-mode graphic is drawn to the graphics device, it cannot be altered or re-used. This means that if you wish to re-create the graphic on a different device, you must re-issue the IDL commands to create the graphic.
- When you add a new item to an existing direct-mode graphic (using a routine like **OPLOT** or **XYOUTS**), the new item is drawn in front of the existing items.

IDL Object Graphics

Versions of IDL beginning with version 5.0 include Object Graphics in addition to Direct Graphics. The salient features of Object Graphics are:

- Object graphics are device independent. There is no concept of a current graphics device when using object-mode graphics; any graphics object can be displayed on any physical device for which a destination object can be created.
- Object graphics are object-oriented. Graphic objects are meant to be created and re-used; you may create a set of graphic objects, modify their attributes, draw them to a window on your computer screen, modify their attributes again, then draw them to a printer device without reissuing all of the IDL commands used to create the objects. Graphics objects also encapsulate functionality; this means that individual objects include method routines that provide functionality specific to the individual object.
- Object graphics are rendered in three dimensions. Rendering implies many operations not needed when drawing Direct Graphics, including calculation of normal vectors for lines and surfaces, lighting considerations, and general object overhead. As a result, the time needed to render a given object—a surface, say—will often be longer than the time taken to draw the analogous image in Direct Graphics.
- Object Graphics use a programmer's interface. Unlike Direct Graphics, which are well suited for both programming and interactive, ad hoc use, Object Graphics are designed to be used in programs that are compiled and run. While it is still possible to create and use graphics objects directly from the IDL command line, the syntax and naming conventions make it more convenient to build a program offline than to create graphics objects on the fly.
- Because Object Graphics persist in memory, there is a greater need for the programmer to be cognizant of memory issues and memory leakage. Efficient design—remembering to destroy unused object references and cleaning up—will avert most problems, but even the best designs can be memory-intensive if large numbers of graphic objects (or large datasets) are involved.



Chapter 17: Direct Graphics Plotting

The following topics are covered in this chapter:

Overview	244	Logarithmic Scaling	269
Plotting Keyword Parameters	245	Multiple Plots on a Page	271
Direct Graphics Coordinate Systems	246	Specifying the Location of the Plot	273
Annotation – The XYOUTS Procedure ...	255	Plotting Missing Data	275
Plotting Symbols	257	Date/Time Plotting	276
Polygon Filling	261	Using the AXIS Procedure	287
Tick Marks	265	Using the CURSOR Procedure	291

Overview

IDL includes several routines that can be used to display data in a variety of plot formats, including general x versus y , contour, mesh surface, and perspective plots. The routines allow users to display information in a manner that can be easily understood during data analysis.

Optional keyword parameters and system variables enable users to change certain specifications of the routines, such as scaling, style, and colors, for custom or specialized plots.

This chapter provides examples of scientific graphics in which one variable is plotted as a function of another. The routines for the display of functions of two variables, `CONTOUR`, `SHADE_SURF`, and `SURFACE`, are explained in detail in [“Plotting Multi-Dimensional Arrays”](#) in Chapter 18.

Plotting with iTools

Beginning with IDL 6.0, you can also use the *IDL Intelligent Tools* (iTools) to display plot data. The iTools provide an easy-to-use interface that allows you to manipulate your data after it has been displayed. See [“Introducing the iTools”](#) in Chapter 1 of the *iTool User’s Guide* manual for more on iTools.

Running the Example Code

The examples in this chapter are all written to take advantage of IDL Direct Graphics. Examples and techniques using IDL Object Graphics are contained in the later chapters of this manual.

Some of the example code used in this chapter is part of the IDL distribution. All of the files mentioned are located in the `examples/doc` subdirectory of the IDL distribution. By default, this directory is part of IDL’s path; if you have not changed your path, you will be able to run the examples as described here. See [!PATH](#) in the *IDL Reference Guide* for information on IDL’s path.

Plotting Keyword Parameters

The IDL plotting procedures are designed to produce acceptable results for most applications with a minimum amount of effort. The large number of keyword parameters, described in the *IDL Reference Guide*, in combination with plotting and graphic system variables, allow users to customize the graphics produced by IDL. Most of these keyword parameters pertain to advanced programming. The major keyword parameters are described and illustrated by example in this chapter.

Correspondence with System Variables

Many of the keyword parameters correspond directly to fields in the system variables !P, !X, !Y, or !Z. When specifying a keyword parameter name and value in a call that value affects only the current call, the corresponding system-variable field is not changed. Changing the value of a system-variable field changes the default for that particular parameter and remains in effect until explicitly changed. The system variables involving graphics and their corresponding keywords are detailed in [Appendix D, “System Variables”](#) in the *IDL Reference Guide* manual.

Example—The COLOR Keyword Parameter

The keyword parameter COLOR corresponds to the field COLOR of the system-variable structure !P and is referenced as !P.COLOR. To set the color of a plot to color-index 12, use the following statement:

```
PLOT, X, Y, COLOR = 12
```

Future plots are not affected and are drawn with color index !P.COLOR, which is normally set to the number of available colors minus one.

The interpretation of the color index varies among the devices supported by IDL. With color video displays, this index selects a color (normally a red, green, blue (RGB) triple stored in a device table). You can control the color selected by each color index with the TVLCT procedure which loads the device color tables.

Other devices have a fixed color associated with each color index. With plotters, for example, the correspondence between colors and color index is established by the order of the pens in the carousel.

To change the default color of future plots, use a statement such as:

```
!P.COLOR = 12
```

which sets the default color to color-index 12. You can override this default at any time by including the COLOR keyword in the graphic routine call.

Direct Graphics Coordinate Systems

You can specify coordinates to IDL in one of the following coordinate systems:

DATA Coordinates

This coordinate system is established by the most recent PLOT, CONTOUR, or SURFACE procedure. This system usually spans the plot window, the area bounded by the plot axes, with a range identical to the range of the plotted data. The system can have two or three dimensions and can be linear, logarithmic, or semi-logarithmic. The mechanisms of converting from one coordinate system to another are described below. See “[CONVERT_COORD Function](#)” on page 247.

DEVICE Coordinates

This coordinate system is the physical coordinate system of the selected plotting device. Device coordinates are integers, ranging from (0, 0) at the bottom-left corner to ($V_x - 1$, $V_y - 1$) at the upper-right corner. V_x and V_y are the number of columns and rows addressed by the device. These numbers are stored in the system variable !D as !D.X_SIZE and !D.Y_SIZE.

NORMAL Coordinates

The normalized coordinate system ranges from zero (0) to one (1) over each of the three axes.

Almost all of the IDL graphics procedures accept parameters in any of these coordinate systems. Most procedures use the data coordinate system by default. Routines beginning with the letters TV are notable exceptions. They use device coordinates by default. You can explicitly specify the coordinate system to be used by including one of the keyword parameters /DATA, /DEVICE, or /NORMAL in the call.

Two-Dimensional Coordinate Conversion

The system variables !D, !P, !X, !Y, and !Z contain the information necessary to convert from one coordinate system to another. The relevant fields of these system variables are explained below, and formulae are given for conversions to and from each coordinate system. See [Chapter 18, “Plotting Multi-Dimensional Arrays”](#) for a discussion of three-dimensional coordinates.

In the following discussion, D is a data coordinate, N is a normalized coordinate, and R is a raw device coordinate.

The fields `!D.X_VSIZE` and `!D.Y_VSIZE` always contain the size of the visible area of the currently selected display or drawing surface. Let V_x and V_y represent these two sizes.

The field `!X.S` is a two-element array that contains the parameters of the linear equation, converting data coordinates to normalized coordinates. `!X.S[0]` is the intercept, and `!X.S[1]` is the slope. `!X.TYPE` is 0 for a linear x -axis and 1 for a logarithmic x -axis. The y - and z -axes are handled in the same manner, using the system variables `!Y` and `!Z`.

Also, let D_x be the data coordinate, N_x the normalized coordinate, R_x the device coordinate, V_x the device X size (in device coordinates), and $X_i = !X.S_i$ (the scaling parameter).

With the above variables defined, the linear two-dimensional coordinate conversions for the x coordinate can be written as follows:

Coordinate Conversion	Linear	Logarithmic
Data to normal	$N_x = X_0 + X_1 D_x$	$N_x = X_0 + X_1 \log D_x$
Data to device	$R_x = V_x (X_0 + X_1 D_x)$	$R_x = V_x (X_0 + X_1 \log D_x)$
Normal to device	$R_x = N_x V_x$	$R_x = N_x V_x$
Normal to data	$D_x = (N_x - X_0) / X_1$	$D_x = 10^{(N_x - X_0) / X_1}$
Device to data	$D_x = (R_x / V_x - X_0) / X_1$	$D_x = 10^{(R_x / V_x - X_0) / X_1}$
Device to normal	$N_x = R_x / V_x$	$N_x = R_x / V_x$

Table 17-1: Equations for X-axis Coordinate Conversion

The y - and z -axis coordinates are converted in exactly the same manner, with the exception that there is no z device coordinate and that logarithmic z -axes are not permitted.

CONVERT_COORD Function

The `CONVERT_COORD` function provides a convenient means of computing the above transformations. It can convert coordinates to and from any of the above systems. The keywords `DATA`, `DEVICE`, or `NORMAL` specify the input system.

The output coordinate system is specified by one of the keywords `TO_DATA`, `TO_DEVICE`, or `TO_NORMAL`. For example, to convert the endpoints of a line from data coordinates (0, 1) to (5, 7) to device coordinates, use the following statement:

```
D = CONVERT_COORD([0, 5], [1, 7], /DATA, /TO_DEVICE)
```

On completion, the variable *D* is a (3, 2) vector, containing the *x*, *y*, and *z* coordinates of the two endpoints.

X Versus Y Plots—PLOT and OPLOT

This section illustrates the use of the basic *x* versus *y* plotting routines, `PLOT` and `OPLOT`. `PLOT` produces linear-linear plots by default, and can produce linear-log, log-linear, or log-log plots with the addition of the `XLOG` and `YLOG` keywords.

Data used in these examples are from a fictitious study of Pacific Northwest Salmon fisheries. In the example, we suppose that data were collected in the years 1967, 1970, and from 1975 to 1983. The following IDL statements create and initialize the variables `SOCKEYE`, `COHO`, `CHINOOK`, and `HUMPBAC`, which contain fictitious fish population counts, in thousands, for the 11 observations:

```
SOCKEYE=[463, 459, 437, 433, 431, 433, 431, 428, 430, 431, 430]
COHO=[468, 461, 431, 430, 427, 425, 423, 420, 418, 421, 420]
CHINOOK=[514, 509, 495, 497, 497, 494, 493, 491, 492, 493, 493]
HUMPBAC=[467, 465, 449, 446, 445, 444, 443, 443, 443, 443, 445]
; Construct a vector in which each element contains
; the year of the sample:
YEAR = [1967, 1970, INDGEN(9) + 1975]
```

If you prefer not to enter the data by hand, run the batch file `plot01` with the following command at the IDL prompt:

```
@plot01
```

See “[Running the Example Code](#)” on page 244 if IDL does not find the batch file.

The following IDL commands create a plot of the population of Sockeye salmon, by year:

```
PLOT, YEAR, SOCKEYE, $
    TITLE='Sockeye Population', XTITLE='Year', $
    YTITLE='Fish (thousands)'
```

The `PLOT` procedure, which produces an *x* versus *y* plot on a new set of axes, requires one or two parameters: a vector of *y* values or a vector of *x* values followed by a vector of *y* values. The first attempt at making a plot produces the figure shown

below. Note that the three titles, defined by the keywords TITLE, XTITLE, and YTITLE, are optional.

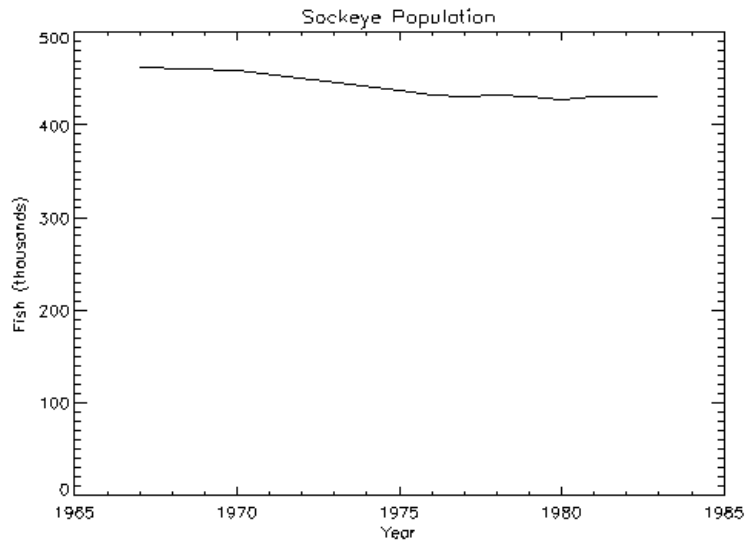


Figure 17-1: Initial Population Plot

Axis Scaling

The fluctuations in the data are hard to see because the scores range from 428 to 463, and the plot's y-axis is scaled from 0 to 500. Two factors cause this effect. By default, IDL sets the minimum y-axis value of linear plots to zero if the y data are all positive. The maximum axis value is automatically set by IDL from the maximum y data value. In addition, IDL attempts to produce from three to six tick-mark intervals that are in increments of an integer power of 10 times 2, 2.5, 5, or 10. In this example, this rounding effect causes the maximum axis value to be 500, rather than 463.

The YNOZERO keyword parameter inhibits setting the y-axis minimum to zero when given positive, nonzero data. The figure below illustrates the data plotted using

this keyword. The y-axis now ranges from 420 to 470, and IDL creates tick-mark intervals of 10.

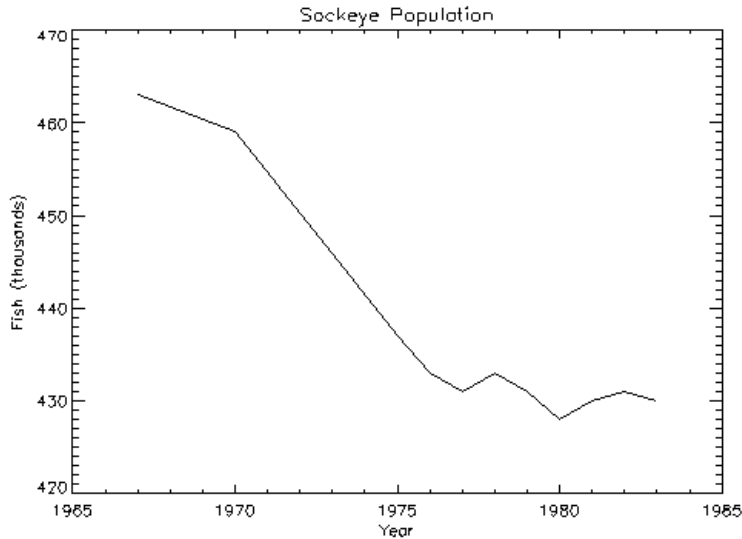


Figure 17-2: Properly Scaled Plot

```

;Define variables:
@plot01
PLOT, YEAR, SOCKEYE, /YNOZERO, $
    TITLE='Sockeye Population', XTITLE='Year', $
    YTITLE='Fish (thousands)'

```

Multiline Titles

The graph-text positioning command `!C`, starts a new line of text output. Titles containing more than one line of text are easily produced by separating each line with this positioning command.

In the above example, the main title could have been displayed on two centered lines by changing the keyword parameter `TITLE` to the following statement:

```
TITLE = 'Sockeye!CPopulation'
```

Note

When using multiple line titles you may find that the default margins are inadequate, causing the titles to run off the page. In this case, set the [XY]MARGIN keywords or increase the values of !X.MARGIN or !Y.MARGIN.

Range Keyword

The range of the x , y , or z axes can be explicitly specified with the [XYZ] RANGE keyword parameter. The argument of the keyword parameter is a two-element vector containing the minimum and maximum axis values.

As explained above, IDL attempts to produce even tick intervals, and the axis range selected by IDL may be slightly larger than that given with the RANGE keyword. To obtain the exact specified interval, set the axis style parameter to one (`YSTYLE = 1`).

The effect of the YNOZERO keyword is identical to that obtained by including the keyword parameter `YRANGE = [MIN(Y), MAX(Y)]` in the call to PLOT. You can make /YNOZERO the default in subsequent plots by setting bit 4 of !Y.STYLE to one (`!Y.STYLE = 16`).

See [STYLE](#) in the *IDL Reference Guide* for details on the STYLE field of the axis system variables !X, !Y, and !Z. Briefly: Other bits in the STYLE field extend the axes by providing a margin around the data, suppress the axis and its notation, and suppress the box-style axes by drawing only left and bottom axes.

For example, to constrain the x-axis to the years 1975 to 1983, the keyword parameter `XRANGE = [1975, 1983]` is included in the call to `PLOT`. The following figure illustrates the result.

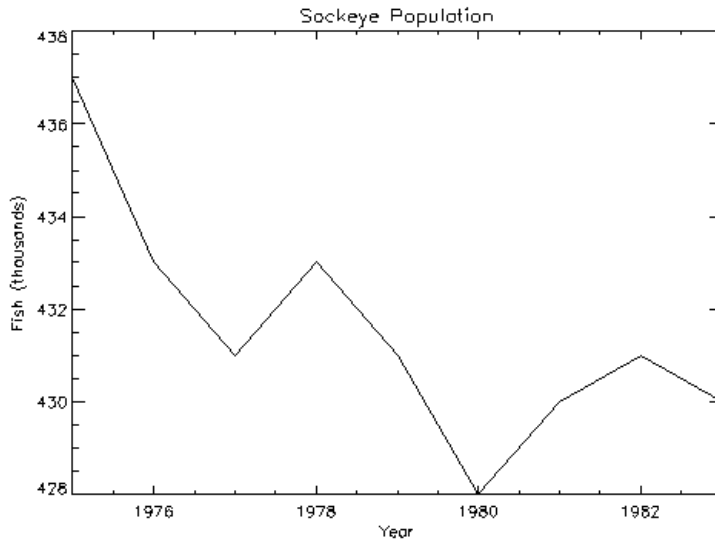


Figure 17-3: Plot with X-Axis Range of 1975 to 1983

Note that the *x*-axis actually extends from 1974 to 1984, as IDL elected to make five tick-mark intervals, each spanning two years. If, as explained above, the *x*-axis style is set to one, the plot will exactly span the given range. The call combining all these options is as follows:

```

; Define variables:
@plot01
PLOT, YEAR, SOCKEYE, /YNOZERO, $
    TITLE='Sockeye Population', XTITLE = 'Year', $
    YTITLE = 'Fish (thousands)', XRANGE = [1975, 1983], /XSTYLE

```

Note

The keyword parameter syntax `/XSTYLE` is synonymous with the expression `XSTYLE = 1`. Setting a keyword parameter to 1 is often referred to as simply setting the keyword.

Overplotting

Additional data can be added to existing plots with the OPLOTT procedure. Each call to PLOT establishes the plot window (the rectangular area enclosed by the axes), the plot region (the box enclosing the plot window and its annotation), the axis types (linear or log), and the scaling. This information is saved in the system variables !P, !X, and !Y and used by subsequent calls to OPLOTT.

Frequently, the color index, line style, or line thickness parameters are changed in each call to OPLOTT to distinguish the data sets. The *IDL Reference Guide* contains a table describing the line style associated with each index.

The figure below illustrates a plot showing all four data sets. Each data set except the first was plotted with a different line style and was produced by a call to OPLOTT. In this example, an (11, 4) array called ALLPTS is defined and contains all the scores for the four categories using the array concatenation operator. Once this array is defined, the IDL array operators and functions can be applied to the entire data set, rather than explicitly referencing the particular sample.

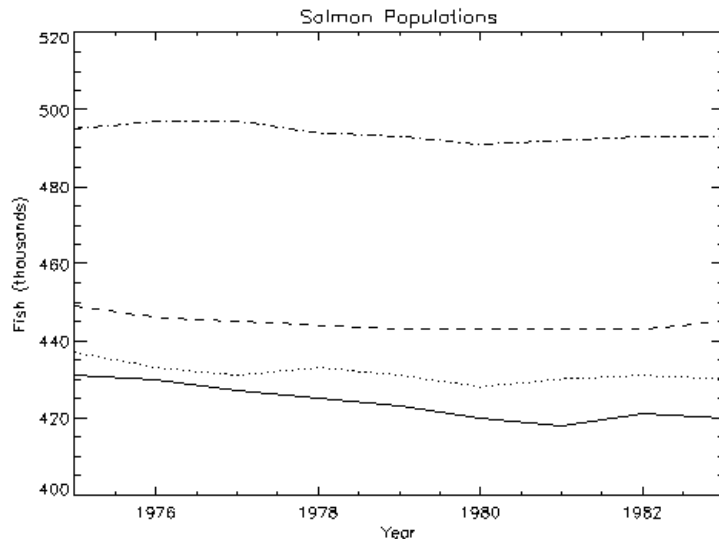


Figure 17-4: Overplotting Using Different Linestyles

First, we define an n -by-4 array containing all four sample vectors. (This array is also defined by the `plot01` batch file.)

```
ALLPTS = [[COHO], [SOCKEYE], [HUMPBACK], [CHINOOK]]
```

The plot in the preceding figure was produced with the following statements:

```
; Define variables:
@plot01
; Plot first graph. Set the y-axis min and max
; from the min and max of all data sets. Default linestyle is 0.
PLOT, YEAR, COHO, YRANGE = [MIN(ALLPTS), MAX(ALLPTS)], $
    TITLE='Salmon Populations', XTITLE = 'Year', $
    YTITLE = 'Fish (thousands)', XRANGE = [1975, 1983], $
    /XSTYLE
; Loop for the three remaining scores, varying the linestyle:
FOR I = 1, 3 DO OPLOT, YEAR, ALLPTS[*], I], LINE = I
```

Annotation – The XYOUTS Procedure

An obvious problem with the previous figure is that each line should be labeled showing what it depicts. The `XYOUTS` procedure is used to write graphic text at a given location. The call to `XYOUTS` to write a string starting at location (x, y) is as follows:

```
XYOUTS, X, Y, STRING
```

See `XYOUTS` in the *IDL Reference Guide* for a complete list of keywords available when adding graphic text to a plot.

The next figure illustrates one method of annotating each graph with its name. The plot was produced exactly as was the previous figure, except that the x -axis range was extended to the year 1990 to allow room for the titles. To accomplish this, the keyword parameter `XRANGE = [1967, 1990]` was added to the call to `PLOT`. A string vector, `NAMES`, containing the names of each sample population also is defined.

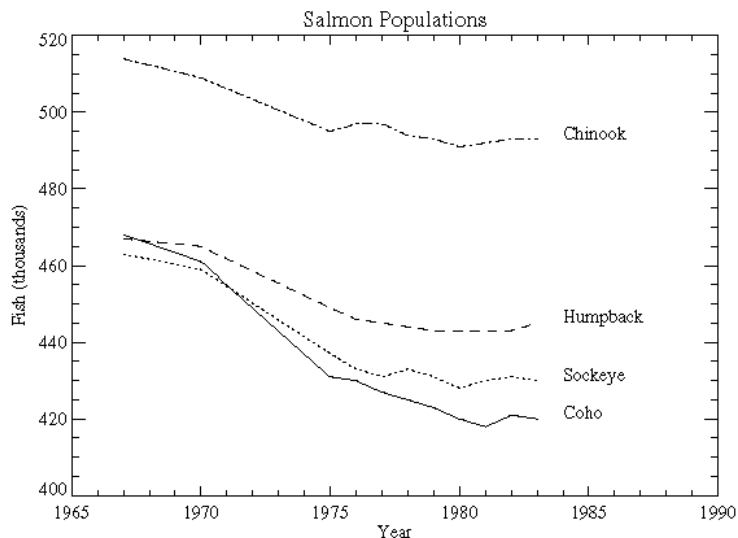


Figure 17-5: Example of Annotating Each Line

First, we define an array containing names for each of the lines plotted. (This array is also defined by the `plot01` batch file.)

```
NAMES=[ 'Coho', 'Sockeye', 'Humpback', 'Chinook' ]
```

The plot was produced with the following statements:

```
; Define variables:
@plot01
; Index of last point:
N1 = N_ELEMENTS(YEAR) - 1
; Plot first graph. Set the y-axis min and max
; from the min and max of all data sets. Default linestyle is 0.
PLOT, YEAR, COHO, YRANGE = [MIN(ALLPTS), MAX(ALLPTS)], $
    TITLE='Salmon Populations', XTITLE = 'Year', $
    YTITLE = 'Fish (thousands)', XRANGE = [1965, 1990], $
    /XSTYLE
; Loop for the three remaining scores, varying the linestyle:
FOR I = 1, 3 DO OPLOT, YEAR, ALLPTS[* , I], LINE = I
; Append the title of each graph on the right:
FOR I = 0, 3 DO XYOUTS, 1984, ALLPTS[N1, I], NAMES[I]
```

Font Selection

The previous figure also illustrates the use of a PostScript font (Times-Roman, in this case) for the titles and annotations. Note that PostScript fonts can only be used when the current graphics devices is set to PostScript.

This font was selected by first setting the default font, controlled by the system variable !P.FONT, to the hardware-font index of zero, and then calling the DEVICE procedure to set the Times-Roman font. To recreate the plot using this font on your system, inspect the batch file `plot02`, located in the `examples/doc` subdirectory of the IDL distribution. Note that running this batch files creates a PostScript file named `plot.ps` in your current working directory. See [“Running the Example Code”](#) on page 244 if IDL does not find the batch file.

Warning

Because not all devices have selectable hardware fonts, default hardware fonts vary. Use of other PostScript fonts and their bold, italic, oblique, and other variants is described in [Appendix H, “Fonts”](#) in the *IDL Reference Guide* manual.

Plotting Symbols

Each data point can be marked with a symbol and/or connected with lines. The value of the keyword parameter `PSYM` selects the marker symbol, as described in the *IDL Reference Guide*. For example, a value of 1 marks each data point with the plus sign (+), 2 is an asterisk (*), etc. Setting `PSYM` to minus the symbol number marks the points with a symbol and connects them with lines. A value of `-1` marks points with a plus sign (+) and connects them with lines. Note also that setting `PSYM` to a value of 10 produces histogram style plots in which a horizontal line is drawn across each x bin.

Frequently, when data points are plotted against the results of a fit or model, symbols are used to mark the data points while the model is plotted using a line. The figure below illustrates this, fitting the Sockeye population values to a quadratic function of the year. The IDL function `POLY_FIT` is used to calculate the quadratic.

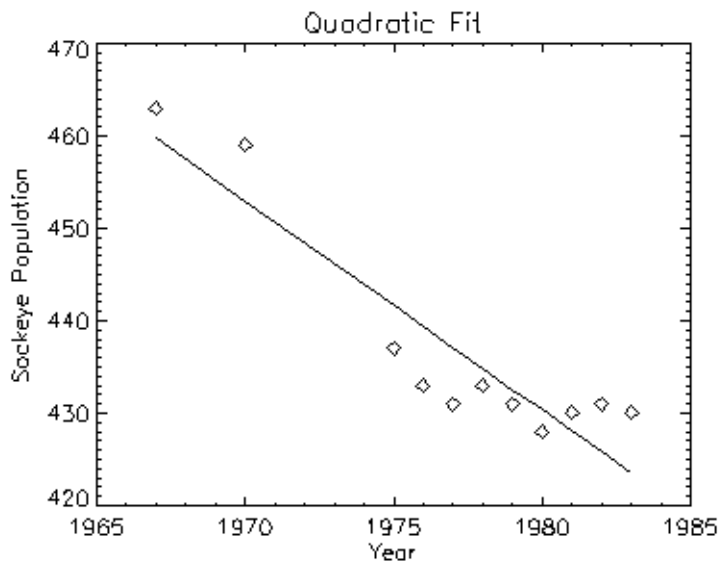


Figure 17-6: Plotting with Predefined Marker Symbols

The statements used to construct the above plot are as follows:

```
; Define variables.  
@plot01  
; Use the LINFIT function to fit the data to a line:
```

```

coeff = LINFIT(YEAR, SOCKEYE)
;YFIT is the fitted line:
YFIT = coeff[0] + coeff[1]*YEAR
; Plot the original data points with PSYM = 4, for diamonds:
PLOT, YEAR, SOCKEYE, /YNOZERO, PSYM = 4, $
    TITLE = 'Quadratic Fit', XTITLE = 'Year', $
    YTITLE = 'Sockeye Population'
; Overplot the smooth curve using a plain line:
OPLOT, YEAR, YFIT

```

Alternatively, you can run the following batch file to create the plot:

```
@plot03
```

See “[Running the Example Code](#)” on page 244 if IDL does not find the batch file.

Defining Your Own Plotting Symbols

The `USERSYM` procedure allows you to define your own symbols by supplying the coordinates of the lines used to draw the symbol. The symbol you define can be drawn using lines or it can be filled using the polygon filling operator. `USERSYM` accepts two vector parameters: a vector of x values and a vector of y values. The coordinate system you use to define the symbol’s shape is centered on each data point, and each unit is approximately the size of a character. For example, to define the simplest symbol, use a one character-wide dash centered over the data point:

```
USERSYM, [-.5, .5], [0, 0]
```

The color and line thickness used to draw the symbols are also optional keyword parameters of `USERSYM`. The following code illustrates the use of `USERSYM` to define a new symbol—a filled circle:

```

; Make a vector of 16 points, A[i] = 2pi/16:
A = FINDGEN(17) * (!PI*2/16.)
; Define the symbol to be a unit circle with 16 points,
; and set the filled flag:
USERSYM, COS(A), SIN(A), /FILL

```

Using the variables defined in the above example, we then create the plot, specifying 8 (user-defined) for the `PSYM` keyword to `PLOT`:

```

PLOT, YEAR, SOCKEYE, /YNOZERO, PSYM = 8, $
    TITLE = 'Quadratic Fit', XTITLE = 'Year', $
    YTITLE = 'Sockeye Population'
; Overplot the smooth curve using a plain line:
OPLOT, YEAR, YFIT

```

The following figure shows the result of this code:

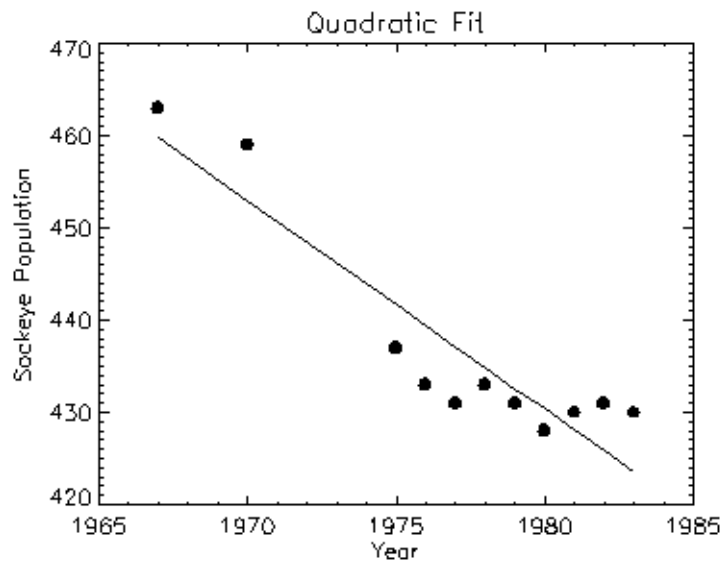


Figure 17-7: Plotting with User-defined Plotting Symbols

See [USERSYM](#) in the *IDL Reference Guide* for additional details.

Histogram Mode

Using the keyword PSYM=10 with the PLOT routines draws graphs in the histogram mode, connecting points with vertical and horizontal lines. This next figure illustrates the comparison between the distribution of the IDL normally distributed random number function (RANDOMN) to the theoretical normal distribution.

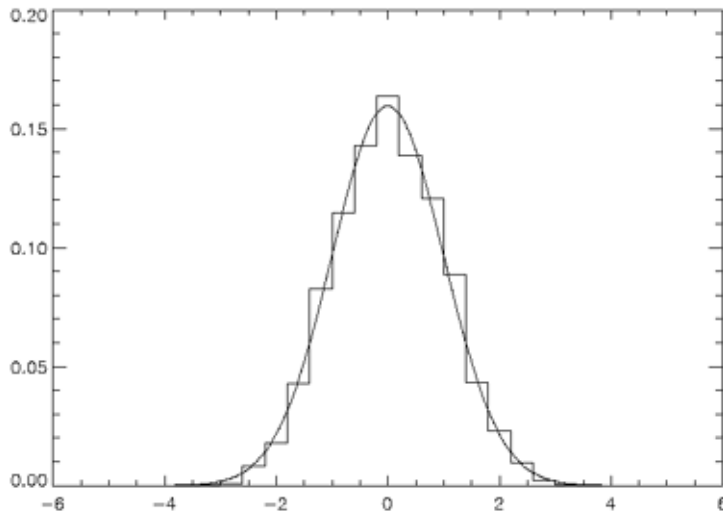


Figure 17-8: Histogram Mode

The plot was produced by the following IDL commands:

```

; Two-hundred values ranging from -5 to 4.95:
X = FINDGEN(200) / 20. - 5.
; Theoretical normal distribution, scale so integral is one:
Y = 1/SQRT(2.*!PI) * EXP(-X^2/2) * (10./200)
; Approximate normal distribution with RANDOMN,
; then form the histogram.
H = HISTOGRAM(RANDOMN(SEED, 2000), $
BINSIZE = 0.4, MIN = -5., MAX = 5.)/2000.
; Plot the approximation using "histogram mode."
PLOT,FINDGEN(26) * 0.4 - 4.8, H, PSYM = 10
; Overplot the actual distribution:
OPLOT, X, Y * 8.

```

Polygon Filling

Many scientific graphs use region filling to highlight the difference between two or more curves, to illustrate boundaries, etc. The IDL `POLYFILL` procedure fills the interior of arbitrary polygons given a list of vertices. The interior of the polygon can be filled with a solid color or with some devices, a user-defined fill pattern contained in a rectangular array.

The figure below illustrates a simple example of polygon filling by filling the region under the Chinook population graph with a color index of 25 percent the maximum, then filling the region under the Sockeye population graph with 50 percent of the maximum index. Because the Chinook populations are always higher than the Sockeye populations, the graph appears as two distinct regions.

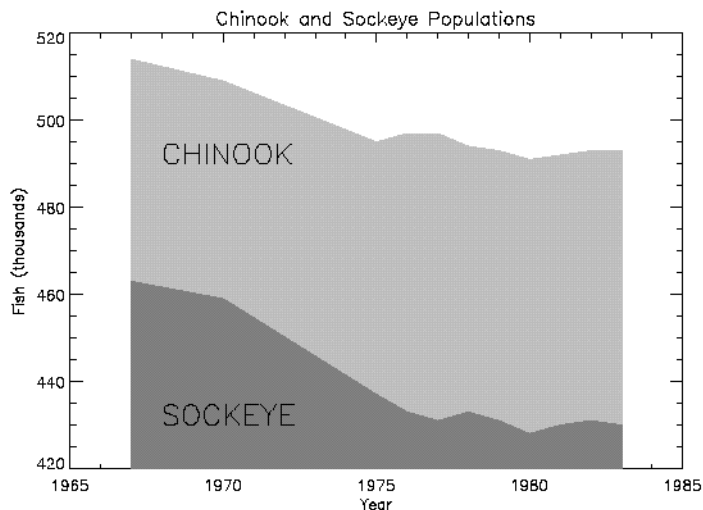


Figure 17-9: Filling Regions Using `POLYFILL`

The program that produced this figure is shown below. It first draws a plot axis with no data, using the `NODATA` keyword. The minimum and maximum y values are directly specified with the `YRANGE` keyword. Because the y-axis range does not always exactly include the specified interval (see “[X Versus Y Plots—PLOT and OPLOT](#)” on page 248), the variable `MINVAL`, is set to the current y-axis minimum, `!Y.CRANGE[0]`. Next, the upper Chinook population region is shaded with a polygon that contains the vertices of the Chinook samples, preceded and followed by

points on the x -axis, (YEAR[0], MINVAL), and (YEAR[n-1], MINVAL). The polygon for the Sockeye samples is drawn using the same method with a different color. Finally, the XYOUTS procedure is used to annotate the two regions.

Enter the following IDL commands to create the plot:

```

; Define variables:
@plot01
; Draw axes, no data, set the range:
PLOT, YEAR, CHINOOK, YRANGE = [MIN(SOCKEYE), MAX(CHINOOK)], $
    /NODATA, TITLE='Sockeye and Chinook Populations', $
    XTITLE='Year', YTITLE='Fish (thousands)'
; Make a vector of x values for the polygon by duplicating
; the first and last points:
PXVAL = [YEAR[0], YEAR, YEAR[N1]]
; Get y value along bottom x-axis:
MINVAL = !Y.CRANGE[0]
; Make a polygon by extending the edges down to the x-axis:
POLYFILL, PXVAL, [MINVAL, CHINOOK, MINVAL], $
    COL = 0.75 * !D.N_COLORS
; Same with second polygon.
POLYFILL, PXVAL, [MINVAL, SOCKEYE, MINVAL], $
    COL = 0.50 * !D.N_COLORS
; Label the polygons:
XYOUTS, 1968, 430, 'SOCKEYE', SIZE=2
XYOUTS, 1968, 490, 'CHINOOK', SIZE=2

```

Alternatively, you can run the following batch file to create the plot:

```
@plot04
```

See [“Running the Example Code”](#) on page 244 if IDL does not find the batch file.

Bar Charts

Bar (or box) charts are used in business-style graphics and are useful in comparing a small number of measurements within a few discrete data sets. Although not designed as a tool for business graphics, IDL can produce many business-style plots with little effort.

The following example produces a box-style chart showing the four salmon populations as boxes of differing colors or shading. The commands used to draw the next figure are shown below with annotation. You do not need to type these commands in yourself; they are collected in the files `plot05.pro`, which contains

the two procedures, and `plot06`, which contains the found in the `examples/doc` subdirectory of the IDL distribution.

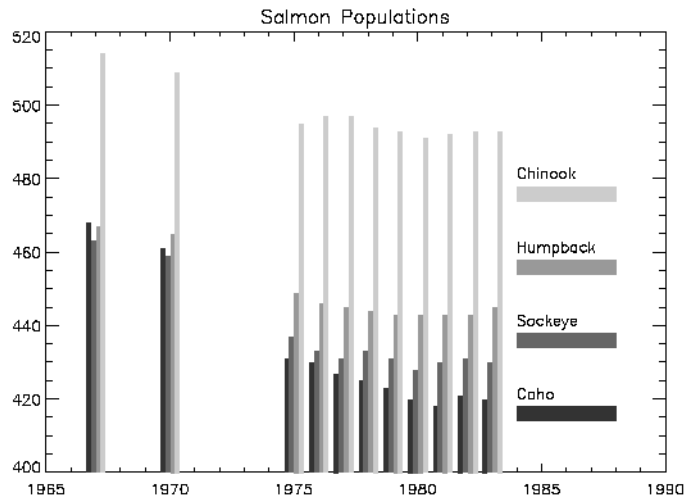


Figure 17-10: Bar Chart Drawn with POLYFILL

First, we define a procedure called `BOX`, which draws a box given the coordinates of two diagonal corners:

```

; Define a procedure that draws a box, using POLYFILL,
; whose corners are (X0, Y0) and (X1, Y1):
PRO BOX, X0, Y0, X1, Y1, color
  ; Call POLYFILL:
  POLYFILL, [X0, X0, X1, X1], [Y0, Y1, Y1, Y0], COL = color
END

```

Next, create a procedure to draw the bar graph:

```

PRO BARGRAPH, minval
  ; Define variables:
  @plot01
  ; Width of bars in data units:
  del = 1./5.
  ; The number of colors used in the bar graph is
  ; defined by the number of colors available on your system:
  ncol=!D.N_COLORS/5
  ; Create a vector of color indices to be used in this procedure:
  colors = ncol*INDGEN(4)+ncol
  ; Loop for each sample:

```

```

FOR iscore = 0, 3 DO BEGIN
; The y value of annotation. Vertical separation is 20 data
; units:
yannot = minval + 20 *(iscore+1)
; Label for each bar:
XYOUTS, 1984, yannot, names[iscore]
; Bar for annotation:
BOX, 1984, yannot - 6, 1988, yannot - 2, colors[iscore]
; The x offset of vertical bar for each sample:
xoff = iscore * del - 2 * del
; Draw vertical box for each year's sample:
FOR iyr=0, N_ELEMENTS(year)-1 DO $
    BOX, year[iyr] + xoff, minval, $
    year[iyr] + xoff + del, $
    allpts[iyr, iscore], $
    colors[iscore]
ENDFOR
END

```

Enter the following at the IDL prompt to compile these two procedures from the IDL distribution:

```
.run plot5.pro
```

To create the bar graph on your screen, enter the following commands.

```

; Load a color table:
LOADCT, 39

```

As in the previous example, the PLOT procedure is used to draw the axes and to establish the scaling using the NODATA keyword.

```

PLOT, year, CHINOOK, YRANGE = [MIN(allpts),MAX(allpts)], $
    TITLE = 'Salmon Populations', /NODATA, $
    XRANGE = [year[0], 1990]
; Get the y value of the bottom x-axis:
minval = !Y.CRANGE[0]
; Create the bar chart:
BARGRAPH, minval

```

Alternatively, you can run the following batch file to create the plot:

```
@plot06
```

See [“Running the Example Code”](#) on page 244 if IDL does not find the batch file.

Tick Marks

You have almost complete control of the number, style, placement, thickness, and annotation of the tick marks. The following plotting keyword parameters and their corresponding system variable fields affect the tick marks:

[XYZ]GRIDSTYLE

The index of the line style to be used for plot tick marks and grids (i.e., when `TICKLEN` is set to 1.0). See [\[XYZ\]GRIDSTYLE](#) in the *IDL Reference Guide* for more information.

[XYZ]MINOR

The number of minor-tick intervals. If set to zero, the default, IDL automatically determines the number of minor ticks in each major tick-mark interval. Setting this parameter to 1 suppresses the minor ticks, and setting it to a positive, nonzero number, n , produces n minor-tick intervals, and $n-1$ minor-tick marks. See [\[XYZ\]MINOR](#) in the *IDL Reference Guide* for more information.

[XYZ]THICK

The thickness of the x , y , or z axes and their tick marks. This parameter is set with the field `THICK` in the axes system variables, `!X`, `!Y`, and `!Z` (e.g., `!X.THICK` controls the x -axis thickness). There are no keyword parameters affecting the axis thickness. See [\[XYZ\]THICK](#) in the *IDL Reference Guide* for more information.

[XYZ]TICKFORMAT

Set this keyword to a format string or a string containing the name of a function that returns a string to be used to format the axis tick mark labels. See [\[XYZ\]TICKFORMAT](#) in the *IDL Reference Guide* for more information.

TICKLEN

The length of each major-tick mark, expressed as a fraction of the window size in the tick mark's direction. The default value is 0.02. A length of 1.0 produces a grid. A value of -0.02 makes tick marks that extend away from the plot. Individual axis ticks can be controlled with the `[XYZ]TICKLEN` keyword. See [TICKLEN](#) in the *IDL Reference Guide* for more information.

[XYZ]TICKNAME

A string array containing the annotation of each major-tick mark. If omitted or if a given string element contains the null string, IDL labels the tick mark with its value. To suppress the tick labels, supply a string array of one-character long, blank strings, i.e., `REPLICATE(' ', N)`. Null strings force IDL to number the tick mark with its value. See [\[XYZ\]TICKNAME](#) in the *IDL Reference Guide* for more information.

Note

If there are n tick-mark intervals, there are $n + 1$ tick marks and labels.

[XYZ]TICKS

The number of major tick-mark intervals. If set to zero or omitted, IDL produces between three and six intervals. See [\[XYZ\]TICKS](#) in the *IDL Reference Guide* for more information.

[XYZ]TICKV

The data values of each tick mark. You can directly specify these values, producing graphs with arbitrary tick marks. If you do this, IDL scales the axis from the first tick value to the last unless you directly specify a range. As above, be sure to provide $n + 1$ tick values. See [\[XYZ\]TICKV](#) in the *IDL Reference Guide* for more information.

Example: Specifying Tick Marks

The following figure shows a box chart illustrating the direct specification of the x -axis tick values, number of ticks, and tick names. Building upon the previous program, this program shows each of the four scores for the year 1967, the first year

in our data. It uses the BOX procedure from the previous example to draw a rectangle for each sample.

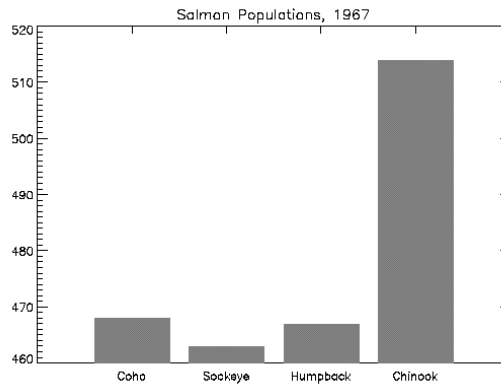


Figure 17-11: Controlling Tick Marks and Their Annotation

Using the data and variables from above, the following commands create the box chart:

Enter the following command at the IDL prompt to compile the BOX and BARGRAPH procedures (discussed in the previous example) from the IDL distribution:

```
.run plot05.pro
```

Enter the following commands to create the box chart:

```
; Define variables:
@plot01
; Tick x values, 0.2, 0.4, 0.6, 0.8:
XVAL = FINDGEN(4)/5. + .2
; Make a vector of scores from first year, corresponding to
; the name vector from above:
YVAL = [COHO[0], SOCKEYE[0], HUMPBACK[0], CHINOOK[0]]
; Make the axes with no data. Force x range to [0, 1],
; centering xval, which also contains the tick values.
; Force three tick intervals making four tick marks.
; Specify the tick names from the names vector:
PLOT, XVAL, YVAL, /YNOZERO, XRANGE = [0,1], XTICKV = XVAL, $
      XTICKS = 3, XTICKNAME = NAMES, /NODATA, $
      TITLE = 'Salmon Populations, 1967'
; Draw the boxes, centered over the tick marks.
; !Y.CRANGE[0] is the y value of the bottom x-axis.
FOR I = 0, 3 DO BOX, XVAL[I] - .08, !Y.CRANGE[0], $
      XVAL[I] + 0.08, YVAL[I], 128
```

Alternatively, you can run the following batch file to create the plot:

```
@plot07
```

See [“Running the Example Code”](#) on page 244 if IDL does not find the batch file.

More Tick Mark Examples

See [“Multiple Plots on a Page”](#) on page 271 for more examples of ways you can control where axes are drawn, tick mark length, and placement.

Logarithmic Scaling

The XLOG, YLOG, and ZLOG keywords can be used with the PLOT routine to get any combination of linear and logarithmic axes. The OPLOT procedure uses the same scaling and transformation as did the most recent plot.

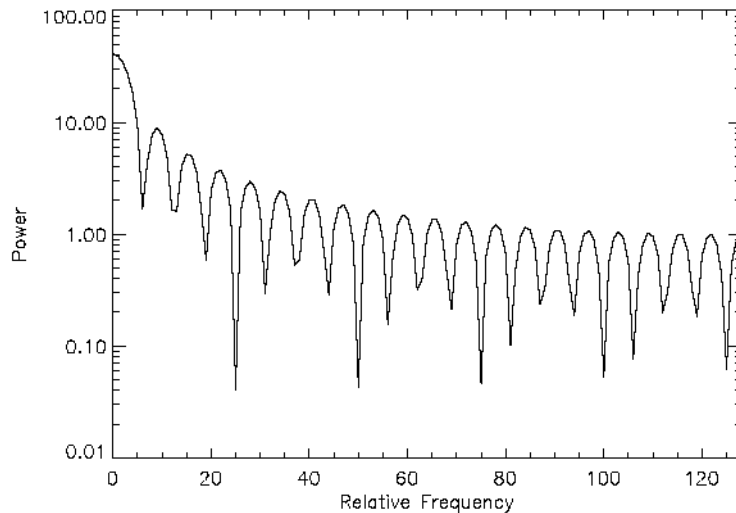


Figure 17-12: Example of Logarithmic Scaling

The figure illustrates using PLOT to make a linear-log plot. It was produced with the following statements:

```

; Create data array:
X = FLTARR(256)
; Make a step function. Array elements 80 through 120 are set to 1:
X[80:120] = 1
; Make a filter:
FREQ = FINDGEN(256)
; Make the filter symmetrical about the value x = 128:
FREQ = FREQ < (256-FREQ)
; Second order Butterworth, cutoff frequency = 20.
FIL = 1./(1+(FREQ/20)^2)
; Plot with a logarithmic x-axis. Use exact axis range:
PLOT, /YLOG, FREQ, ABS(FFT(X,1)), $
      XTITLE = 'Relative Frequency', YTITLE = 'Power', $
      XSTYLE = 1

```

```
; Plot graph:  
OPLOT, FREQ, FIL
```

Alternatively, you can run the following batch file to create the plot:

```
@plot08
```

See [“Running the Example Code”](#) on page 244 if IDL does not find the batch file.

Multiple Plots on a Page

Plots can be ganged on the display or page in the horizontal and/or vertical directions using the system variable field `!P.MULTI`. IDL sets the plot window to produce the given number of plots on each page and moves the window to a new sector at the beginning of each plot. If the page is full, it is first erased. If more than two rows or columns of plots are produced, IDL decreases the character size by a factor of 2.

`!P.MULTI` controls the output of multiple plots. Set `!P.MULTI` equal to an integer vector in which:

- The first element of the vector contains the number of empty sectors remaining on the page. The display is erased if this field is zero when a new plot is begun.
- The second element of the vector contains the number of plots per page in the horizontal direction.
- The third element contains the number of plots per page in the vertical direction.
- The fourth element contains the number of plots stacked in the Z dimension.
- The fifth element controls the order in which plots are drawn. Set the fifth element equal to zero to make plots from left to right (column major), and top to bottom. Set the fifth element equal to one to make plots from top to bottom, left to right (row major).

Omitting any of the five elements from the vector is the same as setting that element equal to zero.

For example, to set up IDL to stack two plots vertically on each page, use the following statement:

```
!P.MULTI = [0, 1, 2]
```

Note that the first element, `!P.MULTI (0)`, is set to zero to cause the next plot to begin a new page. To make four plots per page with two columns and two rows, use the following statement:

```
!P.MULTI = [0, 2, 2]
```

To reset to the default of one plot per page, set the value of `!P.MULTI` to 0, as shown in the following statement:

```
!P.MULTI = 0
```

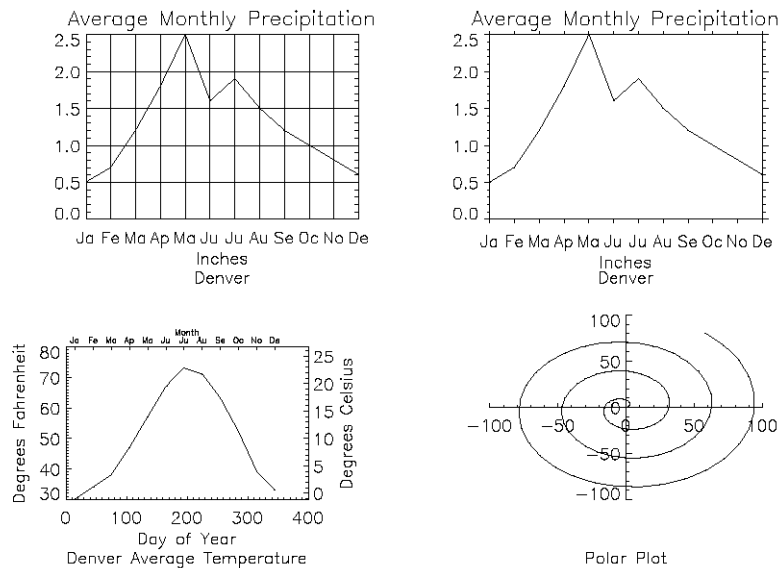


Figure 17-13: Multiple Plots Per Page, Various Tick Marks, and Multiple Axes

This figure shows four plots in a single window. For details, inspect the batch file `plot09` in the `examples/doc` subdirectory of the IDL distribution. Note the following features of the plots in the figure:

1. The plot in the upper left has grid-style tick marks. This is accomplished by setting the `TICKLEN` keyword equal to 1.0
2. The plot in the upper right has outward-facing tick marks. This is accomplished by setting the `TICKLEN` keyword to a negative value.
3. The plot in the lower left corner has different axes on left and right, top and bottom. This is accomplished by drawing the top and right axes separately, using the `AXIS` procedure.
4. The plot in the lower right uses no default axes at all. The centered axes are drawn with calls to the `AXIS` procedure.

Specifying the Location of the Plot

The plot-data window is the region of the page or screen enclosed by the axes. The plot region is the box enclosing the plot-data window and the titles and tick annotation.

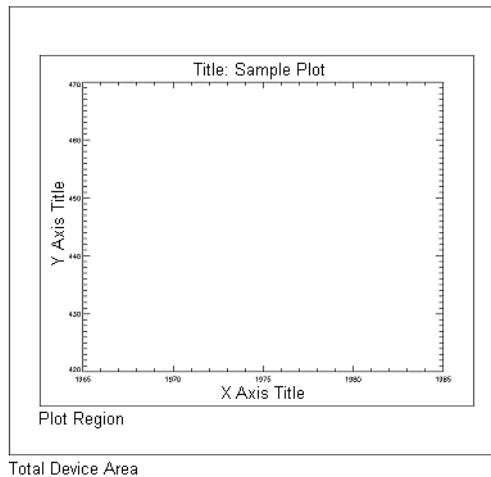


Figure 17-14: The Plot-Data Window, Plot Region, and Device Area Relationship

The figure illustrates the relationship of the plot-data window, plot region, and the entire device area. These areas are determined by the following system variables and keyword parameters, in order of decreasing precedence:

POSITION

The POSITION keyword is accepted by the CONTOUR, MAP_SET, PLOT, SHADE_SURF, and SURFACE routines. Its value is a four-element vector (six elements for three-dimensional plots) containing the position of the axis endpoints: $[x_0, y_0, x_1, y_1]$. Coordinates are specified in normalized coordinates or in device coordinates if the DEVICE keyword is present.

!P.POSITION

!P.POSITION is the system variable equivalent of the POSITION keyword. Its value is a four-element vector in the same form as above containing the normalized coordinates of the plot-data window. !P.POSITION is ignored if x_0 is equal to x_1 , (that is, if !P.POSITION[0] EQ !P.POSITION[2]), which is the default.

!P.REGION

The !P.REGION system variable is another four-element vector in the same form as above containing the normalized coordinates of the plot region, the rectangle enclosing the plot-data window and annotation. It is ignored if !P.REGION [0] is equal to !P.REGION[2].

!P.MULTI

!P.MULTI controls the number of plots per page. It is described in “[Multiple Plots on a Page](#)” on page 271.

[XYZ]MARGIN

The [XYZ]MARGIN keywords are accepted by the AXIS, CONTOUR, PLOT, SHADE_SURF, and SURFACE routines. The value of each of these keywords is a 2-element array specifying the margin on the left and right sides (XMARGIN) or the top and bottom (YMARGIN) of the plot window, in units of character size. Default margins are 10 and 3 for the x -axis, and 4 and 2 for the y -axis. The ZMARGIN keyword is present for consistency and is currently ignored.

![XYZ]MARGIN

![XYZ]MARGIN are the system variable equivalents of the [XYZ]MARGIN keywords.

Plotting Missing Data

The `MAX_VALUE` and `MIN_VALUE` keywords to `PLOT` can be used to create missing data plots wherein bad data values are not plotted. Data values greater than the value of the `MAX_VALUE` keyword or less than the value of the `MIN_VALUE` keyword are treated as missing and are not plotted. The following code creates a dataset with bad data values and plots it with and without these keywords:

```

; Make a 100-element array where each element is
; set equal to its index:
A = FINDGEN(100)
; Set 20 random point in the array equal to 400.
; This simulates "bad" data values above the range
; of the "real" data.
A(RANDOMU(SEED, 20)*100)=400
; Set 20 random point in the array equal to -10.
; This simulates "bad" data values below the range
; of the "real" data.
A(RANDOMU(SEED, 20)*100)=-10
; Plot the dataset with the bad values. Looks pretty bad!
PLOT, A
; Plot the dataset, but don't plot any value over 101.
; The resulting plot looks better, but still shows spurious values:
PLOT, A, MAX_VALUE=101
; This time leave out both high and low spurious values.
; The resulting plot more accurately reflects the "real" data:
PLOT, A, MAX_VALUE=101, MIN_VALUE=0

```

The following plotting routines allow you to set maximum and minimum values in this manner: [CONTOUR](#), [PLOT](#), [SHADE_SURF](#), [SURFACE](#).

In addition to the maximum and minimum values specified with the `MAX_VALUE` and `MIN_VALUE` keywords, these plotting routines treat the IEEE floating-point value NaN (Not A Number) as missing data automatically. (For more information on NaN, see “[Special Floating-Point Values](#)” in *Building IDL Applications*.)

Date/Time Plotting

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

Julian Dates and Times

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

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

Table 17-2: Example Julian Dates

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

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

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

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000, respectively.

Precision of Date/Time Data

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

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

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

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

; Get machine characteristics:
machine = MACHAR(/DOUBLE)

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

; Convert to seconds:
PRINT, precision*86400d0
```

How to Generate Date/Time Data

The **TIMEGEN** function returns an array of double precision floating point values that represent date/time in terms of Julian dates. The first value of the returned array

corresponds to a start date/time, and each subsequent value corresponds to the start date/time plus that array element's one-dimensional subscript multiplied by a step size for a given date/time unit. Unlike the other array generation routines in IDL, TIMEGEN includes a START keyword, which is necessary if the starting date/time is originally provided in calendar (month, day, year) form.

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

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

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

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

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

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

2. Using the calendar format codes:

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

The resulting calendar dates are printed out as follows:

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

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

Displaying Date/Time Data on an Axis in Direct Graphics

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

Displaying Date/Time Data on a Plot Display

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

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

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

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

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

where %I represents minutes and %S represents seconds.

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

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

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

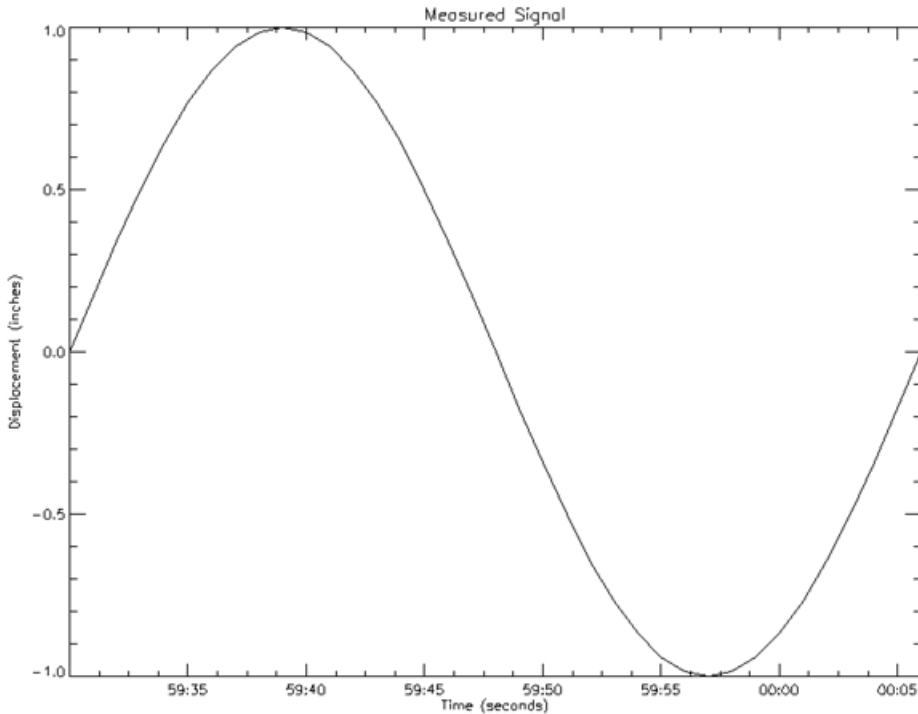


Figure 17-15: Displaying Date/Time data with PLOT

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

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

where %H represents hours, %D represents days, %M represents months, and %Y represents years. Notice DATE_FORMAT is specified with a three element vector.

Date/time data can be displayed on an axis with three levels. The format of these levels are specified through this vector.

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

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

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

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

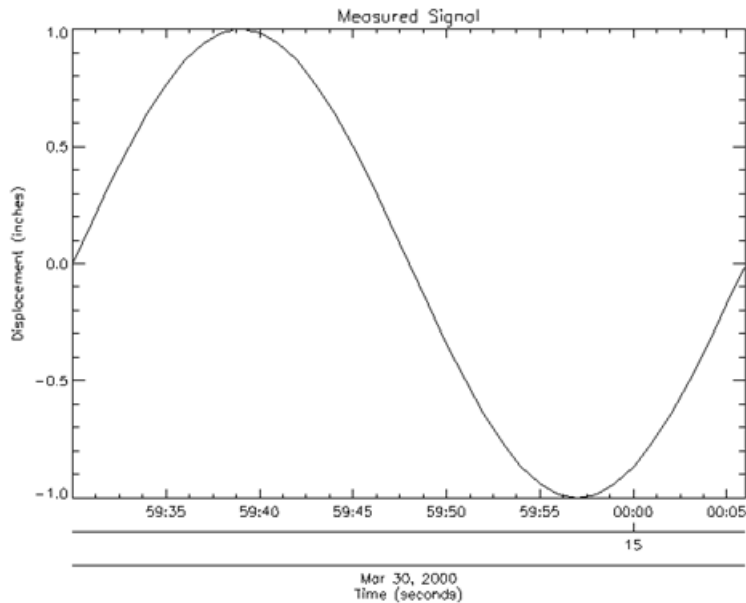


Figure 17-16: Displaying Three Levels of Date/Time data with PLOT

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

Displaying Date/Time Data on a Contour Display

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

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

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

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

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

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

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

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

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

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

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

As in the plot example, the `POSITION` keyword is set to allow the resulting display to contain all three levels and the title of the date/time axis. The `YTICKUNITS` keyword is set to note the unit of each level. And the `YTICKINTERVAL` keyword is set to place the major tick marks at every five second interval.

This example also contains the `YTICKLAYOUT` keyword. By default, this keyword is set to 0, which provides the date/time layout shown in the plot example.

In this example, `YTICKLAYOUT` is set to 2, which rotates and boxes the tick labels to provide the following results:

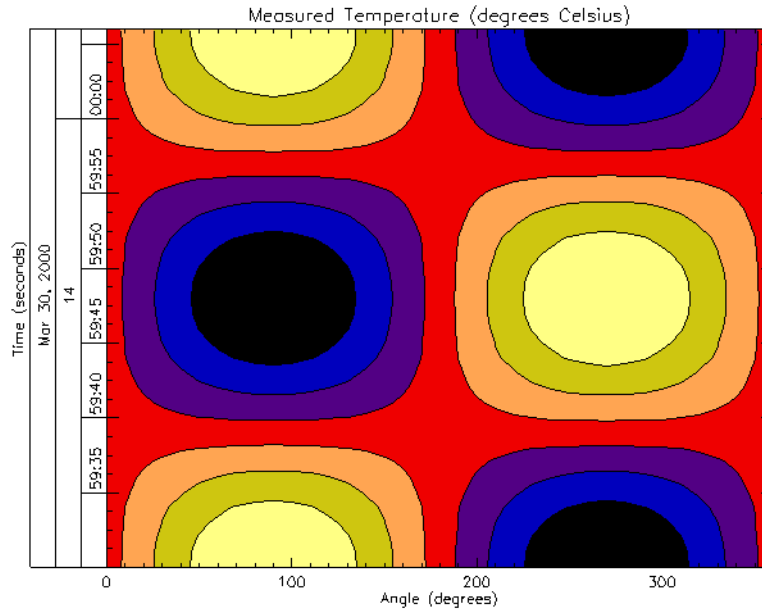


Figure 17-17: Displaying Date/Time Data with `CONTOUR`

Using System Variables to Display Date/Time Data

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

Keywords	System Variables
<code>[XYZ]TICKUNITS</code>	<code>![XYZ].TICKUNITS</code>
<code>[XYZ]TICKINTERVAL</code>	<code>![XYZ].TICKINTERVAL</code>
<code>[XYZ]TICKLAYOUT</code>	<code>![XYZ].TICKLAYOUT</code>

Table 17-3: Relationship Between Keywords and System Variables

Usually, keywords are used more frequently than system variables, but system variables are better when trying to establish a consistent display style. For example,

we could have established a date/time axis style with these system variables before producing our previous displays:

```

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

```

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

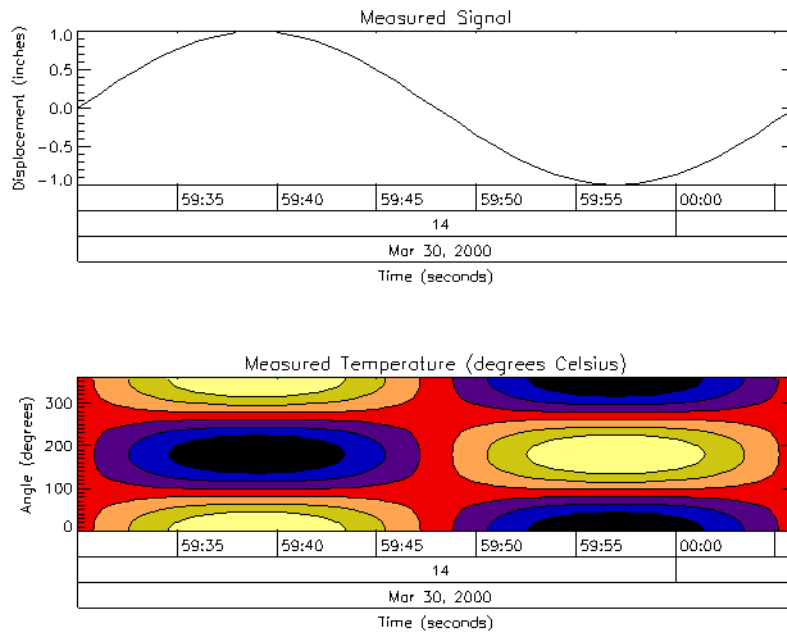


Figure 17-18: Date/Time Axis Style Established With System Variables

Using the AXIS Procedure

The `AXIS` procedure draws and annotates an axis. It optionally saves the scaling established by the axis for use by subsequent graphics procedures. It can be used to add additional axes to plots or to draw axes at a specified position.

The `AXIS` procedure accepts the set of plotting keyword parameters that govern the scaling and appearance of the axes. Additionally, the keyword parameters `XAXIS`, `YAXIS`, and `ZAXIS` specify the orientation and position (if no position coordinates are present) of the axis. The value of these parameters are 0 for the bottom or left axis and 1 for the top or right. The tick marks and their annotation extend away from the plot window. For example, specify `YAXIS = 1` to draw a y-axis on the right of the window.

The optional keyword parameter `SAVE` saves the data-scaling parameters established for the axis in the appropriate axis system variable, `!X`, `!Y`, or `!Z`. The call to `AXIS` is as follows:

```
AXIS[ [, X, Y], Z]
```

where `X`, `Y`, and optionally `Z` specify the coordinates of the axis. Any of the coordinate systems can be used by including the appropriate coordinate keyword in the call. The coordinate corresponding to the axis direction is ignored. When specifying an `x`-axis, the `x`-coordinate parameter is ignored, but must be present if there is a `y` coordinate.

Example: The AXIS Procedure

The figure shown below illustrates using `AXIS` to draw axes with a different scale, opposite the main `x`- and `y`-axes. The plot is produced using `PLOT` with the bottom and left axes annotated and scaled in units of days and degrees Fahrenheit. The `XMARGIN` and `YMARGIN` keyword parameters are specified to allow additional room around the plot window for the new axes. The keyword parameters `XSTYLE = 8` and `YSTYLE = 8` inhibit drawing the top and right axes.

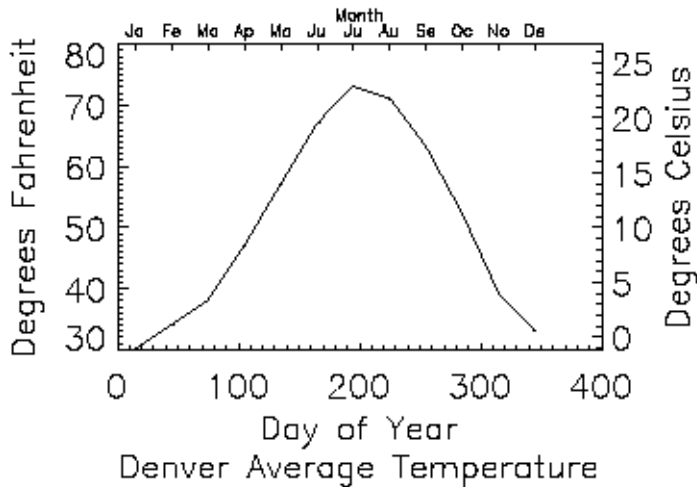


Figure 17-19: A plot created with the `AXIS` procedure

Next, the `AXIS` procedure is called to draw the top, `XAXIS = 1`, axis, labeled in months. Eleven tick intervals with 12 tick marks are drawn. The x value of each monthly tick mark is the day of the year that is approximately the middle of the month. Tick-mark names come from the `MONTH` string array.

The right y -axis, `YAXIS = 1`, is drawn in the same manner. The new y -axis range is set by converting the original y -axis minimum and maximum values, saved by `PLOT` in `!Y.CRANGE`, from Fahrenheit to Celsius, using the formula $C = 5(F-32)/9$. The keyword parameter `YSTYLE = 1` forces the y -axis range to match the given range exactly. The program is as follows:

```

; Plot the data, omit right and top axes:
PLOT, DAY, TEMP, /YNOZERO, $
    SUBTITLE = 'Denver Average Temperature', $
    XTITLE = 'Day of Year', $
    YTITLE = 'Degrees Fahrenheit', $
    XSTYLE=8, YSTYLE=8, XMARGIN=[8, 8], YMARGIN=[4, 4]
; Draw the top x-axis, supplying labels, etc.
; Make the characters smaller so they will fit:
AXIS, XAXIS=1, XTICKS=11, XTICKV=DAY, XTICKN=MONTH, $
    XTITLE='Month', XCHARSIZE = 0.7
; Draw the right y-axis. Scale the current y-axis minimum
; values from Fahrenheit to Celsius and make them

```



```

; the new min and max values. Set YSTYLE=1 to make axis exact.
AXIS, YAXIS=1, YRANGE = (!Y.CRANGE-32)*5./9., YSTYLE = 1, $
  YTITLE = 'Degrees Celsius'

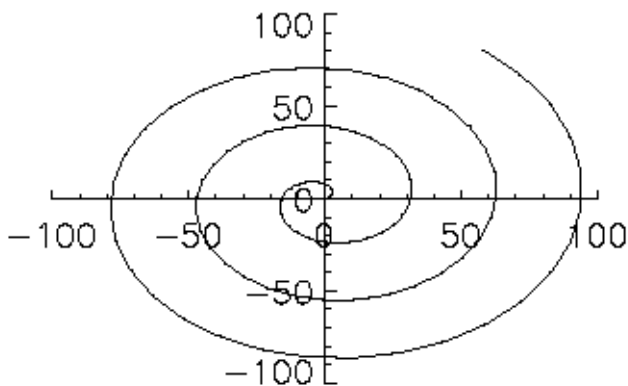
```

The code above is included in the batch file `plot09` in the `examples/doc` subdirectory of the IDL distribution.

Using AXIS with Polar Plots

If the `POLAR` keyword parameter is set, the IDL `PLOT` procedure converts its coordinates from polar to Cartesian coordinates when plotting. The first parameter to plot is the radius, R , and the second is the angle θ (expressed in radians). Polar plots are produced using the standard axis and label styles, with box axes enclosing the plot area.

The following figure illustrates using `AXIS` to draw centered axes, dividing the plot window into the four quadrants centered about the origin. This method uses `PLOT` to plot the polar data and to establish the coordinate scaling, but suppresses the axes. Next, two calls to `AXIS` add the x - and y -axes, drawn through data coordinate $(0, 0)$.



Polar Plot

Figure 17-20: Using `AXIS` for polar plots

```

; Make a radius vector:
R = FINDGEN(100)
; Make a vector:
THETA = R/5
; Plot the data, suppressing the axes by setting their styles to 4:

```

```
PLOT, R, THETA, SUBTITLE='Polar Plot', XSTY=4, YSTY=4, /POLAR
  AXIS, 0, 0, XAX=0
  ; Draw the x and y axes through (0, 0):
  AXIS, 0, 0, YAX=0
```

The code above is included in the batch file `plot09` in the `examples/doc` subdirectory of the IDL distribution.

Using the CURSOR Procedure

The CURSOR procedure reads the position of the interactive graphics cursor of the current graphics device. It enables the graphic cursor on the device, optionally waits for the user to move it and/or press a locator button to terminate the operation (or type a character if the device has no buttons), and then reports the cursor position.

Note, however, that CURSOR should not be used with draw widgets, created by the WIDGET_DRAW function. If you need to find the position of the mouse or status of mouse buttons in a draw widget, set the BUTTON_EVENTS and MOTION_EVENTS keywords to WIDGET_DRAW, then examine the events returned by your draw widget. See [WIDGET_DRAW](#) in the *IDL Reference Guide* for more information.

The CURSOR procedure is called as follows:

```
CURSOR, X, Y [, WAIT]
```

where x and y are the named variables that receive the cursor position. Normally, the position is reported in data coordinates, but the DATA, DEVICE, and NORMAL keywords can be used to explicitly specify the coordinate system.

See [CURSOR](#) in the *IDL Reference Guide* for details.

When CURSOR returns, the `button` field of the system variable !MOUSE is set to the button status. Each mouse button is assigned a bit in the `button` field. Bit 0 is the leftmost button (value = 1), bit 1 is the middle button (value = 2), and bit 3 is the rightmost button (value = 4) for the typical three-button mouse. See [!MOUSE](#) in the *IDL Reference Guide* for details.

Simple Interactive Examples

The following two programs demonstrate simple applications of the interactive graphics cursor and the CURSOR procedure. The code for both routines is located in the file `plot10.pro`, located in the `examples/doc` subdirectory of the IDL distribution. You can also create either routine at the IDL command line by starting with the `.RUN` command and entering each line individually.

The first routine is a simple drawing program. Straight lines are connected to positions marked with the left or middle mouse buttons until the right button is pressed.

```

PRO DRAW
  ; Start with a blank screen:
  ERASE
  ; Get the initial point in normalized coordinates:
  CURSOR, X, Y, /NORMAL, /DOWN
  ; Repeat until right button is pressed. Get the second point.
  ; Draw the line. Make the current second point be the new first.
  WHILE (!MOUSE.button NE 4) DO BEGIN
    CURSOR, X1, Y1, /NORM, /DOWN
    PLOTS,[X,X1], [Y,Y1], /NORMAL
    X = X1 & Y = Y1
  ENDWHILE
END

```

The second simple procedure can be used to label plots using the cursor to position the text:

```

;Text is the string to be written on the screen:
PRO LABEL, TEXT
  ; Ask the user to mark the position:
  PRINT, 'Use the mouse to mark the text position:'
  ; Get the cursor position after pressing any button:
  CURSOR, X, Y, /NORMAL, /DOWN
  ; Write the text at the specified position.
  ; The NOCLIP keyword is used to ensure that
  ; the text will appear even if it is outside
  ; the plotting region.
  XYOUTS, X, Y, TEXT, /NORMAL, /NOCLIP
END

```

To place annotation on a device with an interactive pointer, call this procedure with the command:

```

ANNOTATE, 'Text for label'

```

Next, move the pointer device (mouse, cursor, or joystick) to the desired spot, and press the locator button. Consider how you might augment the LABEL procedure to allow you to specify the size and font of the annotation text.



Chapter 18: Plotting Multi- Dimensional Arrays

The following topics are covered in this chapter:

Overview	294	Three-Dimensional Graphics	313
Contour Plots	295	Three-Dimensional Transformations	323
Overlaying Images and Contour Plots	301	Shaded Surfaces	329
Additional Contour Options	306	Volume Visualization	332
The SURFACE Procedure	310	References	336

Overview

This chapter describes the facilities for drawing representations of two-dimensional arrays. The two intrinsic procedures for the display of arrays are `CONTOUR` and `SURFACE`.

`CONTOUR` and `SURFACE` both use line graphics to depict the value of a two-dimensional array. As its name implies, `CONTOUR` draws contour plots.

`SURFACE` depicts the surface created by interpreting each element value as an elevation. These three-dimensional, wire-mesh surface plots can have almost any rotation about the x - and z -axes (the data z -axis must project parallel to the device's y -axis).

Three-dimensional graphics, coordinate systems, and transformations also are included in this chapter. Almost all of the information concerning coordinate systems, keyword parameters, and system variables discussed in [Chapter 17, “Direct Graphics Plotting”](#), apply to `CONTOUR` and `SURFACE` as well.

Surface and Contour Plots with iTools

Beginning with IDL 6.0, you can also use the *IDL Intelligent Tools* (iTools) to display two-dimensional array data. The iTools provide an easy-to-use interface that allows you to manipulate your data after it has been displayed. See [“Introducing the iTools”](#) in Chapter 1 of the *iTool User's Guide* manual for more on iTools.

Running the Example Code

The examples in this chapter are all written to take advantage of IDL Direct Graphics. Examples and techniques using IDL Object Graphics are contained in the later chapters of this manual.

Some of the example code used in this chapter is part of the IDL distribution. All of the files mentioned are located in the `examples/doc` subdirectory of the IDL distribution. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See [“!PATH”](#) in Appendix D of the *IDL Reference Guide* manual for information on IDL's path.

Contour Plots

The CONTOUR procedure draws contour plots from data stored in a rectangular array. In its simplest form, CONTOUR makes a contour plot given a two-dimensional array of z values. In more complicated forms, CONTOUR accepts, in addition to z , arrays containing the x and y locations of each column, row, or point, plus many keyword parameters. In more sophisticated applications, the output of CONTOUR can be projected from three dimensions to two dimensions, superimposed over an image, or combined with the output of SURFACE. The basic call to CONTOUR is as follows:

```
CONTOUR, Z
```

where Z is a two-dimensional array. This call labels the x - and y -axes with the subscript along each dimension. For example, when contouring a 10×20 array, the x -axis ranges from 0 to 9, and the y -axis ranges from 0 to 19.

You can explicitly specify the x and y locations of each cell as follows:

```
CONTOUR, Z, X, Y
```

where the X and Y arrays can be either vectors or two-dimensional arrays of the same size as Z . If they are vectors, the element $z_{i,j}$ has a coordinate location of (x_i, y_j) . Otherwise, if the x and y arrays are two-dimensional, the element $z_{i,j}$ has the location $(x_{i,j}, y_{i,j})$. Thus, vectors should be used if the x location of $z_{i,j}$ does not depend upon j and the y location of $z_{i,j}$ does not depend upon i .

Dimensions must be compatible. In the one-dimensional case, X must have a dimension equal to the number of columns in Z , and Y must have a dimension equal to the number of rows in Z . In the two-dimensional case, all three arrays must have the same dimensions.

IDL uses linear interpolation to determine the x and y locations of the contour lines that pass between grid elements. The cells must be regular in that the x and y arrays must be monotonic over rows and columns, respectively. The lines describing the quadrilateral enclosing each cell and whose vertices are $(x_{i,j}, y_{i,j})$, $(x_{i+1,j}, y_{i+1,j})$, $(x_{i+1,j+1}, y_{i+1,j+1})$, and $(x_{i,j+1}, y_{i,j+1})$ must intersect only at the four corners and the quadrilateral must not contain other nodes.

See [CONTOUR](#) in the *IDL Reference Guide* for a complete list of CONTOUR's parameters and keywords.

Contouring Methods

In order to provide a wide range of options, CONTOUR uses one of two contouring algorithms. The algorithm used depends on the keywords specified, and is one of the two following methods.

Cell Drawing

The first algorithm, used by default, examines each array cell and draws all contours emanating from that cell before proceeding to the next cell. This method is efficient in terms of computer resources, but does not allow options such as contour labeling or smoothing.

Contour Following

The second method searches for each contour line, then follows the line until it reaches a boundary or closes. This method gives better looking results with dashed linestyles and allows contour labeling and bi-cubic spline interpolation, but requires more computer time. The contour following method is used if any of these keywords are specified: `C_ANNOTATION`, `C_CHARSIZE`, `C_LABELS`, `CLOSED`, `FOLLOW`, `PATH_FILENAME`, or `DOWNHILL`.

Note

Due to their differing algorithms, these two methods will often draw slightly different, yet correct, contour maps for the same data. This difference is a direct result of the fact that there is often more than one valid way to draw contours and should not be a cause for concern.

Example: Maroon Bells Peaks

Digital elevation data of the Maroon Bells area, near Aspen, Colorado, are used to illustrate the CONTOUR procedure. The data set was obtained from a United States Geological Survey Digital Elevation Model tape. This data provides terrain elevation data over a 7.5-minute square (approximately 11×13.7 kilometers at the latitude of Maroon Bells), with 30-meter sampling.

The data are contained in a 360×460 array `A`, sampled in 30-meter square intervals, measured in Universal Transverse Mercator (UTM) coordinates. The rectangular array is not completely filled with data because the 7.5-minute square is not perfectly oriented to the UTM grid system. Missing data are represented as zeroes. Elevation measurements range from 2658 to 4241 meters or from 8720 to 13,914 feet.

The Maroon Bells data is used in a number of examples in this chapter, and is included in an IDL SAVE file called `marbells.dat` in the `examples/data` subdirectory of the IDL distribution. To restore the save file, issue the following commands at the IDL prompt (change the path separator characters as necessary for your platform):

```
CD, !DIR+' /examples/data'  
RESTORE, 'marbells.dat'
```

The batch file `cntour01`, located in the `examples/doc` subdirectory of the IDL distribution, restores this data and defines several variables used in the examples in this chapter.

This command creates an IDL variable named `elev` that contains the 360 x 460 integer array.

The figure below is the result of applying the `CONTOUR` procedure to the data, using the default settings:

```
CONTOUR, elev
```

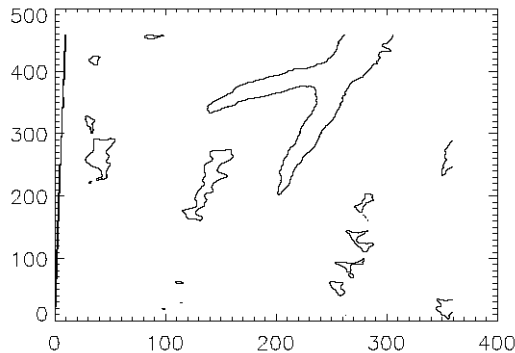


Figure 18-1: Simple Contour Plot of Maroon Bells

A number of problems are apparent with this simple contour plot.

- IDL selected six contour levels, by default, for the elevation from 0 to 4241; that's roughly 4241 divided into 7 intervals or approximately 605 meters in elevation between contour levels. The levels are 605, 1250, ..., 3635 meters, even though the range of valid data is from 2658 to 4241 meters. This is because the missing data values of 0 were considered when selecting the intervals. It is generally more appropriate to select contour levels only within the range of valid data.

- The vertical contours along the left edge are an invalid artifact due to contouring missing data and should not be present.
- For most display systems and for contour intervals of approximately 200 meters, the data has too many samples in the x - y direction. This oversampling has two adverse effects: the contours appear jagged, and a large number of short vectors are produced.
- The axes are labeled by point number, but should be in UTM coordinates.
- It is difficult to visualize the terrain and to discern maxima from minima because each contour is drawn with the same type of line.

The above problems are readily solved using the following simple techniques:

- Specify the contour levels directly using the LEVELS keyword parameter. Selecting contour intervals of 250 meters, at elevation levels of [2750, 3000, 3250, 3500, 3750, 4000], results in six levels within the range of valid data.
- Change the missing data value to a value well above the maximum valid data value, then use the MAX_VALUE keyword parameter to exclude missing points. In this example, we set missing data values to one million with the following statement:

```
elev(WHERE(elev EQ 0)) = 32767
```

Note

32767 is the maximum allowable 16-bit integer.

- Use the REBIN function to decrease the sampling in x and y by a factor of 5:

```
new = REBIN(elev, 360/5, 460/5)
```

This smooths the contours, because the call to REBIN averages $5^2=25$ bins when resampling. The number of vectors transmitted to the display also are decreased by a factor of approximately 25. The variable B is now a 72×92 array.

Care is taken in the second step to ensure that the missing data are not confused with valid data after REBIN is applied. As in this example, REBIN averages bins of $5^2=25$ elements, the missing data value must be set to a value of at least 25 times the maximum valid data value. After applying REBIN, any cell with a missing original data point will have a value of at least $106/25 = 40000$, well over the largest valid data value of approximately 4,500.

The x and y vectors are constructed containing the UTM coordinates for each row and column. From the USGS data tape, the UTM coordinate of the lower-left corner of the array is (326,850: 4,318,500) meters. As the data spacing is 30 meters in both

directions, the x and y vectors, in kilometers, are easily formed using the FINDGEN function, as shown in the example below.

Contour levels at each multiple of 500 meters (every other level) are drawn with a solid linestyle, while levels that fall between are drawn with a dotted line. In addition, the 4000-meter contour is drawn with a triple thick line, emphasizing the top contour.

The result of these improvements is shown in the figure below.

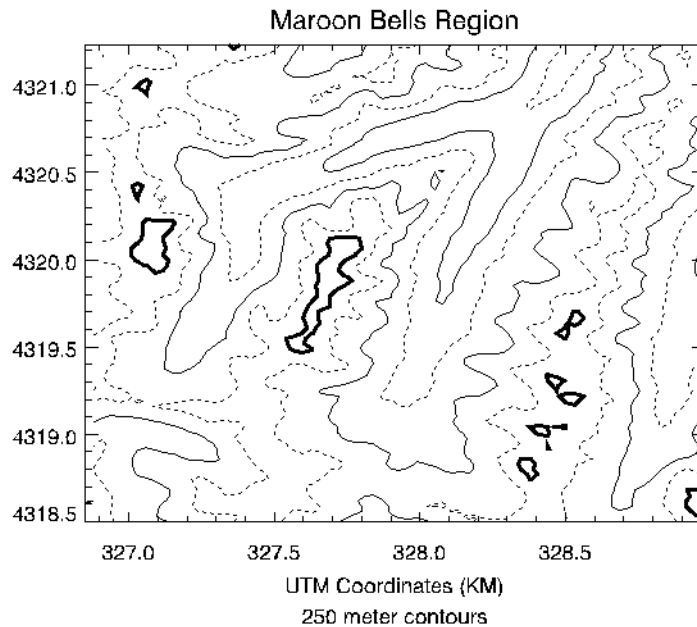


Figure 18-2: Improved Contour Plot

This figure was produced with the following IDL statements:

```

; Restore variables:
@cntour01
; Set missing data points to a large value:
elev (WHERE (elev EQ 0)) = 1E6
; REBIN down to a 72 x 92 matrix:
new = REBIN(elev, 360/5, 460/5)
; Make the x and y vectors specifying the position
; of each column and row.
X = 326.850 + .030 * FINDGEN(72)
Y = 4318.500 + .030 * FINDGEN(92)
; Make the plot, specifying the contour levels,

```

```
; missing data value, linestyle, etc.  
; Set the STYLE keywords to 1, obtaining exact axes.  
CONTOUR, new, X, Y, LEVELS = 2750 + FINDGEN(6) * 250., $  
  XSTYLE = 1, YSTYLE = 1, YMARGIN = 5, MAX_VALUE = 5000, $  
  C_LINestyle = [1, 0], $  
  C_THICK = [1, 1, 1, 1, 1, 3], $  
  TITLE = 'Maroon Bells Region', $  
  SUBTITLE = '250 meter contours', $  
  XTITLE = 'UTM Coordinates (KM)'
```

If you prefer not to enter the code by hand, run the batch file `cntour02` with the following command at the IDL prompt:

```
@cntour02
```

See [“Running the Example Code”](#) on page 294 if IDL does not find the batch file.

The figure below illustrates the data displayed as a grayscale image. Higher elevations are white. This image demonstrates that contour plots do not always provide the best qualitative visualization of many two-dimensional data sets.



Figure 18-3: Maroon Bells Data Displayed as an Image

Overlaying Images and Contour Plots

Superimposing an image and its contour plot combines the best of both worlds: the image allows easy visualization and the contour lines provide a semi-quantitative display. The programs presented in the rest of this section are for advanced computing only.

A combined contour and image display, such as that discussed in this section, can be created with the `IMAGE_CONT` procedure. The following material is intended to illustrate the ways in which images and graphics can be combined using IDL.

The technique used to overlay plots and images depends on whether or not the device is able to represent pixels of variable size, as does PostScript, or if it has pixels of a fixed size. If the device does not have scalable pixels, the image must be resized to fit within the plotting area if it is not already of a size suitable for viewing. This leads to three separate cases that are illustrated in the following examples.

Overlaying with Scalable Pixels

Certain devices, notably PostScript, can display pixels of varying sizes. With these devices, it is easy to set the size and position of an image so that it exactly overlays the plot window. In creating the next figure, the actual dimensions of the contour plot window (contained in the `!X.WINDOW` and `!Y.WINDOW` system variables) were used to calculate the new size of the image.

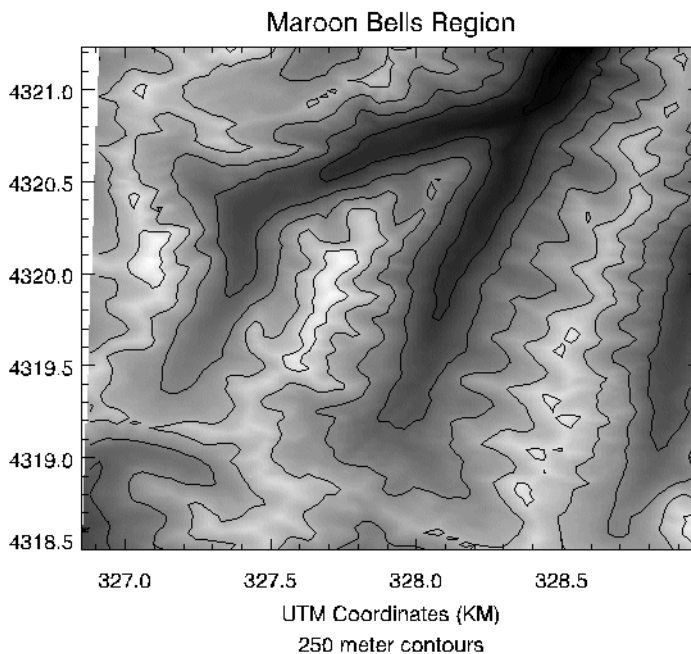


Figure 18-4: Overlay of Image and Contour Plots

Note

In order to do this successfully, you must establish the size of the plot window before scaling the image. This means that you must make a call to `CONTOUR` before displaying the image, to set the window size, and another call to `CONTOUR` after displaying the image, to draw the contour lines on top of the image data.

Inspect the batch file `cntour03` located in the `examples/doc` subdirectory of the IDL distribution to see how the figure was created. Note that the aspect ratio of the image was changed to fit that of the plot window. To retain the original image aspect ratio, the plot window must be resized to an identical aspect ratio using the `POSITION` keyword parameter.

Overlaying with Fixed Pixels

If the pixel size is fixed (as is true on most displays), we can either resize the image to fit the plotting window or size the plotting window to fit the image dimensions.

Method 1: Scale the Image to Fit the Display

We can use the CONGRID function to create an image of the same size as the plotting window. The REBIN function also can be used to resample the original image if the plot window dimensions are an integer multiple or factor of the original image dimensions. REBIN is always faster than CONGRID. The following IDL procedure creates an image of the same size as the window, displays it, then overlays the contour plot.

```

; Restore variables:
@cntour01
; Set missing data points to a large value:
elev (WHERE (elev EQ 0)) = 1E6
; REBIN down to a 72 x 92 matrix.
new = REBIN(elev, 360/5, 460/5)
; Scale image intensities:
image = BYTSCL(elev, MIN=2658, MAX=4241)
; Before displaying the image, use the CONTOUR command
; to create the appropriate plot window.
; The plot window must be created before re-sizing
; the image data.
; Use the NODATA keyword to inhibit actually drawing
; the contour plot.
CONTOUR, new, X, Y, LEVELS = 2750 + FINDGEN(6) * 250., $
    MAX_VALUE = 5000, XSTYLE = 1, YSTYLE = 1, $
    TITLE = 'Maroon Bells Region', $
    SUBTITLE = '250 meter contours', $
    XTITLE = 'UTM Coordinates (KM)', /NODATA
; Get size of plot window in device pixels.
PX = !X.WINDOW * !D.X_VSIZE
PY = !Y.WINDOW * !D.Y_VSIZE
; Desired size of image in pixels.
SX = PX[1] - PX[0] + 1
SY = PY[1] - PY[0] + 1
; Display the image with its lower-left corner at
; the origin of the plot window and with its size
; scaled to fit the plot window.
TVSCL, CONGRID(image, SX, SY), PX[0], PY[0])
CONTOUR, new, X, Y, LEVELS = 2750 + FINDGEN(6) * 250., $
    MAX_VALUE = 5000, XSTYLE = 1, YSTYLE = 1, $
    TITLE = 'Maroon Bells Region', $
    SUBTITLE = '250 meter contours', $

```

```

        XTITLE = 'UTM Coordinates (KM)', /NOERASE
; Write the contours over the image, being sure
; to use the exact axis styles so that the contours
; fill the plot window. Inhibit erasing.

```

If you prefer not to enter the code by hand, run the batch file `cntour04` with the following command at the IDL prompt:

```
@cntour04
```

See “[Running the Example Code](#)” on page 294 if IDL does not find the batch file.

Method 2: Scale the Display to Fit the Image

If the image is already close to the proper display size, it is simpler and more efficient to change the plot window size to that of the image. The following procedure displays the image at the window origin, then sets the plot window to the image size, leaving its origin unchanged.

```

; Restore variables:
@cntour01
; Set missing data points to a large value:
elev (WHERE (elev EQ 0)) = 1E6
; REBIN down to a 72 x 92 matrix.
new = REBIN(elev, 360/5, 460/5)
; Scale image intensities.
image = BYTSCL(elev, MIN=2658, MAX=4241)
; Get size of plot window in device pixels.
PX = !X.WINDOW * !D.X_VSIZE
PY = !Y.WINDOW * !D.Y_VSIZE
; Get the size of the image.
SZ = SIZE(image)
; Display the image with its lower-left corner
; at the origin of the plot window.
TVSCL, image, PX[0], PY[0]
; Write the contours over the image, being sure to use
; the exact axis styles so that the contours fill the plot
; window. Inhibit erasing.
CONTOUR, new, X, Y, XSTYLE = 1, YSTYLE = 1, $
    POSITION = [PX[0], PY[0], PX[0]+SZ[1]-1, PY[0]+SZ[2]-1], $
    LEVELS = 2750 + FINDGEN(6) * 250., MAX_VALUE = 5000, $
    TITLE = 'Maroon Bells Region', $
    SUBTITLE = '250 meter contours', $
    XTITLE = 'UTM Coordinates (KM)', /NOERASE, /DEVICE

```


If you prefer not to enter the code by hand, run the batch file `cntour05` with the following command at the IDL prompt:

```
@cntour05
```

See “[Running the Example Code](#)” on page 294 if IDL does not find the batch file.

Additional Contour Options

In addition to the abilities of CONTOUR demonstrated above, there are several options that depend upon the use of the contour following algorithm. These options are as follows:

Labeling Contours

The C_ANNOTATION, C_CHARSIZE, and C_LABELS keywords are used to control contour labeling. Using them, possibly in conjunction with the LEVELS keyword, it is possible to specify which contours should be labeled, the size of the labels, and the actual labels that should be used.

In the following discussion, a variable named DATA is contoured. This variable contains uniformly distributed random numbers obtained using the following statement:

```
SEED = 20 & DATA = RANDOMU(SEED, 6, 6)
```

To label contours using the defaults for label size and contours to label, it is sufficient to select the FOLLOW keyword. In this case, CONTOUR labels every other contour using the default label size (three-fourths of the plot axis label size). Each contour is labeled with its value.

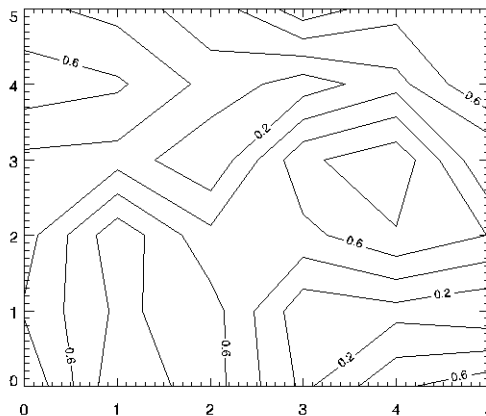


Figure 18-5: Simple Labeled Contour Plot

The preceding figure was produced using the following statement:

```
CONTOUR, /FOLLOW, DATA
```

The `C_CHARSIZE` keyword is used to specify the size of the characters used for labeling in the same manner that `SIZE` is used to control plot axis label size. The `C_LABELS` keyword can be used to select the contours to be labeled. For example, suppose that we want to contour the variable `DATA` at 0.2, 0.5, and 0.8, and we want all three levels labeled. In addition, we wish to make each label larger, and use hardware fonts. This can be accomplished with the statement below.

```
CONTOUR, LEVEL=[0.2, 0.5, 0.8], C_LABELS=[1, 1, 1], $
      C_CHARSIZE = 1.25, DATA, FONT = 0
```

The result is the plot on the left in the figure below.

Finally, it is possible to specify the text to be used for the contour labels using the `C_ANNOTATION` keyword, as shown in the statements below.

```
CONTOUR, LEVEL=[0.2, 0.5, 0.8], C_LABELS=[1, 1, 1], $
      C_ANNOTATION = ["Low", "Medium", "High"], DATA, FONT=0
```

The result is the plot on the right in the figure below.

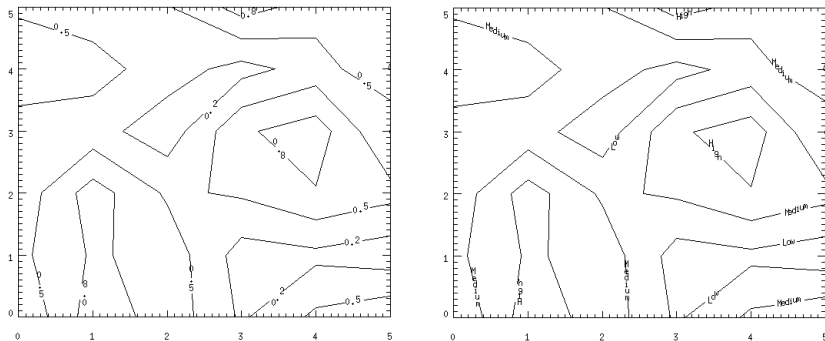


Figure 18-6: Label Size and Levels Specified (left), Explicitly Specified Labels (right)

Smoothing Contours

The `MIN_CURVE_SURF` function can be used to smoothly interpolate both regularly and irregularly sampled surfaces before contouring. This function replaces the older `SPLINE` keyword to `CONTOUR`, which was inaccurate and is no longer

supported. `MIN_CURVE_SURF` interpolates the entire surface to a relatively fine grid before drawing the contours.

See [CONTOUR](#) in the *IDL Reference Guide* for an example using the `MIN_CURVE_SURF` function. See also [MIN_CURVE_SURF](#) in the *IDL Reference Guide* for further details.

The following short example shows the difference between a smoothed and an unsmoothed contour plot:

```
;Create a simple dataset:
data = RANDOMU(seed, 7, 7)
;Plot the unsmoothed data:
CONTOUR, data
;Plot the smoothed data:
CONTOUR, MIN_CURVE_SURF(data)
```

Filling Contours

Set the `FILL` keyword to produce a filled contour plot. The contours are filled with solid or line-filled polygons. For solid polygons, use the `C_COLOR` keyword to specify the color index of the polygons for each contour level. For line fills, use `C_ORIENTATION`, `C_SPACING`, `C_COLOR`, `C_LINESTYLE`, and/or `C_THICK` to specify attributes for the lines.

If the current device is not a pen plotter, each polygon is erased to the background color before the fill lines are drawn, to avoid superimposing one pattern over another.

The `FILL` keyword replaces the use of the `PATH_FILENAME` keyword and `POLYFILL` procedure from previous versions of IDL. Setting the `FILL` keyword also closes any open contours before filling.

The following example illustrates various filled contour plot options.

```
; Create a simple, random dataset for contouring:
data = RANDOMU(seed, 7, 7)
; Create a basic, solid-color, filled CONTOUR plot
; with 6 evenly-spaced levels.
CONTOUR, data, NLEVELS=6, /FILL
; Overplot contour outlines:
CONTOUR, data, NLEVELS=6, /NOERASE
```

Instead of solid colors, contours can be filled with lines:

```
; Create a vector of orientations for the fill lines:
ANGLES = [0, 45, -45]
; Create a vector of colors to use:
C = [70, 120, 200, 255]
; Create contours filled with lines.
CONTOUR, data, NLEVELS=10, C_ORIENT=ANGLES, C_COLORS=C
; Overplot contour outlines:
CONTOUR, data, NLEVELS=10, /NOERASE
```

There are many other controls for filled contour plots. The `C_COLORS`, `C_LINestyle`, `C_SPACING`, and `C_THICK` keywords can also be used to control the type of fill. For more information see [CONTOUR](#) in the *IDL Reference Guide*.

Indicating Direction of Grade

Setting the `DOWNHILL` keyword creates short, perpendicular tick marks along each contour that point in the downhill (decreasing elevation) direction. These marks make the direction of the grade readily apparent. For example:

```
CONTOUR, data, /DOWNHILL
```

The SURFACE Procedure

The SURFACE procedure draws wire mesh representations of functions of x and y , just as CONTOUR draws their contours. Parameters to SURFACE are similar to CONTOUR. SURFACE accepts a two-dimensional array of z (elevation) values, and optionally x and y parameters indicating the location of each z element.

Note

The grid defined by the x and y parameters must be regular, or nearly regular, or errors in hidden line removal will result. Also, the rotation must project the data z -axis so that it is parallel to the drawing surface's y -axis or errors in hidden line removal will result.

SURFACE projects the three-dimensional array of points into two dimensions after rotating about the z - and then the x -axes. Each point is connected to its neighbors by lines. Hidden lines are suppressed. The rotation about the x - and z -axes can be specified with keywords or a complete three-dimensional transformation matrix can be stored in the field !P.T for use by SURFACE. Details concerning the mechanics of three-dimensional projection and rotation are covered in the next section.

The following IDL code illustrates the most basic call to SURFACE. It produces a two-dimensional Gaussian function, then calls SURFACE to produce the figure below.

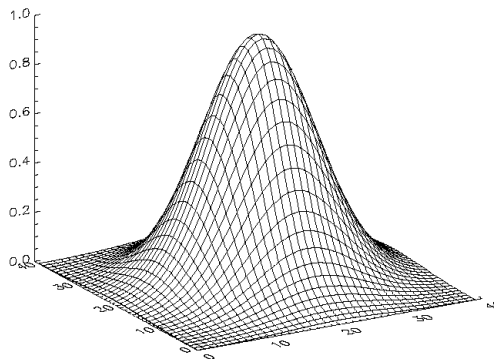


Figure 18-7: Simple SURFACE Plot of a Gaussian

```

; Create a 40 by 40 array in which each element is
; equal to the Euclidean distance from the center:
Z = SHIFT(DIST(40), 20, 20)
; Make Gaussian with a 1/e width of 10:
Z = EXP(-(Z/10)^2)
; Call SURFACE to display plot:
SURFACE, Z

```

In the example above, the DIST function creates an (n, n) array in which each element is set to its Euclidean distance from the origin.

SURFACE Keyword Parameters

In addition to the standard graphics keyword parameters, SURFACE accepts a number of unique keyword parameters. See [SURFACE](#) in the *IDL Reference Guide* for details.

Example

The figures below illustrate the application of the SURFACE procedure to the Maroon Bells data used in the first section of this chapter. As with CONTOUR, it is often useful to reduce the number of individual data values, so that the surface is not obscured by excessive detail.

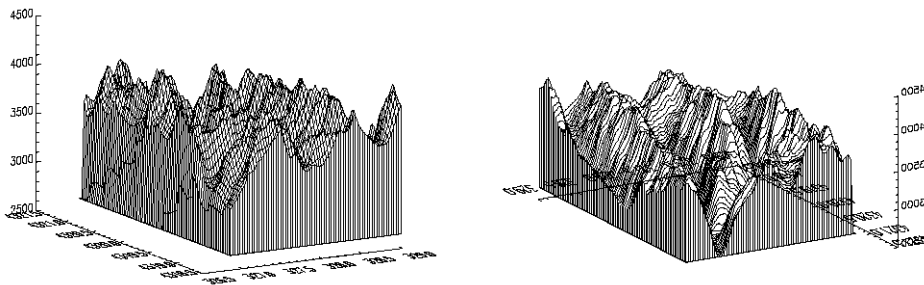


Figure 18-8: Maroon Bells Surface Plots

The left illustration in the figure above was produced by the following statements:

```

; Restore variables.
@cntour01
; Resize the original data into a 72 x 92 array, setting
; all data values which are less than 2650 (the lowest
; elevation we wish to show) to 2650.
surf = REBIN(elev > 2650, 360/5, 460/5)
; Display the surface, drawing a skirt down to 2650 meters:
SURFACE, surf, X, Y, SKIRT = 2650

```

Alternatively, run the batch file `surf01` with the following command at the IDL prompt:

```
@surf01
```

See [“Running the Example Code”](#) on page 294 if IDL does not find the batch file.

The right illustration in the figure shows the Maroon Peaks area looking from the back row to the front row (north to the south) of the Maroon Peaks area. This perspective on the data is created by setting the angle of rotation around the z -axis to 210 degrees (setting `AZ = 210`), and increasing the azimuth from the default 30 degrees to 45 (setting `AX = 45`). Also, only the horizontal lines are drawn because the `/HORIZONTAL` keyword is present in the following call:

```
SURFACE, surf, X, Y, SKIRT = 2650, /HORIZ, AZ = 210, AX = 45
```

Because the axes are rotated 210 degrees about the original z -axis, the annotation is reversed and the x -axis is behind and obscured by the surface. This undesirable effect can be eliminated by reversing the minimum and maximum values of the X and Y ranges used when drawing the surface:

```

; As above, but reverse the data rather than the axes:
SURFACE, surf, X, Y, SKIRT = 2650, /HORIZONTAL, AX = 45, $
  YRANGE = [MAX(Y), MIN(Y)], XRANGE=[MAX(X), MIN(X)]

```


Three-Dimensional Graphics

Points in xyz space are expressed by vectors of homogeneous coordinates. These vectors are translated, rotated, scaled, and projected onto the two-dimensional drawing surface by multiplying them by transformation matrices. The geometrical transformations used by IDL, and many other graphics packages, are taken from Chapters 7 and 8 of Foley and Van Dam (1982). The reader is urged to consult this book for a detailed description of homogeneous coordinates and transformation matrices since this section presents only an overview.

Homogeneous Coordinates

A point in homogeneous coordinates is represented as a four-element column vector of three coordinates and a scale factor $w \neq 0$. For example:

$$P(wx, wy, wz, w) \equiv P(x/w, y/w, z/w, 1) \equiv (x, y, z)$$

One advantage of this approach is that translation, which normally must be expressed as an addition, can be represented as a matrix multiplication. Another advantage is that homogeneous coordinate representations simplify perspective transformations. The notion of rows and columns used by IDL is opposite that of Foley and Van Dam (1982). In IDL, the column subscript is first, while in Foley and Van Dam (1982) the row subscript is first. This changes all row vectors to column vectors and transposes matrices.

Right-Handed Coordinate System

The coordinate system is right-handed so that when looking from a positive axis to the origin, a positive rotation is counterclockwise. As usual, the x -axis runs across the display, the y -axis is vertical, and the positive z -axis extends out from the display to the viewer. For example, a 90-degree positive rotation about the z -axis transforms the x -axis to the y -axis.

Transformation Matrices

Note

For most applications, it is not necessary to create, manipulate, or to even understand transformation matrices. The procedure T3D, explained below, implements most of the common transformations.

Transformation matrices, which post-multiply a point vector to produce a new point vector, must be (4, 4). A series of transformation matrices can be concatenated into a single matrix by multiplication. If A_1 , A_2 , and A_3 are transformation matrices to be applied in order, and the matrix A is the product of the three matrices, the following applies.

$$((P \bullet A_1) \bullet A_2) \bullet A_3 \equiv P \bullet ((A_1 \bullet A_2) \bullet A_3) = P \bullet A$$

IDL stores the concatenated transformation matrix in the system variable field !P.T.

Each of the operations of translation, scaling, rotation, and shearing can be represented by a transformation matrix.

Translation

The transformation matrix to translate a point by (D_x, D_y, D_z) is shown below.

$$\begin{bmatrix} 1 & 0 & 0 & D_x \\ 0 & 1 & 0 & D_y \\ 0 & 0 & 1 & D_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scaling by factors of S_x , S_y , and S_z about the x -, y -, and z -axes respectively, is represented by the matrix below.

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

Rotation about the x -, y -, and z -axes is represented respectively by the following three matrices:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x & 0 \\ 0 & \sin \theta_x & \cos \theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos \theta_z & -\sin \theta_z & 0 & 0 \\ \sin \theta_z & \cos \theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

T3D Procedure

The IDL procedure T3D creates and accumulates transformation matrices, storing them in the system variable field !P.T. The procedure can be used to create a transformation matrix composed of any combination of translation, scaling, rotation, perspective projection, oblique projection, and axis exchange. Transformations are applied in the order of the keyword descriptions below.

RESET

Set this keyword to reset the transformation matrix to the identity matrix to begin a new accumulation of transformations. If this keyword is not present, the current transformation matrix !P.T is post-multiplied by the new transformation. The final transformation matrix is always stored back in !P.T.

TRANSLATE

This keyword argument accepts a 3-element vector. The viewpoint is translated by the three-element vector $[T_x, T_y, T_z]$.

SCALE

This keyword argument accepts a 3-element vector. The viewing area is scaled by factor $[S_x, S_y, S_z]$.

ROTATE

This keyword accepts a 3-element vector. The viewing area is rotated about each axis by the amount $[\theta_x, \theta_y, \theta_z]$, in degrees.

PERSPECTIVE

A scalar (p) indicating the z distance of the center of the projection in the negative direction. Objects are projected into the xy plane, at $z = 0$, and the eye is at point $(0, 0, -p)$.

OBLIQUE

A two-element vector, $[d, \alpha]$, specifying the parameters for an oblique projection. Points are projected onto the xy -plane at $z = 0$ as follows:

$$x_0 = x + z(d \cos \alpha)$$

$$y_0 = y + z(d \sin \alpha)$$

An oblique projection is a parallel projection in which the normal to the projection plane is the z -axis, and the unit vector $(0, 0, 1)$ is projected to $(d \cos \alpha, d \sin \alpha)$ where α is expressed in degrees.

XYEXCH

If set, exchanges the x - and y -axes.

XZEXCH

If set, exchanges the x - and z -axes.

YZEXCH

If set, exchanges the y - and z -axes.

Example: The Transformation Created by SURFACE

The SURFACE procedure creates a transformation matrix from its keyword parameters AX and AZ as follows:

1. Starting with the identity transformation, SURFACE translates the center of the normalized cube to the origin.
2. SURFACE rotates 90 degrees about the x -axis to make the $+z$ -axis of the data the $+y$ axis of the display. The $+y$ data axis extends from the front of the display to the rear.
3. SURFACE rotates AZ degrees about the y -axis. This rotates the result counter-clockwise, as seen from above the page.
4. SURFACE rotates AX degrees about the x -axis, tilting the data towards the viewer.
5. The procedure then removes the translation applied in the first step and scales the data so that the data are still contained within the normal coordinate unit cube after transformation.

These transformations can be created using T3D as shown below. The [SCALE3](#) procedure, documented in the IDL Reference Guide, mimics the transformation matrix created by [SURFACE](#) using the following method:

```

; Translate to move center of cube to origin.
T3D, /RESET, TRANSLATE = [-.5, -.5, -.5]
; Rotate 90 degrees about x-axis, so +z axis is now +y.
; Then rotate AZ degrees about y-axis.
T3D, ROTATE = [-90, AZ, 0]
; Rotate AX about x axis:
T3D, ROTATE = [AX, 0, 0]
; Restore origin.
T3D, TRANSLATE = [0.5, 0.5, 0.5]

```

The SCALE3 procedure, scales the unit cube by a fixed factor, $1/\text{SQRT}(3)$ to ensure that the corners of the rotated cube fit within the drawing area. If requested, it also will set the data scaling. Animations involving rotations or the SURFACE procedure

should have their scaling and viewing transformation set by SCALE3 rather than the obsolete SURFR procedure, so that the scaling does not vary between frames.

Three-Dimensional Coordinate Conversion

To convert from a three-dimensional coordinate to a two-dimensional coordinate, IDL follows these steps:

- Data coordinates are converted to three-dimensional normalized coordinates. To convert the x coordinate from data to normalized coordinates, use the formula $N_x = X_0 + X_1 D_x$. The same process is used to convert the y and z coordinates using !Y.S and !Z.S.
- The three-dimensional normalized coordinate, $P = (N_x, N_y, N_z)$, whose homogeneous representation is $(N_x, N_y, N_z, 1)$, is multiplied by the concatenated transformation matrix !P.T:

$$P' = P \bullet !P.T$$

- The vector P' is scaled by dividing by w , and the normalized two-dimensional coordinates are extracted:

$$N'_x = P'_x / P'_w \text{ and } N'_y = P'_y / P'_w$$

- The normalized xy coordinate is converted to device coordinates as described in “[Two-Dimensional Coordinate Conversion](#)” in Chapter 17.

The CONVERT_COORD function performs the above process when converting to and from coordinate systems when the T3D keyword is specified. For example, if a three-dimensional coordinate system is established, then the device coordinates of the data point (0, 1, 2) can be computed as follows:

```
D = CONVERT_COORD(0, 1, 2, /TO_DEVICE, /T3D, /DATA)
```

On completion, the three-element vector D will contain the desired device coordinates. The process of converting from three-dimensional to two-dimensional coordinates also can be written as an IDL function. This function accepts a three-dimensional data coordinate, returns a two-element vector containing the coordinate transformed to two-dimensional normalized coordinates using the current transformation matrix:

```
FUNCTION CVT_TO_2D, X, Y, Z
  ; Make a homogeneous vector of normalized 3D coordinates:
  P = [!X.S[0] + !X.S[1] * X, !Y.S[0] + !Y.S[1] * Y, $
      !Z.S[0] + !Z.S[1] * Z, 1]
  ; Transform by !P.T:
  P = P # !P.T
  ; Return the scaled result as a two-element,
```

```

; two-dimensional, xy vector:
RETURN, [P[0] / P[3], P[1] / P[3]]
END

```

Establishing a Three-Dimensional Coordinate System

Usually, scaling parameters for coordinate conversion are set up by the higher-level procedures. To set up your own three-dimensional coordinate system with a given transformation matrix and x , y , z data range, follow these steps:

- Establish the scaling from your data coordinates to normalized coordinates—the (0, 1) cube. Assuming your data are contained in the range (X_{min} , Y_{min} , Z_{min}) to (X_{max} , Y_{max} , Z_{max}), set the data scaling system variables as follows:

```

!X.S = [ -Xmin, 1 ] / (Xmax - Xmin)
!Y.S = [ -Ymin, 1 ] / (Ymax - Ymin)
!Z.S = [ -Zmin, 1 ] / (Zmax - Zmin)

```

- Establish the transformation matrix that determines the view of the unit cube. This can be done by either calling T3D, as explained above or by directly manipulating !P.T yourself. If you wish to simply mimic the rotations provided by the SURFACE procedure, call the SCALE3 procedure (which can also be used to perform the previous step).

Example

This example draws four views of a simple house. The procedure HOUSE defines the coordinates of the front and back faces of the house. The data-to-normal coordinate scaling is set, as shown above, to a volume about 25 percent larger than that enclosing the house. The PLOTS procedure is called to draw lines describing and connecting the front and back faces. XYOUTS is called to label the front and back faces.

The commands shown after the definition of the HOUSE procedure contain four sequences of calls to T3D to establish the coordinate transformation, each followed by a call to HOUSE. If you prefer not to enter the IDL code by hand, run the batch file showhaus with the following command at the IDL prompt:

```
@showhaus
```

See “Running the Example Code” on page 294 if IDL does not find the batch file.

```

PRO HOUSE
  ; X coordinates of 10 vertices. First 5 are front face,
  ; second 5 are back face. The range is 0 to 16.
  house_x = [0, 16, 16, 8, 0, 0, 16, 16, 8, 0]
  ; The corresponding y values range from 0 to 16.
  house_y = [0, 0, 10, 16, 10, 0, 0, 10, 16, 10]
  ;The z values range from 30 to 54.
  house_z = [54, 54, 54, 54, 54, 30, 30, 30, 30, 30]
  ; Define max and min xy values to scale.
  ; Slightly larger than data range.
  min_x = -4 & max_x = 20.
  ; Set x data scale to range from -4 to 20.
  !X.S = [-(-4), 1.]/(20 - (-4))
  ; Same for y.
  !Y.S = !X.S
  ; The z range is from 10 to 70.
  !Z.S = [-10, 1.]/(70 - 10)
  ; Indices of front face.
  face = [INDGEN(5), 0]
  ; Draw front face.
  PLOTS, house_x[face], house_y[face], $
    house_z[face], /T3D, /DATA
  ; Draw back face.
  PLOTS, house_x[face + 5], house_y[face + 5], $
    house_z[face + 5], /T3D, /DATA
  ; Connecting lines from front to back.
  FOR I = 0, 4 DO PLOTS, [house_x[i], house_x[i + 5]], $
    [house_y[i], house_y[i + 5]], $
    [house_z[i], house_z[i + 5]], /T3D, /DATA
  ; Annotate front peak.
  XYOUTS, house_x[3], house_y[3], Z = house_z[3], 'Front', $
    /T3D, /DATA, SIZE = 2
  ; Annotate back.
  XYOUTS, house_x[8], house_y[8], Z = house_z[8], 'Back', $
    /T3D, /DATA, SIZE = 2
END

```

The HOUSE procedure could be called from the IDL command line to produce a number of different plots. For example:

```

  ; Set up no rotation, scale, and draw house.
  T3D, /RESET & HOUSE
  ; Create a handy constant.
  H = [0.5, 0.5, 0.5]
  ; Straight projection after rotating 30 degrees about x and y axes.
  T3D, /RESET, TRANS = -H, ROT = [30, 30, 0] & $
    T3D, TR = H & HOUSE
  ; No rotation, oblique projection, z factor = 0.5, angle = 45.

```



```

T3D, /RESET, TRANS = -H, ROT=[0, 0, 0], OBLIQUE=[.5, -45] & $
    T3D, TR = H & HOUSE
; Rotate 6 degrees about x and y, then apply perspective.
T3D, /RESET, TR=-H, ROT=[-6, 6, 0], PERS=4 & $
    T3D, TR=H & HOUSE

```

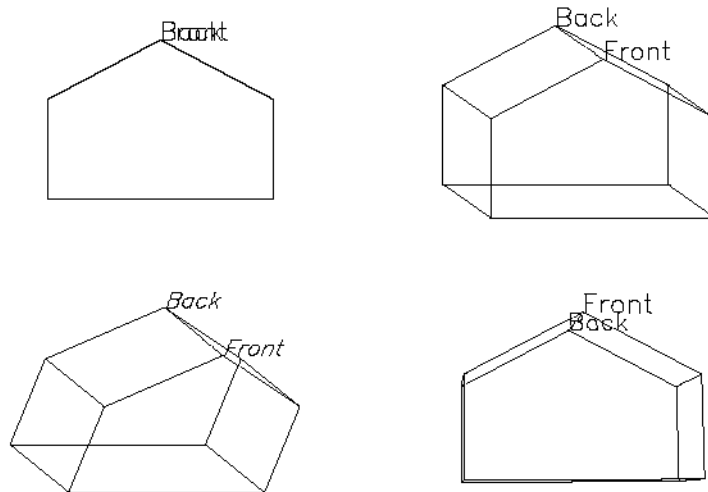


Figure 18-9: Illustration of Different Three-Dimensional Transformations

The figure illustrates the different transformations. The four rotations are:

- Upper left: no rotation, plain projection
- Upper right: oblique projection, factor = 0.5, angle = -45
- Bottom left: rotation of 30 degrees about both the x - and y -axes, plain projection
- Bottom right: rotation of -6 degrees about the x -axis and $+6$ degrees about the y -axis, and perspective projection with the eye at 4.

Rotating the House

A common procedure for visualizing three-dimensional data is to animate the data by rotating it about one or more axes. To make an animation of the house in the preceding example with the XINTERANIMATE procedure, use the following example.

```
; Initialize animation: set frame size and number of frames.
sizx = 300
sizy = 300
nframes = 16
XINTERANIMATE, SET=[sizx, sizy, nframes]
; Rotate about the z axis. Draw the house. Save the window.
FOR i = 0, nframes - 1 DO BEGIN $
    SCALE3, AX = 75, AZ = i * 360. / nframes & $
    ERASE & $
    HOUSE & $
    SCALE3, AX = 75, AZ = i * 360. / nframes & $
    XINTERANIMATE, FRAME=i, WINDOW=!D.WINDOW & $
ENDFOR
; Show the animation.
XINTERANIMATE
```

In the above example, `SCALE3` rather than `SCALE3D` is used to maintain the same scaling in all rotations. If you prefer not to enter the IDL code by hand, run the batch file `animhaus` with the following command at the IDL prompt:

```
@animhaus
```

See [“Running the Example Code”](#) on page 294 if IDL does not find the batch file.

Three-Dimensional Transformations

The `CONTOUR` and `PLOT` procedures output their results using the three-dimensional coordinate transformation contained in `!P.T` when the keyword `T3D` is specified. Note that `!P.T` must contain a valid transformation matrix prior to using the `T3D` keyword.

`PLOT` and its variants output graphs in the xy -plane at the normal coordinate z value given by the keyword `ZVALUE`. If this keyword is not specified, the plot is drawn at the bottom of the unit cube at $z = 0$.

`CONTOUR` draws its axes at $z = 0$ and its contours at their z data value if `ZVALUE` is not specified. If `ZVALUE` is present, `CONTOUR` draws both the axes and contours in the xy -plane at the given z value.

Combining `CONTOUR` and `SURFACE`

It is easy to combine the results of `SURFACE` with the other IDL graphics procedures. The keyword parameter `SAVE` causes `SURFACE` to save the graphic transformation it used in `!P.T`. Then, when either `CONTOUR` or `PLOT` is called with the keyword parameter `T3D`, its output is transformed with the same projection. For example, the figure below illustrates how `SURFACE` and `CONTOUR` can be combined. In essence, this is a combination of figures from 2 previous sections.

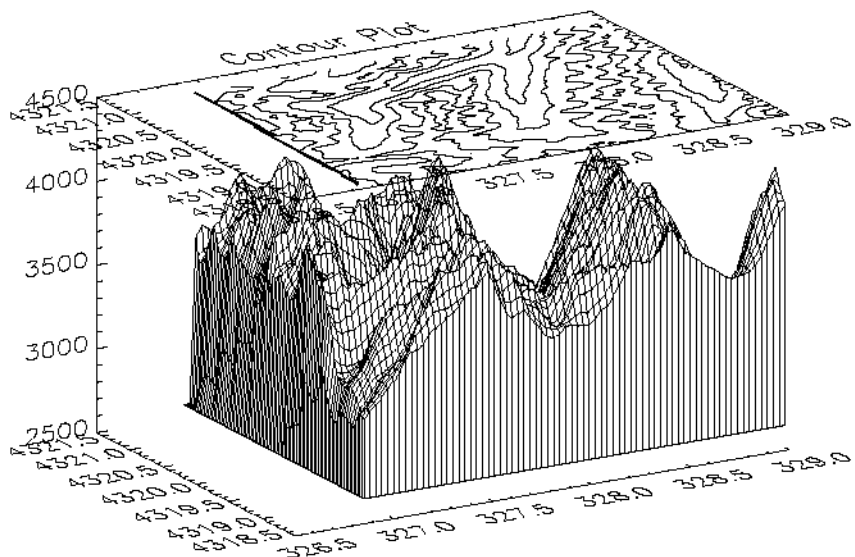


Figure 18-10: Combining `CONTOUR` with `SURFACE`, Maroon Bells Data

Using the same variables as in the earlier sections of this chapter, the figure was produced with the following statements:

```

; Restore variables.
@cntour01
; Resize the original data into a 72 x 92 array,
; setting all data values which are less than
; 2650 (the lowest elevation we wish to show) to 2650.
surf = REBIN(elev > 2650, 360/5, 460/5)
; Make the mesh.
SURFACE, surf, X, Y, SKIRT=2650, /SAVE
; Specify T3D to align with SURFACE, at ZVALUE of 1.0.
; Suppress clipping as the plot is outside the normal plot window.
CONTOUR, surf, X, Y, /T3D, /NOERASE, TITLE = 'Contour Plot', $
    MAX_VAL = 5000., ZVALUE = 1.0, /NOCLIP, $
    LEVELS = 2750. + FINDGEN(6) * 250

```

More Complicated Transformations

The figure below illustrates the application of three-dimensional transforms to the output of CONTOUR and PLOT. Using the two-dimensional Gaussian array **z** defined in “[The SURFACE Procedure](#)” on page 310, it draws a three-dimensional contour plot with the contours stacked above the axes in the *z* direction. It then plots the sum of the columns, also a Gaussian, in the *xz*-plane, and the sum of the rows in the *yz* plane.

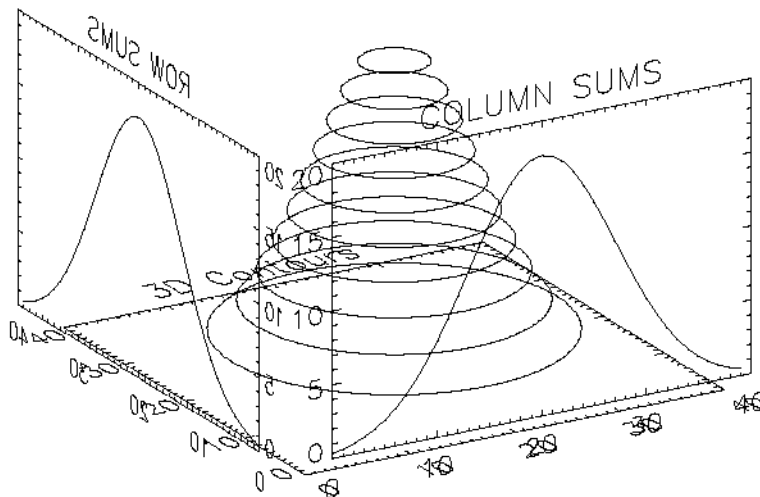


Figure 18-11: PLOT and CONTOUR with a Three-dimensional Transform

It was constructed as follows:

- First, the SCALE3 procedure is called to establish the default three- to two-dimensional transformation used by SURFACE, as explained above. The default rotations are 30 degrees about both the x - and z -axes.
- Next, a vector, POS, defining the cube containing the plot window is defined in normalized coordinates. The cube extends from 0.1 to 1.0 in the x and y directions and from 0 to 1 in the z direction. Each call to CONTOUR and PLOT must explicitly specify this window to align the plots. This is necessary because the default margins around the plot window are different in each direction.
- CONTOUR is called to draw the stacked contours with the axes at $z = 0$. Clipping is disabled to allow drawing outside the default plot window, which is only two-dimensional.
- The procedure T3D is called to exchange the y - and z -axes. The original xyz coordinate system is now xzy .
- PLOT is called to draw the column sums which appear in front of the contour plot. The expression $Z\#REPLICATE(1., N_y)$ creates a row vector containing the sum of each row in the two-dimensional array z . The NOERASE and NOCLIP keywords are specified to prevent erasure and clipping. This plot appears in the xz -plane because of the previous axis exchange.
- T3D is called again to exchange the x - and z -axes. This makes the original xyz coordinate system, which was converted to xzy , now correspond to yzx .
- PLOT is called to produce the column sums in the yz -plane in the same manner as the first plot. The original x -axis is drawn in the y -plane, and the y -axis is in the z -plane. One unavoidable side effect of this method is that the annotation of this plot is backwards. If the plot is transformed so the letters read correctly, the x -axis of the plot would be reversed in relation to the y -axis of the contour plot.

The IDL code used to draw the figure is as follows:

```

; Create the Z variable:
Z = SHIFT(DIST(40), 20, 20)
Z = EXP(-(Z/10)^2)
; NX and NY are the X and Y dimensions of the Z array:
NX = (SIZE(Z))(1)
NY = (SIZE(Z))(2)
; Set up !P.T with default SURFACE transformation.
SCALE3
; Define the three-dimensional plot
; window: x = 0.1 to 1, Y=0.1 to 1, and z = 0 to 1.
POS=[.1, .1, 1, 1, 0, 1]

```

```
; Make the stacked contours. Use 10 contour levels.
CONTOUR, Z, /T3D, NLEVELS=10, /NOCLIP, POSIT=POS, CHARSIZE=2
; Swap y and z axes. The original xyz system is now xzy:
T3D, /YZEXCH
; Plot the column sums in front of the contour plot:
PLOT, Z#REPLICATE(1., NY), /NOERASE, /NOCLIP, /T3D, $
    TITLE='COLUMN SUMS', POSITION = POS, CHARSIZE = 2
; Swap x and z—original xyz is now yzx:
T3D, /XZEXCH
; Plot the row sums along the right side of the contour plot:
PLOT, REPLICATE(1., NX)#Z, /NOERASE, /T3D, /NOCLIP, $
    TITLE = 'ROW SUMS', POSITION = POS, CHARSIZE = 2
```

If you prefer not to enter the IDL code by hand, run the batch file `cntour06` with the following command at the IDL prompt:

```
@cntour06
```

See [“Running the Example Code”](#) on page 294 if IDL does not find the batch file.

Combining Images with Three-Dimensional Graphics

Images are combined with three-dimensional graphics, as shown in the figure below, using the transformation techniques described above.

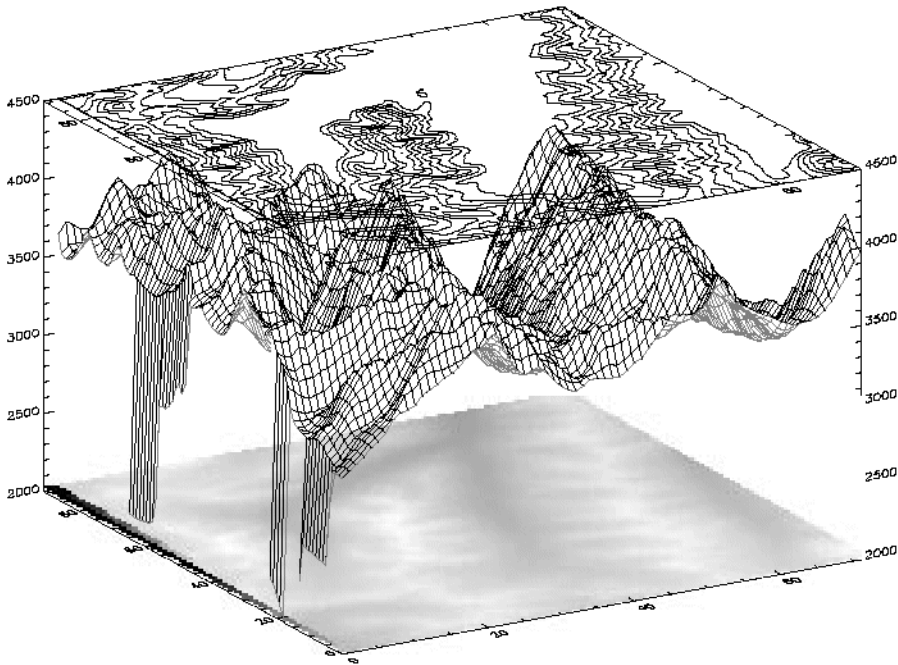


Figure 18-12: Using *SHOW3* to Overlay an Image, Surface Mesh, and Contour

The rectangular image must be transformed so that it fits underneath the mesh drawn by *SURFACE*. The general approach is as follows:

- Use *SURFACE* to establish the general scaling and geometrical transformation. Draw no data, as the graphics made by *SURFACE* will be over-written by the transformed image.
- For each of the four corners of the image, translate the data coordinate, which is simply the subscript of the corner, into a device coordinate. The data coordinates of the four corners of an (m, n) image are $(0, 0)$, $(m-1, 0)$, $(0, n-1)$, and $(m-1, n-1)$. Call this data coordinate system (x, y) . Using a procedure or function similar to *CVT_TO_2D* (see “[Three-Dimensional Coordinate Conversion](#)” on page 318) convert to device coordinates, which in this discussion are called (U, V) .

- The image is transformed from the original xy coordinates to a new image in UV coordinates using the POLY_2D function. POLY_2D accepts an input image and the coefficients of a polynomial in UV giving the xy coordinates in the original image. The equations for x and y are below.

$$X = S_{0,0} + S_{1,0}U + S_{1,0}V + S_{1,1}UV$$

$$Y = T_{0,0} + T_{1,0}U + T_{1,0}V + T_{1,1}UV$$

We solve for the four unknown S coefficients using the four equations relating the x corner coordinates to their U coordinates. The T coefficients are similarly found using the y and V coordinates. This can be done using matrix operators and inversion or more simply, with the procedure POLY_WARP.

- The new image is a rectangle that encloses the quadrilateral described by the UV coordinates. Its size is specified in the formula below:

$$(\text{MAX}(U) - \text{MIN}(U) + 2, \text{MAX}(V) - \text{MIN}(V) + 1)$$

- POLY_2D is called to form the new image which is displayed at device coordinate ($\text{MIN}(U)$, $\text{MIN}(V)$).
- SURFACE is called once again to display the mesh surface over the image.
- Finally, CONTOUR is called with ZVALUE set to 1.0, placing the contour above both the image and the surface.

The SHOW3 procedure performs these operations. It should be examined for details of how images and graphics can be combined.

The following IDL commands were used to create the previous image:

```

; Restore variables:
@cntour01
; Reduce the size of elev array:
new = REBIN(elev, 360/5, 460/5)
; Create an array of levels for CONTOUR:
levs = (FINDGEN(10)*100)+3500
; Use SHOW3. Note the use of keywords E_SURFACE
; and E_CONTOUR to pass values to the SURFACE and
; CONTOUR routines used within SHOW3.
SHOW3, new, E_SURFACE={min:2000}, E_CONTOUR={levels:levs}

```


Shaded Surfaces

The `SHADE_SURF` procedure creates a shaded representation of a surface made from regularly gridded elevation data. The shading information can be supplied as a parameter or computed using a light-source model. Displays are easily constructed depicting the surface elevation of a variable shaded as a function of itself or another variable. This procedure is similar to the `SURFACE` routine, but it renders the visible surface as a shaded image rather than a mesh.

Parameters are identical to those of the `SURFACE` procedure. See [SHADE_SURF](#) in the *IDL Reference Guide* for details.

Shading Method

The shading applied to each polygon, defined by its four surrounding elevations, can be either constant over the entire cell or interpolated. Constant shading takes less time because only one shading value needs to be computed for the entire polygon. Interpolated shading gives smoother results. The Gouraud method of interpolation is used: the shade values are computed at each elevation point, coinciding with each polygon vertex. The shading is then interpolated along each edge, finally, between edges along each vertical scan line.

Light-source shading is computed using a combination of depth cueing, ambient light, and diffuse reflection, adapted from Foley and Van Dam (1982, Chapter 19):

$$I = I_a + dI_p(\mathbf{L} \cdot \mathbf{N})$$

where

I_a	Term due to ambient light. All visible objects have at least this intensity, which is approximately 20 percent of the maximum intensity.
$I_p(\mathbf{L} \cdot \mathbf{N})$	Term due to diffuse reflection. The reflected light is proportional to the cosine of the angle between the surface normal vector \mathbf{N} and the vector pointing to the light source, \mathbf{L} . I_p is approximately 0.9.
d	Term for depth cueing, causing surfaces further away from the observer to appear dimmer. The normalized depth is $d=(z+2)/3$, ranging from zero for the most distant point to one for the closest.

Shading Parameters

Parameters affecting the method of shading interpolation, light source direction, and rejection of hidden faces are set with the `SET_SHADING` procedure. Defaults are Gouraud interpolation, light-source direction `[0, 0, 1]`, and rejection of hidden faces enabled. See the description of `SET_SHADING` in the *IDL Reference Guide* for a more complete description of the parameters.

Note

The `REJECT` keyword has no effect on the output of `SHADE_SURF`—it is used only with solids.

Examples Using `SHADE_SURF`

The following figure illustrates the application of `SHADE_SURF`, with light-source shading, to the two-dimensional Gaussian (also drawn as a mesh in [Figure 18-7](#)). This figure was produced by the following statements.

```

; Create a 40-by-40 array in which each element
; is equal to the Euclidean distance from the center.
Z = SHIFT(DIST(40), 20, 20)
; Make Gaussian with a 1/e width of 10:
Z = EXP(-(Z/10)^2)
SHADE_SURF, Z

```

The right half of the following figure shows the use of an array of shades, which in this case is simply the surface elevation scaled into the range of bytes.

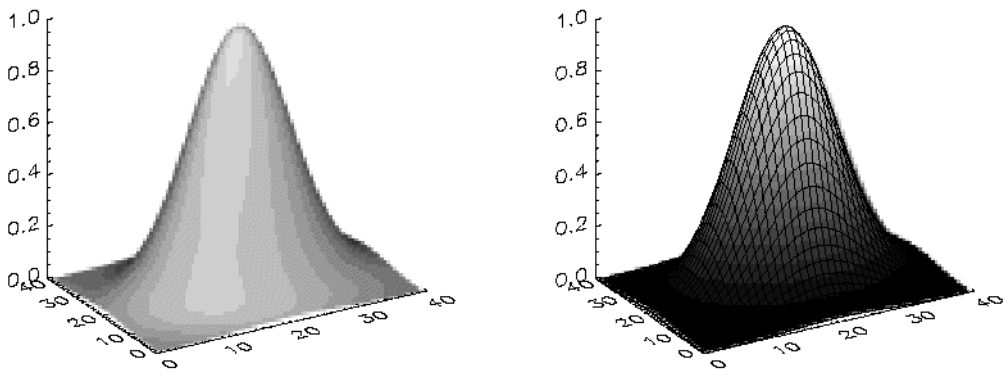


Figure 18-13: Shaded Representations of a Two-Dimensional Gaussian

The output of SURFACE is superimposed over the shaded image with the statements below.

```
; Show Gaussian with shades created by scaling
; elevation into the range of bytes.
SHADE_SURF, Z, SHADES=BYTSCL(Z, TOP = !D.TABLE_SIZE)
; Draw the mesh surface over the shaded figure.
; Suppress the axes:
SURFACE, Z, XST = 4, YST = 4, ZST = 4, /NOERASE
```

The next figure shows the Maroon Bells data as a light-source shaded surface (this data is also shown in the right half of [Figure 18-8](#)). It was produced by the following statements:

```
; Restore variables.
@cntour01
SHADE_SURF, elev, AZ=210, AX=45, XST=4, YST=4, ZST=4
```

The AX and AZ keywords specify the orientation. The axes are suppressed by the axis-style keyword parameters; as in this orientation, the axes are behind the surface.

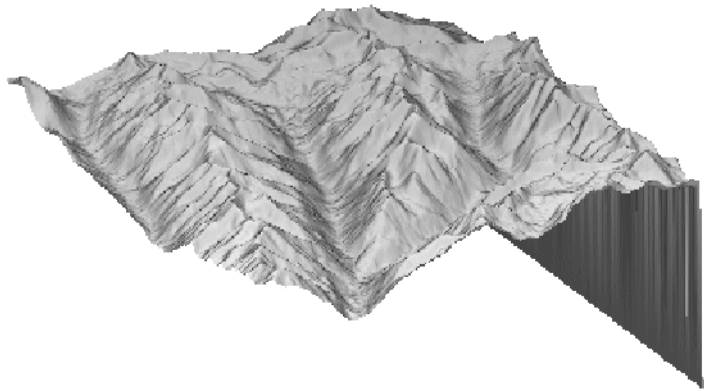


Figure 18-14: Maroon Bells Data Shown as a Shaded Surface

Volume Visualization

A common problem in data visualization is how to display a constant density surface (also known as an isosurface), given a three-dimensional grid of density measurements. In medical imaging, stacking a series of two-dimensional images created by computed tomography or magnetic resonance creates a grid of density measurements that can be contoured to display the surfaces of anatomical structures. Atmospheric scientists create three-dimensional grids of water densities that can be contoured at the proper density level to show the surface of clouds. It is relatively easy to produce these surfaces using the `SHADE_VOLUME` procedure in conjunction with the `POLYSHADE` function.

`SHADE_VOLUME` accepts a three-dimensional grid of densities and a contour level. It outputs the set of polygons that describe the surface of the contour. The polygons are described by a $(3, n)$ array of vertices and a polygon list array that contains the vertices belonging to each polygon. Given a volume array with dimensions of (D_0, D_1, D_2) , the resulting vertex coordinates range between 0 and $D_0 - 1$ in x , 0 and $D_1 - 1$ in y , and 0 and $D_2 - 1$ in z . Keyword parameters to `SHADE_VOLUME` include the following:

LOW

A flag indicating which side of the contour surface is to be viewed: 1 for the high side and 0 for the low (the default). If the contour to be viewed encloses high data values, as in the [“Cloud Example”](#) data, set the `LOW` keyword parameter to 1.

SHADES

An array of shading values for each volume element (voxel). On completion, `SHADE_VOLUME` replaces this array with the interpolated shading for each vertex of the surface.

These polygons are then fed to the `POLYSHADE` function to produce the shaded surface representation. It must be noted that the maximum volume size and polygon complexity are limited by the amount of available memory, as these routines store the density measurements, vertex list, and polygon list in memory.

Cloud Example

This next figure, produced by the following IDL code, shows the three-dimensional contour surface of the precipitating region of a thunderstorm simulated by a three-dimensional cloud model.



Figure 18-15: A 3-dimensional Contour Surface of a Cloud's Precipitating Region

The data were provided by the National Center for Atmospheric Research. The original data are contained in an array called `clouds`, a (55, 55, 32) element floating-point array. Each array element contains the amount of water contained in the corresponding volume of air.

```
; Restore the data:
RESTORE, FILEPATH('clouds3d.dat', SUBDIR=['examples','data'])
; Create the contour surface polygons (v and p)
; at density 0.1, from clouds. Show the low side:
SHADE_VOLUME, clouds, 0.1, v, p, /LOW
; Obtain the dimensions of the volume.
; Variables S[1], S[2], and S[3] now contain
; the number of columns, rows, and slices in the volume:
```

```

s = SIZE(clouds)
; Use SCALE3 to establish the three-dimensional
; transformation matrix. Rotate 45 degrees about the z-axis:
SCALE3, XRANGE=[0,S[1]], YRANGE=[0,S[2]], $
      ZRANGE=[0,S[3]], AX=0, AZ=45
; Render and display the polygons:
TV, POLYSHADE(v, p, /T3D)

```

If you prefer not to enter the IDL code by hand, run the batch file `cclouds` with the following command at the IDL prompt:

```
@cclouds
```

See “[Running the Example Code](#)” on page 294 if IDL does not find the batch file.

The shaded volume can be viewed from different rotations by changing the three-dimensional transformation matrix, `!P.T`, and calling `POLYSHADE` for each view. The following code displays 20 views of the volume, each separated by 18 degrees.

```

; Define number of views:
nframes = 20
FOR i = 0, nframes - 1 DO BEGIN & $
  ; Translate the center of the (0, 1) unit cube
  ; to (0,0) and rotate about the x-axis:
  T3D, TR=[-.5, -.5, -.5], ROT=[0, 360./NFRAMES, 0] & $
  ; Translate the center back to (0.5, 0.5, 0.5):
  T3D, TR = [.5, .5, .5] & $
  ; Show the surface:
  TV, POLYSHADE(v, p, /T3D) & $
ENDFOR

```

The animation rate of the above loop will not be very fast, especially with a larger number of polygons. Each image could be saved for rapid replay by writing it to a disk file. Given enough memory and/or display resources, the `XINTERANIMATE` procedure could be used to animate the views.

Volume Visualization Tools

IDL also includes two interactive volume visualization tools:

SLICER3

`SLICER3`, implemented using IDL Direct Graphics, is a tool used to view isosurfaces and slices of volume data. See [SLICER3](#) in the *IDL Reference Guide* for more information.

XVOLUME

XVOLUME, implemented using IDL Object Graphics, is a utility for viewing and interactively manipulating volumes and isosurfaces. The utility provides a graphical interface for manipulating the volume orientation, adjusting the color table and opacity, viewing image planes and contours, and adjusting the color, opacity, and threshold value of an isosurface. See [XVOLUME](#) in the *IDL Reference Guide* for more information.

References

Foley, J.D., and A. Van Dam (1982), *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Co.



Chapter 19: Map Projections

The following topics are covered in this chapter:

Overview	338	Azimuthal Projections	348
The MAP_SET Procedure	340	Cylindrical Projections	358
The MAP_GRID Procedure	343	Pseudocylindrical Projections	363
The MAP_CONTINENTS Procedure	344	Putting Data on Maps	366
Graphics Techniques for Mapping	345	High-Resolution Continent Outlines	368
Map Projections Described	347	References	370

Overview

The IDL mapping package contains the following procedures:

MAP_SET

This procedure establishes the coordinate conversion mechanism for mapping points on a globe's surface to points on a plane, according to one of 16 possible projections. This procedure also sets up the clipping parameters of the region to be mapped, the center of the map, and the polar rotation. Mapping transformation values are stored in the !MAP system variable. MAP_SET must be called to set up a map projection before any other mapping routines are called. See [MAP_SET](#) in the *IDL Reference Guide* for more information.

MAP_GRID

This procedure draws the graticule of parallels and meridians (grid lines) according to the specifications established by MAP_SET. See [MAP_GRID](#) in the *IDL Reference Guide* for more information.

MAP_CONTINENTS

This procedure draws continental or other boundaries over a map projection established by MAP_SET. Continents, coastlines, rivers, and political borders can be drawn in either low or high resolution. Continents may also be filled with solid colors. See [MAP_CONTINENTS](#) in the *IDL Reference Guide* for more information.

MAP_IMAGE and MAP_PATCH

These functions return an image warped to fit the current map projection. See [MAP_IMAGE](#) and [MAP_PATCH](#) in the *IDL Reference Guide* for more information.

MAP_PROJ_INIT

This function establishes the coordinate conversion mechanism for mapping points on a globe's surface to points on a plane, according to either one of the IDL projections or one of the General Cartographic Transformation Package (GCTP) map projections. Unlike MAP_SET, this function does not modify the !MAP system variable, but rather returns a !MAP structure variable that can be used by the map transformation functions MAP_PROJ_FORWARD and MAP_PROJ_INVERSE. See [MAP_PROJ_INIT](#) in the *IDL Reference Guide* for more information.

MAP_PROJ_FORWARD, and MAP_PROJ_INVERSE

These functions transform map coordinates between latitude/longitude and Cartesian (X, Y) coordinates. Both functions can use the map transformation values from either the !MAP system variable or a !MAP structure created by MAP_PROJ_INIT. See [MAP_PROJ_FORWARD](#) and [MAP_PROJ_INVERSE](#) in the *IDL Reference Guide* for more information.

Example Graphics

The examples in this chapter are all written to take advantage of IDL Direct Graphics. Examples and techniques using IDL Object Graphics are contained in the later chapters of this manual.

The MAP_SET Procedure

The MAP_SET procedure establishes the axis type and coordinate conversion mechanism for mapping points on the Earth's surface, expressed in latitude and longitude, to points on a plane, according to one of 16 possible map projections. Many other keywords are available to control various graphics options. For information on all the available keywords, see [MAP_SET](#) in the *IDL Reference Guide* for more information.

You can select the map projection, the map center, polar rotation, and geographical limits. The system variable !MAP retains the information needed to effect coordinate conversions to the plane and inversely from the projection plane to points on the earth in latitude and longitude. Do not change the values of the fields in !MAP directly. You can plot the graticule and continental boundaries with MAP_SET by setting the GRID and CONTINENT keywords. The procedure has the calling sequence:

```
MAP_SET[ , P0lat, P0lon, Rot]
```

where the keywords are described as follows:

P_{0lat}

P_{0lat} is the latitude of the point on the Earth's surface at the center of the projection plane. Latitude is measured in degrees North of the equator, where $-90^\circ \leq P_{0lat} \leq 90^\circ$. If P_{0lat} is not set, the default value is zero.

P_{0lon}

P_{0lon} is the longitude of the point on the Earth's surface to be mapped to the center of the map projection. Longitude is measured in degrees east of the Greenwich meridian and $-180^\circ \leq P_{0lon} \leq 180^\circ$. If P_{0lon} is not set, the default value is zero.

Rot

Rot is the angle through which the North direction should be rotated around the line L between the Earth's center and the point (P_{0lat}, P_{0lon}) . Rot is measured in degrees with the positive direction being clockwise rotated around L . Rot should satisfy $-180 \leq Rot \leq 180$.

If the center is at the North Pole, the North direction is in the direction of $P_{0lon} + 180$ degrees. If the origin is at the South Pole, then North is in the direction of P_{0lon} . The default value of Rot is zero.

MAP_SET Keywords

MAP_SET accepts many keywords that customize the projection attributes of the map. A few of the important ones are described below. See [MAP_SET](#) in the *IDL Reference Guide* for descriptions of all the keywords.

CONTINENTS

Set this keyword to plot the continental boundaries.

GRID

Set this keyword to draw the grid of parallels and meridians.

ISOTROPIC

Set this keyword to produce a map that has the same scale in the *X* and *Y* directions.

LIMIT

Set this keyword to a four- or eight-element vector. The four-element vector, [Lat_{min} , Lon_{min} , Lat_{max} , Lon_{max}], specifies the boundaries of a simple region to be mapped. (Lat_{min} , Lon_{min}) and (Lat_{max} , Lon_{max}) are the latitudes and longitudes of two points diagonal from each other on the region's boundary. For more complex regions or projections, the eight-element vector, [Lat_0 , Lon_0 , Lat_1 , Lon_1 , Lat_2 , Lon_2 , Lat_3 , Lon_3] specifies four points located, respectively, on the left, top, right and bottom edges of the map.

Note

If the longitude range in LIMIT is less than or equal to 180 degrees, map clipping is performed in lat/lon coordinates before the transform. If the longitude range is greater than 180 degrees, map clipping is done in Cartesian coordinates after the transform. For non-cylindrical projections, clipping after the transformation to Cartesian coordinates means that some lat/lon points that fall outside the bounds specified by LIMIT may not be clipped. This occurs when the transformed lat/lon points fall inside the cartesian clipping rectangle.

SCALE

Set this keyword to construct an isotropic map with the given scale, set to the ratio of 1:scale. If SCALE is not specified, the map is fit to the window. The typical scale for global maps is in the ratio of between 1:100 million and 1:200 million. For continents, the typical scale is in the ratio of approximately 1:50 million. For

example, `SCALE=100E6` sets the scale at the center of the map to 1:100 million, which is in the same ratio as 1 inch to 1578 miles (1 cm to 1000 km).

The MAP_GRID Procedure

MAP_GRID draws the graticule of parallels and meridians according to the specifications established by MAP_SET. The MAP_SET procedure should be called before MAP_GRID to establish the projection type, the center of the projection, polar rotation, and geographical limits. Latitude and/or longitude lines can be drawn in different line styles, colors, and spacings. See [MAP_GRID](#) in the *IDL Reference Guide* for more information on all the available options.

The MAP_CONTINENTS Procedure

MAP_CONTINENTS draws the projection of the continental boundaries, according to the specifications established by MAP_SET. MAP_SET should be called before MAP_CONTINENTS to establish the projection type, the center of the projection, polar rotation, and geographical limits. See [MAP_CONTINENTS](#) in the *IDL Reference Guide* for more information on all the available options.

Graphics Techniques for Mapping

Standard graphics techniques are insufficient when projecting areas on a sphere to a two-dimensional surface for two reasons. First, two points on a sphere are connected by two different lines. Second, areas may wrap around the edges of cylindrical and pseudo-cylindrical projections.

Graphical entities on the surface of a sphere can be properly represented on any map by using a combination of the following four stages: splitting, 3D clipping, projection, and rectangular clipping. The `MAP_SET` procedure automatically sets up the proper mapping technique to best fit the projection selected by the user.

Warning

For proper rendering, splitting, and clipping, polygons must be traversed in counter-clockwise order when observed from outside the sphere. If this requirement is not met, the exterior, instead of the interior, of the polygons may be filled. Also, vectors connecting the points spanning the singular line for cylindrical projections will be drawn in the wrong direction if polygons are not traversed in the correct order.

Splitting

The splitting stage is used for cylindrical and pseudo-cylindrical projections. The singular line, one half of a great circle line, is located opposite the center of the projection; points on this line appear on both edges of the map. The singular line is the intersection of the surface of the sphere with a plane passing through the center of projection, one of the poles of projections, and the center of the sphere.

3D Clipping

Map graphics are clipped to one side of an arbitrary clipping plane in one or more clipping stages. For example, to draw a hemisphere centered on a given point, the clipping plane passes through the center of the sphere and has a normal vector that coincides with the given point.

Projection

In the projection stage, a point expressed in latitude and longitude is transformed to a point on the mapping plane.

Rectangular Clipping

After the map graphics have been projected onto the mapping plane, a conventional rectangular clipping stage ensures that the graphics are properly bounded and closed in the rectangular display area.

Map Projections Described

In the following sections, the available IDL projections are discussed in detail. The projections are grouped within three categories: azimuthal, cylindrical, and pseudo-cylindrical.

Note

The General Cartographic Transformation Package (GCTP) map projections are not described here. Documentation for the GCTP package is available from the US Geologic Survey at <http://mapping.usgs.gov>.

Note

In this text, the plane of the projection is referred to as the UV plane with horizontal axis u and vertical axis v .

Azimuthal Projections

With azimuthal projections, the UV plane is tangent to the globe. The point of tangency is projected onto the center of the plane and its latitude and longitude are P_{0lat} and P_{0lon} , respectively. Rot is the angle between North and the v -axis.

Important characteristics of azimuthal maps include the fact that directions or azimuths are correct from the center of the projection to any other point, and great circles through the center are projected to straight lines on the plane.

The IDL mapping package includes the following azimuthal projections: orthographic, stereographic, gnomonic, azimuthal equidistant, Aitoff, Lambert's azimuthal equal area, Hammer-Aitoff, and satellite.

Orthographic Projection

The orthographic projection was known by the Egyptians and Greeks 2000 years ago. This projection looks like a globe because it is a perspective projection from infinite distance. As such, it maps one hemisphere of the globe into the UV plane. Distortions are greatest along the rim of the hemisphere where distances and land masses are compressed.

The following statements are used to produce an orthographic projection centered over Eastern Spain at a scale of 70 million to 1:

```
MAP_SET, /ORTHOGRAPHIC, 40, 0, SCALE=70e6, /CONTINENTS, $
/GRID, LONDEL=15, LATDEL=15, $
TITLE = 'Oblique Orthographic'
```

The output of these statements is shown in the following figure.

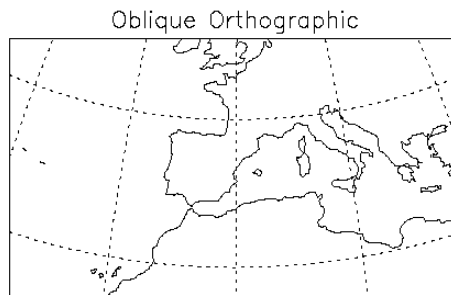


Figure 19-1: Orthographic Projection

Stereographic Projection

The stereographic projection is a true perspective projection with the globe being projected onto the UV plane from the point P on the globe diametrically opposite to the point of tangency. The whole globe except P is mapped onto the UV plane. There is great distortion for regions close to P , since P maps to infinity.

The stereographic projection is the only known perspective projection that is also conformal. It is frequently used for polar maps. For example, a stereographic view of the north pole has the south pole as its point of perspective.

The following statement uses the stereographic projection to draw the hemisphere centered on the equator at longitude -105 degrees and produces an equatorial stereographic map:

```
MAP_SET, /STEREO, 0, -105, /ISOTROPIC, $  
  /GRID, LATDEL = 20, LONDEL = 20, /HORIZON, /CONTINENT, $  
  TITLE = 'Equatorial Stereographic'
```

The output of this statement is shown in the following figure:

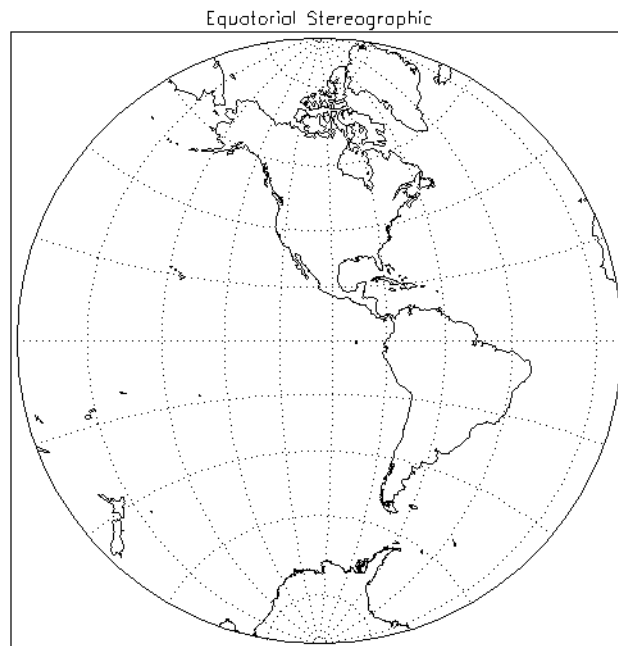


Figure 19-2: An Azimuthal Projection

Since the `LATDEL` and `LONDEL` keywords are set to 20, parallels and meridians are spaced 20 degrees apart. The `GRID` and `CONTINENT` keywords signal that the grid and continents should be drawn.

Gnomonic Projection

The gnomonic projection (also called Central or Gmonic) projects all great circles to straight lines. The gnomonic projection is the perspective, azimuthal projection with point of perspective at the center of the globe. Hence, with the gnomonic projection, the interior of a hemispherical region of the globe is projected to the UV plane with the rim of the hemisphere going to infinity. Except at the center, there is great distortion of shape, area, and scale. The default clipping region for the gnomonic projection is a circle with a radius of 60 degrees at the center of projection.

```
MAP_SET, /GNOMIC, 40, -105, LIMIT = [20, -130, 70, -70], $
      /ISOTROPIC, /GRID, /CONTINENT, $
      TITLE = 'Oblique Gnomonic'
```

This projection is centered around the point at latitude 40 degrees and longitude -105 degrees. The region on the globe that is mapped lies between 20 degrees and 70 degrees of latitude and -130 degrees and -70 degrees of longitude.

The output of these statements is shown in the following figure:

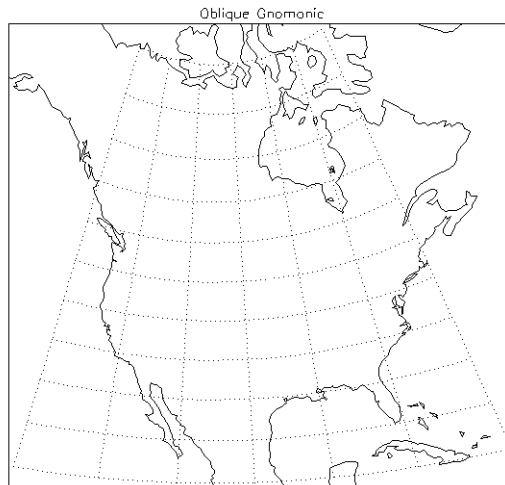


Figure 19-3: A Gnomonic Projection

Azimuthal Equidistant Projection

The azimuthal equidistant projection is also not a true perspective projection, because it preserves correctly the distances between the tangent point and all other points on the globe. The point P opposite the tangent point is mapped to a circle on the UV plane, and hence, the whole globe is mapped to the plane. There is infinite distortion close to the outer rim of the map, which is the circular image of P .

If the keyword `LIMIT` is not set, the whole globe is mapped to the UV plane.

```
MAP_SET, /AZIMUTHAL, /ISOTROPIC, -90, $  
      /GRID, LONDEL=20, LATDEL=20, /CONTINENT, $  
      /HORIZON, TITLE = 'Polar Azimuthal'
```

It is centered at the South Pole and shows the entire globe.

The output of these statements is shown in the following figure:

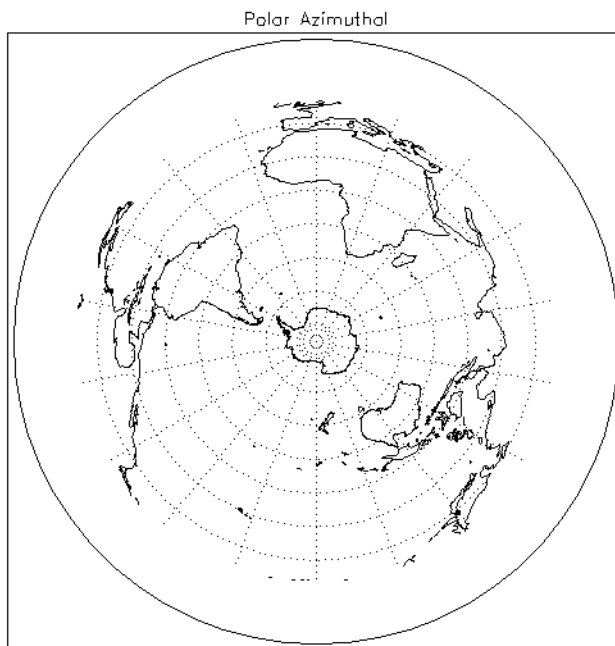


Figure 19-4: An Azimuthal Equidistant Projection

Aitoff Projection

The Aitoff projection modifies the equatorial aspect of one hemisphere of the azimuthal equidistant projection, described above. Lines parallel to the equator are stretched horizontally and meridian values are doubled, thereby displaying the world as an ellipse with axes in a 2:1 ratio. Both the equator and the central meridian are represented at true scale; however, distances measured between the point of tangency and any other point on the map are no longer true to scale.

An Aitoff projection centered on the international dateline can be produced by the command:

```
MAP_SET, 0, 180, /Aitoff, /GRID, /CONTINENTS, /ISOTROPIC, $
      TITLE= 'Aitoff Projection'
```

The output of these statements is shown in the following figure:

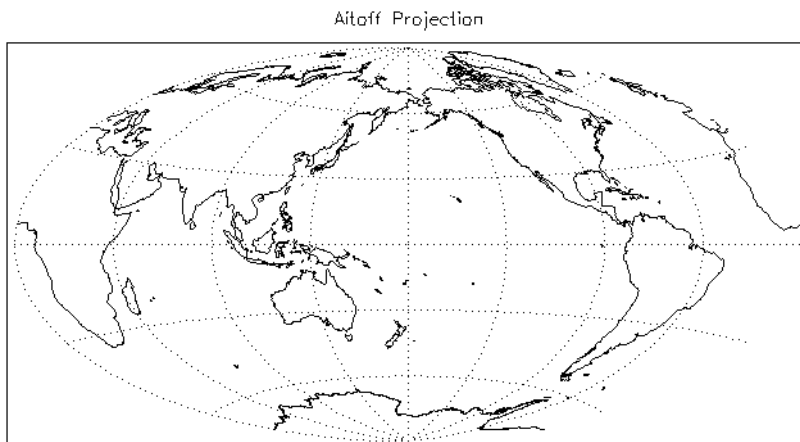


Figure 19-5: An Aitoff Projection

Lambert's Equal Area Projection

Lambert's equal area projection adjusts projected distances in order to preserve area. Hence, it is not a true perspective projection. Like the stereographic projection, it maps to infinity the point *P* diametrically opposite the point of tangency. Note also that to preserve area, distances between points become more contracted as the points become closer to *P*. Lambert's equal area projection has less overall scale variation than the other azimuthal projections.

```
MAP_SET, /LAMBERT, 90, 0, -105, /ISOTROPIC, $
```



```
/GRID, LATDEL=20, LONDEL=20, $  
/CONTINENTS, E_CONTINENTS={FILL:1}, /HORIZON, $  
TITLE = 'Polar Lambert'
```

The output of these statements is shown in the following figure:

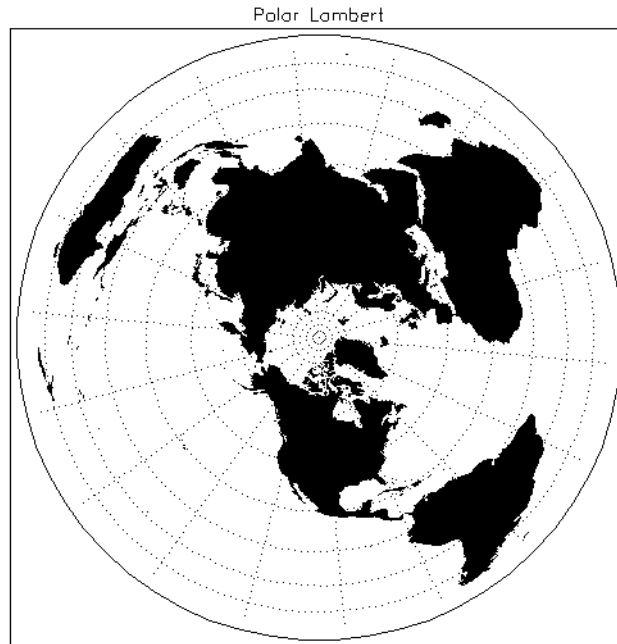


Figure 19-6: A Lambert's Equal Area Projection

Note

This map shows the Northern Hemisphere rotated counterclockwise 105 degrees, filling the continents with a solid color

Hammer-Aitoff Projection

Although the Hammer-Aitoff projection is not truly azimuthal, it is included in this section because it is derived from the equatorial aspect of Lambert's equal area projection limited to a hemisphere (in the same way Aitoff's projection is derived from the equatorial aspect of the azimuthal equidistant projection). In this derivation, the hemisphere is represented inside an ellipse with the rest of the world in the lunes of the ellipse.

Because the Hammer-Aitoff projection produces an equal area map of the entire globe, it is useful for visual representations of geographically related statistical data and distributions. Astronomers use this projection to show the entire celestial sphere on one map in a way that accurately depicts the relative distribution of the stars in different regions of the sky.

A Hammer-Aitoff projection centered on the international dateline can be produced by the command:

```
MAP_SET, 0, 180, /HAMMER, /GRID, /CONTINENTS, /ISOTROPIC, $
      /HORIZON, TITLE= 'Hammer-Aitoff Projection'
```

The output of these statements is shown in the following figure:

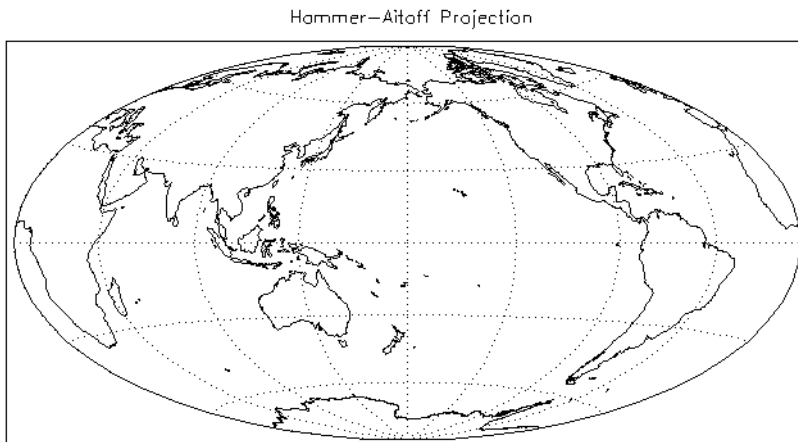


Figure 19-7: The Hammer-Aitoff Projection

Satellite Projection

The satellite projection, also called the General Perspective projection, simulates a view of the globe as seen from a camera in space. If the camera faces the center of the globe, the projection is called a Vertical Perspective projection (note that the orthographic, stereographic, and gnomonic projections are special cases of this projection), otherwise the projection is called a Tilted Perspective projection.

The globe is viewed from a point in space, with the viewing plane touching the surface of the globe at the point directly beneath the satellite (the sub-satellite point). If the projection plane is perpendicular to the line connecting the point of projection and the center of the globe, a Vertical Perspective projection results. Otherwise, the

projection plane is horizontally turned Γ degrees clockwise from the north, then tilted ω degrees downward from horizontal.

For the satellite projection, P_{0Lat} and P_{0Lon} represent the latitude and longitude of the sub-satellite point. Three additional parameters, P , Ω , and Γ (supplied as a three-element vector argument to the SAT_P keyword), are required where:

- P is the distance of the point of perspective (camera) from the center of the globe, expressed in units of the radius of the globe.
- Ω is the downward tilt of the camera, in degrees from the new horizontal. If both Γ and Ω are 0, a Vertical Perspective projection results.
- Γ is the angle, expressed in degrees clockwise from north, of the rotation of the projection plane.

Note

Since all meridians and parallels are oblique lines or arcs, the LIMIT keyword must be supplied as an eight-element vector representing four points that delineate the limits of the map. The extent of the map limits, when expressed in latitude/longitude is a complicated polygon, rather than a simple quadrilateral.

The map in the accompanying figure, which shows the eastern seaboard of the United States from an altitude of about 160km, above Newburgh, NY, was produced with the code that follows.

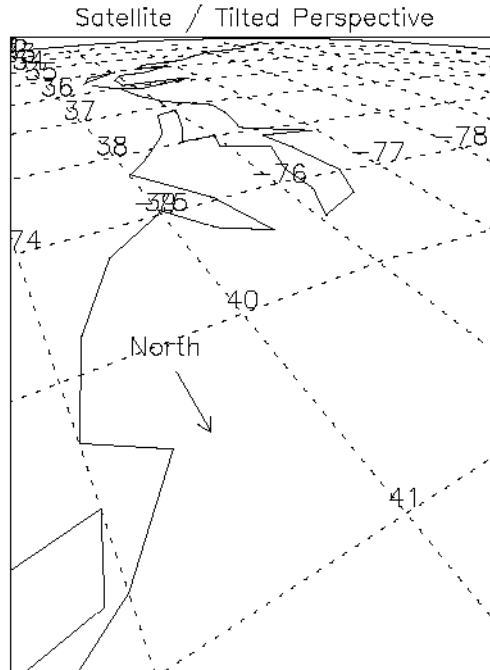


Figure 19-8: Satellite Projection

The parameters for this satellite projection are:

- Center of projection = 41.5N latitude, -74W longitude
- P (altitude) = 1.025 = $(1.0 + 160 / 6371\text{km})$
- Γ (rotation of projection plane) = 150 degrees
- Ω (tilt of projection plane) = 55 degrees
- The eight element LIMIT keyword array specifies the latitude/longitude locations of points at the bottom, left, top, and right of the map respectively.
- The HORIZON keyword draws a horizon line.

Example: Labeling and Drawing Projections

Labeling and drawing a vector on a satellite projection.

```
MAP_SET, /SATELLITE, SAT_P=[1.0251, 55, 150], 41.5, -74., $
      /ISOTROPIC, /HORIZON, $
      LIMIT=[39, -74, 33, -80, 40, -77, 41,-74], $
      /CONTINENTS, TITLE='Satellite / Tilted Perspective'
; Set up the satellite projection:
MAP_GRID, /LABEL, LATLAB=-75, LONLAB=39, LATDEL=1, LONDEL=1
; Get North vector:
p = convert_coord(-74.5, [40.2, 40.5], /TO_NORM)
; Draw North arrow:
ARROW, p(0,0), p(1,0), p(0,1), p(1,1), /NORMAL
XYOUTS, -74.5, 40.1, 'North', ALIGNMENT=0.5
```

Cylindrical Projections

A cylindrical projection maps the globe to a cylinder which is formed by wrapping the UV plane around the globe with the u -axis coinciding with a great circle. The parameters P_{olat} , P_{olon} , and Rot determine the great circle that passes through the point $C=(P_{olat}, P_{olon})$. In the discussions below, this great circle is sometimes referred to as EQ. Rot is the angle between North at the map's center and the v -axis (which is perpendicular to the great circle). The cylinder is cut along the line parallel to the v -axis and passing through the point diametrically opposite to C. It is then rolled out to form a plane.

The cylindrical projections in IDL include: Mercator, Transverse Mercator, cylindrical equidistant, Miller, Lambert's conformal conic, and Alber's equal-area conic.

Mercator Projection

Mercator's projection is partially developed by projecting the globe onto the cylinder from the center of the globe. This is a partial explanation of the projection because vertical distances are subjected to additional transformations to achieve conformity—that is, local preservation of shape. To properly use the projection, the user should be aware that the two points on the globe 90 degrees from the central great circle (e.g., the North and South Poles in the case that the selected great circle is the equator) are mapped to infinite distances. By default, the keyword `LIMIT` is set to `[-80, -180, 80, 180]` because of the great distortions around the poles when the equator is selected.

The following statement produces a simple Mercator projection:

```
MAP_SET, /MERCATOR, 0, 0, /ISOTROPIC, $
      /GRID, /CONTINENTS, $
      TITLE = 'Simple Mercator'
```

The result of this statement is shown in the upper-left corner of the following figure.

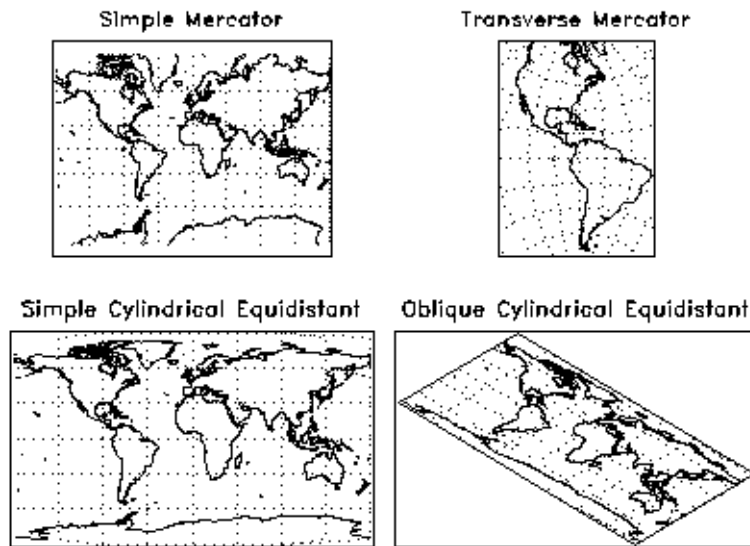


Figure 19-9: Cylindrical Projections

Latitudes range from -80 degrees to 80 degrees.

Transverse Mercator Projection

The Transverse Mercator (also called the *UTM*, and *Gauss-Krueger* in Europe) projection rotates the equator of the Mercator projection 90 degrees so that it follows a specified central meridian. In other words, the Transverse Mercator involves projecting the Earth onto a cylinder which is always in contact with a meridian instead of with the Equator.

The central meridian intersects two meridians and the Equator at right angles; these four lines are straight. All other meridians and parallels are complex curves which are concave toward the central meridian. Shape is true only within small areas and the areas increase in size as they move away from the central meridian. Most other IDL projections are scaled in the range of ± 1 to $\pm 2\pi$; the UV plane of the Transverse Mercator projection is scaled in meters. The conformal nature of this projection and its use of the meridian makes it useful for north-south regions.

The Clarke 1866 ellipsoid is used for the default, but its parameters can be altered with the ELLIPSOID keyword.

Example: The UTM Map

To create a UTM map, centered near London, with a scale of 10 million to one, type the following:

```
MAP_SET, /TRANSVERSE, 51, 0, SCALE=10e6, $
/GRID, LATDEL=2.5, LONDEL=2.5, /LABEL, LONLAB=48, $
/CONTINENTS, E_CONT={COUNTRIES:1, COASTS:1}, $
TITLE='UTM Projection'
```

When the eccentricity of the Earth is not important, global scale Transverse Mercator projections can be easily created using the Mercator projection with the CENTRAL_AZIMUTH keyword set to 90 degrees, and setting *Rot* to rotate the map 90 degrees. For example, to create the Transverse Mercator map showing North and South America, with a central meridian of -90 degrees West and centered on the Equator, shown in the upper-right corner of the figure in the “[Mercator Projection](#)” section. It is produced by the following statement:

```
MAP_SET, /MERCATOR, 0, -75, 90, CENTRAL_AZIMUTH=90, $
/ISOTROPIC, LIMIT= [32,-130, 70,-86, -5,-34, -58, -67], $
/GRID, LATDEL=15, LONDEL=15, /CONTINENTS, $
TITLE = 'Transverse Mercator'
```

Cylindrical Equidistant Projection

The cylindrical equidistant projection is one of the simplest projections to construct. If EQ is the equator, this projection simply lays out horizontal and vertical distances on the cylinder to coincide numerically with their measurements in latitudes and longitudes on the sphere. Hence, the equidistant cylindrical projection maps the entire globe to a rectangular region bounded by

$$-180 \leq u \leq 180$$

and

$$-90 \leq v \leq 90$$

If EQ is the equator, meridians and parallels will be equally spaced parallel lines.

The following code is used to produce a simple cylindrical equidistant projection and an oblique cylindrical equidistant projection as shown in the lower-left and lower-right sections of the figure under the “[Mercator Projection](#)” heading:


```
MAP_SET, /CYLINDRICAL, 0, 0, /GRID, /CONTINENTS, $
TITLE = 'Simple Cylindrical Equidistant'
```

Now rotate the projection by 45%:

```
MAP_SET, /CYLINDRICAL, 0, 0, 45, $
/GRID, /CONTINENT, /HORIZON, $
TITLE='Oblique Cylindrical Equidistant'
```

Miller Cylindrical Projection

The Miller projection is a simple mathematical modification of the Mercator projection, incorporating some aspects of cylindrical projections. It is not equal-area, conformal or equidistant along the meridians. Meridians are equidistant from each other, but latitude parallels are spaced farther apart as they move away from the Equator, thereby keeping shape and area distortion to a minimum. The meridians and parallels intersect each other at right angles, with the poles shown as straight lines. The Equator is the only line shown true to scale and free of distortion.

Conic Projection

The Lambert's conformal conic with two standard parallels is constructed by projecting the globe onto a cone passing through two parallels. Additional scaling achieves conformity. The pole under the cone's apex is transformed to a point, and the other pole is mapped to infinity. The scale is correct along the two standard parallels. Parallels are projected onto circles and meridians onto equally spaced straight lines. The `STANDARD_PARALLELS` keyword specifies the latitudes of one or two standard parallels.

The following statement produces the map shown in the accompanying figure, which features North America with standard parallels at 20 degrees and 60 degrees:

```
MAP_SET, /CONIC, 40, -80, STANDARD_PARALLELS=[20,60], $
/ISOTROPIC, LIMIT=[0, -260, 90, 100], $
/GRID, LATDEL=15, LONDEL=20, /CONTINENT, $
TITLE= 'Lambert's Conic'
```

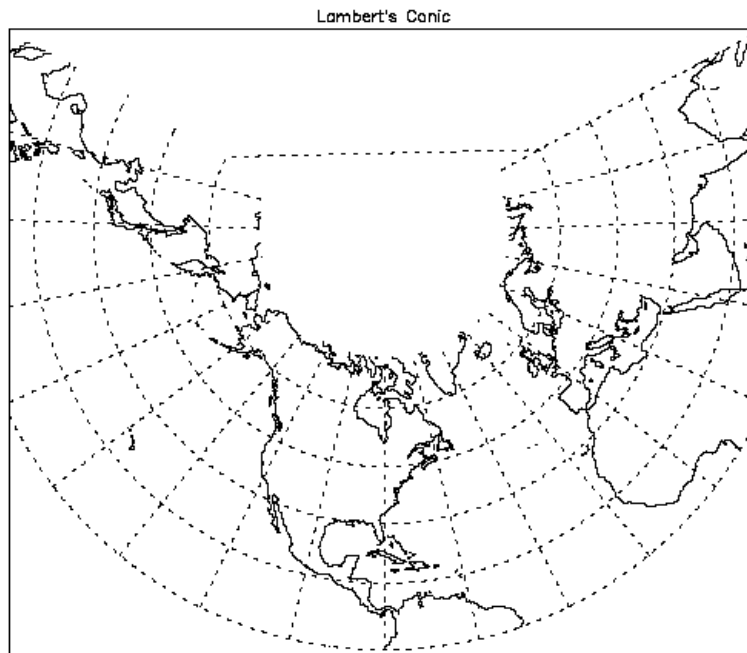


Figure 19-10: Lambert's Conformal Conic with Standard Parallels at 20° and 60°

Albers Equal-Area Conic Projection

The Albers Equal-Area Conic is like most other conics in that meridians are equally spaced radii, parallels are concentric arcs of circles and scale is constant along any parallel. To maintain equal area, the scale factor along meridians is the reciprocal of the scale factor along parallels, with the scale along the parallels between the two standard parallels too small, and the scale beyond the standard parallels too large. Standard parallels are correct in scale along the parallel, as well as in every direction.

The Albers projection is particularly useful for predominantly east-west regions. Any keywords for the Lambert conformal conic also apply to the Albers conic.

Pseudocylindrical Projections

Pseudocylindrical projections are distinguished by the fact that in their simplest form, lines of latitude are parallel straight lines and meridians are curved lines.

Robinson Cylindrical

This pseudocylindrical projection was designed by Arthur Robinson in 1963 for Rand McNally. It is suitable for World maps and is a compromise to best fulfill a number of conflicting requirements, including an uninterrupted format, minimal shearing, minimal apparent area-scale distortion for major continents, and simplicity. It was designed to make the world look right. Since its introduction, it has been adopted by the National Geographic Society for many of their world maps.

Each individual parallel is equally divided by the meridians. The poles are represented by lines rather than points to avoid compressing the northern land masses.

Note

The central meridian should always be 0 degrees longitude to retain the correct balance of shapes, sizes, and relative positions.

The next statement produces the Robinson projection shown in the lower-left corner of the figure which follows.

```
MAP_SET, /ROBINSON, 0, 0, /ISOTROPIC, /GRID, $
      /HORIZON, E_CONTINENTS={FILL:1}, TITLE='Robinson'
```

Sinusoidal Projection

With the sinusoidal projection, the central meridian is a straight line and all other meridians are equally spaced sinusoidal curves. The scaling is true along the central meridian as well as along all parallels.

The sinusoidal projection is one of the easiest projections to construct. The formulas below from Snyder (1987) give the relationship between the latitude ϕ and longitude λ of a point on the globe and its image on the UV plane.

$$u = \lambda \cos \phi$$

$$v = \phi$$

The parameters P_{OLat} and Rot of the `MAP_SET` procedure must be zero. If they are not, an error message results and the procedure `MAP_SET` will reset both of these parameters to zero and continue. By default, P_{OLon} (the central longitude) is zero, but the user can set it to any other value between -180 and 180 . If the keyword `LIMIT` is undefined, the entire globe is the region selected for mapping.

The following statements produces the sinusoidal map of the whole globe centered at longitude 0 degrees and latitude 0 degrees:

```
MAP_SET, /SINUSOIDAL, /ISOTROPIC, $
        /CONTINENTS, TITLE='Sinusoidal'
MAP_GRID, LONDEL=20, /HORIZON
```

The result of these statements is shown in the upper-left corner of the following figure.

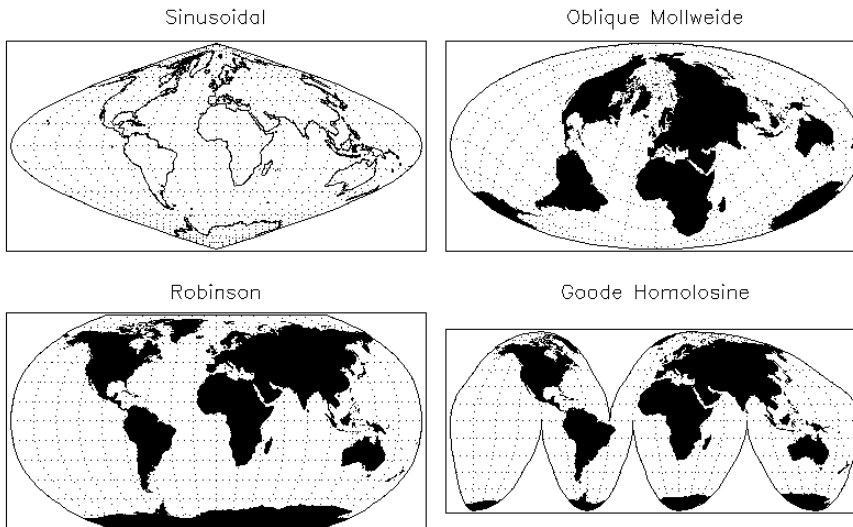


Figure 19-11: Pseudocylindrical Projections

Mollweide Projection

With the Mollweide projection, the central meridian is a straight line, the meridians 90 degrees from the central meridian are circular arcs and all other meridians are elliptical arcs. The Mollweide projection maps the entire globe onto an ellipse in the UV plane. The circular arcs encompass a hemisphere and the rest of the globe is contained in the lunes on either side.

If the keyword `LIMIT` is not set, the whole globe will be mapped to the plane. The following statement produces a Mollweide projection in oblique form, as illustrated in the upper-right corner of the previous figure:

```
MAP_SET, /MOLLWEIDE, 45, 0, /ISOTROPIC, $
      /GRID, LATDEL=20, LONDEL=20, $
      /HORIZON, E_CONTINENTS={FILL:1}, $
      TITLE='Oblique Mollweide'
```

Since the center of the projection is not on the equator, parallels of latitude are not straight lines, just as they are not straight lines with an oblique Mercator or cylindrical equidistant projection.

Goode's Homolosine Projection

The Goode interrupted Homolosine projection, developed by J. Paul Goode, in 1923, is designed for World maps to show the continents with minimal scale and shape distortion. This is accomplished by interrupting the projection and choosing several central meridians to coincide with large land masses. This projection is a fusion of the Sinusoidal projection between the latitudes of 44.7 degrees North and South, and the Mollweide projection between these parallels and the poles.

The following statement produced the example of Goode's Homolosine projection in the lower-right corner of the previous figure:

```
MAP_SET, /GOODESHOMOSINE, 0, 0, /ISOTROPIC, /GRID, $
      LATDEL=15, LONDEL=20, /HORIZON, E_CONTINENTS={FILL:1}, $
      TITLE='Goode Homolosine'
```

Putting Data on Maps

The procedures `PLOT`, `OPlot`, `PLOTS`, `XYOUTS`, and `CONTOUR` can be used to display and annotate geographical data on maps created by the routines `MAP_SET`, `MAP_GRID`, and `MAP_CONTINENTS`. The `MAP_IMAGE` procedure can be used to warp regularly-gridded images to map projections.

Example—Using `CONTOUR` with `MAP_SET`

The following simple example creates a `CONTOUR` plot over a Mollweide map projection and then over a polar stereographic projection. The resulting map is shown below.

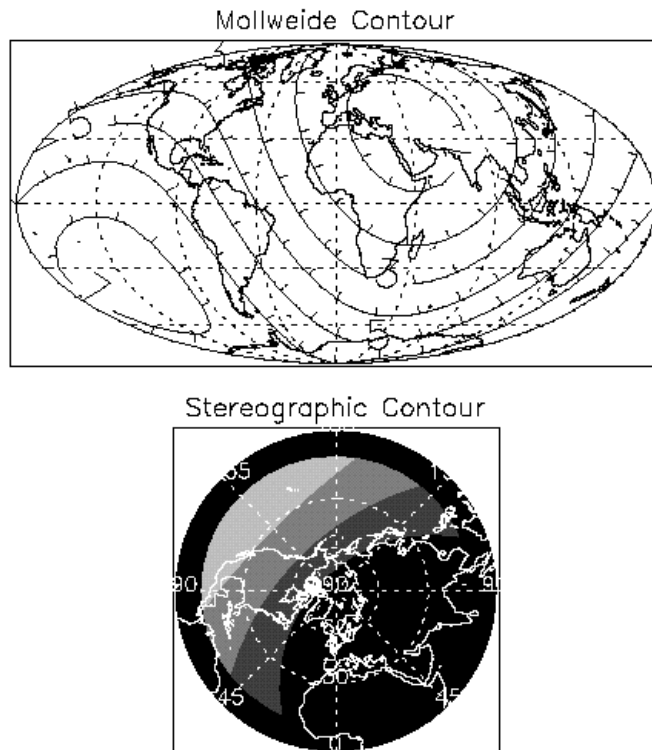


Figure 19-12: Combining `CONTOUR` with `MAP_SET`

```

; Make a 10 degree latitude/longitude grid covering the Earth:
lat = REPLICATE(10., 37) # FINDGEN(19) - 90.
lon = FINDGEN(37) # REPLICATE(10, 19)
; Convert lat and lon to Cartesian coordinates:
X = COS(!DTOR * lon) * COS(!DTOR * lat)
Y = SIN(!DTOR * lon) * COS(!DTOR * lat)
Z = SIN(!DTOR * lat)
; Create the function to be plotted, set it equal
; to the distance squared from (1,1,1):
F = (X-1.)^2 + (Y-1.)^2 + (Z-1.)^2
MAP_SET, /MOLLWEIDE, 0, 0, /ISOTROPIC, $
      /HORIZON, /GRID, /CONTINENTS, $
      TITLE='Mollweide Contour'
CONTOUR, F, lon, lat, NLEVELS=7, $
      /OVERPLOT, /DOWNHILL, /FOLLOW
; Fill the contours over the northern hemisphere and
; display in a polar stereographic projection:
MAP_SET, /STEREO, 90, 0, $
      /ISOTROPIC, /HORIZON, E_HORIZON={FILL:1}, $
      TITLE='Stereographic Contour'
; Display points in the northern hemisphere only:
CONTOUR, F(*,10:*), lon(*,10:*), lat(*,10:*), $
      /OVERPLOT, /FILL, NLEVELS=5
MAP_GRID, /LABEL, COLOR=255
MAP_CONTINENTS, COLOR=255

```

Limitations

Filling contours or polygons over maps that cover more than a hemisphere will produce incorrect results. This is because of the ambiguity between polygons that enclose an area, and those that enclose the entire surface of the sphere outside the area; and because of the ambiguity of determining the clockwise-ness of polygons on a sphere that cover more than a hemisphere.

High-Resolution Continent Outlines

IDL supports two different datasets that contain continent outlines and other geographical and political boundaries. The default data set is a low-resolution continental outline database that is automatically installed when you install IDL. The high-resolution database was adapted from the 1993 CIA World Map database by Thomas Oetli of the Swiss Meteorological Institute. The high-resolution outlines are found in an optional data set that may not have been installed when your copy of IDL was first installed.

To access the high-resolution data set, simply set the `HIRES` keyword when calling `MAP_CONTINENTS` with the `COASTS`, `COUNTRIES`, `FILL_CONTINENTS`, or `RIVERS` keywords. You can also get high-resolution continent boundaries by calling `MAP_SET` with the `HIRES` and `CONTINENTS` keywords set. See [MAP_CONTINENTS](#) in the *IDL Reference Guide* for an example of using the high-resolution outlines.

Resolution of Map Databases

Data points in the CIA World Map database are approximately one kilometer apart. Note, however, that in the case of the coast and river databases, actual distances between the data points may be much smaller because of convolutions in the coastline or riverbed.

Data points in the low-resolution map database are either a subset of the high-resolution database (rivers and country boundaries) or are based on the continental map database used in previous versions of IDL (the file `supmap.dat` in the `resource/maps` subdirectory of the IDL distribution). Data points in the low-resolution database are approximately 10 kilometers apart.

Neither of the map databases is intended for high-precision work.

The following table compares the low-resolution and high-resolution map databases:

Feature	Low-Resolution	High-Resolution
Coastlines, islands, and lakes (including continental outlines)	Data in file <code>supmap.dat</code> .	Entire CIA World Map

Table 19-1: Comparison of Low- and High-resolution Map Databases

Feature	Low-Resolution	High-Resolution
Continental polygons	Data extracted from <code>supmap.dat</code> .	Every 20th point of CIA World Map.
Rivers	Every 250th point of the CIA World Map.	Entire CIA World Map.
National boundaries	Every 100th point of CIA World Map.	Entire CIA World Map.

Table 19-1: Comparison of Low- and High-resolution Map Databases

References

Greenwood, David (1964), *Mapping*, University of Chicago Press, Chicago.

Pearson, Frederick II (1990), *Map Projections: Theory and Applications*, CRC Press, Inc., Boca Raton.

Snyder, John P. (1987), *Map Projections—A Working Manual*, U.S. Geological Survey Professional Paper 1395, U.S. Government Printing Office, Washington, D.C.



Chapter 20: Image Display Routines

The following topics are found in this chapter:

Overview	372	Color Tables	381
Images	373	TrueColor Displays	389
Imaging Routines	374	Controlling the Device Cursor	393
Image Display	375	References	394
Reading from the Display Device	379		

Overview

IDL provides a powerful environment for image processing and display. The routines described in this chapter provide the interface between IDL and the image display system. This chapter describes these image display and control routines and provides examples of their use.

Image Display with iTools

Beginning with IDL 6.0, you can also use the *IDL Intelligent Tools* (iTools) to display image data. The iTools provide an easy-to-use interface that allows you to manipulate your data after it has been displayed. See [“Introducing the iTools”](#) in Chapter 1 of the *iTool User’s Guide* manual for more on iTools.

Graphics Used in Examples

The examples in this chapter are all written to take advantage of IDL Direct Graphics.

Images

An image consists of a two-dimensional array of pixels. The value of each pixel represents the intensity and/or color of that position in the scene. Images of this form are known as sampled or raster images, because they consist of a discrete grid of samples. Such images come from many different sources and are a common form of representing scientific and medical data.

Imaging Routines

The following IDL routines are used for the display and manipulation of images:

TVCRS

This procedure manipulates the image device cursor. TVCRS allows the cursor to be enabled and disabled, as well as allowing it to be positioned.

TV

This procedure displays images on the image display.

TVSCL

This procedure scales the intensity values of the image into the range of the display device, then displays the result on the image display.

TVLCT

This procedure loads a new color table into the display device.

TVRD

This function reads image pixels back from the display device.

In addition, most routines used for plotting and graphics can be used with the display of images as well. These routines are described in [Chapter 17, “Direct Graphics Plotting”](#) and [Chapter 20, “Image Display Routines”](#). For example, to overlay an image and its contour plot, the output of the [CONTOUR](#) procedure is combined with that of [TV](#). The [CURSOR](#) routine, described in [“Using the CURSOR Procedure”](#) on page 291, reads the position of the interactive pointing device and may also be used to determine the location of image pixels.

Image Display

The TV and TVSCL procedures display images on the screen. These procedures use the same arguments and keywords and differ only in that TVSCL scales the image into the intensity range of the display device, while TV displays the image directly. They have the form:

```
TV, IMAGE[, POSITION]
TV, IMAGE[, X, Y[, CHANNEL]]
TVSCL, IMAGE[, POSITION]
TVSCL, IMAGE[, X, Y[, CHANNEL]]
```

where the arguments and keywords are as follows:

IMAGE

A vector or two-dimensional matrix to be displayed as an image. If not already of byte type, it is converted prior to use.

X, Y

If present, these arguments specify the lower-left coordinate of the displayed image.

POSITION

Position number of the image. Image positions are discussed in detail below.

CHANNEL

Some image display devices are capable of storing more than a single image or can combine three single color images to form a TrueColor image. CHANNEL specifies the memory channel to be written. It is assumed to be zero if not specified. This parameter is ignored on display systems that have only one memory channel.

If no optional parameters are present, IMAGE is output to the display with its lower-left corner at coordinate (0, 0). The optional parameters can be used to specify the screen position of the image in a variety of ways.

Image Orientation

The screen coordinate system for image displays puts the origin, (0, 0), at the lower-left corner of the device. The upper-right corner has the coordinate ($xsize-1$, $ysize-1$), where $xsize$ and $ysize$ are the dimensions of the visible area of the display. The descriptions of the image display routines that follow assume a display size of 512×512 , although other sizes may be used.

The system variable !ORDER controls the order in which the image is written to the screen. Images are normally output with the first row at the bottom, i.e., in bottom-to-top order, unless !ORDER is 1, in which case images are written on the screen from top to bottom. The ORDER keyword also can be specified with TV and TVSCL. It works in the same manner as !ORDER except that its effect only lasts for the duration of the single call—the default reverts to that specified by !ORDER.

An image can be displayed with any of the eight possible combinations of axis reversal and transposition by combining the display procedures with the ROTATE function.

Image Position

Image positions run from the left of the screen to the right and from the top of the screen to the bottom. If a position number is used instead of x and y , the position of the image is calculated from the dimensions of the image (using integer arithmetic) as follows:

x_{size}, y_{size} = size of display or window

x_{dim}, y_{dim} = dimensions of array

$N_x = x_{size}/x_{dim}$ = images across screen

$x = x_{dim} \text{Position}_{\text{modulo}N_x}$ = starting x

$y = y_{size} - y_{dim} (1 + \text{Position}/N_x)$ = starting y

For example, when displaying 128×128 images on a 512×512 display, the position numbers run from 0 to 15 as follows:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Image Size

Most image devices have a fixed number of display pixels. Common sizes are 512×512 , 1280×1024 , and 900×1152 (for Sun workstations). Such pixels have a fixed size which cannot be changed. For such devices, the area written on the screen is the same size as the dimensions of the image array. One-dimensional vectors are considered row vectors. The x and y parameters specify the coordinates of the lower-left corner of the area written on the display.

There are some devices, however, that have the ability to place an image with any number of pixels into an area of arbitrary size. PostScript devices are a notable

example. Such devices are said to have scalable pixels, because there is no direct connection between the number of pixels in the image and the physical space it occupies in the displayed image. When the current image device has scalable pixels, IDL sets the first bit of !D.FLAGS. The following IDL statement can be used to determine if the current device has scalable pixels:

```
SP = !D.FLAGS AND 1
```

SP will be nonzero if the device has scalable pixels. When displaying an image on a device with scalable pixels, the default uses the entire display surface for the image. The XSIZE and YSIZE keywords can be used to override this default and specify the width and height that should be used.

The XSIZE and YSIZE keywords also should be used when positioning images with the POSITION argument to TV or TVSCL. POSITION normally uses the size of the image in pixels to determine the placement of the image, but this is not possible for devices with scalable pixels. Instead, the default for such devices is to assume a single position that fills the entire available display surface. However, if XSIZE and YSIZE are specified, POSITION will use them to determine image placement.

Examples

```
; Set all display memory to 100:
TV, REPLICATE(100B, 512, 512)
; Define a 50 column by 100 row array:
ABC = BYTARR(50, 100)
; Display array ABC starting at location x = 300, y=400.
; Display pixels in columns 300 to 349, and
; rows 400 to 499 are zeroed.
TV, ABC, 300, 400
; Display image divided by 2 at position number 12:
TV, ABC/2, 12
; Output image to memory channel 2, lower-left
; corner at (256, 256).
TV, A, 256, 256, 2
; Assume file one contains a sequence of 64 x 64 byte arrays:
AA = ASSOC(1, BYTARR(64, 64))
; Display 64 images from file, from left to right and
; top to bottom, filling a 512 x 512 area:
FOR I = 0, 63 DO TV, AA[I], I
```

Image Scaling

An image can be contrast enhanced so any subrange of pixel values are scaled to fill the entire range of displayed brightnesses using a variety of methods.

For example, if the image A contains an object superimposed on a varying background and the pixel values in the object range from a value of S to the brightest value in the image the IDL statement:

```
TVSCL, A > S
```

will use the entire range of display brightnesses to display the object. The expression $A > S$ results in an image in which each pixel in A less than S is set to S . Thus, S becomes the new minimum intensity. The TVSCL procedure scales the new image into the available number of color-table entries before loading it into the display. Again, the image A is not changed.

Another method that is more efficient, although slightly obscure, is to use the BYTSCL function to scale the array as follows:

```
TV, BYTSCL(A, MIN = S, TOP = !D.TABLE_SIZE)
```

This method is more efficient because the value S is known and avoids scanning the array for the minimum and maximum values. Also, one less array operation is required.

If the object in A has values from 2.6 to 9.4, the statements

```
;Slow method.
TVSCL, A > 2.6 < 9.4
;Faster method.
TV, BYTSCL(A, MIN=2.6, MAX=9.4, TOP = !D.TABLE_SIZE)
```

will truncate the image so 2.6 is the new minimum and 9.4 is the new maximum before scaling and display.

Some examples of using the TVSCL function follow.

```
;Display square root of image:
TVSCL, SQRT(A)
;Display unsharp masked image:
TVSCL, A - SMOOTH(A, 3)
;Display scaled sum at position number 12:
TVSCL, A + B, 12
```

Reading from the Display Device

The TVRD function reads the contents of the display device memory back into an IDL variable. One use for this capability is to build up a complex display using many IDL statements, and then read the resulting image back as a single unit for storage in a file.

The TVRD function returns the contents of the specified rectangular portion of the display subsystem's memory. The coordinate (x_0, y_0) is the starting coordinate of the data to be read, and N_x, N_y is the size of the rectangle in columns and rows. This results in a byte array of dimensions $N_x \times N_y$.

A Note on Reading Data from Windows

On some systems, when backing store is provided by the window system (RETAIN=1), reading data from a window using TVRD() may cause unexpected results. For example, data may be improperly read from the window even when the image displayed on screen is correct. Having IDL provide the backing store (RETAIN=2) ensures that the window contents will be read properly.

The TVRD function has the form:

```
TVRD([X0, [Y0, [NX, [NY, Channel]]]])
```

where the arguments are described as follows.

X₀

Specifies the starting column of data to read.

Y₀

Specifies the starting row of data to read.

N_X

The number of columns to read.

N_Y

The number of rows to read.

Channel

The memory channel to be read. It is assumed to be zero if not specified. This parameter is ignored on display systems that only have one memory channel.

If the system variable !ORDER is set to zero, then data are read from the bottom up; otherwise, data are read in the top-down direction.

Example

The following statement inverts the 100×100 area of the display starting at (200, 300):

```
;Reverse area:  
TV, NOT TVRD(200, 300, 100, 100)
```

Ability to Read from Display

Not all image devices are able to support reading pixels back from device memory. If the current device has this ability, IDL sets the eighth bit of !D.FLAGS.

```
; Determine if the current device allows reading  
; from display memory:  
TEST = !D.FLAGS AND 128
```

TEST will be nonzero if the device allows such operations.

Color Tables

There are numerous systems for the measuring and specification of color. Most systems are three-dimensional in nature. For a complete discussion of color systems, refer to Foley and Van Dam (1982, Chapter 17). Parts of this discussion are taken from that chapter.

Most devices capable of displaying color use the RGB (red, green, and blue) color system. Other common color systems include the Munsell, HSV (hue, saturation, and value), HLS (hue, lightness, and saturation), and CMY (cyan, magenta, and yellow) color systems. Algorithms exist to convert colors from one system to another. IDL accepts color specifications in the RGB, HLS, or HSV color systems.

The RGB color system, as implemented in IDL, uses a three-dimensional Cartesian coordinate system with the value of each color ranging from 0 to 255. Each displayable color is a point within this cube, shown in [Figure 20-1](#) (after Foley and Van Dam). The origin, (0, 0, 0), where each color coordinate is 0, is black. The point at (255, 255, 255) is white and represents an additive mixture of the full intensity of each of the three colors. Points along the main diagonal—where the intensities of each of the three primary colors are equal—are shades of gray. The color yellow is represented by the coordinate (255, 255, 0), or a mixture of 100% red, plus 100% green, and no blue.

Typically, digital display devices represent each component of an RGB color coordinate as an n -bit integer in the range of 0 to $2^n - 1$. Each displayable color is an RGB coordinate triple of n -bit numbers yielding a palette containing 2^{3n} total colors. Therefore, for 8-bit colors, each color coordinate can range from 0 to 255, and the total palette contains 2^{24} or 16,777,216 colors.

A display with an m -bit pixel can represent 2^m colors simultaneously, given enough pixels. In the case of 8-bit colors, 24-bit pixels are required to represent all colors. The more common case is a display with 8 bits per pixel which allows the display of $2^8 = 256$ colors selected from the much larger palette.

If there are not enough bits in a pixel to represent all colors, $m < 2^{3n}$, a color translation table is used to associate the value of a pixel with a color triple. This table is an array of color triples with an element for each possible pixel value. Given 8-bit pixels, a color table containing $2^8 = 256$ elements is required. The color table element with an index of i specifies the color for pixels with a value of i .

To summarize, given a display with an n -bit color representation and an m -bit pixel, the color translation table, C , is a 2^m long array of RGB triples:

$$C_i = \{r_i, g_i, b_i\}, \quad 0 \leq i < 2^m$$

$$0 \leq r_i, g_i, b_i < 2^n$$

Objects containing a value, or color index, of i are displayed with a color of C_i .

The IDL `COLOR_CONVERT` procedure can be used to convert color triples to and from the RGB color system and the HLS and HSV systems.

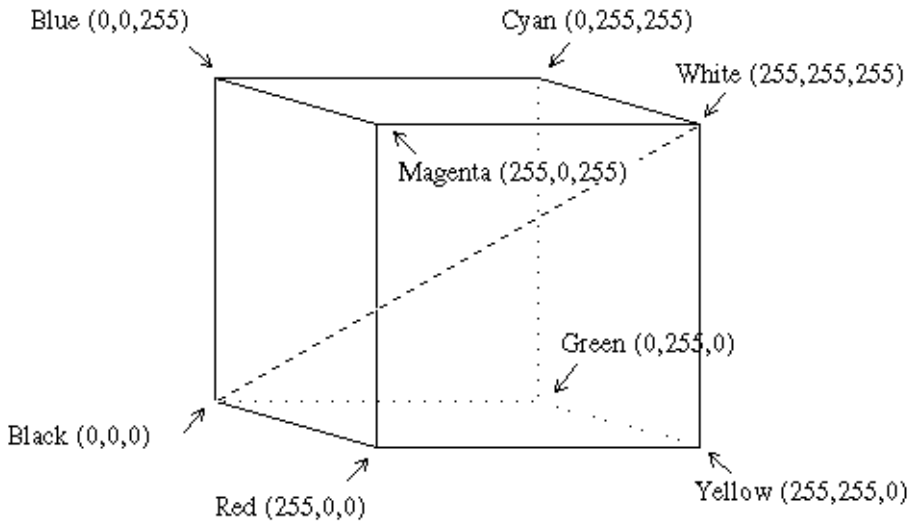


Figure 20-1: RGB Color Cube. (Note: grays are on the main diagonal.)

You can display TrueColor images on pseudo-color displays by using the `COLOR_QUAN` function. This function creates a pseudo-color palette for displaying the TrueColor image and then maps the TrueColor image to the new palette. See [COLOR_QUAN](#) in the *IDL Reference Guide* for more information.

Loading Color Tables

IDL maintains its own internal color table which is read and written by the `TVLCT` procedure. When this table is modified, it is loaded into the currently selected graphics output device. A call to this procedure has the form:

```
TVLCT, V1, V2, V3 [, Start]
```

where the arguments and keywords are as follows:

V_1 , V_2 , and V_3

The vectors containing the intensity or value of each color for each index in the RGB, HLS, or HSV color systems. Standard devices have an 8-bit color representation so the color values should range from 0 to 255. These vectors can contain up to 2^m elements (usually 256), assuming the display contains m bit pixels.

Start

The starting index in the color translation table into which V_1 , V_2 , and V_3 are loaded. If not specified, a value of 0 is used, causing the tables to be loaded starting at the first element of the translation vectors. The *Start* argument can be used to change only part of the color table.

In addition, the following keyword parameters can also be present:

GET

Returns the RGB values from the internal color table into the three variables.

HLS

Indicates that the parameters specify color using the HLS color system. The plain argument parameters are in the order H-L-S. Hue is expressed in degrees, and the lightness and saturation range from 0 to 1.

HSV

Indicates that the parameters specify color using the HSV color system. The plain argument parameters are in the order H-S-V. As above, hue is in degrees, and the saturation and value range from 0 to 1.

Example

This example creates a graph with the axes drawn in white, then successively adds red, green, blue, and yellow lines. As there are five distinct colors, plus one color for the background, a six-element color table is created. Usually, color index 0 represents black (0, 0, 0). We arbitrarily choose color index 1 to be white (1, 1, 1), 2 as red (1, 0, 0), 3 as green (0, 1, 0), 4 as blue (0, 0, 1), and 5 as yellow (1, 1, 0). The display must have at least 3 bits per pixel to represent six colors simultaneously, and an 8-bit color table is assumed.

```

;Specify the red component of each color:
RED = [0, 1, 1, 0, 0, 1]
;Specify the green component of each color:
GREEN = [0, 1, 0, 1, 0, 1]
;Specify the blue component of each color:
BLUE = [0, 1, 0, 0, 1, 0]
;Load the first six elements of the color table:
TVLCT, 255 * RED, 255 * GREEN, 255 * BLUE
;Draw the axes in white, color index 1:
PLOT, COLOR = 1, /NODATA,...
;Draw in red:
OPLOT, COLOR = 2, ...
;Draw in green:
OPLOT, COLOR = 3, ...
;Draw in blue.
OPLOT, COLOR = 4, ...
;Draw in yellow:
OPLOT, COLOR = 5, ... ...

```

The `INDGEN` function is handy when creating larger color tables in which each color's intensity can be expressed as a function of its index:

```

; Straight line, A[I] = I:
A = INDGEN(256)
; Display image with a linear red scale, disable green and blue:
TVLCT, A, A * 0, A * 0
; Display with linear black and white scale:
TVLCT, A, A, A
; Warm body temperature scale. Red is linear,
; green starts at 128, and blue starts at 192:
TVLCT, A, 2 * (A - 128) > 64, 4 * (A - 192) > 0

```

Color Table Procedures

The following IDL procedures are used to manipulate color tables:

LOADCT

Load predefined color tables. `LOADCT` has one parameter: the index of the predefined color table to be loaded. There are 40 pre-defined color tables in the file

colors1.tbl, which is supplied with IDL. To obtain a menu listing the available color tables, call `LOADCT` with no parameters. Standard tables are listed below.

Number	Name	Number	Name
0	Black & White Linear	21	Hue Sat Value 1
1	Blue/White Linear	22	Hue Sat Value 2
2	Green-Red-Blue-White	23	Purple-Red + Stripes
3	Red Temperature	24	Beach
4	Blue-Green-Red-Yellow	25	Mac Style
5	Standard Gamma-II	26	Eos A
6	Prism	27	Eos B
7	Red-Purple	28	Hardcandy
8	Green/White Linear	29	Nature
9	Green/White Exponential	30	Ocean
10	Green-Pink	31	Peppermint
11	Blue-Red	32	Plasma
12	16 Level	33	Blue-Red 2
13	Rainbow	34	Rainbow 2
14	Steps	35	Blue Waves
15	Stern Special	36	Volcano
16	Haze	37	Waves
17	Blue-Pastel-Red	38	Rainbow18
18	Pastels	39	Rainbow + white
19	Hue Sat Lightness 1	40	Rainbow + black
20	Hue Sat Lightness 2		

Table 20-1: Predefined Color Tables

XLOADCT

This procedure provides a widget interface to `LOADCT`. Pre-defined color tables can be loaded and manipulated using this tool. Tables can be stretched and Gamma corrected interactively using this procedure.

XPALETTE

This widget procedure allows you to create your own color tables using a set of three sliders. This procedure can interpolate the space between color indices (to create smooth color transitions) or edit individual colors.

MODIFYCT

Saves color tables for later use by `LOADCT`.

HSV

Makes and loads color tables based on the HSV color system. A spiral through the single-ended HSV cone is traced. The color representation of pixel values is linearly interpolated from beginning and ending values of hue, saturation, and value.

HLS

Makes and loads color tables based on the HLS color system which is based on the Otswald color system. As with the HSV procedure, spirals are interpolated in the three-dimensional color space.

PSEUDO

Generates and loads a pseudo-color table based on the LHB (lightness, hue, and brightness) system.

STRETCH

Linearly expands the entire range of the last color table loaded to cover a given range of pixel values. `STRETCH` has two parameters: the pixel value to be displayed with color index 0 and the pixel value to be displayed with the maximum color index:

```
STRETCH, LOW, HIGH
```

Example

```
; Expand the color tables so that pixels in  
; the range of 100 to 150 fill the entire color range:  
STRETCH, 100, 150
```

To revert to a normal color table, call `STRETCH` with no parameters.

Note

The window-oriented procedures will not work without a window system.

Obtaining the Color Tables

All of the IDL color-table procedures maintain the current color table in a common block called `COLORS`, defined as follows:

```
COMMON COLORS, R_orig, G_orig, B_orig, R_curr, G_curr, B_curr
```

The variables are integer vectors of length equal to the number of color indices. Your program can access these variables by defining the common block. The convention is that routines that modify the current color table should read it from `R_orig`, `G_orig`, and `B_orig`, then load the color table using `TVLCT` and leave the resulting color table in `R_curr`, `G_curr`, and `B_curr`.

Color Tables—Switching Between Devices

Use the `SET_PLOT` procedure to direct the graphics output to different devices. Because devices have differing capabilities and not all are capable of representing the same number of colors, the treatment of color tables when switching devices is somewhat tricky.

After selecting a new graphics output device, `SET_PLOT` will perform one of the following color-table actions depending upon which keyword parameters are specified:

- The default is to do nothing. The problem with this treatment is that the internal color tables incorrectly reflect the state of the device's color tables until `TVLCT` is called (usually via `LOADCT`).
- If the `COPY` keyword parameter is set, the internal color tables are copied into the device. This is straightforward if both devices have the same number of color indices. If the new device has more colors than the old device, some color indices will be invalid. If the new device has less colors than the old, not all the colors are saved. This is the preferred method if you are displaying graphics and each color index is explicitly loaded.

- When the INTERPOLATE keyword is set, the new device's table is loaded by interpolating the old color table to span the new number of color indices. This method works best when displaying images with continuous color ranges.

TrueColor Displays

IDL supports the use of some TrueColor displays with 24 bits per pixel. TrueColor displays have multiple channels. That is, they store information about each primary color component (red, green, and blue) of a pixel separately. A TrueColor display with n bits per memory channel can display 2^{3n} simultaneous colors, as opposed to the 2^n simultaneous colors available with a normal indexed (pseudo) color display. Images can be transferred to and from each individual memory channel, or to all channels simultaneously.

The X Window visuals TrueColor and DirectColor are among the TrueColor devices supported by IDL.

Configuration

The TrueColor display is configured as a single display with three channels:

Channel Number	Output
0	All colors
1	Red
2	Green
3	Blue

Table 20-2: TrueColor Display Channels

Lookup Tables

Warning

Not all TrueColor display systems have writable color lookup tables.

Each output channel, red, green, or blue, is routed through its own 8-bit deep, 256 element lookup table. The lookup tables can be used to compensate for color inaccuracies generated by the display hardware or present in the acquisition process. Initially, each lookup table is loaded with a linear ramp, mapping its input directly to its output.

As the TrueColor lookup tables are of the same size and number of elements as those on a pseudo-color display, operation of the TVLCT procedure, which loads the lookup tables, is unchanged.

Furthermore, if the same image is loaded into each channel, operation of the display mimics that of a standard 8-bit deep pseudo-color display. Most, but not all, IDL image processing procedures written for a standard color display will run on a TrueColor display without modification. The routines that transfer images to the display, TV and TVSCL, write the same 8-bit data to each channel (channel 0) if no channel parameter is present. The function TVRD, which reads data from the display, returns the maximum value contained in the three-color channels for each pixel if no channel parameter is present.

Color Indices

The color index specifier can range from 0 to $2^{24}-1$. The system variable field !D.N_COLORS, which contains the number of colors, is set to 2^{24} on a TrueColor display. The system variable field, !D.TABLE_SIZE, contains the number of RGB color table elements.

The low 8 bits, bits 0 to 7, of the color index are written to the red channel; bits 8 to 15 are written to the green; and bits 16 to 23 are written to the blue. For example, a given RGB, the index is $R + 256(G + 256B)$. To create a plot with a given color (assuming linear lookup tables), use the following statement:

```
PLOT, X, Y, COLOR = R + 256L * (G + 256L * B)
```

TrueColor Images

Images can be transferred to and from the display in either 8-bit or 24-bit mode. The CHANNEL parameter specifies the source or destination channel number for 8-bit images, and the TRUE keyword indicates for 24-bit images the method of channel interleaving. If neither keyword parameter is present, the 8-bit image is written to all three-color channels, yielding the same effect as if the channel parameter is specified as 0.

For example, to transfer three 8-bit images contained in the arrays R, G, and B to their respective channels, use the following statements:

```
;Load red in channel 1:
TV, R, 0, 0, 1
;Load green in channel 2:
TV, G, 0, 0, 2
;Load blue in channel 3:
TV, B, 0, 0, 3
```

The position parameters (0, 0 above) can be altered to write to any location in the window.

For 24-bit images, the RGB data can be interleaved by pixel, by line, or by image. Use the TRUE parameter to specify the method of interleaving. A c column by l line TrueColor image is dimensioned as follows:

TRUE Value	Dimensions	Interleaving
1	$(3, c, l)$	Pixel
2	$(c, 3, l)$	Line
3	$(c, l, 3)$	Image

Table 20-3: Values for the TRUE Keyword

For example, to write a TrueColor, line interleaved image contained in the variable t , with its lower-left corner at coordinate (100, 200), use the following statement:

```
TV, T, 100, 200, TRUE = 2
```

Reading Images

To read from the display to an IDL variable or expression, use the TVRD function with either the CHANNEL parameter or TRUE keyword parameter. The calling sequence for TVRD is:

$$\text{Result} = \text{TVRD}([X_0, Y_0, N_x, N_y, \text{Channel}])$$

where (X_0, Y_0) specifies the window coordinate of the lower-left corner of the rectangle to be read, and (N_x, N_y) contains the number of columns and rows to read. Note that all parameters to TVRD are optional. If no arguments are supplied, the entire area of the display device is returned.

When used without the TRUE parameter, TVRD returns an (N_x, N_y) byte image read from the indicated channel. If the channel number is not specified or is zero, the maximum RGB value of each pixel is returned, approximating the luminance.

If present and nonzero, the TRUE keyword indicates that a TrueColor image is to be read and specifies the index of the dimension over which color is interleaved. The result is a $(3, N_x, N_y)$ pixel interleaved array if TRUE is 1; or an $(N_x, 3, N_y)$ line interleaved array if TRUE is 2; or an $(N_x, N_y, 3)$ image interleaved array if TRUE is 3.

Some examples of TVRD follow.

```
; Read a 512 × 512 image, starting at (0, 0),  
; from the red channel into R:  
R = TVRD(0, 0, 512, 512, 1)  
; Read a TrueColor 512 × 512, line interleaved image,  
; starting at (0, 0) into T. The variable T is  
; now dimensioned (512, 3, 512):  
T = TVRD(0, 0, 512, 512, TRUE = 2)  
; Read the maximum RGB value of each pixel into L:  
L = TVRD(0, 0, 512, 512)
```


Controlling the Device Cursor

The TVCRS function manipulates the cursor of the image display. Normally, the cursor is disabled and is not visible. TVCRS with one argument allows the cursor to be enabled or disabled. While TVCRS with two parameters enables the cursor and places it on pixel location (x, y) . TVCRS has the form

```
TVCRS[ , ON_OFF]  
TVCRS[ , X, Y]
```

where the arguments and keywords are as follows:

ON_OFF

Specifies whether the cursor should be on or off. If present and nonzero, the cursor is enabled. If ON_OFF is zero or no parameters are specified, the cursor is turned off.

X

The column to which the cursor will be set.

Y

The row to which the cursor will be set.

TVCRS also takes various keywords that affect how it positions the cursor. Notably, the keywords DATA, DEVICE, and NORMAL specify the coordinate system. See the entry for [TVCRS](#) in the *IDL Reference Guide* for details.

References

Foley, J.D., and A. Van Dam (1982), *Fundamentals of Interactive Computer Graphics*, Addison-Wesley Publishing Company.



Chapter 21: Signal Processing

The following topics are covered in this chapter:

Overview	396	The Wavelet Transform	417
Digital Signals	397	Convolution	418
Signal Analysis Transforms	400	Correlation and Covariance	419
The Fourier Transform	401	Digital Filtering	420
Interpreting FFT Results	402	Finite Impulse Response (FIR) Filters ...	421
Displaying FFT Results	403	FIR Filter Implementation	425
Using Windows	409	Infinite Impulse Response Filters	427
Aliasing	413	Routines for Signal Processing	431
FFT Algorithm Details	414	References	433
The Hilbert Transform	415		

Overview

A signal, by definition, contains information. Any signal obtained from a physical process also contains noise. It is often difficult or impossible to make sense of the information contained in a digital signal by looking at it in its raw form—that is, as a sequence of real values at discrete points in time. Signal analysis transforms offer natural, meaningful, alternate representations of the information contained in a signal.

This chapter describes IDL's digital signal processing tools. Most of the procedures and functions mentioned here work in two or more dimensions. For simplicity, only one dimensional signals are used in the examples.

Running the Example Code

The examples in this chapter are all written to take advantage of IDL Direct Graphics. Examples and techniques using IDL Object Graphics are contained in the later chapters of this manual.

The example code used in this chapter is part of the IDL distribution. All of the files mentioned are located in the `examples/doc` subdirectory of the IDL distribution. By default, this directory is part of IDL's path; if you have not changed your path, you will be able to run the examples as described here. See “[!PATH](#)” in Appendix D of the *IDL Reference Guide* manual for information on IDL's path.

Digital Signals

A one-dimensional digital signal is a sequence of data, represented as a vector in an array-oriented language like IDL. The term digital actually describes two different properties:

1. The signal is defined only at discrete points in time as a result of sampling, or because the instrument which measured the signal is inherently discrete-time in nature. Usually, the time interval between measurements is constant.
2. The signal can take on only discrete values.

In this discussion, we assume that the signal is sampled at a time interval. The concepts and techniques presented here apply equally well to any type of signal—the independent variable may represent time, space, or any abstract quantity.

The following IDL commands create a simulated digital signal $u(k)$, sampled at an interval `delt`. This simulated signal will be used in examples throughout this chapter. The simulated signal contains 1024 time samples, with a sampling interval of 0.02 seconds. The signal contains a DC component and components at 2.8, 6.5, and 11.0 cycles per second.

Enter the following commands at the IDL prompt to create the simulated signal:

```
N = 1024
delt = 0.02
u = -0.3 $
    + 1.0 * SIN(2 * !PI * 2.8 * delt * FINDGEN(N)) $
    + 1.0 * SIN(2 * !PI * 6.25 * delt * FINDGEN(N)) $
    + 1.0 * SIN(2 * !PI * 11.0 * delt * FINDGEN(N))
```

Alternately, you can run the following batch file to create the signal:

```
@sigprc01
```

See [“Running the Example Code”](#) on page 396 if IDL does not find the batch file.

Because the signal is digital, the conventional way to display it is with a histogram (or step) plot. To create a histogram plot, set the PSYM keyword to the PLOT routine equal to 10. A section of the example signal $u(k)$ is plotted in the figure below.

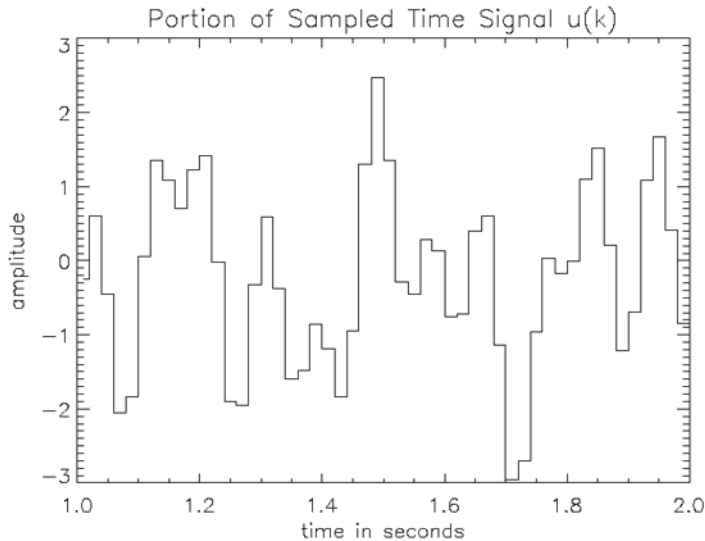


Figure 21-1: Histogram plot of sample signal $u(k)$.

Note

When the number of sampled data points is large, the steps in the histogram plot are too small to see. In such cases you should not plot in histogram mode.

Enter the following commands at the IDL prompt to create the plot:

```

; Compute time data sequence u.
@sigprc01
; Vector of discrete times:
t = delt * FINDGEN(N)
; Beginning of plot time range:
t1 = 1.0
; End of plot time range:
t2 = 2.0
PLOT, t+delt/2, u, PSYM=10, XRANGE=[t1,t2], $
    XTITLE='time in seconds', YTITLE='amplitude', $
    TITLE='Portion of Sampled Time Signal u(k)'
```

Alternately, you can run the following batch file to create the plot:

```
@sigprc02
```

See “[Running the Example Code](#)” on page 396 if IDL does not find the batch file.

Signal Analysis Transforms

Most signals can be decomposed into a sum of discrete (usually sinusoidal) signal components. The result of such decomposition is a frequency spectrum that can uniquely identify the signal. IDL provides three transforms to decompose a signal and prepare it for analysis: the Fourier transform, the Hilbert transform, and the wavelet transform.

The Fourier Transform

The Discrete Fourier Transform (DFT) is the most widely used method for determining the frequency spectra of digital signals. This is due to the development of an efficient algorithm for computing DFTs known as the Fast Fourier Transform (FFT).

The discrete Fourier transform, $v(m)$, of an N -element, one-dimensional function, $u(k)$, is defined as:

$$v(m) = \frac{1}{N} \sum_{k=0}^{N-1} u(k) \exp[-j2\pi mk/N]$$

The inverse transform is defined as:

$$u(k) = \sum_{m=0}^{N-1} v(m) \exp[j2\pi mk/N]$$

IDL implements the Fast Fourier Transform in the FFT function. You can find details on using IDL's FFT function in the following sections and in "FFT" in the *IDL Reference Guide* manual.

Interpreting FFT Results

Just as the sampled time data represents the value of a signal at discrete points in time, the result of a (forward) Fast Fourier Transform represents the spectrum of the signal at discrete frequencies. These discrete frequencies are a function of the frequency index (m), the number of samples collected (N), and the sampling interval (δ):

$$f(m) = \frac{m}{N\delta}$$

The frequencies for which the FFT of a sampled signal are defined are sometimes called frequency bins, which refers to the histogram-like nature of a discrete spectrum. The width of each frequency bin is $1/(N * \delta)$.

Due to the complex exponential in the definition of the DFT, the spectrum has a cyclic dependence on the frequency index m . That is:

$$v(m + pN) = v(m)$$

for $p =$ any integer.

The frequency spectrum computed by IDL's FFT function for a one-dimensional time sequence is stored in a vector with indices running from 0 to $N-1$, which is also a valid range for the frequency index m . However, the frequencies associated with frequency indices greater than $N/2$ are above the Nyquist frequency and are not physically meaningful for sampled signals. Many textbooks choose to define the range of the frequency index m to be from $-(N/2 - 1)$ to $N/2$ so that it is (nearly) centered around zero. From the cyclic relation above with $p = -1$:

$$v(-(N/2 - 1)) = v(N/2 + 1 - N) = v(N/2 + 1)$$

$$v(-(N/2 - 2)) = v(N/2 + 2 - N) = v(N/2 + 2)$$

...

$$v(-2) = v(N - 2 - N) = v(N - 2)$$

$$v(-1) = v(N - 1 - N) = v(N - 1)$$

This index shift is easily accomplished in IDL with the `SHIFT` function. See [“Real and Imaginary Components”](#) on page 403 for an example.

Displaying FFT Results

Depending on the application, there are many ways to display spectral data, the result of the (forward) FFT function.

Real and Imaginary Components

The most direct way is to plot the real and imaginary parts of the spectrum as a function of frequency index or as a function of the corresponding frequencies. The following figure displays the real and imaginary parts of the spectrum $v(m)$ of the sampled signal $u(k)$ for frequencies from $-(N/2 - 1)/(N * \delta)$ to $(N/2)/(N * \delta)$ cycles per second.

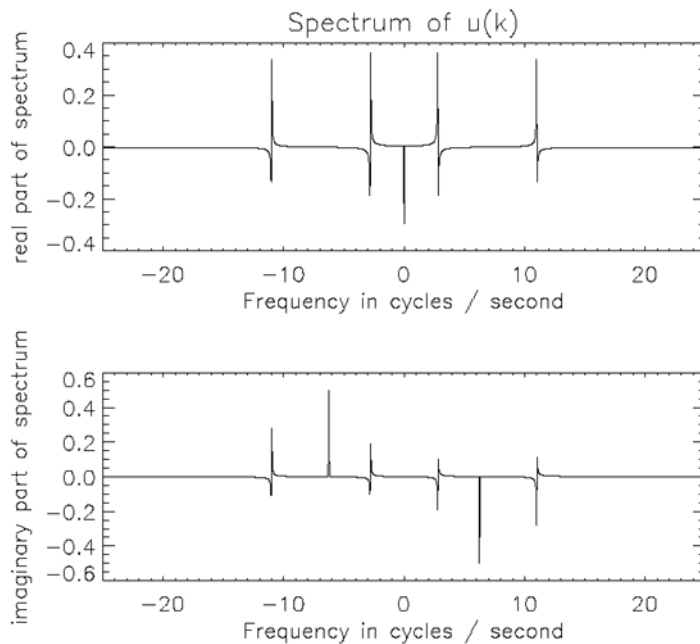


Figure 21-2: Real and Imaginary parts of the sample signal.

Enter the following commands at the IDL prompt to create the plot:

```

;Compute time sequence data:
@sigprc01
; Compute spectrum v:
V = FFT(U)
M = (INDGEN(N)-(N/2-1))
; Frequencies corresponding to m in cycles/second:
F = M / (N*delt)
; Set up for two plots in window:
!P.MULTI = [0, 1, 2]
PLOT, F, FLOAT(SHIFT(V,N/2-1)), $
  YTITLE='real part of spectrum', $
  XTITLE='Frequency in cycles / second', $
  XRANGE=[-1,1]/(2*delt), XSTYLE=1, $
  TITLE='Spectrum of u(k)'
PLOT, F, IMAGINARY(SHIFT(V,N/2-1)), $
  YTITLE='imaginary part of spectrum', $
  XTITLE='Frequency in cycles / second', $
  XRANGE=[-1,1]/(2*delt), XSTYLE=1
!P.MULTI = 0

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc03
```

See “[Running the Example Code](#)” on page 396 if IDL does not find the batch file.

IDL’s FFT function always returns a single- or double-precision complex array with the same dimensions as the input argument. In the case of a forward FFT performed on a one-dimensional vector of N real values, the result is an N -element vector of complex quantities, which takes $2N$ real values to represent. It would seem that there is twice as much information in the spectral data as there is in the time sequence data. This is not the case. For a real valued time sequence, half of the information in the frequency sequence is redundant. Specifically:

```

; 1 redundant value:
IMAGINARY(v(0)) = 0.0
; 1 redundant value:
IMAGINARY(v(N/2)) = 0.0

```

and

```

; for m=1 to N/2-1, N-2 redundant values:
v(N-m) = CONJ(v(m))

```

so that exactly N of the single- or double-precision values used to represent the frequency spectrum are redundant. This redundancy is evident in the previous figure. Notice that the real part of the spectrum is an even function (symmetric about zero),

and the imaginary part of the spectrum is an odd function (anti-symmetric about zero). This is always the case for the spectra of real-valued time sequences.

Because of the redundancy in such spectra, it is common to display only half of the spectrum of a real time sequence. That is, only the spectral values with frequency indices from 0 to $N/2$, which correspond to frequencies from 0 to $1/(2 * \delta)$, the Nyquist frequency. This vector of positive frequencies is generated in IDL with the following command:

```
; f = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]  
F = FINDGEN(N/2+1)/(N*delt)
```

Magnitude and Phase

It is also common to display the magnitude and phase of the spectrum, which have physical significance, as opposed to the real and imaginary parts of the spectrum, which do not have physical significance. Since there is a one-to-one correspondence between a complex number and its magnitude and phase, no information is lost in the transformation from a complex spectrum to its magnitude and phase. In IDL, the magnitude is easily determined with the absolute value (ABS) function, and the phase with the arc-tangent (ATAN) function. By one widely used convention, the magnitude of the spectrum is plotted in decibels (dB) and the phase is plotted in

degrees, against frequency on a logarithmic scale. The magnitude and phase of our sample signal are plotted in the figure below.

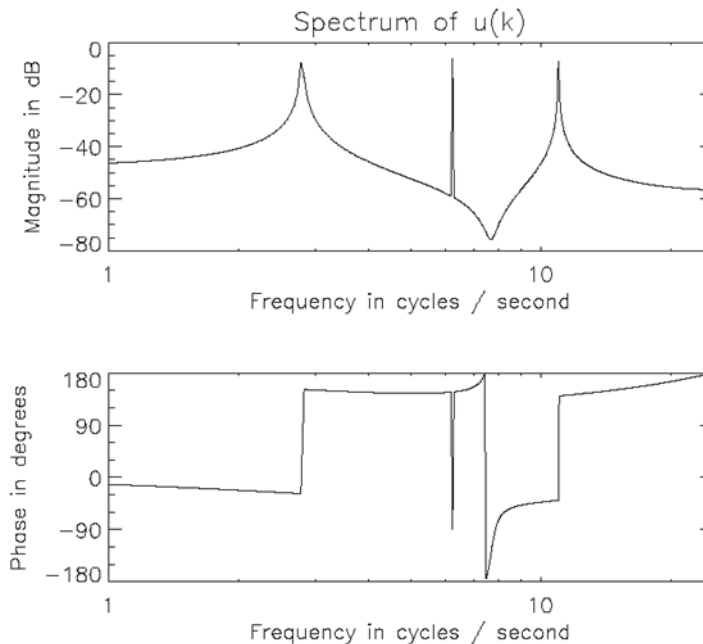


Figure 21-3: Magnitude and phase of the sample signal.

Enter the following commands at the IDL prompt to create the plot:

```

; Compute time sequence data:
@sigprc01
; Compute spectrum v:
V = FFT(U)
; F = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]:
F = FINDGEN(N/2+1) / (N*delt)
; Magnitude of first half of v:
mag = ABS(V(0:N/2))
; Phase of first half of v:
phi = ATAN(V(0:N/2), /PHASE)
; Set up for two plots in window:
!P.MULTI = [0, 1, 2]
; Create log plots of magnitude in dB and phase in degrees:
PLOT, F, 20*ALOG10(mag), YTITLE='Magnitude in dB', $
      XTITLE='Frequency in cycles / second', /XLOG, $
      XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1, $

```

```

    TITLE='Spectrum of u(k)'
    PLOT, F, phi!/DTOR, YTITLE='Phase in degrees', $
        YRANGE=[-180,180], YSTYLE=1, YTICKS=4, YMINOR=3, $
        XTITLE='Frequency in cycles / second', /XLOG, $
        XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1
    !P.MULTI = 0

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc04
```

See “[Running the Example Code](#)” on page 396 if IDL does not find the batch file.

Using a logarithmic scale for the frequency axis has the advantage of spreading out the lower frequencies, while higher frequencies are crowded together. Note that the spectrum at zero frequency (DC) is lost completely on a semi-logarithmic plot.

The previous figure shows the strong frequency components at 2.8, 6.25, and 11.0 cycles/second as peaks in the magnitude plot, and as discontinuities in the phase plot. The magnitude peak at 6.25 cycles/second is a narrow spike, as would be expected from the pure sine wave component at that frequency in the time data sequence. The peaks at 2.8 and 11.0 cycles/second are more spread out, due to an effect known as smearing or leakage. This effect is a direct result of the definition of the DFT and is not due to any inaccuracy in the FFT. Smearing is reduced by increasing the length of the time sequence, or by choosing a sample size which includes an integral number of cycles of the frequency component of interest. There are an integral number of cycles of the 6.25 cycles/second component in the time sequence used for this example, which is why the peak at that frequency is sharper.

The apparent discontinuity in the phase plot at around 7.45 cycles/second is an anomaly known as phase wrapping. It is a result of resolving the phase from the real and imaginary parts of the spectrum with the arctangent function (ATAN), which returns principal values between -180 and $+180$ degrees.

Power Spectrum

Finally, for many applications, the phase information is not useful. For these, it is often customary to plot the power spectrum, which is the square of the magnitude of the complex spectrum. The resulting plot is shown in the figure below.

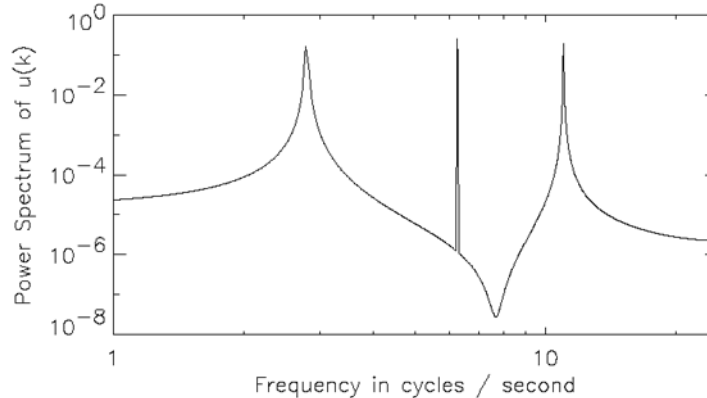


Figure 21-4: Power spectrum of the sample signal.

Enter the following commands at the IDL prompt to create the plot:

```

; Compute time sequence data.
@sigprc01
; Compute spectrum v:
V = FFT(U)
; F = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]:
F = FINDGEN(N/2+1) / (N*delt)
; Create log-log plot of power spectrum:
PLOT, F, ABS(V(0:N/2))^2, YTITLE='Power Spectrum of u(k)', $
  /YLOG, XTITLE='Frequency in cycles / second', /XLOG, $
  XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc05
```

See [“Running the Example Code”](#) on page 396 if IDL does not find the batch file.

Using Windows

The smearing or leakage effect mentioned previously is a direct consequence of the definition of the Discrete Fourier Transform and of the fact that a finite time sample of a signal often does not include an integral number of some of the frequency components in the signal. The effect of this truncation can be reduced by increasing the length of the time sequence or by employing a windowing algorithm. IDL's HANNING function computes two windows which are widely used in signal processing: the Hanning window and the Hamming window.

Hanning Window

The Hanning window is defined as:

$$w(k) = \frac{1}{2} \left(1 - \cos\left(\frac{2\pi k}{N}\right) \right)$$

The resulting vector is multiplied element-by-element with the sampled signal vector before applying the FFT. For example, the following IDL command computes the Hanning window and then applies the FFT function:

```
v_n = FFT(HANNING(N)*U)
```

The power spectrum of the Hanning windowed signal shows the mitigation of the truncation effect (see the figure below).

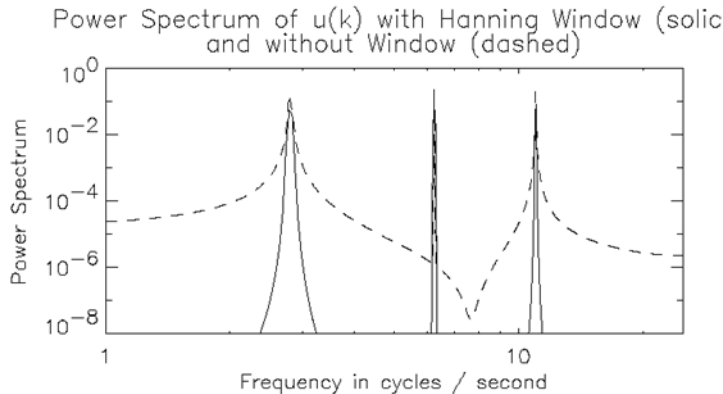


Figure 21-5: Power spectrum of the sample signal after applying a Hanning window.

Enter the following commands at the IDL prompt to create the plot:

```

; Compute time sequence data:
@sigprc01
; F = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]:
F = FINDGEN(N/2+1) / (N*delt)
v_n = FFT(HANNING(N)*U)
;Create a log-log plot of power spectrum:
PLOT, F, ABS(v_n(0:N/2))^2, YTITLE='Power Spectrum', $
  /YLOG, YRANGE=[1.0e-8,1.0], YSTYLE=1, YMARGIN=[4,4], $
  XTITLE='Frequency in cycles / second', /XLOG, $
  XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1, $
  TITLE='Power Spectrum of u(k) with Hanning Window ' $
  +'(solid)!Cand without Window (dashed) '
; Overplot without window:
O PLOT, F, ABS(FFT(U))(0:N/2))^2, LINESSTYLE=2

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc06
```

See “[Running the Example Code](#)” on page 396 if IDL does not find the batch file.

Hamming Window

The Hamming window is defined as:

$$w(k) = 0.54 - 0.46 \cos\left(\frac{2\pi k}{N}\right)$$

The resulting vector is multiplied element-by-element with the sampled signal vector before applying the FFT. For example, the following IDL command computes the Hamming window and then applies the FFT function:

```
v_m = FFT(HANNING(N, ALPHA=0.56)*U)
```

The power spectrum of the Hamming windowed signal shows the mitigation of the truncation effect (see the figure below).

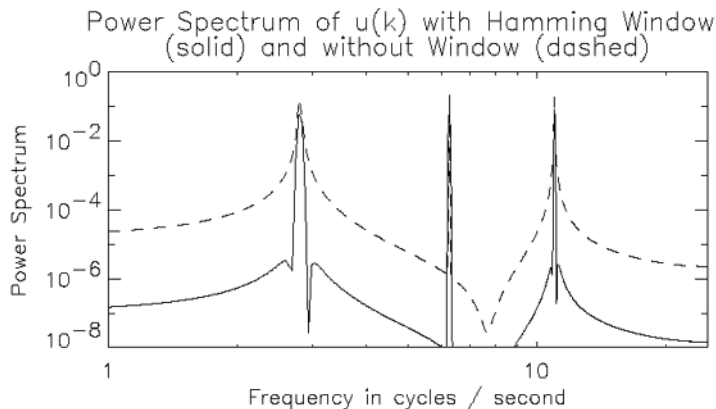


Figure 21-6: Power spectrum of the sample signal after applying a Hamming window.

Enter the following commands at the IDL prompt to create the plot:

```
; Compute time sequence data.
@sigprc01
; F = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]:
F = FINDGEN(N/2+1) / (N*delt)
v_m = FFT(HANNING(N, ALPHA=0.54)*U)
; Create log-log plot of power spectrum:
PLOT, F, ABS(v_m(0:N/2))^2, YTITLE='Power Spectrum', $
    /YLOG, YRANGE=[1.0e-8,1.0], YSTYLE=1, YMARGIN=[4,4], $
    XTITLE='Frequency in cycles / second', $
```

```
  /XLOG, XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1, $  
  TITLE='Power Spectrum of u(k) with Hamming Window'  
  +'!C(solid) and without Window (dashed)'  
  ; Overplot without window:  
  OPLOT, F, ABS((FFT(U))(0:N/2))^2, LINESSTYLE=2
```

Alternately, you can run the following batch file to create the plot:

```
@sigprc07
```

See [“Running the Example Code”](#) on page 396 if IDL does not find the batch file.

Aliasing

Aliasing is a well known phenomenon in sampled data analysis. It occurs when the signal being sampled has components at frequencies higher than the Nyquist frequency, which is equal to half the sampling frequency. Aliasing is a consequence of the fact that after sampling, every periodic signal at a frequency greater than the Nyquist frequency looks exactly like some other periodic signal at a frequency less than the Nyquist frequency.

For example, suppose we add a 30 cycle per second periodic component to our sampled data sequence $u(t)$. The power spectrum of the augmented signal is shown in the figure below.

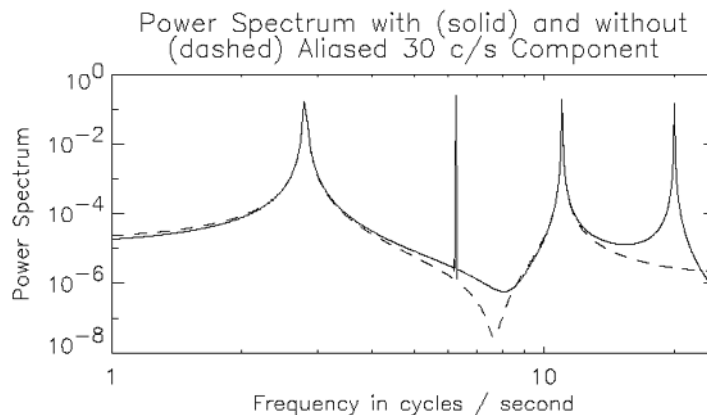


Figure 21-7: Power spectrum of the sample signal after adding a 30 cycles per second component.

Because the frequency of the new component is above the Nyquist frequency of 25 cycles per second ($25 = 1/(2 \cdot \Delta t)$), the power spectrum shows the contribution of the new component as an alias at 20 cycles per second.

To prevent aliasing, frequency components of a signal above the Nyquist frequency must be removed before sampling.

You can run the following batch file to create the plot:

```
@sigprc08
```

See [“Running the Example Code”](#) on page 396 if IDL does not find the batch file.

FFT Algorithm Details

IDL's implementation of the fast Fourier transform is based on the Cooley-Tukey algorithm. The algorithm takes advantage of the fact that the discrete Fourier transform (DFT) of a discrete time series with an even number of points is equal to the sum of two DFTs, each half the length of the original. For data lengths that are a power of 2, this algorithm is used recursively, each iteration subdividing the data into smaller sets to be transformed. In the IDL FFT, this method is also extended to powers of 3 and 5. If the number of points in the original time series does not contain powers of 2, 3, or 5, the original data are still subdivided into data sets with lengths equal to the prime factors of N . The resulting subdivisions with lengths equal to prime numbers other than 2, 3, or 5 must be transformed using a slow DFT. The slow DFT is mathematically equivalent to the FFT, but requires N^2 operations instead of $N \log_2(N)$.

This implementation means that the FFT function is fastest when the number of points is rich in powers of 2, 3, or 5. The slowest case is when the number of samples is a large prime number. In this case, a significant improvement in efficiency can be gained by padding the data set with zeros to increase the number of data points to a power of 2, 3, or 5.

For real input data of even lengths, the FFT algorithm also takes advantage of the fact that the real array can be packed into a complex array of half the length, and unpacked at the end, thus cutting the running time in half.

The Hilbert Transform

The Hilbert transform is a time-domain to time-domain transformation which shifts the phase of a signal by 90 degrees. Positive frequency components are shifted by +90 degrees, and negative frequency components are shifted by -90 degrees. Applying a Hilbert transform to a signal twice in succession shifts the phases of all of the components by 180 degrees, and so produces the negative of the original signal. IDL's HILBERT function accepts both real and complex valued signals as inputs; the imaginary part of the result is zero for real inputs.

In optics and signal analysis, the Hilbert transform of the time signal $r(t)$ is known as the quadrature function of $r(t)$, which is used to form a complex function known as the analytic signal. The analytic signal is defined as:

$$\hat{r}(t) = r(t) - jH(r(t))$$

where j is the square root of -1 and H is the Hilbert function.

The projection of the analytic signal onto the plane defined by the real axis and the time axis is the original signal. The projection onto the plane defined by the imaginary axis and the time axis is the Hilbert transform of the original signal.

The following example plots the complex analytic signal of a periodic time signal with a slowly varying amplitude.

Analytic Signal for $r(t)$ Using Hilbert Transform

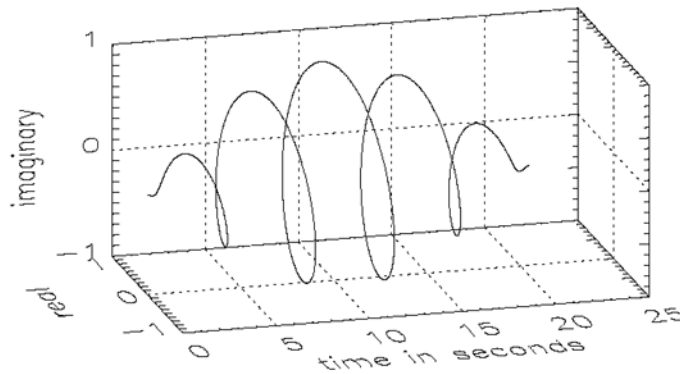


Figure 21-8: Analytic signal for $r(t)$.

Enter the following commands at the IDL prompt to create the plot:

```

; Number of time samples in data set:
N = 1024
; Sampling interval in seconds:
delt = 0.02
; Vector of discrete times:
T = delt * FINDGEN(N)
f1 = 5.0 / ((n-1)*delt)
f2 = 0.5 / ((n-1)*delt)
R = SIN(2*!PI*f1*T) * SIN(2*!PI*f2*T)
PLOT_3DBOX, T, R, -FLOAT(HILBERT(R)), $
  AX=40, AZ=15, XTICKS=5, XCHARSIZE=2, $
  XTITLE = 'time in seconds', YTICKS=2, YCHARSIZE=2, $
  YTITLE = 'real', YMARGIN=[4,8], ZTICKS=2, ZCHARSIZE=2, $
  ZTITLE = 'imaginary'
XYOUTS, 0.5, 0.95, /NORMAL, ALIGNMENT=0.5, SIZE=1.5, $
  'Analytic Signal for r(t) Using Hilbert Transform'

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc09
```

See “[Running the Example Code](#)” on page 396 if IDL does not find the batch file.

The Wavelet Transform

Like the discrete Fourier transform, the discrete wavelet transform (DWT) is a linear operation that defines a forward and inverse relationship between the time-domain and the frequency-domain, also called the wavelet domain. This relationship is expressed through the use of basis functions. In the case of the DFT, trigonometric sines and cosines of varying angles are used. In the case of the DWT, the basis functions are more complicated and usually called mother functions or wavelets. Also like the DFT, the DWT is orthogonal, making many operations computationally efficient. For example, the inverse wavelet transform, when viewed as a matrix operator, is simply the transpose of the forward transform.

Most of the usefulness of wavelets relies on the fact that wavelet transforms can usefully be severely truncated—that is, they can be effectively turned into sparse expressions. This property is a result of the simultaneous compact representation of the wavelet basis functions in the time and frequency domains. See “WTN” in the *IDL Reference Guide* manual for an example using the wavelet transform.

Convolution

Discrete convolution in digital signal processing is used (among other things) to smooth sampled signals using a weighted moving average. It also has many applications outside of signal processing.

IDL has two functions for doing discrete convolution: `BLK_CON` and `CONVOL`. `BLK_CON` takes advantage of the fact that the convolution of two signals is the Inverse Fourier transform of the product of the Fourier transforms of the two signals. `BLK_CON` is faster than `CONVOL`, but not as flexible. Among the many applications for discrete convolution is the implementation of digital filters. See the example in the [“Finite Impulse Response \(FIR\) Filters”](#) on page 421.

Correlation and Covariance

Correlation and covariance (which is correlation with any non-zero mean values of the signals removed beforehand) are closely related to convolution. They are useful in analyzing signals with random components. Autocorrelation and autocovariance of signals are computed with the `A_CORRELATE` function, and crosscorrelation and crosscovariance are computed with the `C_CORRELATE` function. See [“Time-Series Analysis”](#) on page 485 for details.

Digital Filtering

Digital filters can be implemented on a computer to remove unwanted frequency components (noise) from a sampled signal. Two broad classes of filters are Finite Impulse Response (FIR) or Moving Average (MA) filters, and Infinite Impulse Response (IIR) or AutoRegressive Moving Average (ARMA) filters. Both of these classes of filters are described below.

Finite Impulse Response (FIR) Filters

Digital filters that have an impulse response which reaches zero in a finite number of steps are (appropriately enough) called Finite Impulse Response (FIR) filters. An FIR filter can be implemented non-recursively by convolving its impulse response (which is often used to define an FIR filter) with the time data sequence it is filtering. FIR filters are somewhat simpler than Infinite Impulse Response (IIR) filters, which contain one or more feedback terms and must be implemented with difference equations or some other recursive technique.

IDL's `DIGITAL_FILTER` function computes the impulse response of an FIR filter based on Kaiser's window, which in turn is based on the modified Bessel function. The Kaiser filter is "nearly optimum in the sense of having the largest energy in the mainlobe for a given peak sidelobe level" [Jackson, Leland B., *Digital Filters and Signal Processing*]. The `DIGITAL_FILTER` function constructs lowpass, highpass, bandpass, or bandstop filters.

The figure below plots a bandstop filter which suppresses frequencies between 7 cycles per second and 15 cycles per second for data sampled every 0.02 seconds.

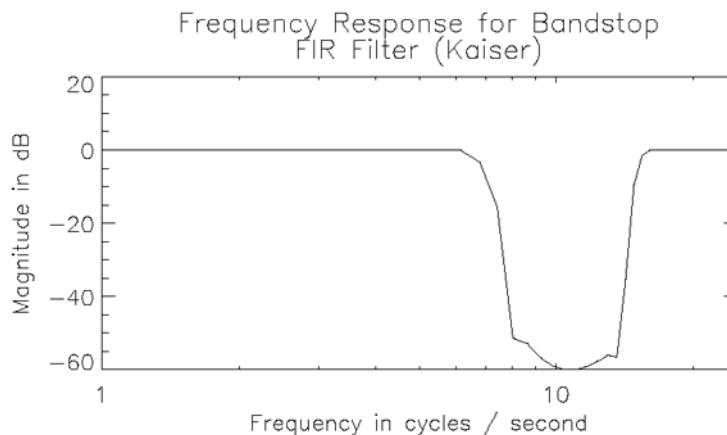


Figure 21-9: Bandstop FIR filter.

Enter the following commands at the IDL prompt to create the plot:

```

; Sampling period in seconds:
delt = 0.02
; Frequencies above f_low will be passed:
f_low = 15.
; Frequencies below f_high will be passed:
f_high = 7.
; Ripple amplitude will be less than -50 dB:
a_ripple = 50.
; The order of the filter:
nterms = 40
; Compute the impulse response = the filter coefficients:
bs_ir_k = DIGITAL_FILTER(f_low*2*delt, f_high*2*delt, $
    a_ripple, nterms)
; The frequency response of the filter is the FFT of its
; impulse response:
nfilt = N_ELEMENTS(bs_ir_k)
; where nfilt = number of points in impulse response.
; Scale frequency response by number of points:
bs_fr_k = FFT(bs_ir_k) * nfilt
; Create a log plot of magnitude in decibels:
f_filt = FINDGEN(nfilt/2+1) / (nfilt*delt)
; Magnitude of bandstop filter transfer function:
mag = ABS(bs_fr_k(0:nfilt/2))
PLOT, f_filt, 20*ALOG10(mag), YTITLE='Magnitude in dB', $
    XTITLE='Frequency in cycles / second', /XLOG, $
    XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1, $
    TITLE='Frequency Response for Bandstop!CFIR Filter (Kaiser)'

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc10
```

See [“Running the Example Code”](#) on page 396 if IDL does not find the batch file.

Other FIR filters can be designed based on the Hanning and Hamming windows (see [“Using Windows”](#) on page 409), or any other user-defined window function. The design procedure is simple:

1. Compute the impulse response of an ideal filter using the inverse FFT
2. Apply a window to the impulse response. The modified impulse response defines the FIR filter.

The figure below shows the plot using the same sampling period and frequency cutoffs as above, and the corresponding ideal filter is constructed in the frequency domain using the Hanning window.

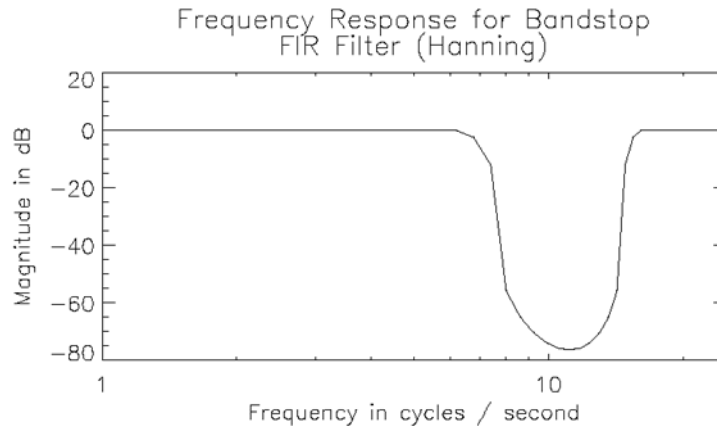


Figure 21-10: Bandstop filter using Hanning Window.

Enter the following commands at the IDL prompt to create the plot:

```

; Sampling period in seconds:
delt = 0.02
; Frequencies above f_low will be passed:
f_low = 15.
; Frequencies below f_high will be passed:
f_high = 7.
; The length of the filter:
nfilt = 81
f_filt = FINDGEN(nfilt/2+1) / (nfilt*delt)
; Pass frequencies greater than f_low and less than f_high:
ideal_fr = (f_filt GT f_low) OR (f_filt LT F_high)
; Convert from byte to floating point:
ideal_fr = FLOAT(ideal_fr)
; Replicate to obtain values for negative frequencies:
ideal_fr = [ideal_fr, REVERSE(ideal_fr[1:*])]
; Now use an inverse FFT to get the impulse response
; of the ideal filter:
ideal_ir = FLOAT(FFT(ideal_fr, /INVERSE))
; Ideal_fr is an even function, so the result is real.
; Scale by the # of points:
ideal_ir = ideal_ir / nfilt
; Shift it before applying the window:

```

```

ideal_ir = SHIFT(ideal_ir, nfilt/2)
; Apply a Hanning window to the shifted ideal impulse response.
; These are the coefficients of the filter:
bs_ir_n = ideal_ir * HANNING(nfilt)
; The frequency response of the filter is the FFT
; of its impulse response. Scale by the number of points:
bs_fr_n = FFT(bs_ir_n) * nfilt
; Create a log plot of magnitude in decibels
; Magnitude of Hanning bandstop filter transfer function:
mag = ABS(bs_fr_n(0:nfilt/2))
PLOT, f_filt, 20 * ALOG10(mag), YTITLE='Magnitude in dB', $
    XTITLE='Frequency in cycles / second', /XLOG, $
    XRANGE=[1.0, 1.0 / (2.0 * delt)], XSTYLE=1, $
    TITLE='Frequency Response for Bandstop!CFIR Filter (Hanning)'

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc11
```

See [“Running the Example Code”](#) on page 396 if IDL does not find the batch file.

FIR Filter Implementation

The simplest FIR filter to apply to a signal is the rectangular or boxcar filter, which is implemented with IDL's `SMOOTH` function, or the closely related `MEDIAN` function.

Applying other FIR filters to signals is straightforward since the filter is non-recursive. The filtered signal is simply the convolution of the impulse response of the filter with the original signal. The impulse response of the filter is computed with the `DIGITAL_FILTER` function or by the procedure in the previous section.

IDL's `BLK_CON` function provides a simple and efficient way to convolve a filter with a signal. Using $u(k)$ from the previous example and the bandstop filter created above creates the plot shown in the figure below.

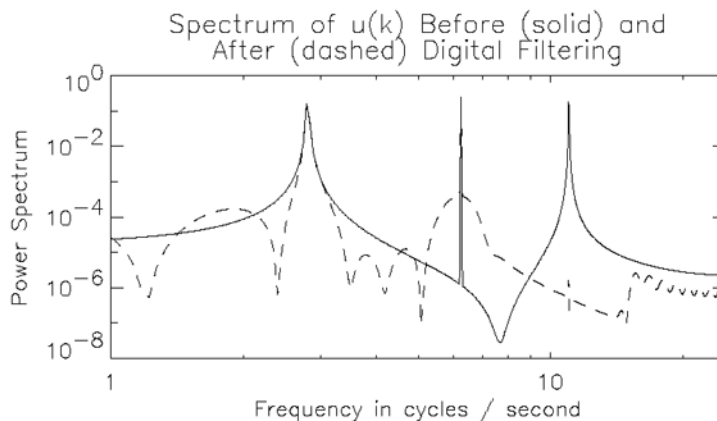


Figure 21-11: Digital signal before and after filtering.

The frequency response of the filtered signal shows that the frequency component at 11.0 cycles / second has been filtered out, while the frequency components at 2.8 and 6.25 cycles / second, as well as the DC component, have been passed by the filter.

Enter the following commands at the IDL prompt to create the plot:

```
; Compute time data sequence u:
@sigprc01
; Compute the Kaiser filter coefficients
; with the sampling period in seconds:
delt = 0.02
```

```

; Frequencies above f_low will be passed:
f_low = 15.
; Frequencies below f_high will be passed:
f_high = 7.
; Ripple amplitude will be less than -50 dB:
a_ripple = 50.
; The order of the filter:
nterms = 40
; Compute the impulse response = the filter coefficients:
bs_ir_k = DIGITAL_FILTER(f_low*2*delt, f_high*2*delt, $
    a_ripple, nterms)
; Convolve the Kaiser filter with the signal:
u_filt = BLK_CON(bs_ir_k, u)
; Spectrum of original signal:
v = FFT(u)
; Spectrum of the filtered signal:
v_filt = FFT(u_filt)
; Create a log-log plot of power spectra.
; F = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]
F = FINDGEN(N/2+1) / (N*delt)
PLOT, F, ABS(v(0:N/2))^2, YTITLE='Power Spectrum', /YLOG, $
    XTITLE='Frequency in cycles / second', /XLOG, $
    XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1, $
    TITLE='Spectrum of u(k) Before (solid) and!CAfter $
        (dashed) Digital Filtering'

```

Alternately, you can run the following batch file to create the plot:

```
@sigprc12
```

See “[Running the Example Code](#)” on page 396 if IDL does not find the batch file.

Infinite Impulse Response Filters

Digital filters which must be implemented recursively are called Infinite Impulse Response (IIR) filters because, theoretically, the response of these filters to an impulse never settles to zero. In practice, the impulse response of many IIR filters approaches zero asymptotically, and may actually reach zero in a finite number of samples due to the finite word length of digital computers.

One method of designing digital filters starts with the Laplace transform representation of an analog filter with the required frequency response. For example, the Laplace transform representation (or continuous transfer function) of a second order notch filter with the notch at f_0 cycles per second is:

$$\frac{y(s)}{u(s)} = \frac{\left(\frac{f_0}{2\pi} + s^2\right)}{\left(1 + 2s\left(\frac{f_0}{2\pi}\right) + s^2\right)}$$

where s is the Laplace transform variable. Then the continuous transfer function is converted to the equivalent discrete transfer function using one of several techniques. One of these is the bilinear (Tustin) transform, where

$$(2/\delta) * (z-1) / (z+1)$$

is substituted for the Laplace transform variable s . In this expression, z is the unit delay operator.

For the notch filter above, the bilinear transformation yields the following discrete transfer function:

$$\frac{y(z)}{u(z)} = \frac{\left(\frac{1+c^2}{2} - 2cz + \frac{1+c^2}{2}z^2\right)}{(c^2 - 2cz + z^2)}$$

where $c = (1 - \pi * f_0 * \delta) / (1 + \pi * f_0 * \delta)$.

Enter the following IDL statements to compute the coefficients of the discrete transfer function:

```

delt = 0.02
; Notch frequency in cycles per second:
f0 = 6.5
c = (1.0-!PI*F0*delt) / (1.0+!PI*F0*delt)
b = [(1+c^2)/2, -2*c, (1+c^2)/2]
a = [ c^2, -2*c, 1]

```

Alternately, you can run the following batch file to compute the coefficients:

```
@sigprc13
```

See “[Running the Example Code](#)” on page 396 if IDL does not find the batch file.

IIR Filter Implementation

Since an Infinite Impulse Response filter contains feedback loops, its output at every time step depends on previous outputs, and the filter must be implemented recursively with difference equations. The discrete transfer function

$$y(z) = \left(\frac{b_0 + b_1 z + \dots + b_{nb} z^{nb}}{a_0 + a_1 z + \dots + a_{na} z^{na}} \right) u(z)$$

is implemented with the difference equation

$$y(k) = \frac{(b_0 u(k-nb) + b_1 u(k-nb+1) + \dots + b_{nb} u(k) - a_0 y(k-na) - a_1 y(k-na+1) - \dots - a_{na-1} y(k-1))}{a_{na}}$$

An IIR filter is stable if the absolute values of the roots of the denominator of the discrete transfer function $a(z)$ are all less than one. The impulse response of a stable IIR filter approaches zero as the time index k approaches infinity. The frequency response function of a stable IIR filter is the Discrete Fourier Transform of the filter’s impulse response.

The figure below plots the impulse and frequency response functions of the notch filter defined above using recursive difference equations.

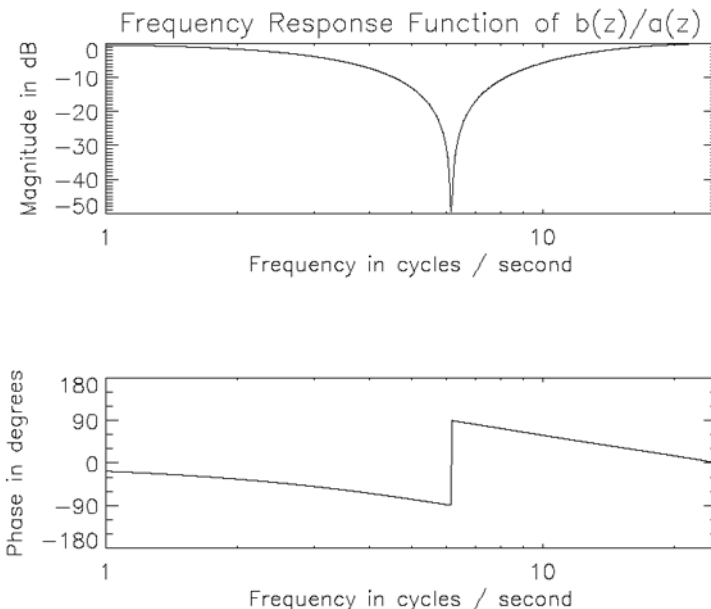


Figure 21-12: Impulse and frequency response of a notch filter.

Enter the following commands at the IDL prompt to create the plot:

```

; Load the coefficients for the notch filter:
@sigprc13
; Degree of denominator polynomial:
na = N_ELEMENTS(A)-1
; Degree of numerator polynomial:
nb = N_ELEMENTS(B)-1
N = 1024L
; Create an impulse U:
U = FLTARR(N)
U[0] = FLOAT(N)
Y = FLTARR(N)
Y[0] = B[2]*U[0] / A[na]
; Recursively compute the filtered signal:
FOR K = 1, N-1 DO $
    Y(K) = ( TOTAL( B[nb-K>0:nb]*U[K-nb>0:K] ) $
            - TOTAL( A[na-K>0:na-1]*Y[K-na>0:K-1] ) ) / A[na]
; Compute spectrum V:

```

```

V = FFT(Y)
; F = [0.0, 1.0/(N*delt), ... , 1.0/(2.0*delt)]
F = FINDGEN(N/2+1) / (N*delt)
; Magnitude of first half of V:
mag = ABS(V(0:N/2))
; Phase of first half of V:
phi = ATAN(V(0:N/2), /PHASE)
; Create log plots of magnitude in decibels and phase in degrees.
; Set up for two plots in window:
!P.MULTI = [0, 1, 2]
PLOT, F, 20*ALOG10(mag), YTITLE='Magnitude in dB', $
    XTITLE='Frequency in cycles / second', /XLOG, $
    XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1, $
    TITLE='Frequency Response Function of b(z)/a(z)'
PLOT, F, phi!/DTOR, YTITLE='Phase in degrees', $
    YRANGE=[-180,180], YSTYLE=1, YTICKS=4, YMINOR=3, $
    XTITLE='Frequency in cycles / second', /XLOG, $
    XRANGE=[1.0,1.0/(2.0*delt)], XSTYLE=1
!P.MULTI = 0

```

Note

Because the impulse response approaches zero, IDL may warn of floating-point underflow errors. This is an expected consequence of the digital implementation of an Infinite Impulse Response filter.

Alternately, you can run the following batch file to create the plot:

```
@sigprc14
```

See [“Running the Example Code”](#) on page 396 if IDL does not find the batch file.

The same code could be used to filter any input sequence $u(k)$.

Routines for Signal Processing

Below is a brief description of IDL routines for signal processing. More detailed information is available in the *IDL Reference Guide*.

Routine	Description
A_CORRELATE	Computes autocorrelation.
BLK_CON	Convolve input signal with impulse-response sequence.
C_CORRELATE	Computes cross correlation.
CONVOL	Convolve two vectors or arrays.
CORRELATE	Computes the linear Pearson correlation.
DIGITAL_FILTER	Calculates coefficients of a non-recursive, digital filter.
FFT	Returns the Fast Fourier Transform of an array.
HANNING	Creates Hanning and Hamming windows.
HILBERT	Constructs a Hilbert transform.
INTERPOL	Performs linear interpolation on vectors.
M_CORRELATE	Computes multiple correlation coefficient.
MEDIAN	Returns the median value of an array or applies a median filter.
P_CORRELATE	Computes partial correlation coefficient.
R_CORRELATE	Computes rank correlation.
SAVGOL	Returns coefficients of Savitzky-Golay smoothing filter.
SMOOTH	Smooths with a boxcar average.
TS_COEF	Computes the coefficients for autoregressive time-series.
TS_DIFF	Computes the forward differences of a time-series.
TS_FCAST	Computes future or past values of a stationary time-series.
TS_SMOOTH	Computes moving averages of a time-series.

Table 21-1: Signal Processing Routines in IDL

Routine	Description
WTN	Returns wavelet transform of the input array.

Table 21-1: Signal Processing Routines in IDL

References

- Bracewell, Ronald N., *The Fourier Transform and Its Applications*, New York: McGraw-Hill, 1978. ISBN 0-07-007013-X
- Chen, Chi-Tsong, *One-Dimensional Digital Signal Processing*, New York: Marcel Dekker, Inc., 1979. ISBN 0-8247-6877-9
- Jackson, Leland B., *Digital Filters and Signal Processing*, Boston: Kluwer Academic Publishers, 1986. ISBN 0-89838-174-6
- Mayeda, Wataru, *Digital Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1993. ISBN 0-13-211301-5
- Morgera, Salvatore D. and Krishna, Hari, *Digital Signal Processing: Applications to Communications and Algebraic Coding Theories*, Boston: Academic Press, 1989. ISBN 0-12-506995-2
- Oppenheim, Alan V. and Schaffer, Ronald W., *Discrete-time signal processing*, Englewood Cliffs, NJ: Prentice-Hall, 1989. ISBN 0-13-216292-X
- Peled, Abraham and Liu, Bede, *Digital Signal Processing*, New York: John Wiley & Sons, Inc., 1976. ISBN 0-471-01941-0
- Press, William H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5
- Proakis, John G. and Manolakis, Dimitris G., *Digital Signal Processing: Principles, Algorithms, and Applications*, New York: Macmillan Publishing Company, 1992. ISBN 0-02-396815-X
- Rabiner, Lawrence R. and Gold, Bernard, *Theory and application of digital signal processing*, Englewood Cliffs, NJ: Prentice-Hall, 1975. ISBN 0-139-14101-4
- Strang, Gilbert and Nguyen, Truong, *Wavelets and Filter Banks*, Wellesley, MA: Wellesley-Cambridge Press, 1996. ISBN 0-961-40887-1



Chapter 22: Mathematics

The following topics are covered in this chapter:

IDL's Numerical Recipes Functions	437	Integration	465
Accuracy & Floating-Point Operations . . .	438	Linear Systems	470
Arrays and Matrices	440	Nonlinear Equations	478
Correlation Analysis	446	Optimization	480
Curve and Surface Fitting	450	Sparse Arrays	482
Eigenvalues and Eigenvectors	453	Time-Series Analysis	485
Gridding and Interpolation	459	Multivariate Analysis	488
Hypothesis Testing	462	References	494

This chapter documents IDL's mathematics and statistics procedures and functions. These include Numerical Recipes™ algorithms published in *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition).

This chapter also includes introductory discussions of the following topics and an overview of the way IDL handles the particular problems involved:

- Arrays and Matrices
- Correlation Analysis
- Curve and Surface Fitting
- Eigenvalues and Eigenvectors
- Gridding and Interpolation
- Hypothesis Testing
- Integration
- Linear Systems
- Nonlinear Equations
- Optimization
- Sparse Arrays
- Time Series Analysis

References are provided at the end of each section for a more detailed description and understanding of the topic.

Research Systems, Inc. is extremely interested in the accuracy of its algorithms. Bug reports, documentation errors and suggestions for future mathematics and statistics enhancements can be sent to RSI via:

Internet: support@RSInc.com

Fax: (303) 786-9909

IDL's Numerical Recipes Functions

IDL includes a number of routines based on algorithms published in *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition). Routines derived from Numerical Recipes are noted as such in the *IDL Reference Guide* and in the IDL Online Help.

In IDL versions up to and including IDL version 3.6, mathematics functions based on Numerical Recipes algorithms required that input be in column-major format. This is no longer the case. Routines based on Numerical Recipes algorithms have been reworked and renamed, so that all IDL functions now expect input arrays to be in row-major format (composed of row vectors).

Note

To maintain compatibility with IDL programs based on earlier versions, the old routines (using the older input convention) are still available. No alterations need be made to existing code as a result of this change in IDL. We recommend that all new IDL programs take advantage of the new names and input convention.

Accuracy & Floating-Point Operations

In a computer, real numbers are represented with finite precision. While in most cases it is safe to assume that the result of an arithmetical operation done on your computer is correct, it is important to remember that this finite-precision representation leads to unavoidable errors, especially when floating-point numbers, which are digital approximations to real numbers, are involved.

To understand why floating-point numbers are inherently inaccurate, consider the following:

- Floating-point numbers must be made to fit in a space (a string of binary digits in a computer's memory register) that can only hold an integer and a scaling factor.
- Floating-point numbers are represented by strings of a limited number of bits, but represent numbers much larger or smaller than that number of digits can be made to express.

In other words, floating-point values are finite-precision approximations of infinitely precise numbers.

Roundoff Error

When working with floating-point arithmetic, it is helpful to consider the quantity known as the machine accuracy or the floating-point accuracy of your particular computer. This is the smallest number that, when added to 1.0, produces a floating-point result that is different from 1.0.

A useful way of thinking about machine accuracy is to consider it to be the fractional accuracy to which floating-point numbers are represented. In other words, the machine accuracy roughly corresponds to a change of the least significant bit of the floating-point mantissa—precisely what can happen if a number with more significant digits than fit in the floating-point mantissa is rounded to fit the space available. Generally speaking, every floating-point arithmetic operation introduces an error at least equal to the machine accuracy into the result. This error is known as roundoff error.

Roundoff errors are cumulative. Depending on the algorithm you are using, a calculation involving n arithmetic operations might have a total roundoff error between $\text{SQRT}(n)$ times the machine accuracy and n times the machine accuracy.

Note that the machine accuracy is not the same as the smallest floating-point number your computer can represent. To find these and other machine-dependent quantities for your own computer, see [MACHAR](#) in the *IDL Reference Guide*.

Truncation Error

Another type of error is also present in some numerical algorithms. Truncation error is the error introduced by the process of numerically approximating a continuous function by evaluating it at a finite number of discrete points. Often, accuracy can be increased (again at some cost of computation time) by increasing the number of discrete points evaluated.

For example, consider the process of calculating

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Obviously, the answer becomes more accurate as n approaches infinity. When performing the actual computation, however, a cutoff value must be specified for n . Increasing n reduces truncation error at the expense of computational effort.

Several IDL routines allow you to specify cutoff values in such cases (see, for example, [INT_2D](#) of the *IDL Reference Guide*). When writing your own routines in IDL, it is important to consider this trade-off between accuracy and computational time.

Routines for Mathematical Error Assessment

Below is a brief description of IDL routines for checking math error status and machine characteristics. More detailed information is available in the *IDL Reference Guide*.

CHECK_MATH	Returns and clears accumulated math error status.
FINITE	Returns True if its argument is finite.
MACHAR	Determines and returns machine-specific parameters affecting floating-point arithmetic.

Table 22-1: *Mathematical Error Assessment Routines in IDL*

Arrays and Matrices

IDL provides a variety of mechanisms for working with multidimensional data sets. Understanding these mechanisms requires a familiarity with linear algebra and the concept of a two-dimensional data set.

Note

There are two terms commonly used to refer to two-dimensional data sets: *array* and *matrix*. People who work with images tend to call two-dimensional data sets arrays, while mathematicians tend to call two-dimensional data sets matrices. The terms are interchangeable, but the different conventions assumed by people who use them may lead to confusion.

Consider a two-dimensional data set, with dimensions m and n . In a computer, the data from this data set is stored in a unidimensional set of memory addresses; what makes the data “two-dimensional” is the way the individual elements are indexed by the software that accesses the data in memory. This topic is discussed in detail in “[Columns, Rows, and Array Majority](#)” in Chapter 7 of the *Building IDL Applications* manual; if you are unsure of your understanding of the process of mapping multidimensional data into unidimensional computer memory, please read that section carefully.

There are two possible ways to depict a two-dimensional data set on paper — row by row or column by column. For example, the standard mathematical representation of an $m \times n$ data set is shown in [Figure 22-1](#), with m rows and n columns:

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \cdots & A_{0,n-1} \\ A_{1,0} & A_{1,1} & \cdots & A_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ A_{m-1,0} & A_{m-1,1} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

Figure 22-1: An $m \times n$ array represented in mathematical notation.

Here, the first dimension (m) represents the row index, and the second dimension (n) represents the column index. Thus, if the data set is represented using this notation, the term `Array[3, 2]` refers to an element that is four rows down from the top row and three columns to the right of the leftmost row. (Note that indices are zero-based.)

Figure 22-2 depicts the standard image-processing representation of the same data set, with m columns and n rows:

$$\begin{bmatrix} A_{0,0} & A_{1,0} & \cdots & A_{m-1,0} \\ A_{0,1} & A_{1,1} & \cdots & A_{m-1,1} \\ \cdots & \cdots & \cdots & \cdots \\ A_{0,n-1} & A_{1,n-1} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

Figure 22-2: An $m \times n$ array represented in image-processing notation.

Here, the first dimension (m) represents the column index, and the second dimension (n) represents the row index. Thus, if the data set is represented using this notation, the term `Array[3, 2]` refers to an element that is four columns to the right of the leftmost column and three rows down from the top row. This is the representation used by IDL.

It is important to understand that these are two views of the *same* data; all that has changed is the notational convention applied. Why is this notational convention important? Because when reading or writing data in a two-dimensional data set, performance improves if elements that are contiguous in the computer's memory are accessed consecutively. Incrementing the index of the first dimension by one shifts one "slot" in computer memory, whereas incrementing the index of the second dimension by one shifts a number of "slots" at least as large as the size of the first dimension.

Note

The terms *column-major* and *row-major* are commonly used to define which dimension of a two-dimensional array represents the column index and which represents the row index. These terms are defined and discussed in detail in "Columns, Rows, and Array Majority" in Chapter 7 of the *Building IDL Applications* manual.

Transposing Arrays

You should be aware that many numerical algorithms — especially those that are written in a row-major language such as C or C++ — assume data is indexed (row, column). Since IDL assumes data is indexed (column, row), it is important to keep

this distinction in mind. In order to work with data indexed (row, column), you can use IDL's **TRANSPOSE** function to interchange the order of the indices.

Note that it is possible for an array to be indistinguishable from its transpose. In this case the number of columns and rows are identical and there is a symmetry between the rows of the array and the columns of its transpose. Arrays satisfying this condition are said to be *symmetric*. When dealing with symmetric arrays the use of the **TRANSPOSE** function is unnecessary, since $A^T = A$.

Multiplying Arrays

IDL has two operators used to multiply arrays. To illustrate the difference between the two operators, consider the following two arrays:

```
;A 3-column by 2-row array:
A = [ [0, 1, 2],$
      [3, 4, 5] ]

;A 2-column by 3-row array:
B = [ [0, 1],$
      [2, 3],$
      [4, 5] ]
```

The # Operator

The # operator computes array elements by multiplying the columns of the first array by the rows of the second array. The resulting array has the same number of columns as the first array and the same number of rows as the second array. The second array must have the same number of columns as the first array has rows.

For example, consider the arrays defined above:

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}, B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

We obtain the elements of $A \# B$ as follows:

$$\begin{bmatrix} A_{0,0}B_{0,0} + A_{0,1}B_{1,0} & A_{1,0}B_{0,0} + A_{1,1}B_{1,0} & A_{2,0}B_{0,0} + A_{2,1}B_{1,0} \\ A_{0,0}B_{0,1} + A_{0,1}B_{1,1} & A_{1,0}B_{0,1} + A_{1,1}B_{1,1} & A_{2,0}B_{0,1} + A_{2,1}B_{1,1} \\ A_{0,0}B_{0,2} + A_{0,1}B_{1,2} & A_{1,0}B_{0,2} + A_{1,1}B_{1,2} & A_{2,0}B_{0,2} + A_{2,1}B_{1,2} \end{bmatrix}$$

Or, using the actual values from the arrays:

$$\begin{bmatrix} (0)(0) + (3)(1) & (1)(0) + (4)(1) & (2)(0) + (5)(1) \\ (0)(2) + (3)(3) & (1)(2) + (4)(3) & (2)(2) + (5)(3) \\ (0)(4) + (3)(5) & (1)(4) + (4)(5) & (2)(4) + (5)(5) \end{bmatrix}$$

Therefore, when we issue the following command:

```
PRINT, A#B
```

IDL prints:

```
   3         4         5
   9        14        19
  15        24        33
```

Tip

If one or both of the arrays are also transposed, such as `TRANSPPOSE(A) # B`, it is more efficient to use the [MATRIX_MULTIPLY](#) function, which does the transpose simultaneously with the multiplication.

Note on the Definition of Matrix Multiplication

While the definition of the IDL `#` operator may *appear* to be at odds with the standard mathematical definition of matrix multiplication — namely, that the operator multiplies each row of the first matrix by each column of the second matrix — this is a case of slightly imprecise terminology. The confusion arises from the mappings of the words “row” and “column” — which refer to elements in a two-dimensional entity called an *array* or a *matrix* — to the one-dimensional vector of values stored in computer memory. In reality, what the matrix multiplication operator does is multiply the elements of the first *dimension* of the first array/matrix by the elements of the second *dimension* of the second array/matrix. IDL’s convention is to consider the first dimension to be the column and the second dimension to be the row, whereas the standard mathematical convention considers the first dimension to be the row and the second dimension to be the column. For a more complete discussion of this topic, see “[Columns, Rows, and Array Majority](#)” in Chapter 7 of the *Building IDL Applications* manual.

The ## Operator

The `##` operator computes array elements by multiplying the rows of the first array by the columns of the second array. The resulting array has the same number of rows as

the first array and the same number of columns as the second array. The second array must have the same number of rows as the first array has columns.

For example, consider the arrays defined above:

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}, B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}$$

We obtain the elements of $A \## B$ as follows:

$$\begin{bmatrix} A_{0,0}B_{0,0} + A_{1,0}B_{0,1} + A_{2,0}B_{0,2} & A_{0,0}B_{1,0} + A_{1,0}B_{1,1} + A_{2,0}B_{1,2} \\ A_{0,1}B_{0,0} + A_{1,1}B_{0,1} + A_{2,1}B_{0,2} & A_{0,1}B_{1,0} + A_{1,1}B_{1,1} + A_{2,1}B_{1,2} \end{bmatrix}$$

Or, using the actual values from the arrays:

$$\begin{bmatrix} (0)(0) + (1)(2) + (2)(4) & (0)(1) + (1)(3) + (2)(5) \\ (3)(0) + (4)(2) + (5)(4) & (3)(1) + (4)(3) + (5)(5) \end{bmatrix}$$

Therefore, when we issue the following command:

```
PRINT, A##B
```

IDL prints:

```
10          13
28          40
```

Multiplying Vectors

When using the # and ## operators to multiply vectors, note the following:

- For $A \# B$, where A and B are vectors, IDL performs $A \# \text{TRANSPPOSE}(B)$. In this case, $C = A \# B$ is a matrix with $C_{ij} = A_i B_j$. Mathematically, this is equivalent to the *outer product*, usually denoted by $A \otimes B$.
- For $A \## B$, where A and B are vectors, IDL performs $\text{TRANSPPOSE}(A) \## B$. In this case, $C = A \## B$ is a matrix with $C_{ij} = B_i A_j$.

- To compute the *dot product*, usually denoted by $A \cdot B$, use `TRANSPPOSE(A) # B`.

Notes on the # and ## Operators

Note the following with regard to the array multiplication operators:

- The # and ## operators are order specific.
- $A \# B = B \## A$
- $A \# B = (B^T \# A^T)^T$

Routines for Multiplying Arrays

The `MATRIX_MULTIPLY` and `MATRIX_POWER` routines are also available:

- `MATRIX_MULTIPLY` calculates the value of the # operator applied to two (possibly transposed) arrays. See “[MATRIX_MULTIPLY](#)” in the *IDL Reference Guide* manual for details.
- `MATRIX_POWER` computes the product of a matrix with itself. See “[MATRIX_POWER](#)” in the *IDL Reference Guide* manual for details.

Correlation Analysis

Given two n -element sample populations, X and Y , it is possible to quantify the degree of fit to a linear model using the correlation coefficient. The correlation coefficient, r , is a scalar quantity in the interval $[-1.0, 1.0]$, and is defined as the ratio of the covariance of the sample populations to the product of their standard deviations.

$$r = \frac{\text{covariance of X and Y}}{(\text{standard deviation of X})(\text{standard deviation of Y})}$$

or

$$r = \frac{\frac{1}{N-1} \sum_{i=0}^{N-1} \left(x_i - \left[\sum_{k=0}^{N-1} \frac{x_k}{N} \right] \right) \left(y_i - \left[\sum_{k=0}^{N-1} \frac{y_k}{N} \right] \right)}{\sqrt{\frac{1}{N-1} \sum_{i=0}^{N-1} \left(x_i - \left[\sum_{k=0}^{N-1} \frac{x_k}{N} \right] \right)^2} \sqrt{\frac{1}{N-1} \sum_{i=0}^{N-1} \left(y_i - \left[\sum_{k=0}^{N-1} \frac{y_k}{N} \right] \right)^2}}$$

The correlation coefficient is a direct measure of how well two sample populations vary jointly. A value of $r = +1$ or $r = -1$ indicates a perfect fit to a positive or negative linear model, respectively. A value of r close to $+1$ or -1 indicates a high degree of correlation and a good fit to a linear model. A value of r close to 0 indicates a poor fit to a linear model.

Correlation Example

The following sample populations represent a perfect positive linear correlation.

```
X = [-8.1, 1.0, -14.3, 4.2, -10.1, 4.3, 6.3, 5.0, 15.1, -2.2]
Y = [-9.8, -0.7, -16.0, 2.5, -11.8, 2.6, 4.6, 3.3, 13.4, -3.9]
;Compute the correlation coefficient of X and Y.
PRINT, CORRELATE(X, Y)
```

IDL prints:

```
1.00000
```

The following sample populations represent a high negative linear correlation.

```
X = [ 1.8, -2.7, 0.7, -0.5, -1.3, -0.9, 0.6, -1.5, 2.5, 3.0]
Y = [-4.7, 9.8, -3.7, 2.8, 5.1, 3.9, -3.6, 5.8, -7.3, -7.4]
;Compute the correlation coefficient of X and Y:
PRINT, CORRELATE(X, Y)
```

IDL prints:

```
-0.979907
```

The following sample populations represent a poor linear correlation.

```
X = [-1.8, 0.1, -0.1, 1.9, 0.5, 1.1, 1.9, 0.3, -0.2, -1.0]
Y = [ 1.5, -1.0, -0.6, 1.1, 0.7, -0.7, 1.1, -0.1, 0.6, -0.1]
;Compute the correlation coefficient of X and Y:
PRINT, CORRELATE(X, Y)
```

IDL prints:

```
0.0322859
```

Notes on Interpreting the Correlation Coefficient

When interpreting the value of the correlation coefficient, it is important to remember the following two caveats:

1. Although a high degree of correlation (a value close to +1 or -1) indicates a good mathematical fit to a linear model, its applied interpretation may be completely nonsensical. For example, there may be a high degree of correlation between the number of scientists using IDL to study atmospheric phenomena and the consumption of alcohol in Russia, but the two events are clearly unrelated.
2. Although a correlation coefficient close to 0 indicates a poor fit to a linear model, it does not mean that there is no correlation between the two sample populations. It is possible that the relationship between X and Y is accurately described by a nonlinear model. See “[Curve and Surface Fitting](#)” on page 450 for further details on fitting data to linear and nonlinear models.

Multiple Linear Models

The fundamental principles of correlation that apply to the linear model of two sample populations may be extended to the multiple-linear model. The degree of relationship between three or more sample populations may be quantified using the multiple correlation coefficient. The degree of relationship between two sample

populations when the effects of all other sample populations are removed may be quantified using the partial correlation coefficient. Both of these coefficients are scalar quantities in the interval [0.0, 1.0]. A value of +1 indicates a perfect linear relationship between populations. A value close to +1 indicates a high degree of linear relationship between populations; whereas a value close to 0 indicates a poor linear relationship between populations. (Although a value of 0 indicates no linear relationship between populations, remember that there may be a nonlinear relationship.)

Partial Correlation Example

Define the independent (X) and dependent (Y) data.

```
X = [[0.477121, 2.0, 13.0], $
      [0.477121, 5.0, 6.0], $
      [0.301030, 5.0, 9.0], $
      [0.000000, 7.0, 5.5], $
      [0.602060, 3.0, 7.0], $
      [0.698970, 2.0, 9.5], $
      [0.301030, 2.0, 17.0], $
      [0.477121, 5.0, 12.5], $
      [0.698970, 2.0, 13.5], $
      [0.000000, 3.0, 12.5], $
      [0.602060, 4.0, 13.0], $
      [0.301030, 6.0, 7.5], $
      [0.301030, 2.0, 7.5], $
      [0.698970, 3.0, 12.0], $
      [0.000000, 4.0, 14.0], $
      [0.698970, 6.0, 11.5], $
      [0.301030, 2.0, 15.0], $
      [0.602060, 6.0, 8.5], $
      [0.477121, 7.0, 14.5], $
      [0.000000, 5.0, 9.5]]
Y = [97.682, 98.424, 101.435, 102.266, 97.067, 97.397, $
      99.481, 99.613, 96.901, 100.152, 98.797, 100.796, $
      98.750, 97.991, 100.007, 98.615, 100.225, 98.388, $
      98.937, 100.617]
```

Compute the multiple correlation of Y on the first column of X . The result should be 0.798816.

```
PRINT, M_CORRELATE(X[0,*], Y)
```

IDL prints:

```
0.798816
```

Compute the multiple correlation of Y on the first two columns of X . The result should be 0.875872.


```
PRINT, M_CORRELATE(X[0:1,*], Y)
```

IDL prints:

```
0.875872
```

Compute the multiple correlation of Y on all columns of X . The result should be 0.877197.

```
PRINT, M_CORRELATE(X, Y)
```

IDL prints:

```
0.877197
;Define the five sample populations.
X0 = [30, 26, 28, 33, 35, 29]
X1 = [0.29, 0.33, 0.34, 0.30, 0.30, 0.35]
X2 = [65, 60, 65, 70, 70, 60]
X3 = [2700, 2850, 2800, 3100, 2750, 3050]
Y = [37, 33, 32, 37, 36, 33]
```

Compute the partial correlation of $X1$ and Y with the effects of $X0$, $X2$ and $X3$ removed.

```
PRINT, P_CORRELATE(X1, Y, REFORM([X0,X2,X3], 3, N_ELEMENTS(X1)))
```

IDL prints:

```
0.996017
```

Routines for Computing Correlations

Below is a brief description of IDL routines for computing correlations. More detailed information is available in the *IDL Reference Guide*.

A_CORRELATE	Computes autocorrelation.
C_CORRELATE	Computes cross correlation.
CORRELATE	Computes the linear Pearson correlation.
M_CORRELATE	Computes multiple correlation coefficient.
P_CORRELATE	Computes partial correlation coefficient.
R_CORRELATE	Computes rank correlation.

Table 22-2: Correlation Routines in IDL

Curve and Surface Fitting

The problem of curve fitting may be stated as follows:

Given a tabulated set of data values $\{x_i, y_i\}$ and the general form of a mathematical model (a function $f(x)$ with unspecified parameters), determine the parameters of the model that minimize an error criterion. The problem of surface fitting involves tabulated data of the form $\{x_i, y_i, z_i\}$ and a function $f(x, y)$ of two spatial dimensions.

For example, we can use the **CURVEFIT** routine to determine the parameters A and B of a user-supplied function $f(x)$, such that the sums of the squares of the residuals between the tabulated data $\{x_i, y_i\}$ and function are minimized. We will use the following function and data:

$$f(x) = a(1 - e^{-bx})$$

$$x_i = [0.25, 0.75, 1.25, 1.75, 2.25]$$

$$y_i = [0.28, 0.57, 0.68, 0.74, 0.79]$$

First we must provide a procedure written in IDL to evaluate the function, f , and its partial derivatives with respect to the parameters a_0 and a_1 :

```

PRO funct, X, A, F, PDER
  F = A[0] * (1.0 - EXP(-A[1] * X))
  ; If the function is called with four parameters,
  ; calculate the partial derivatives:
  IF N_PARAMS() GE 4 THEN BEGIN
    ; PDER's column dimension is equal to the number of
    ; elements in xi and its row dimension is equal to
    ; the number of parameters in the function F:
    pder = FLTARR(N_ELEMENTS(X), 2)
    ; Compute the partial derivatives with respect to
    ; a0 and place in the first row of PDER:
    pder[* , 0] = 1.0 - EXP(-A[1] * X)
    ; Compute the partial derivatives with respect to
    ; a1 and place in the second row of PDER:
    pder[* , 1] = A[0] * x * EXP(-A[1] * X)
  ENDIF
END

```

Note

The function will not calculate the partial derivatives unless it is called with four parameters. This allows the calling routine (in this case **CURVEFIT**) to avoid the extra computation in cases when the partial derivatives are not needed.

Next, we can use the following IDL commands to find the function's parameters:

```

;Define the vectors of tabulated:
X = [0.25, 0.75, 1.25, 1.75, 2.25]
;data values:
Y = [0.28, 0.57, 0.68, 0.74, 0.79]
;Define a vector of weights:
W = 1.0 / Y
;Provide an initial guess of the function's parameters:
A = [1.0, 1.0]
;Compute the parameters a0 and a1:
yfit = CURVEFIT(X, Y, W, A, SIGMA_A, FUNCTION_NAME = 'funct')
;Print the parameters, which are returned in A:
PRINT, A

```

IDL prints:

```
0.787386  1.71602
```

Thus the nonlinear function that best fits the data is:

$$f(x) = 0.787386 (1 - e^{-1.71602x})$$

Routines for Curve and Surface Fitting

Below is a brief description of IDL routines for curve and surface fitting. More detailed information is available in the *IDL Reference Guide*.

COMFIT	Fits paired data using one of six common filtering functions.
CRVLENGTH	Computes the length of a curve.
CURVEFIT	Fits multivariate data with a user-supplied function.
GAUSS2DFIT	Fits a 2D elliptical Gaussian equation to rectilinearly gridded data.
GAUSSFIT	Fits the sum of a Gaussian and a quadratic.
GRID_TPS	Uses thin plate splines to interpolate a set of values over a regular 2D grid, from irregularly sampled data values.
KRIG2D	Interpolates set of points using kriging.
LADFIT	Fits paired data using least absolute deviation method.

Table 22-3: Curve and Surface Fitting Routines in IDL

<code>LINFIT</code>	Fits by minimizing the Chi-square error statistic.
<code>LMFIT</code>	Does a non-linear least squares fit.
<code>MIN_CURVE_SURF</code>	Interpolates points with a minimum curvature surface or a thin-plate-spline surface. Useful with <code>CONTOUR</code> .
<code>POLY_FIT</code>	Performs a least-square polynomial fit.
<code>REGRESS</code>	Computes fit using multiple linear regression.
<code>SFIT</code>	Performs polynomial fit to a surface.
<code>SVDFIT</code>	Multivariate least squares fit using SVD method.
<code>TRIGRID</code>	Interpolates irregularly-gridded data to a regular grid from a triangulation.

Table 22-3: Curve and Surface Fitting Routines in IDL

Eigenvalues and Eigenvectors

Consider a system of equations that satisfies the array-vector relationship $Ax = \lambda x$, where A is an n -by- n array, x is an n -element vector, and λ is a scalar. A scalar λ and nonzero vector x that simultaneously satisfy this relationship are referred to as an eigenvalue and an eigenvector of the array A , respectively. The set of all eigenvectors of the array A is then referred to as the eigenspace of A . Ideally, the eigenspace will consist of n linearly-independent eigenvectors, although this is not always the case.

IDL computes the eigenvalues and eigenvectors of a real symmetric n -by- n array using Householder transformations and the QL algorithm with implicit shifts. The eigenvalues of a real, n -by- n nonsymmetric array are computed from the upper Hessenberg form of the array using the QR algorithm. Eigenvectors are computed using inverse subspace iteration.

Although it is not practical for numerical computation, the problem of computing eigenvalues and eigenvectors can also be defined in terms of the determinant function. The eigenvalues of an n -by- n array A are the roots of the polynomial defined by $\det(A - \lambda I)$, where I is the identity matrix (an array with 1s on the main diagonal and 0s elsewhere) with the same dimensions as A . By expressing eigenvalues as the roots of a polynomial, we see that they can be either real or complex. If an eigenvalue is complex, its corresponding eigenvectors are also complex.

The following examples demonstrate how to use IDL to compute the eigenvalues and eigenvectors of real, symmetric and nonsymmetric n -by- n arrays. Note that it is possible to check the accuracy of the computed eigenvalues and eigenvectors by algebraically manipulating the definition given above to read $Ax - \lambda x = 0$; in this case 0 denotes an n -element vector, all elements of which are zero.

Symmetric Array with n Distinct Real Eigenvalues

Example

To compute eigenvalues and eigenvectors of a real, symmetric, n -by- n array, begin with a symmetric array A .

Note

The eigenvalues and eigenvectors of a real, symmetric n -by- n array are real numbers.

```
A = [[ 3.0,  1.0, -4.0], $
      [ 1.0,  3.0, -4.0], $
```

```

    [-4.0, -4.0, 8.0]]
; Compute the tridiagonal form of A:
TRIRED, A, D, E
; Compute the eigenvalues (returned in vector D) and
; the eigenvectors (returned in the rows of the array A):
TRIQL, D, E, A
; Print eigenvalues:
PRINT, D

```

IDL prints:

```
2.00000  4.76837e-07  12.0000
```

The exact values are: [2.0, 0.0, 12.0].

```

;Print the eigenvectors, which are returned as row vectors in A:
PRINT, A

```

IDL prints:

```

0.707107  -0.707107  0.00000
-0.577350  -0.577350  -0.577350
-0.408248  -0.408248  0.816497

```

The exact eigenvectors are:

$$\begin{bmatrix} 1/\sqrt{2} & -1/\sqrt{2} & 0 \\ -1/\sqrt{3} & -1/\sqrt{3} & -1/\sqrt{3} \\ -1/\sqrt{6} & -1/\sqrt{6} & 2/\sqrt{6} \end{bmatrix}$$

Nonsymmetric Array with n Distinct Real and Complex Eigenvalues

Example

To compute the eigenvalues and eigenvectors of a real, nonsymmetric n -by- n array, begin with an array A . In this example, there are n distinct eigenvalues and n linearly-independent eigenvectors.

```

A = [[ 1.0, 0.0,  2.0], $
      [ 0.0, 1.0, -1.0], $
      [-1.0, 1.0,  1.0]]
; Reduce to upper Hessenberg format:
hes = ELMHES(A)
; Compute the eigenvalues:
evals = HQR(hes)
; Print the eigenvalues:
PRINT, evals

```

IDL prints:

```

( 1.00000, -1.73205)( 1.00000,  1.73205)
( 1.00000,  0.00000)

```

Note

The three eigenvalues are distinct, and that two are complex. Note also that complex eigenvalues of an n -by- n real, nonsymmetric array always occur in complex conjugate pairs.

```

; Initialize a variable to contain the residual:
residual = 1
; Compute the eigenvectors and the residual for each
; eigenvalue/eigenvector pair, using double-precision arithmetic:
evecs = EIGENVEC(A, evals, /DOUBLE, RESIDUAL=residual)
; Print the eigenvectors, which are returned as
; row vectors in evecs:
PRINT, evecs[* ,0]

```

IDL prints:

```

( 0.68168704,  0.18789033)( -0.34084352, -0.093945164)
( 0.16271780, -0.59035830)
PRINT, evecs[* ,1]

```

IDL prints:

```

( 0.18789033,  0.68168704)( -0.093945164, -0.34084352)
(-0.59035830,  0.16271780)
PRINT, evecs[* ,2]

```

IDL prints:

```

( 0.70710678,  0.0000000)( 0.70710678,  0.0000000)
(-2.3570226e-21,  0.0000000)

```

We can check the accuracy of these results using the relation $Ax - \lambda x = 0$. The array contained in the variable specified by the `RESIDUAL` keyword contains the result of this computation.

```

PRINT, residual

```

IDL prints:

```
( -1.2021898e-07, 1.1893681e-07)( 6.0109490e-08, -5.9468404e-08)
( 1.0300230e-07, 1.0411269e-07)
( 1.1893681e-07, -1.2021898e-07)( -5.9468404e-08, 6.0109490e-08)
( 1.0411269e-07, 1.0300230e-07)
( 0.0000000, 0.0000000)( 0.0000000, 0.0000000)
```

The results are all zero to within machine precision.

Repeated Eigenvalues

Example

To compute the eigenvalues and eigenvectors of a real, nonsymmetric n -by- n array, begin with an array A . In this example, there are fewer than n distinct eigenvalues, but n independent eigenvectors are available.

```
A = [[8.0, 0.0, 3.0], $
      [2.0, 2.0, 1.0], $
      [2.0, 0.0, 3.0]]
; Reduce A to upper Hessenberg form and compute the eigenvalues.
; Note that both operations can be combined into a single command.
evals = HQR(ELMHES(A))
; Print the eigenvalues:
PRINT, evals
```

IDL prints:

```
( 9.00000, 0.00000) ( 2.00000, 0.00000)
( 2.00000, 0.00000)
```

Note

The three eigenvalues are real, but only two are distinct.

```
; Initialize a variable to contain the residual:
residual = 1
; Compute the eigenvectors and residual, using
; double-precision arithmetic:
evecs = EIGENVEC(A, evals, /DOUBLE, RESIDUAL=residual)
; Print the eigenvectors:
PRINT, evecs[* ,0]
```

IDL prints:

```
( 0.90453403, 0.0000000)( 0.30151134, 0.0000000)
( 0.30151134, 0.0000000)
PRINT, evecs[* ,1]
```

IDL prints:


```
( -0.27907279,  0.0000000)( -0.78140380,  0.0000000)
(  0.55814557,  0.0000000)
PRINT, evecs[*,2]
```

IDL prints:

```
( -0.27907279,  0.0000000)( -0.78140380,  0.0000000)
(  0.55814557,  0.0000000)
```

We can compute an independent eigenvector for the repeated eigenvalue (2.0) by perturbing it slightly, allowing the algorithm `EIGENVEC` to recognize the eigenvalue as distinct and to compute a linearly-independent eigenvector.

```
newresidual = 1
evecs[*,2] = EIGENVEC(A, evals[2]+1.0e-6, /DOUBLE, $
  RESIDUAL = newresidual)
PRINT, evecs[*,2]
```

IDL prints:

```
( -0.33333333,  0.0000000)(  0.66666667,  0.0000000)
(  0.66666667,  0.0000000)
```

Once again, we can check the accuracy of these results by checking that each element in the residuals—for both the original eigenvectors and the perturbed eigenvector—is zero to within machine precision.

Example 4: The So-called Defective Case

In the so-called defective case, there are fewer than n distinct eigenvalues and fewer than n linearly-independent eigenvectors. Begin with an array A :

```
A = [[2.0, -1.0], $
      [1.0,  0.0]]
; Reduce A to upper Hessenberg form and compute the eigenvalues.
; Note that both operations can be combined into a single command.
evals = HQR(ELMHES(A))
; Print the eigenvalues:
PRINT, evals
```

IDL prints:

```
(  1.00000,  0.00000)(  1.00000,  0.00000)
```

Note

The two eigenvalues are real, but not distinct.

```

;Compute the eigenvectors, using double-precision arithmetic:
evecs = EIGENVEC(A, evals, /DOUBLE)
;Print the eigenvectors:
PRINT, evecs[* ,0]

```

IDL prints:

```

( 0.70710678, 0.0000000)( 0.70710678, 0.0000000)
PRINT, evecs[* ,1]

```

IDL prints:

```

( 0.70710678, 0.0000000)( 0.70710678, 0.0000000)

```

We attempt to compute an independent eigenvector using the method described in the previous example:

```

evecs[* ,1] = EIGENVEC(A, evals[1]+1.0e-6, /DOUBLE)
PRINT, evecs[1,*]

```

IDL prints:

```

( 0.70710678, 0.0000000)( 0.70710678, 0.0000000)

```

In this example, n independent eigenvectors do not exist. This situation is termed the defective case and cannot be resolved analytically or numerically.

Routines for Computing Eigenvalues and Eigenvectors

Below is a brief description of IDL routines for computing eigenvalues and eigenvectors. More detailed information is available in the *IDL Reference Guide*.

EIGENQL	Computes eigenvalues and eigenvectors of a real, symmetric array.
EIGENVEC	Computes eigenvectors of a real, non-symmetric array.
ELMHES	Reduces nonsymmetric array to upper Hessenberg form.
HQR	Returns all eigenvalues of an upper Hessenberg array.
TRIQL	Determines eigenvalues and eigenvectors of tridiagonal array.
TRIRED	Reduces a real, symmetric array to tridiagonal form.

Table 22-4: Eigenvalue and Eigenvector Routines in IDL

Gridding and Interpolation

Given a set of tabulated data in n -dimensions with each dimension being described as follows:

1. $\{x_i, y_i = f(x_i)\}$,
2. $\{x_i, y_i, z_i = f(x_i, y_i)\}$, or
3. $\{x_i, y_i, z_i, w_i = f(x_i, y_i, z_i)\}$

it is possible to calculate intermediate values of the function f using interpolation. IDL includes a variety of routines to solve this type of problem.

The determination of intermediate values is based upon an interpolating function that establishes a relationship between the tabulated data points. Different algorithms employ different types of interpolating functions suitable for different types of data trends.

Unlike curve-fitting algorithms, interpolation requires that the interpolating function be an exact fit at each of the tabulated data points. Interpolation does not use any type of error analysis and its accuracy depends upon the behavior of the interpolating function between successive data points. Polynomial, spline, and nearest-neighbor are among the interpolation methods used in IDL. Kriging is another interpolation method, one which does not require an exact fit at each tabulated data point. Kriging applies a weighting to each of the tabulated data points based on spatial variance and trends among the points. Weights are computed by combining calculations of spatial continuity and anisotropy within either an exponential or spherical semivariogram model.

Gridding, a topic closely related to interpolation, is the problem of creating uniformly-spaced planar data from irregularly-spaced data. IDL handles this type of problem by constructing a Delaunay triangulation. This method is highly accurate and has great utility since many of IDL's graphics routines require uniformly-gridded data. Extrapolation, the estimation of values outside the range of tabulated data, is also possible using this method.

Routines for Gridding and Interpolation

Below is a brief description of IDL routines for gridding and interpolation. More detailed information is available in the *IDL Reference Guide*.

Routine	Description
BILINEAR	Computes array using bilinear interpolation.
GRID_TPS	Uses thin plate splines to interpolate a set of values over a regular 2D grid, from irregularly sampled data values.
GRID3	Creates a regularly-gridded 3D dataset from a set of scattered 3D nodes.
INTERPOL	Performs linear interpolation on vectors.
INTERPOLATE	Returns an array of interpolates.
KRIG2D	Interpolates set of points using kriging.
MIN_CURVE_SURF	Interpolates points with a minimum curvature surface or a thin-plate-spline surface. Useful with <code>CONTOUR</code> .
POLAR_SURFACE	Interpolates a surface from polar coordinates to rectangular coordinates.
SPH_SCAT	Performs spherical gridding.
SPL_INIT	Establishes the type of interpolating spline.
SPL_INTERP	Performs cubic spline interpolation (Numerical Recipes).
SPLINE	Performs cubic spline interpolation.
SPLINE_P	Performs parametric cubic spline interpolation.
TRI_SURF	Interpolates gridded set of points with a smooth quintic surface.
TRIANGULATE	Constructs Delaunay triangulation of a planar set of points.
TRIGRID	Interpolates irregularly-gridded data to a regular grid from a triangulation.

Table 22-5: Gridding and Interpolation Routines in IDL

Routine	Description
VALUE_LOCATE	Finds the intervals within a given monotonic vector that brackets a given set of one or more search values.
VORONOI	Computes Voronoi polygon given Delaunay triangulation.

Table 22-5: Gridding and Interpolation Routines in IDL (Continued)

Hypothesis Testing

Hypothesis testing tests one or more sample populations for a statistical characteristic or interaction. The results of the testing process are generally used to formulate conclusions about the probability distributions of the sample populations.

Hypothesis testing involves four steps:

- The formulation of a hypothesis.
- The selection and collection of sample population data.
- The application of an appropriate test.
- The interpretation of the test results.

For example, suppose the FDA wishes to establish the effectiveness of a new drug in the treatment of a certain ailment. Researchers test the assumption that the drug is effective by administering it to a sample population and collecting data on the patients' health. Once the data are collected, an appropriate statistical test is selected and the results analyzed. If the interpretation of the test results suggests a statistically significant improvement in the patients' condition, the researchers conclude that the drug will be effective in general.

It is important to remember that a valid or successful test does not prove the proposed hypothesis. Only by disproving competing or opposing hypotheses can a given assumption's validity be statistically established.

One- and Two-sided Tests

In the above example, only the hypothesis that the drug would significantly improve the condition of the patients receiving it was tested. This type of test is called one-sided or one-tailed, because it is concerned with deviation in one direction from the norm (in this case, improvement of the patients' condition). A hypothesis designed to test the improvement or ill-effect of the trial drug on the patient group would be called two-sided or two-tailed.

Parametric and Nonparametric Tests

Tests of hypothesis are usually classified into parametric and nonparametric methods. Parametric methods make assumptions about the underlying distribution from which sample populations are selected. Nonparametric methods make no assumptions about a sample population's distribution and are often based upon magnitude-based ranking, rather than actual measurement data. In many cases it is possible to replace a

parametric test with a corresponding nonparametric test without significantly affecting the conclusion.

The following example demonstrates this by replacing the parametric T-means test with the nonparametric Wilcoxon Rank-Sum test to test the hypothesis that two sample populations have significantly different means of distribution.

Define two sample populations.

```
X = [257, 208, 296, 324, 240, 246, 267, 311, 324, 323, 263, $
     305, 270, 260, 251, 275, 288, 242, 304, 267]
Y = [201, 56, 185, 221, 165, 161, 182, 239, 278, 243, 197, $
     271, 214, 216, 175, 192, 208, 150, 281, 196]
```

Compute the T-statistic and its significance, using IDL's `TM_TEST` function, assuming that X and Y belong to Normal populations with the same variance.

```
PRINT, TM_TEST(X, Y)
```

IDL prints:

```
5.52839 2.52455e-06
```

The small value of the significance (2.52455e-06) indicates that X and Y have significantly different means.

Compute the Wilcoxon Rank-Sum Test, using IDL's `RS_TEST` function, to test the hypothesis that X and Y have the same mean of distribution.

```
PRINT, RS_TEST(X, Y)
```

IDL prints:

```
-4.26039 1.01924e-05
```

The small value of the computed probability (1.01924e-05) requires the rejection of the proposed hypothesis and the conclusion that X and Y have significantly different means of distribution.

Each of IDL's 11 parametric and nonparametric hypothesis testing functions is based upon a well-known and widely-accepted statistical test. Each of these functions returns a two-element vector containing the statistic on which the test is based and its significance. Examples are provided and demonstrate how the result is interpreted.

Routines for Hypothesis Testing

Below is a brief description of IDL routines for hypothesis testing. More detailed information is available in the *IDL Reference Guide*.

Routine	Description
CTL_TEST	Performs chi-square goodness-of-fit test.
FV_TEST	Performs the F-variance test.
KW_TEST	Performs Kruskal-Wallis H-test.
LNP_TEST	Computes the Lomb Normalized Periodogram.
MD_TEST	Performs the Median Delta test.
R_TEST	Runs test for randomness.
RS_TEST	Performs the Wilcoxon Rank-Sum test.
S_TEST	Performs the Sign test.
TM_TEST	Performs t-means test.
XSQ_TEST	Computes Chi-square goodness-of-fit test.

Table 22-6: Hypothesis Testing Routines in IDL

Integration

Numerical methods of approximating integrals are important in many areas of pure and applied science. For a function of a single variable, $f(x)$, it is often the case that the antiderivative $F = \int f(x) dx$ is unavailable using standard techniques such as trigonometric substitutions and integration-by-parts formulas. These standard techniques become increasingly unusable when integrating multivariate functions, $f(x, y)$ and $f(x, y, z)$. Numerically approximating the integral operator provides the only method of solution when the antiderivative is not explicitly available. IDL offers the following numerical methods for the integration of uni-, bi-, and trivariate functions:

- Integration of a univariate function over an open or closed interval is possible using one of several routines based on well known methods developed by Romberg and Simpson.

$$I = \int_{x=a}^{x=b} f(x) dx$$

- The problem of integrating over a tabulated set of data $\{ x_i, y_i = f(x_i) \}$ can be solved using a highly accurate 5-point Newton-Cotes formula. This method is more accurate and efficient than using interpolation or curve-fitting to find an approximate function and then integrating.
- Integration of a bivariate function over a regular or irregular region in the x - y plane is possible using an iterated Gaussian Quadrature routine.

$$I = \int_{x=a}^{x=b} \int_{y=p(x)}^{y=q(x)} f(x, y) dy dx$$

- Integration of a trivariate function over a regular or irregular region in x - y - z space is possible using an iterated Gaussian Quadrature routine.

$$I = \int_{x=a}^{x=b} \int_{y=p(x)}^{y=q(x)} \int_{z=u(x,y)}^{z=v(x,y)} f(x, y, z) dz dy dx$$

Note

IDL's iterated Gaussian Quadrature routines, INT_2D and INT_3D, follow the $dy-dx$ and $dz-dy-dx$ order of evaluation, respectively. Problems not conforming to this standard must be changed as described in the following example.

A Bivariate Function

Example

Suppose that we wish to evaluate

$$\int_{y=0}^{y=4} \int_{x=\sqrt{y}}^{x=2} y \cdot \cos(x^5) dx dy$$

The order of integration is initially described as a $dx-dy$ region in the $x-y$ plane. Using the diagram below, you can easily change the integration order to $dy-dx$.

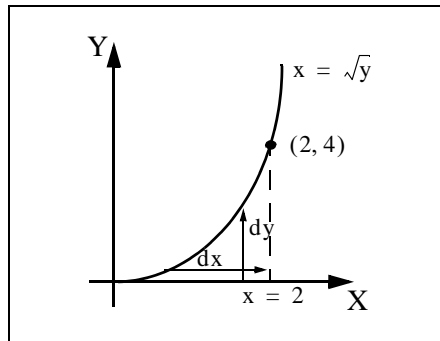


Figure 22-3: The Bivariate Function

The integral is now of the form

$$\int_{x=0}^{x=2} \int_{y=0}^{y=x^2} y \cdot \cos(x^5) dy dx$$

The new expression can be evaluated using the [INT_2D](#) function.

To use INT_2D, we must specify the function to be integrated and expressions for the upper and lower limits of integration. First, we write an IDL function for the integrand, the function $f(x, y)$:

```

FUNCTION fxy, X, Y
  RETURN, Y * COS(X^5)
END

```

Next, we write a function for the limits of integration of the inner integral. Note that the limits of the outer integral are specified numerically, in vector form, while the limits of the inner integral must be specified as an IDL function even if they are constants. In this case, the function is:

```

FUNCTION pq_limits, X
  RETURN, [0.0, X^2]
END

```

Now we can use the following IDL commands to print the value of the integral expressed above. First, we define a variable `AB_LIMITS` containing the vector of lower and upper limits of the outer integral. Next, we call `INT_2D`. The first argument is the name of the IDL function that represents the integrand (`FX`, in this case). The second argument is the name of the variable containing the vector of limits for the outer integral (`AB_LIMITS`, in this case). The third argument is the name of the IDL function defining the lower and upper limits of the inside integral (`PQ_LIMITS`, in this case). The fourth argument (48) refers to the number of transformation points used in the computation. As a general rule, the number of transformation points used with iterated Gaussian Quadrature should increase as the integrand becomes more oscillatory or the region of integration becomes more irregular.

```

ab_limits = [0.0, 2.0]
PRINT, INT_2D('fxy', ab_limits, 'pq_limits', 48)

```

IDL prints:

```
0.055142668
```

This is the exact solution to 9 decimal accuracy.

A Trivariate Function

Example

Suppose that we wish to evaluate

$$\int_{x=-2}^{x=2} \int_{y=-\sqrt{4-x^2}}^{y=\sqrt{4-x^2}} \int_{z=0}^{z=\sqrt{4-x^2-y^2}} z(x^2 + y^2 + z^2)^{3/2} dz dy dx$$

This integral can be evaluated using the INT_3D function. As with INT_2D, we must specify the function to be integrated and expressions for the upper and lower limits of integration. Note that in this case IDL functions must be provided for the upper and lower integration limits of both inside integrals.

For the above integral, the required functions are the integrand $f(x, y, z)$:

```
FUNCTION fxyz, X, Y, Z
  RETURN, Z * (X^2 + Y^2 + Z^2)^1.5
END
```

The limits of integration of the first inside integral:

```
FUNCTION pq_limits, X
  RETURN, [-SQRT(4.0 - X^2), SQRT(4.0 - X^2)]
END
```

The limits of integration of the second inside integral:

```
FUNCTION uv_limits, X, Y
  RETURN, [0.0, SQRT(4.0 - X^2 - Y^2)]
END
```

We can use the following IDL commands to determine the value of the above integral using 6, 10, 20 and 48 transformation points.

For 6 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
  'pq_limits', 'uv_limits', 6)
```

IDL prints:

```
57.417720
```

For 10 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
  'pq_limits', 'uv_limits', 10)
```

IDL prints:

```
57.444248
```

20 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
  'pq_limits', 'uv_limits', 20)
```

IDL prints:

```
57.446201
```

48 transformation points:

```
PRINT, INT_3D('fxyz', [-2.0, 2.0], $
             'pq_limits', 'uv_limits', 48)
```

IDL prints:

```
57.446265
```

The exact solution to 6-decimal accuracy is 57.446267.

Routines for Differentiation and Integration

Below is a brief description of IDL routines for differentiation and integration. More detailed information is available in the *IDL Reference Guide*.

CRVLENGTH	Computes the length of a curve.
DERIV	Performs differentiation using 3-point Lagrangian interpolation.
DERIVSIG	Computes standard deviation of derivative found by DERIV.
INT_2D	Computes the double integral of a bivariate function.
INT_3D	Computes the triple integral of a trivariate function.
INT_TABULATED	Integrates a tabulated set of data.
LSODE	Advances a solution to a system of ordinary differential equations one time-step H.
QROMB	Evaluates integral over a closed interval.
QROMO	Evaluates integral over an open interval.
QSIMP	Evaluates integral using Simpson's rule.
RK4	Solves differential equations using fourth-order Runge-Kutta method.

Table 22-7: Differentiation and Integration Routines in IDL

Linear Systems

IDL offers a variety of methods for the solution of simultaneous linear equations. In order to use these routines successfully, the user should consider both existence and uniqueness criteria and the potential difficulties in finding the solution numerically.

The solution vector x of an n -by- n linear system $Ax = b$ is guaranteed to exist and to be unique if the coefficient array A is invertible. Using a simple algebraic manipulation, it is possible to formulate the solution vector x in terms of the inverse of the coefficient array A and the right-side vector b : $x = A^{-1}b$. Although this relationship provides a concise mathematical representation of the solution, it is never used in practice. Array inversion is computationally expensive (requiring a large number of floating-point operations) and prone to severe round-off errors.

An alternate way of describing the existence of a solution is to say that the system $Ax = b$ is solvable if and only if the vector b may be expressed as a linear combination of the columns of A . This definition is important when considering the solutions of non-square (over- and under-determined) linear systems.

While the invertibility of the coefficient array A may ensure that a solution exists, it does not help in determining the solution. Some systems can be solved accurately using numerical methods whereas others cannot. In order to better understand the accuracy of a numerical solution, we can classify the condition of the system it solves.

The scalar quantity known as the condition number of a linear system is a measure of a solution's sensitivity to the effects of finite-precision arithmetic. The condition number of an n -by- n linear system $Ax = b$ is computed explicitly as $|A||A^{-1}|$ (where $||$ denotes a Euclidean norm). A linear system whose condition number is small is considered well-conditioned and well suited to numerical computation. A linear system whose condition number is large is considered ill-conditioned and prone to computational errors. To some extent, the solution of an ill-conditioned system may be improved using an extended-precision data type (such as double-precision float). Other situations require an approximate solution to the system using its Singular Value Decomposition.

The following two examples show how the singular value decomposition may be used to find solutions when a linear system is over- or underdetermined.

Overdetermined Systems

Example

In the case of the overdetermined system (when there are more linear equations than unknowns), the vector b cannot be expressed as a linear combination of the columns of array A . (In other words, b lies outside of the subspace spanned by the columns of A .) Using IDL's SVDC procedure, it is possible to determine a projected solution of the overdetermined system (b is projected onto the subspace spanned by the columns of A and then the system is solved). This type of solution has the property of minimizing the residual error $E = b - Ax$ in a least-squares sense.

Suppose that we wish to solve the following linear system:

$$\begin{bmatrix} 1.0 & 2.0 \\ 1.0 & 3.0 \\ 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 4.0 \\ 5.0 \\ 6.0 \end{bmatrix}$$

The vector b does not lie in the two-dimensional subspace spanned by the columns of A (there is no linear combination of the columns of A that yield b), and therefore an exact solution is not possible.

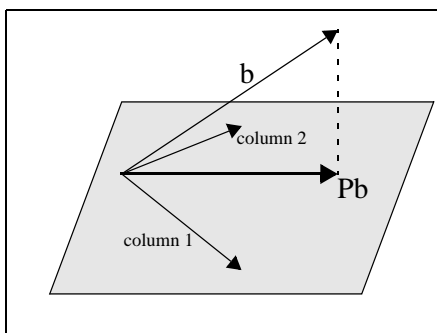


Figure 22-4: Overdetermined System Diagram

It is possible, however, to find a solution to this system that minimizes the residual error by orthogonally projecting the vector b onto the two-dimensional subspace spanned by the columns of the array A . The projected vector is then used as the right-hand side of the system. The orthogonal projection of b onto the column space of A may be expressed with the array-vector product $A(A^T A)^{-1} A^T b$, where $A(A^T A)^{-1} A^T$ is known as the projection matrix, P .

In this example, the array-vector product Pb yields:

$$\begin{bmatrix} 4.0 \\ 5.0 \\ 0.0 \end{bmatrix}$$

and we wish to solve the linear system

$$\begin{bmatrix} 1.0 & 2.0 \\ 1.0 & 3.0 \\ 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 4.0 \\ 5.0 \\ 0.0 \end{bmatrix} \quad \text{where} \quad \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.0 \end{bmatrix}$$

In many cases, the explicit calculation of the projected solution is numerically unstable, resulting in large accumulated round-off errors. For this reason it is best to use singular value decomposition to effect the orthogonal projection of the vector b onto the subspace spanned by the columns of the array A .

The following IDL commands use singular value decomposition to solve the system in a numerically stable manner. Begin with the array A :

```
A = [[1.0, 2.0], $
      [1.0, 3.0], $
      [0.0, 0.0]]
; Define the right-hand side vector B:
B = [4.0, 5.0, 6.0]
; Compute the singular value decomposition of A:
SVDC, A, W, U, V
```

Create a diagonal array WP of reciprocal singular values from the output vector W . To avoid overflow errors when the reciprocal values are calculated, only elements with absolute values greater than or equal to 1.0×10^{-5} are reciprocated.

```
N = N_ELEMENTS(W)
WP = FLTARR(N, N)
FOR K = 0, N-1 DO $
  IF ABS(W(K)) GE 1.0e-5 THEN WP(K, K) = 1.0/W(K)
```

We can now express the solution to the linear system as a array-vector product. (See Section 2.6 of *Numerical Recipes* for a derivation of this formula.)

```
X = V ## WP ## TRANSPOSE(U) ## B
; Print the solution:
PRINT, X
```


IDL Prints:

```
2.00000
1.00000
```

Underdetermined Systems

Example

In the case of the underdetermined system (when there are fewer linear equations than unknowns), a unique solution is not possible. Using IDL's SVDC procedure it is possible to determine the minimal norm solution. Given a vector norm, this type of solution has the property of having the minimal length of all possible solutions with respect to that norm.

Suppose that we wish to solve the following linear system.

$$\begin{bmatrix} 1.0 & 3.0 & 3.0 & 2.0 \\ 2.0 & 6.0 & 9.0 & 5.0 \\ -1.0 & -3.0 & 3.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 5.0 \\ 5.0 \end{bmatrix}$$

Using elementary row operations it is possible to reduce the system to

$$\begin{bmatrix} 1.0 & 3.0 & 3.0 & 2.0 \\ 0.0 & 0.0 & 3.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 3.0 \\ 0.0 \end{bmatrix}$$

It is now possible to express the solution x in terms of x_1 and x_3 :

$$x = \begin{bmatrix} -2 - 3x_1 - x_3 \\ x_1 \\ 1 - x_3/3 \\ x_3 \end{bmatrix}$$

The values of x_1 and x_3 are completely arbitrary. Setting $x_1 = 0$ and $x_3 = 0$ results in one possible solution of this system:

$$x = \begin{bmatrix} -2.0 \\ 0.0 \\ 1.0 \\ 0.0 \end{bmatrix}$$

Another possible solution is obtained using singular value decomposition and results in the minimal norm condition. The minimal norm solution for this system is:

$$x = \begin{bmatrix} -0.211009 \\ -0.633027 \\ 0.963303 \\ 0.110092 \end{bmatrix}$$

Note that this vector also satisfies the solution x as it is expressed in terms of x_1 and x_3 .

The following IDL commands use singular value decomposition to find the minimal norm solution. Begin with the array A :

```
A = [[ 1.0, 3.0, 3.0, 2.0], $
      [ 2.0, 6.0, 9.0, 5.0], $
      [-1.0, -3.0, 3.0, 0.0]]
; Define the right-hand side vector B:
B = [1.0, 5.0, 5.0]
; Compute the decomposition of A:
SVDC, A, W, U, V
```

Create a diagonal array WP of reciprocal singular values from the output vector W . To avoid overflow errors when the reciprocal values are calculated, only elements with absolute values greater than or equal to 1.0×10^{-5} are reciprocated.

```
N = N_ELEMENTS(W)
WP = FLTARR(N, N)
FOR K = 0, N-1 DO $
  IF ABS(W(K)) GE 1.0e-5 THEN WP(K, K) = 1.0/W(K)
```

We can now express the solution to the linear system as a array-vector product. (See Section 2.6 of *Numerical Recipes* for a derivation of this formula.) The solution is expressed in terms of x_1 and x_3 with minimal norm.

```
X = V ## WP ## TRANSPOSE(U) ## B
;Print the solution:
PRINT, X
```

IDL Prints:

```
-0.211009
-0.633027
 0.963303
 0.110092
```

Complex Linear Systems

Example

We can use IDL's `LU_COMPLEX` function to compute the solution to a linear system with real and complex coefficients. Suppose we wish to solve the following linear system:

$$\begin{bmatrix} -1 + 0i & 1 - 3i & 2 + 0i & 3 + 3i \\ -2 + 0i & -1 + 3i & 0 + 1i & 3 + 1i \\ 3 + 0i & 0 + 4i & 0 - 1i & 0 - 3i \\ 2 + 0i & 1 + 1i & 2 + 1i & 2 + 1i \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} 15 - 2i \\ -2 - 1i \\ -20 + 11i \\ -10 + 10i \end{bmatrix}$$

```
;First we define the real part of the complex coefficient array:
re = [[-1, 1, 2, 3], $
      [-2, -1, 0, 3], $
      [3, 0, 0, 0], $
      [2, 1, 2, 2]]
;Next, we define the imaginary part of the coefficient array:
im = [[0, -3, 0, 3], $
      [0, 3, 1, 1], $
      [0, 4, -1, -3], $
      [0, 1, 1, 1]]
; Combine the real and imaginary parts to form
; a single complex coefficient array:
A = COMPLEX(re, im)
; Define the right-hand side vector B:
B = [COMPLEX(15,-2), COMPLEX(-2,-1), COMPLEX(-20,11), $
     COMPLEX(-10,10)]
; Compute the solution using double-precision complex arithmetic:
Z = LU_COMPLEX(A, B, /DOUBLE)
PRINT, TRANSPOSE(Z), FORMAT = '(f5.2, ",", f5.2, "i)'
```

IDL prints:

```
-4.00, 1.00i
 2.00, 2.00i
 0.00, 3.00i
-0.00,-1.00i
```

We can check the accuracy of the computed solution by computing the residual, $Az-b$:

```
PRINT, A##Z-B
```

IDL prints:

```
(      0.00000,      0.00000)
(      0.00000,      0.00000)
(      0.00000,      0.00000)
(      0.00000,      0.00000)
```

Routines for Solving Simultaneous Linear Equations

Below is a brief description of IDL routines for solving simultaneous linear equations. More detailed information is available in the *IDL Reference Guide*.

CHOLDC	Constructs Cholesky decomposition of a matrix.
CHOLSOL	Solves set of linear equations (use with CHOLDC).
COND	Computes the condition number of a square matrix.
CRAMER	Solves system of linear equations using Cramer's rule.
CROSSP	Computes vector cross product.
DETERM	Computes the determinant of a square matrix.
GS_ITER	Solves linear system using Gauss-Seidel iteration.
IDENTITY	Returns an identity array.
INVERT	Computes the inverse of a square array.
LINBCG	Solves a set of sparse linear equations using the iterative biconjugate gradient method.
LU_COMPLEX	Solves complex linear system using LU decomposition.
LUDC	Replaces array with the LU decomposition.
LUMPROVE	Uses LU decomposition to iteratively improve an approximate solution.
LUSOL	Solves a set of linear equations. Use with LUDC.
NORM	Computes Euclidean norm of vector or Infinity norm of array.

Table 22-8: Routines for Solving Simultaneous Linear Equations

SVDC	Computes Singular Value Decomposition of an array.
SVSOL	Solves set of linear equations using back-substitution.
TRACE	Computes the trace of an array.
TRISOL	Solves tridiagonal systems of linear equations.

Table 22-8: Routines for Solving Simultaneous Linear Equations

Nonlinear Equations

The problem of finding a solution to a system of n nonlinear equations, $F(x) = 0$, may be stated as follows:

given $F: \mathbb{R}^n \rightarrow \mathbb{R}^n$, find x_* (an element of \mathbb{R}^n) such that $F(x_*) = 0$

For example:

$$F(x) = \begin{bmatrix} x_0 + x_1 - 3 \\ x_0^2 + x_1^2 - 9 \end{bmatrix}$$

$x_* = [0, 3]$ or $x_* = [3, 0]$

Note

A solution to a system of nonlinear equations is not necessarily unique.

The most powerful and successful numerical methods for solving systems of nonlinear equations are loosely based upon a simple two-step iterative method frequently referred to as Newton's method. This method begins with an initial guess and constructs a solution by iteratively approximating the n -dimensional nonlinear system of equations with an n -by- n linear system of equations.

The first step formulates an n -by- n linear system of equations ($J_s = -F$) where the coefficient array J is the Jacobian (the array of first partial derivatives of F), s is a solution vector, and $-F$ is the negative of the nonlinear system of equations. Both J and $-F$ are evaluated at the current value of the n -element vector x .

$$J(x_k) s_k = -F(x_k)$$

The second step uses the solution s_k of the linear system as a directional update to the current approximate solution x_k of the nonlinear system of equations. The next approximate solution x_{k+1} is a linear combination of the current approximate solution x_k and the directional update s_k .

$$x_{k+1} = x_k + s_k$$

The success of Newton's method relies primarily on providing an initial guess close to a solution of the nonlinear system of equations. In practice this proves to be quite difficult and severely limits the application of this simple two-step method.

IDL provides two algorithms that are designed to overcome the restriction that the initial guess be close to a solution. These algorithms implement a line search which checks, and if necessary modifies, the course of the algorithm at each step ensuring

progress toward a solution of the nonlinear system of equations. IDL's NEWTON and BROYDEN functions are among a class of algorithms known as quasi-Newton methods.

The solution of an n -dimensional system of nonlinear equations, $F(x) = 0$, is often considered a root of that system. As a one-dimensional counterpart to NEWTON and BROYDEN, IDL provides the FX_ROOT and FZ_ROOTS functions.

Routines for Solving Nonlinear Equations

Below is a brief description of IDL routines for solving systems of nonlinear equations. More detailed information is available in the *IDL Reference Guide*.

BROYDEN	Solves nonlinear equations using Broyden's method.
FX_ROOT	Computes real and complex roots of a univariate nonlinear function using an optimal Müller's method.
FZ_ROOTS	Finds the roots of a complex polynomial using Laguerre's method.
NEWTON	Solves nonlinear equations using Newton's method.

Table 22-9: Routines for Solving Nonlinear Equations

Optimization

The problem of finding an unconstrained minimizer of an n -dimensional function, f , may be stated as follows:

given $f: \mathbb{R}^n \rightarrow \mathbb{R}$, find x_* (an element of \mathbb{R}^n) such that $f(x_*)$ is a minimum of f .

For example:

$$f(x) = (x_0 - 3)^4 + (x_1 - 2)^2$$

$$x_* = [3, 2]$$

In minimizing an n -dimensional function f , it is a necessary condition that the gradient at the minimizer x_* , $\nabla f(x_*)$, be the zero vector. Mathematically expressing this condition defines the following system of nonlinear equations.

$$\begin{bmatrix} \frac{\partial f(x)}{\partial x_0} \\ \frac{\partial f(x)}{\partial x_1} \\ \dots \\ \frac{\partial f(x)}{\partial x_{n-1}} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \end{bmatrix}$$

This relation might suggest that finding a minimizer is equivalent to solving a system of linear equations based on the gradient. In most cases, however, this is not true. It is just as likely that a solution, x_* , of $\nabla f(x)=0$ be a maximizer or a local minimizer of f . Thus the gradient alone does not provide sufficient information in determining the role of x_* .

IDL provides two algorithms that do sufficiently determine the global minimizer of an n -dimensional function. IDL's DFPMIN routine is among a class of algorithms known as variable metric methods and requires a user-supplied analytic gradient of the function to be minimized. IDL's POWELL routine implements a direction-set method that does not require a user-supplied analytic gradient. The utility of the POWELL routine is evident as the function to be minimized becomes more complicated and partial derivatives become more difficult to calculate.

Routines for Optimization

Below is a brief description of IDL routines for optimization. More detailed information is available in the *IDL Reference Guide*.

AMOEBA	Minimizes a function using downhill simplex method.
CONSTRAINED_MIN	Minimizes a function using Generalized Reduced Gradient Method.
DFPMIN	Minimizes a function using Davidon-Fletcher-Powell method.
POWELL	Minimizes a function using the Powell method.

Table 22-10: Optimization Routines in IDL

Sparse Arrays

The occurrence of zero elements in a large array is both a computational and storage inconvenience. An array in which a large percentage of elements are zeros is referred to as being *sparse*.

Because standard linear algebra techniques are highly inefficient when dealing with sparse arrays, IDL incorporates a collection of routines designed to handle them effectively. These routines use the row-indexed sparse storage method, which stores the array in structure form, as a vector of data and a vector of indices. The length of each vector is equal to 1 plus the number of diagonal elements of the array plus the number of off-diagonal elements with an absolute magnitude greater than or equal to a specified threshold value. Diagonal elements of the array are always retained even if their absolute magnitude is less than the specified threshold. Sparse array routines that handle array-vector and array-array multiplication, file input/output, and the solution of systems of simultaneous linear equations are included.

Note

For more information on IDL's sparse array storage method, see section 2.7, "Sparse Linear Systems," in *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), published by Cambridge University Press.

When considering using IDL's sparse array routines, remember that the computational savings gained by working in sparse storage format is at least partially offset by the need to first convert the arrays to that format. Although an absolute determination of when to use sparse format is not possible, the example below demonstrates the time savings when solving a 500 by 500 linear system in which approximately 50% of the coefficient array's elements are zeros.

Diagonally-Dominant Array

Example

Create a 500-by-500 element pseudo-random diagonally-dominant floating-point array in which approximately 50% of the elements are zeros. (In a diagonally-dominant array, the diagonal element in a given row is greater than the sum of the absolute values of the non-diagonal elements in that row.)

```
N = 500L
A = RANDOMN(SEED, N, N)*10
; Set elements with absolute magnitude greater than or
; equal to eight to zero:
I = WHERE(ABS(A) GE 8)
```

```

A[I] = 0.0
; Set each diagonal element to the absolute sum of
; its row elements plus 1.0:
diag = TOTAL(ABS(A), 1)
A(INDGEN(N) * (N+1)) = diag + 1.0
; Create a right-hand side vector, b, in which 40% of
; the elements are ones and 60% are twos.
B = [REPLICATE(1.0, 0.4*N), REPLICATE(2.0, 0.6*N)]

```

We now calculate a solution to this system using two different methods, measuring the time elapsed. First, we compute the solution using the iterative biconjugate gradient method and a sparse array storage format. Note that we include everything between the start and stop timer commands as a single operation, so that only computation time (as opposed to typing time) is recorded.

```

;Begin with an initial guess:
X = REPLICATE(1.0, N_ELEMENTS(B))
;Start the timer:
start = SYSTIME(1) & $
;Solve the system:
result1 = LINBCG(SPRSIN(A), B, X) & $
;Stop the timer.
stop = SYSTIME(1)
;Print the time taken, in seconds:
PRINT, 'Time for Iterative Biconjugate Gradient:', stop-start

```

IDL prints:

```

Time for Iterative Biconjugate Gradient      1.1259040

```

Remember that your result will depend on your hardware configuration.

Next, we compute the solution using LU decomposition.

```

;Start the timer:
start = SYSTIME(1) & $
;Compute the LU decomposition of A:
LUDEC, A, index & $
;Compute the solution:
result2 = LUSOL(A, index, B) & $
;Stop the timer:
stop = SYSTIME(1)
;Print the time taken, in seconds:
PRINT, 'Time for LU Decomposition:', stop-start

```

IDL prints:

```

Time for LU decomposition      14.871168

```

Finally, we can compare the absolute error between result1 and result2. The following commands will print the indices of any elements of the two results that

differ by more than 1.0×10^{-5} , or a -1 if the two results are identical to within five decimal places.

```
error = ABS(result1-result2)
PRINT, WHERE(error GT 1.0e-5)
```

IDL prints:

```
-1
```

See the documentation for the WTN function for an example using IDL's sparse array functions with image data.

Note

The times shown here were recorded on a DEC 3000 Alpha workstation running OSF/1; they are shown as examples only. Your times will depend on your specific computing platform.

Routines for Handling Sparse Arrays

Below is a brief description of IDL routines for handling sparse arrays. More detailed information is available in the *IDL Reference Guide*. Note that SPRSIN must be used to convert to sparse storage format before the other routines can be used.

FULSTR	Restores a sparse matrix to full storage mode.
LINBCG	Solves a set of sparse linear equations using the iterative biconjugate gradient method.
READ_SPR	Reads a row-indexed sparse matrix from a file.
SPRSAB	Performs matrix multiplication on sparse matrices.
SPRSAX	Multiplies sparse matrix by a vector.
SPRSIN	Converts matrix to row-index sparse matrix.
SPRSTP	Constructs the transpose of a sparse matrix.
WRITE_SPR	Writes row-indexed sparse array structure to a file.

Table 22-11: Sparse Array Routines in IDL

Time-Series Analysis

A time-series is a sequential collection of data observations indexed over time. In most cases, the observed data is continuous and is recorded at a discrete and finite set of equally-spaced points. An n -element time-series is denoted as $x = (x_0, x_1, x_2, \dots, x_{n-1})$, where the time-indexed distance between any two successive observations is referred to as the sampling interval.

A widely held theory assumes that a time-series is comprised of four components:

- A trend or long term movement.
- A cyclical fluctuation about the trend.
- A pronounced seasonal effect.
- A residual, irregular, or random effect.

Collectively, these components make the analysis of a time-series a far more challenging task than just fitting a linear or nonlinear regression model. Adjacent observations are unlikely to be independent of one another. Clusters of observations are frequently correlated with increasing strength as the time intervals between them become shorter. Often the analysis is a multi-step process involving graphical and numerical methods.

The first step in the analysis of a time-series is the transformation to stationary series. A stationary series exhibits statistical properties that are unchanged as the period of observation is moved forward or backward in time. Specifically, the mean and variance of a stationary time-series remain fixed in time. The sample autocorrelation function is a commonly used tool in determining the stationarity of a time-series. The autocorrelation of a time-series measures the dependence between observations as a function of their time differences or lag. A plot of the sample autocorrelation coefficients against corresponding lags can be very helpful in determining the stationarity of a time-series.

For example, suppose the IDL variable X contains time-series data:

```
X = [ 5.44, 6.38, 5.43, 5.22, 5.28, $  
      5.21, 5.23, 4.33, 5.58, 6.18, $  
      6.16, 6.07, 6.56, 5.93, 5.70, $  
      5.36, 5.17, 5.35, 5.61, 5.83, $  
      5.29, 5.58, 4.77, 5.17, 5.33]
```

The following IDL commands plot both the time-series data and the sample autocorrelation versus the lags.

```
;Set the plotting window to hold two plots:
!P.MULTI=[0,1,2]
;Plot the data:
PLOT, X
```

Compute the sample autocorrelation function for time lagged values 0 – 20 and plot.

```
lag = INDGEN(21)
result = A_CORRELATE(X, lag)
PLOT, lag, result
;Add a reference line at zero:
PLOTS, [0,20], [0,0], /DATA
;Set the plotting window back to a single plot:
!P.MULTI=0
```

The following figure shows the resulting graph.

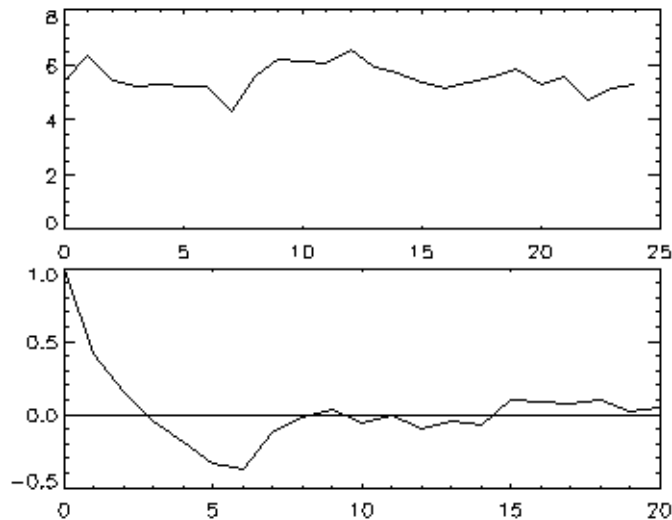


Figure 22-5: The top graph plots time-series data. The bottom graph plots the autocorrelation of that data versus the lag. Because the time-series has a significant autocorrelation up to a lag of seven, it must be considered non-stationary.

Nonstationary components of a time-series may be eliminated in a variety of ways. Two frequently used methods are known as moving averages and forward

differencing. The method of moving averages dampens fluctuations in a time-series by taking successive averages of groups of observations. Each successive overlapping sequence of k observations in the series is replaced by the mean of that sequence. The method of forward differencing replaces each time-series observation with the difference of the current observation and its adjacent observation one step forward in time. Differencing may be computed recursively to eliminate more complex nonstationary components.

Once a time-series has been transformed to stationarity, it may be modeled using an autoregressive process. An autoregressive process expresses the current observation, x_t , as a combination of past time-series values and residual white noise. The simplest case is known as a first order autoregressive model and is expressed as

$$x_t = \phi x_{t-1} + \omega_t$$

The coefficient ϕ is estimated using the time-series data. The general autoregressive model of order p is expressed as

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \dots + \phi_p x_{t-p} + \omega_t$$

Modeling a stationary time-series as a p -th order autoregressive process allows the extrapolation of data for future values of time. This process is known as forecasting.

Routines for Time-Series Analysis

Below is a brief description of IDL routines for time-series analysis. More detailed information is available in the *IDL Reference Guide*.

A_CORRELATE	Computes autocorrelation.
C_CORRELATE	Computes cross correlation.
SMOOTH	Smooths with a boxcar average.
TS_COEF	Computes the coefficients for autoregressive time-series.
TS_DIFF	Computes the forward differences of a time-series.
TS_FCAST	Computes future or past values of a stationary time-series.
TS_SMOOTH	Computes moving averages of a time-series.

Table 22-12: Time-Series Analysis Routines in IDL

Multivariate Analysis

IDL provides a number of tools for analyzing multivariate data. These tools are broadly grouped into two categories: Cluster Analysis and [Principal Components Analysis](#).

Cluster Analysis

Cluster Analysis attempts to construct a sensible and informative classification of an initially unclassified sample population using a set of common variables for each individual. The clusters are constructed so as to group samples with the similar features, based upon a set of variables. The samples (contained in the rows of an input array) are each assigned a cluster number based upon the values of their corresponding variables (contained in the columns of an input array).

In computing a cluster analysis, a predetermined number of cluster centers are formed and then each sample is assigned to the unique cluster which minimizes a distance criterion based upon the variables of the data. Given an m -column, n -row array, IDL's `CLUST_WTS` and `CLUSTER` functions compute n cluster centers and n clusters, respectively. Conceivably, some clusters will contain multiple samples while other clusters will contain none. The choice of clusters is arbitrary; in general, however, the user will want to specify a number less than the default (the number of rows in the input array). The cluster number (the number that identifies the cluster group) assigned to a particular sample or group of samples is not necessarily unique.

It is possible that not all variables play an equal role in the classification process. In this situation, greater or lesser importance may be given to each variable using the `VARIABLE_WTS` keyword to the `CLUST_WTS` function. The default behavior is to assume all variables contained in the data array are of equal importance.

Under certain circumstances, a classification of variables may be desired. The `CLUST_WTS` and `CLUSTER` functions provide this functionality by first transposing the m -column, n -row input array using the `TRANSPOSE` function and then interchanging the roles of variables and samples.

Example of Cluster Analysis

Define an array with 5 variables (columns) and 9 samples (rows):

```
array = [[ 99, 79, 63, 87, 249 ], $
         [ 67, 41, 36, 51, 114 ], $
         [ 67, 41, 36, 51, 114 ], $
         [ 94, 191, 160, 173, 124 ], $
         [ 42, 108, 37, 51, 41 ], $
```



```

[ 67, 41, 36, 51, 114 ], $
[ 94, 191, 160, 173, 124 ], $
[ 99, 79, 63, 87, 249 ], $
[ 67, 41, 36, 51, 114 ] ]
; Compute the cluster weights with four cluster centers:
weights = CLUST_WTS(array, N_CLUSTERS = 4)
; Compute the cluster assignments, for each sample,
; into one of four clusters:
result = CLUSTER(array, weights, N_CLUSTERS = 4)
; Display the cluster assignment and corresponding sample (row):
FOR k = 0, 8 DO $
    PRINT, result[k], array[*, k]

```

IDL prints:

1	99	79	63	87	249
3	67	41	36	51	114
3	67	41	36	51	114
0	94	191	160	173	124
2	42	108	37	51	41
3	67	41	36	51	114
0	94	191	160	173	124
1	99	79	63	87	249
3	67	41	36	51	114

Samples 0 and 7 contain identical data and are assigned to cluster #1. Samples 1, 2, 5, and 8 contain identical data and are assigned to cluster #3. Samples 3 and 6 contain identical data and are assigned to cluster #0. Sample 4 is unique and is assigned to cluster #2.

If this example is run several times, each time computing new cluster weights, it is possible that the cluster number assigned to each grouping of samples may change.

Principal Components Analysis

Principal components analysis is a mathematical technique which describes a multivariate set of data using derived variables. The derived variables are formulated using specific linear combinations of the original variables. The derived variables are uncorrelated and are computed in decreasing order of importance; the first variable accounts for as much as possible of the variation in the original data, the second variable accounts for the second largest portion of the variation in the original data, and so on. Principal components analysis attempts to construct a small set of derived variables which summarize the original data, thereby reducing the dimensionality of the original data.

The principal components of a multivariate set of data are computed from the eigenvalues and eigenvectors of either the sample correlation or sample covariance

matrix. If the variables of the multivariate data are measured in widely differing units (large variations in magnitude), it is usually best to use the sample correlation matrix in computing the principal components; this is the default method used in IDL's PCOMP function.

Another alternative is to standardize the variables of the multivariate data prior to computing principal components. Standardizing the variables essentially makes them all equally important by creating new variables that each have a mean of zero and a variance of one. Proceeding in this way allows the principal components to be computed from the sample covariance matrix. IDL's PCOMP function includes COVARIANCE and STANDARDIZE keywords to provide this functionality.

For example, suppose that we wish to restate the following data using its principal components. There are three variables, each consisting of five samples.

	Var 1	Var 2	Var 3
Sample 1	2.0	1.0	3.0
Sample 2	4.0	2.0	3.0
Sample 3	4.0	1.0	0.0
Sample 4	2.0	3.0	3.0
Sample 5	5.0	1.0	9.0

Table 22-13: Data for Principal Component Analysis

We compute the principal components (the coefficients of the derived variables) to 2 decimal accuracy and store them by row in the following array.

$$\begin{bmatrix} 0.87 & -0.70 & 0.69 \\ 0.01 & -0.64 & -0.66 \\ 0.49 & 0.32 & -0.30 \end{bmatrix}$$

The derived variables $\{z_1, z_2, z_3\}$ are then computed as follows:

$$z_1 = (0.87) \begin{bmatrix} 2.0 \\ 4.0 \\ 4.0 \\ 2.0 \\ 5.0 \end{bmatrix} + (-0.70) \begin{bmatrix} 1.0 \\ 2.0 \\ 1.0 \\ 3.0 \\ 1.0 \end{bmatrix} + (0.69) \begin{bmatrix} 3.0 \\ 3.0 \\ 0.0 \\ 3.0 \\ 9.0 \end{bmatrix}$$

$$z_2 = (0.01) \begin{bmatrix} 2.0 \\ 4.0 \\ 4.0 \\ 2.0 \\ 5.0 \end{bmatrix} + (-0.64) \begin{bmatrix} 1.0 \\ 2.0 \\ 1.0 \\ 3.0 \\ 1.0 \end{bmatrix} + (-0.66) \begin{bmatrix} 3.0 \\ 3.0 \\ 0.0 \\ 3.0 \\ 9.0 \end{bmatrix}$$

$$z_3 = (0.49) \begin{bmatrix} 2.0 \\ 4.0 \\ 4.0 \\ 2.0 \\ 5.0 \end{bmatrix} + (0.32) \begin{bmatrix} 1.0 \\ 2.0 \\ 1.0 \\ 3.0 \\ 1.0 \end{bmatrix} + (-0.30) \begin{bmatrix} 3.0 \\ 3.0 \\ 0.0 \\ 3.0 \\ 9.0 \end{bmatrix}$$

In this example, analysis shows that the derived variable z_1 accounts for 57.3% of the total variance of the original data, the derived variable z_2 accounts for 28.2% of the total variance of the original data, and the derived variable z_3 accounts for 14.5% of the total variance of the original data.

Example of Derived Variables from Principal Components

The following example constructs an appropriate set of derived variables, based upon the principal components of the original data, which may be used to reduce the dimensionality of the data. The data consist of four variables, each containing of twenty samples.

```

; Define an array with 4 variables and 20 samples:
data = [[19.5, 43.1, 29.1, 11.9], $
        [24.7, 49.8, 28.2, 22.8], $
        [30.7, 51.9, 37.0, 18.7], $
        [29.8, 54.3, 31.1, 20.1], $
        [19.1, 42.2, 30.9, 12.9], $

```

```

[25.6, 53.9, 23.7, 21.7], $
[31.4, 58.5, 27.6, 27.1], $
[27.9, 52.1, 30.6, 25.4], $
[22.1, 49.9, 23.2, 21.3], $
[25.5, 53.5, 24.8, 19.3], $
[31.1, 56.6, 30.0, 25.4], $
[30.4, 56.7, 28.3, 27.2], $
[18.7, 46.5, 23.0, 11.7], $
[19.7, 44.2, 28.6, 17.8], $
[14.6, 42.7, 21.3, 12.8], $
[29.5, 54.4, 30.1, 23.9], $
[27.7, 55.3, 25.7, 22.6], $
[30.2, 58.6, 24.6, 25.4], $
[22.7, 48.2, 27.1, 14.8], $
[25.2, 51.0, 27.5, 21.1]]

```

The variables that will contain the values returned by the **COEFFICIENTS**, **EIGENVALUES**, and **VARIANCES** keywords to the **PCOMP** routine must be initialized as nonzero values prior to calling **PCOMP**.

```

coef = 1 & eval = 1 & var = 1
; Compute the derived variables based upon
; the principal components.
result = PCOMP(data, COEFFICIENTS = coef, $
  EIGENVALUES = eval, VARIANCES = var)
; Display the array of derived variables:
PRINT, result, FORMAT = '(4(f5.1, 2x))'

```

IDL prints:

```

 81.4   15.5   -5.5    0.5
102.7   11.1   -4.1    0.6
109.9   20.3   -6.2    0.5
110.5   13.8   -6.3    0.6
 81.8   17.1   -4.9    0.6
104.8    6.2   -5.4    0.6
121.3    8.1   -5.2    0.6
111.3   12.6   -4.0    0.6
 97.0    6.4   -4.4    0.6
102.5    7.8   -6.1    0.6
118.5   11.2   -5.3    0.6
118.9    9.1   -4.7    0.6
 81.5    8.8   -6.3    0.6
 88.0   13.4   -3.9    0.6
 74.3    7.5   -4.8    0.6
113.4   12.0   -5.1    0.6
109.7    7.7   -5.6    0.6
117.5    5.5   -5.7    0.6
 91.4   12.0   -6.1    0.6
102.5   10.6   -4.9    0.6

```

Display the percentage of total variance for each derived variable:

```
PRINT, var
```

IDL prints:

```
0.712422
0.250319
0.0370950
0.000164269
```

Display the percentage of variance for the first two derived variables; the first two columns of the resulting array above.

```
PRINT, TOTAL(var[0:1])
```

IDL prints:

```
0.962741
```

This indicates that the first two derived variables (the first two columns of the resulting array) account for 96.3% of the total variance of the original data, and thus could be used to summarize the original data.

Routines for Multivariate Analysis

Below is a brief description of IDL routines for multivariate analysis. More detailed information is available in the *IDL Reference Guide*.

CLUST_WTS	Computes the cluster weights of an array for cluster analysis.
CLUSTER	Performs cluster analysis.
CTI_TEST	Performs chi-square goodness-of-fit test.
KW_TEST	Performs Kruskal-Wallis H-test.
M_CORRELATE	Computes multiple correlation coefficient.
P_CORRELATE	Computes partial correlation coefficient.
PCOMP	Computes principal components/derived variables.
STANDARDIZE	Computes standardized variables.

Table 22-14: Multivariate Analysis Routines in IDL

References

Accuracy and Floating Point Operations

Burden, Richard L., J. Douglas Faires, and Albert C. Reynolds. *Numerical Analysis*. Boston: PWS Publishing, 1993. ISBN 0-534-93219-3

Stoer, J., and R. Bulirsch. *Introduction to Numerical Analysis*. New York: Springer-Verlag, 1980. ISBN 0-387-90420-4

Press, William H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Correlation Analysis

Harnet, Donald L. *Introduction to Statistical Methods*. Reading, Massachusetts: Addison-Wesley, 1975. ISBN 0-201-02752-6

Neter, John., William Wasserman, and G.A. Whitmore. *Applied Statistics*. Newton, Massachusetts: Allyn and Bacon, 1988. ISBN 0-205-10328-6

Press, William H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Curve and Surface Fitting

Bevington, Philip R. *Data Reduction and Error Analysis for the Physical Sciences*. New York: McGraw-Hill, 1969.

Lancaster, Peter and Kestutis Salkauskas. *Curve and Surface Fitting (An Introduction)*. San Diego: Academic Press, 1986. ISBN 0-124-36060-0

Eigenvalues and Eigenvectors

Press, William H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Strang, Gilbert. *Linear Algebra and Its Applications*. San Diego: Harcourt Brace Jovanovich, 1988. ISBN 0-155-551005-3

Gridding and Interpolation

Lancaster, Peter and Kestutis Salkauskas. *Curve and Surface Fitting (An Introduction)*. San Diego: Academic Press, 1986. ISBN 0-124-36060-0

Press, William H. *et al. Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Hypothesis Testing

Harnett, Donald H. *Introduction to Statistical Methods*. Reading, Massachusetts: Addison-Wesley, 1975. ISBN 0-201-02752-6

Kraft, Charles H. and Constance Van Eeden. *A Nonparametric Introduction to Statistics*. New York: Macmillan, 1968.

Sprent, Peter. *Applied Nonparametric Statistical Methods*. London: Chapman and Hall, 1989. ISBN 0-412-30600-X

Integration

Chapra, Steven C. and Raymond P. Canale. *Numerical Methods for Engineers*. New York: McGraw-Hill, 1988. ISBN 0-070-79984-9

Press, William H. *et al. Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Linear Systems

Golub, Gene H. and Van Loan, Charles F. *Matrix Computations*. Baltimore: Johns Hopkins University Press, 1989. ISBN 0-8018-3772-3

Kreyszig, Erwin. *Advanced Engineering Mathematics*. New York: Wiley & Sons, Inc., 1993. ISBN 0-471-55380-8

Press, William H. *et al. Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Strang, Gilbert. *Linear Algebra and Its Applications*. San Diego: Harcourt Brace Jovanovich, 1988. ISBN 0-155-551005-3

Nonlinear Equations

Dennis, J.E. Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall, 1983. ISBN 0-136-27216-9

Press, William H. *et al. Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Optimization

Dennis, J.E. Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Englewood Cliffs, NJ: Prentice-Hall, 1983. ISBN 0-136-27216-9

Press, William H. *et al. Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Sparse Arrays

Press, William H. *et al. Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1992. ISBN 0-521-43108-5

Time-Series Analysis

Chatfield, C. *The Analysis of Time Series*. London: Chapman and Hall, 1975. ISBN 0-412-31820-2

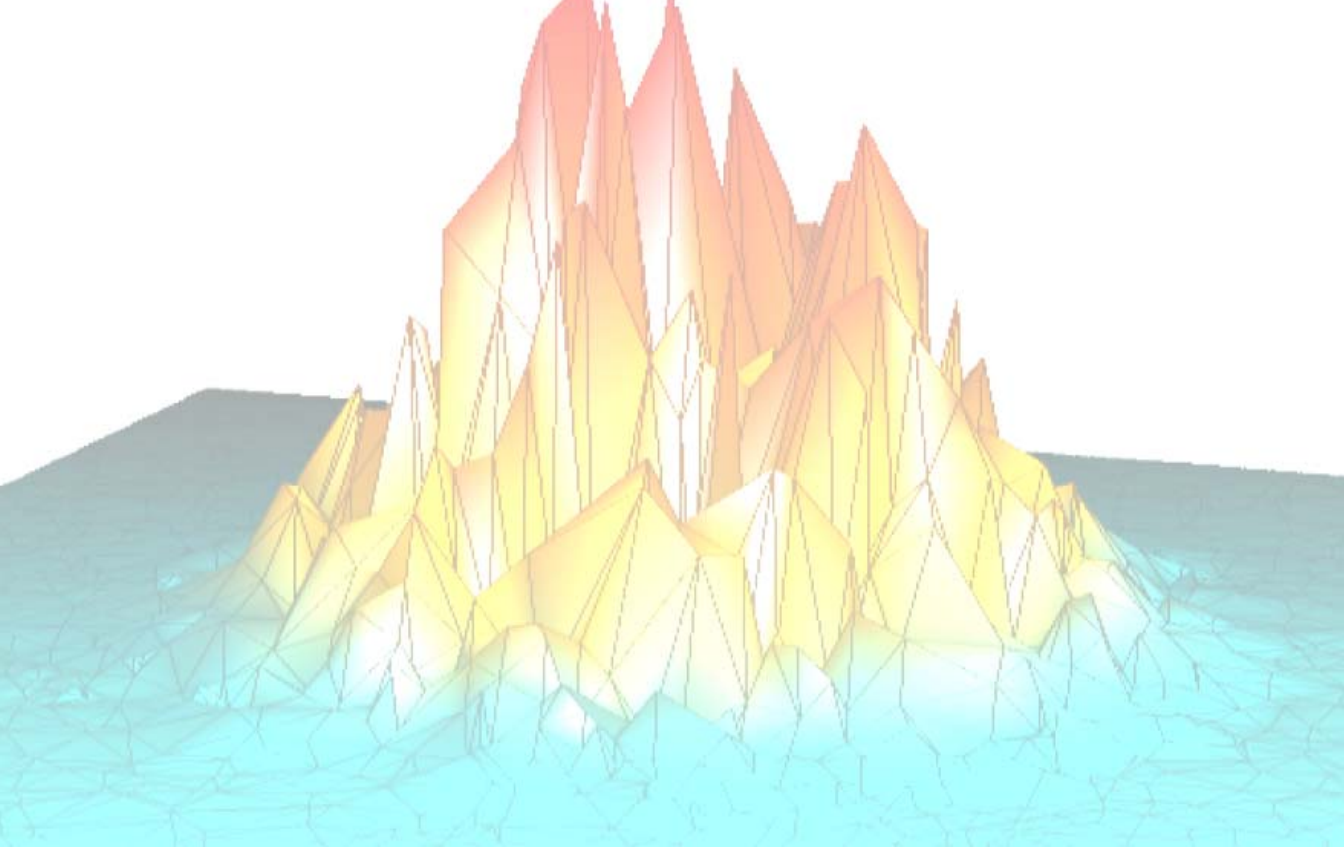
Neter, John., William Wasserman, and G.A. Whitmore. *Applied Statistics*. Newton, Massachusetts: Allyn and Bacon, 1988. ISBN 0-205-10328-6

Multivariate Analysis

Jackson, Barbara Bund. *Multivariate Data Analysis*. Homewood, Illinois: R.D. Irwin, 1983. ISBN 0-256-02848-6

Everitt, Brian S. *Cluster Analysis*. New York: Halsted Press, 1993. ISBN 0-470-22043-0

Kachigan, Sam Kash. *Multivariate Statistical Analysis*. New York: Radius Press, 1991. ISBN 0-942154-91-6



Part IV: Object Graphics



Chapter 23: Object Graphics

This chapter discusses the difference between IDL Direct Graphics and IDL Object Graphics, and provides an overview of the IDL Object Graphics classes.

Overview	500	Attribute Objects	514
Direct versus Object Graphics	502	Helper Objects	515
How to Use Object Graphics	504	Destination Objects	516
Overview of Object Graphics Classes	506	File Format Objects	517
Container Objects	508	Properties of Objects	519
Structure Objects	509	Alpha Channel and Objects	521
Atomic Graphic Objects	510	Undocumented Graphic Object Classes ..	527
Composite Objects	513		

Overview

The IDL Object Graphics system is a collection of pre-defined object classes, each of which is designed to encapsulate a particular visual representation. Actions (such as the modification of attributes, or data picking) may be performed on instances of these object classes by calling corresponding pre-defined methods. These objects are designed for building complex three-dimensional data visualizations.

For example, the `IDLgrAxis` object provides an encapsulation of all of the components associated with a graphical representation of an axis. One of the actions that can be performed on an axis is retrieving the current value of one or more of its attributes (such as its color, tick values, or data range). This action may be performed via the `IDLgrAxis::GetProperty` method. A complete listing of the types of objects included in the Object Graphics system are described beginning in [“Overview of Object Graphics Classes”](#) on page 506.

Object Graphics should be thought of as a collection of building blocks. In order to display something on the screen, the user selects the appropriate set of blocks and puts them together so that as a group they provide a visual result. In this respect, Object Graphics are quite different than Direct Graphics. A single line of code is unlikely to produce a complete visualization. Furthermore, a basic understanding of the IDL object system is required (for instance, how to create an object, how to call a method, how to destroy an object, etc.). Because of the level at which these objects are presented, Object Graphics are aimed at application programmers rather than command line users.

Object Graphics do not interact in any way with the system variables (such as `!P`, `!X`, `!Y`, and `!Z`). Each graphic object is intended to encapsulate all of the information required to fully describe itself. Reliance on external structures is not condoned. The advantage of this approach is that once an object is created, it will always behave in the same way even if the system state is modified by another program, or if the object is moved to another user’s IDL session, where the system state may have been customized in a different way than the state in which the object was originally defined.

Object Graphics are designed for building interactive three-dimensional visualization applications. Direct manipulation tools (such as the `Trackball` object) are provided to aid the application developer. Selection and data picking are also built in, so the developer can spend less time working out data projection issues and more time focusing on domain specific data analysis and visualization features.

Over time, Research Systems, Inc., will continue to build higher-level applications with these objects, applications that are suitable for users who prefer not to become programmers to interact with their data. The IDL Intelligent Tools (iTools) are good examples of currently available applications built using Object Graphics. For more information, see the *iTools User's Guide*.

Additional examples based on Object Graphics can be found in the IDL demo.

Direct versus Object Graphics

Beginning with IDL version 5.0, IDL supports two distinct graphics modes: Direct Graphics and Object Graphics. Direct Graphics rely on the concept of a current graphics device; IDL commands like PLOT or SURFACE create images directly on the current graphics device. Object Graphics use an object-oriented programmers' interface to create graphic objects, which must then be drawn, explicitly, to a destination of the programmers choosing.

IDL Direct Graphics

If you have used versions of IDL prior to version 5.0, you are already familiar with IDL Direct Graphics. The salient features of Direct Graphics are:

- Direct Graphics use a graphics device ('X' for X-windows systems displays, 'WIN' for Microsoft Windows displays, 'PS' for PostScript files, etc.). You switch between graphics devices using the SET_PLOT command, and control the features of the current graphics device using the DEVICE command.
- IDL commands that existed in IDL 4.0 use Direct Graphics. Commands like PLOT, SURFACE, XYOUTS, MAP_SET, etc. all draw their output directly on the current graphics device.
- Once a direct-mode graphic is drawn to the graphics device, it cannot be altered or re-used. This means that if you wish to re-create the graphic on a different device, you must re-issue the IDL commands to create the graphic.
- When you add a new item to an existing direct-mode graphic (using a routine like OPLOT or XYOUTS), the new item is drawn in front of the existing items.

Documentation for IDL Direct Graphics routines is found in the following volumes of the IDL Documentation set: *Using IDL*, *Building IDL Applications*, and the *IDL Reference Guide*.

IDL Object Graphics

Versions of IDL beginning with version 5.0 include Object Graphics in addition to Direct Graphics. The salient features of Object Graphics are:

- Object graphics are rendered in three dimensions. Rendering implies many operations not needed when drawing Direct Graphics, including calculation of normal vectors for lines and surfaces, lighting considerations, and general object overhead. As a result, the time needed to render a given object—a

surface, for example—will often be longer than the time taken to draw the analogous image in Direct Graphics.

- Object graphics are device independent. There is no concept of a current graphics device when using object-mode graphics; any graphics object can be displayed on any physical device for which a destination object can be created.
- Object graphics are object oriented. Graphic objects are meant to be created and re-used; you may create a set of graphic objects, modify their attributes, draw them to a window on your computer screen, modify their attributes again, then draw them to a printer device without reissuing all of the IDL commands used to create the objects. Graphics objects also encapsulate functionality; this means that individual objects include method routines that provide functionality specific to the individual object.
- Object Graphics use a programmers interface. Unlike Direct Graphics, which are well suited for both programming and interactive, ad hoc use, Object Graphics are designed to be used in programs that are compiled and run. While it is still possible to create and use graphics objects directly from the IDL command line, the syntax and naming conventions make it more convenient to build a program off line than to create graphics objects on the fly.
- Because Object Graphics persist in memory, there is a greater need for the programmer to be cognizant of memory issues and memory leakage. Efficient design—remembering to destroy unused object references and cleaning up—will avert most problems, but even the best designs can be memory-intensive if large numbers of graphic objects (or large datasets) are involved.

Explanatory material on IDL's object system is contained in [Chapter 23, “Object Basics”](#) in the *Building IDL Applications* manual. For reference material describing IDL's object classes, see the [“Object Class and Method Reference”](#) in the *IDL Reference Guide* manual.

How to Use Object Graphics

All Object Graphics applications require at least two basic building blocks. These include:

- A destination object - the device (such as a window, memory buffer, file, clipboard, or printer) to which the visualization is to be rendered. For more information, see “[Destination Objects](#)” on page 516
- A view object - the viewport rectangle (within the destination) within which the rendering is to appear (as well as how data should be projected into that rectangle).

For example:

```
;Create a destination object, in this case a window:
oWindow = OBJ_NEW('IDLgrWindow')
;Create a viewport that fills the entire window:
oView = OBJ_NEW('IDLgrView')
;Draw the view within the window:
OWindow->Draw, oView
```

By themselves, a window and a single view are not particularly enlightening, but you will find that these two types of objects are utilized by all Object Graphics applications. To change an attribute of an object, you do not have to create a new instance of that object. Instead, use the SetProperty method on the original object to modify the value of the attribute.

For example, to change the color of the view to gray:

```
;Set the color property of the view:
OView->SetProperty, COLOR=[60,60,60]
;Redraw:
OWindow->Draw, oView
```

If more than one view is to be drawn to the destination, then an additional object is required:

- A scene object - a container of views

For example:

```
; Create a scene and add our original view to it:
OScene = OBJ_NEW('IDLgrScene')
oScene->Add, oView
; Modify our original view so that it covers
; the upper left quadrant of the window.
OView->SetProperty, LOCATION=[0.0,0.5], DIMENSIONS=[0.5,0.5], $
    UNITS=3
```



```

; Create and add a second red view that covers
; the right half of the window.
OView2 = OBJ_NEW('IDLgrView', LOCATION=[0.5,0.0], $
    DIMENSIONS=[0.5,1.0], UNITS=3,COLOR=[255,0,0])
OScene->Add, oView2
; Now draw the scene, rather than the view, to the window:
OWindow->Draw, oScene

```

In the examples so far, the views have been empty canvases. For data visualization applications, these views will need some graphical content. To draw visual representations within the views, two additional types of objects are required:

- A model object - a transformation node
- An atomic graphic object - a graphical representation of data (such as an axis, plot line, or surface mesh). For more information, see [“Atomic Graphic Objects”](#) on page 510.

For example, to include a text label within a view:

```

; Create a model and add it to the original view:
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel
; Create a text object and add it to the model:
oText = OBJ_NEW('IDLgrText', 'Hello World', ALIGNMENT=0.5)
oModel->Add, oText
; Redraw the scene:
OWindow->Draw, oScene

```

Notice that the scene, views, model, and text are all combined together into a self-contained hierarchy. It is the overall hierarchy that is drawn to the destination object.

The transformation associated with the model can be modified to impact the text it contains. For example:

```

; Rotate by 90 degrees about the Z-axis:
oModel->Rotate, [0,0,1], 90
; Redraw:
OWindow->Draw, oScene

```

When the objects are no longer required, they need to be destroyed. Destination objects must be destroyed separately, but the graphic hierarchies can be destroyed in full by simply destroying the root of the hierarchy. For example:

```

OBJ_DESTROY, oWindow
OBJ_DESTROY, oScene

```

In this example, the destruction of the scene will cause the destruction of all of its children (including the views, model, and text).

Overview of Object Graphics Classes

The following sections provide an overview of the different types of objects included in the IDL Object Graphics class library. In order to describe the attributes of the IDL Object Graphics classes, we have grouped the objects into functional categories: [Container Objects](#), [Structure Objects](#), [Atomic Graphic Objects](#), [Composite Objects](#), [Attribute Objects](#), [Helper Objects](#), [Destination Objects](#), and [File Format Objects](#).

Note

These category names are purely descriptive; for example, structure objects contain the `IDLgrModel`, `IDLgrScene`, and `IDLgrView` classes, but no class named `structure`. There is one exception to this rule: the container objects category which includes the `IDL_Container` class.

This chapter does not describe the relationships between object classes. See [Chapter 24, “The Graphics Object Hierarchy”](#) for a discussion of the object tree.

Naming Conventions

In general, object classes shipped with IDL have names of the form:

`IDLxxYyyy`

where `xx` represents the broad functional grouping (*gr* for graphics objects, *db* for database objects, and *an* for analysis, for example). `Yyyy` is the class name itself (such as *Axis* or *Surface*). Object classes that are useful in more than one functional context (container objects, for example) omit the functional grouping code entirely (`IDL_Container`). All object classes shipped with IDL are prepended with the letters `IDL`—we strongly suggest that you do not use this prefix when writing your own object classes, as we will continue to add new object classes using this convention.

The typographical convention used to describe IDL objects is slightly different from that used for non-object functions and procedures. Whereas non-object procedures are presented in upper case letters, object classes and methods use mixed case. For example, we refer to the `PLOT` routine, but to the `IDLgrPlot` object. Method names are also presented in mixed case (`IDLgrAxis::GetProperty`).

Common Methods

In addition to their own specific methods, all object classes shipped with IDL except for the `IDL_Container` class have four methods in common: `Cleanup`, `Init`, `GetProperty`, and `SetProperty`. The `Cleanup` and `Init` methods are life-cycle methods, and cannot be called directly except within a subclass' `Cleanup` or `Init` method. (See “[The Object Lifecycle](#)” in Chapter 23 of the *Building IDL Applications* manual.) The `GetProperty` and `SetProperty` methods allow you to inspect (get) or change (set) the various properties associated with a given object. Properties associated with graphics objects include things like color, location, line style, or data.

Container Objects

IDL's container object, realized in the `IDL_Container` class, provides a way to group disparate IDL objects into single object. Container objects provide a convenient way to move or destroy groups of objects; when a container is destroyed, its `Cleanup` method automatically calls the `Cleanup` methods of all the objects in the container and destroys them as well.

See “[IDL_Container](#)” in the *IDL Reference Guide* manual for details.

Structure Objects

Structure objects create a hierarchy of graphic objects—an object tree. Structure objects also contain the information necessary to transform graphics objects in space. Building an object tree allows you to manipulate groups of graphic objects easily by transforming a single `IDLgrModel` object to which members of the group belong. (See [Chapter 24, “The Graphics Object Hierarchy”](#) for a discussion of the object tree.)

Model

Objects of the `IDLgrModel` class serve as containers for individual graphic objects (plot lines, axes, text, etc.) and for other model objects. Model objects include a three-dimensional transformation matrix that describes how the model and all of its components are positioned in space. Altering the model’s transformation matrix changes the position and orientation of any objects the model contains. If a model object contains another model object, the contained model is positioned according to both its own transformation matrix and that of its container.

See [“IDLgrModel”](#) in the *IDL Reference Guide* manual for further details.

View

Objects of the `IDLgrView` class serve as containers for model objects. A view object can be supplied as the argument to a `Draw` method.

See [“IDLgrView”](#) in the *IDL Reference Guide* manual for further details.

Viewgroup

Objects of the `IDLgrViewgroup` class serve as containers for views. A viewgroup object can be supplied as the argument to a `Draw` method.

See [“IDLgrViewgroup”](#) in the *IDL Reference Guide* manual for further details.

Scene

Objects of the `IDLgrScene` class serve as containers for view and view group objects. A scene object can be supplied as the argument to a `Draw` method.

See [“IDLgrScene”](#) in the *IDL Reference Guide* manual for further details.

Atomic Graphic Objects

Atomic Graphic Objects, or graphics atoms, are the low-level objects used to create images. Graphics atoms have attributes such as size, color, width, or associated color palette. Graphics atoms do not include a transformation matrix and do not contain other objects.

Axis

Objects of the `IDLgrAxis` class are individual axes. One axis object is required for each axis line to be rendered.

See “[IDLgrAxis](#)” in the *IDL Reference Guide* manual for further details.

Contour

Objects of the `IDLgrContour` class are lines representing contour information plotted from user data.

See “[IDLgrContour](#)” in the *IDL Reference Guide* manual for further details.

Image

Objects of the `IDLgrImage` class are two-dimensional arrays of data with an associated mapping of the data values to pixel values.

See “[IDLgrImage](#)” in the *IDL Reference Guide* manual for further details.

Light

Objects of the `IDLgrLight` class are light sources by which atomic graphic objects are illuminated. Light objects are not actually rendered, but are included as graphics atoms (meaning they must be contained in a model object) so that they can be positioned and transformed along with the graphic objects they illuminate. If no light object is included in a particular view, default lighting is supplied.

See “[IDLgrLight](#)” in the *IDL Reference Guide* manual for further details.

Plot

Objects of the `IDLgrPlot` class are individual plot lines, created from a user-supplied vector of dependent data values (and, optionally, a vector of independent data values). Plot objects do not include axes.

See “[IDLgrPlot](#)” in the *IDL Reference Guide* manual for further details.

Polygon

Objects of the `IDLgrPolygon` class are individual polygons, created from a user-supplied array of data values.

See “[IDLgrPolygon](#)” in the *IDL Reference Guide* manual for further details.

Polyline

Objects of the `IDLgrPolyline` class are individual polylines, created from a user-supplied array of data points. Locations of the data points supplied are connected by a single line.

See “[IDLgrPolyline](#)” in the *IDL Reference Guide* manual for further details.

Region of Interest

Objects of the `IDLgrROI` class are representations of a *region of interest*. Regions of interest are described as a set of vertices that may be connected to generate a path or a polygon, or may be treated as separate points.

See “[IDLgrROI](#)” in the *IDL Reference Guide* manual for further details.

Objects of the `IDLgrROIGroup` class are representations of a group of regions of interest.

See “[IDLgrROIGroup](#)” in the *IDL Reference Guide* manual for further details.

Surface

Objects of the `IDLgrSurface` class are individual three-dimensional surfaces, created from a user-supplied array of data values.

See “[IDLgrSurface](#)” in the *IDL Reference Guide* manual for further details.

Text

Objects of the `IDLgrText` class are text strings that can be positioned within the rendering area.

See “[IDLgrText](#)” in the *IDL Reference Guide* manual for further details.

Volume

Objects of the `IDLgrVolume` class map a three-dimensional array of data values to a three-dimensional array of voxel colors, which, when drawn, are projected to two dimensions.

See “[IDLgrVolume](#)” in the *IDL Reference Guide* manual for further details.

Composite Objects

A composite object is an encapsulation of a group of other objects that together provide a commonly useful graphical representation.

Colorbar

Objects of the `IDLgrColorbar` class are annotations that provide information about the data values associated with colors used in a visualization.

See “[IDLgrColorbar](#)” in the *IDL Reference Guide* manual for further details.

Legend

Objects of the `IDLgrLegend` class are annotations that provide information about the meaning of individual data items or lines in a visualization.

See “[IDLgrLegend](#)” in the *IDL Reference Guide* manual for further details.

Attribute Objects

Attribute objects are used when rendering graphic objects, but exist outside the hierarchy of Model-View-Scene objects that are actually rendered.

Font

Objects of the `IDLgrFont` class define the typeface, size, weight, and style of text used when rendering a text object.

See “[IDLgrFont](#)” in the *IDL Reference Guide* manual for further details.

Palette

Objects of the `IDLgrPalette` class define a color lookup table that maps indices to red, green, and blue values.

See “[IDLgrPalette](#)” in the *IDL Reference Guide* manual for further details.

Pattern

Objects of the `IDLgrPattern` class defines which pixels are filled and which are left blank when a graphic object is filled.

See “[IDLgrPattern](#)” in the *IDL Reference Guide* manual for further details.

Symbol

Objects of the `IDLgrSymbol` class define graphical element that can be used when plotting data.

See “[IDLgrSymbol](#)” in the *IDL Reference Guide* manual for further details.

Helper Objects

Helper objects alter data in useful ways or provide other services. They exist outside the hierarchy of Model-View-Scene objects that are actually rendered.

Tessellator

Objects of the `IDLgrTessellator` class convert a simple concave polygon (or a simple polygon with holes) into a number of simple convex polygons (general triangles). Tessellation is useful because IDL's polygon object handles only convex polygons.

See "[IDLgrTessellator](#)" in the *IDL Reference Guide* manual for further details.

TrackBall

Objects of the `TrackBall` class provide a simple interface to allow the user to translate and rotate three-dimensional Object Graphics hierarchies displayed in an IDL `WIDGET_DRAW` window using the mouse.

See "[TrackBall](#)" in the *IDL Reference Guide* manual for further details.

Destination Objects

Destination objects are objects on which object trees can be rendered (displayed on a screen or printed on a printer).

Buffer

Objects of the `IDLgrBuffer` class represent an off-screen, in-memory data area that may serve as a graphics source or destination.

See “[IDLgrBuffer](#)” in the *IDL Reference Guide* manual for further details.

Clipboard

Objects of the `IDLgrClipboard` class send Object Graphics to the operating system’s native clipboard or to a file in bitmap or vector format.

See “[IDLgrClipboard](#)” in the *IDL Reference Guide* manual for further details.

Printer

Objects of the `IDLgrPrinter` class represent a hardcopy graphics destination. By default, printer objects represent the default system printer; you can use the IDL routines `DIALOG_PRINTJOB` and `DIALOG_PRINTERSETUP` to change the printer associated with a printer object.

See “[IDLgrPrinter](#)” in the *IDL Reference Guide* manual for further details.

VRML

Objects of the `IDLgrVRML` class allow you to save the contents of an Object Graphics hierarchy as a VRML 2.0 format file.

See “[IDLgrVRML](#)” in the *IDL Reference Guide* manual for further details.

Window

Objects of the `IDLgrWindow` class represent an on-screen area on a display device in which graphic objects can be rendered.

See “[IDLgrWindow](#)” in the *IDL Reference Guide* manual for further details.

File Format Objects

DICOM

Objects of the `IDLffDICOM` class contain the data for one or more images embedded in a DICOM Part 10 file.

See “[IDLffDICOM](#)” in the *IDL Reference Guide* manual for further details.

DXF

Objects of the `IDLffDXF` class contain geometry, connectivity and attributes for graphics primitives.

See “[IDLffDXF](#)” in the *IDL Reference Guide* manual for further details.

JPEG 2000

Objects of the `IDLffJPEG2000` class provide an interface to files in the JPEG 2000 format.

See “[IDLffJPEG2000](#)” in the *IDL Reference Guide* manual for further details.

Language Catalogs

Objects of the `IDLffLangCat` class provide an interface to IDL language catalog files.

See “[IDLffLangCat](#)” in the *IDL Reference Guide* manual for further details.

MrSID

Objects of the `IDLffMrSID` class are used to query information about and load image data from a MrSID (`.sid`) image file.

See “[IDLffMrSID](#)” in the *IDL Reference Guide* manual for further details.

MPEG

Objects of the `IDLgrMPEG` class allow you to save an array of image frames as an MPEG movie.

See “[IDLgrMPEG](#)” in the *IDL Reference Guide* manual for further details.

Shape Files

Objects of the `IDLffShape` class contain geometry, connectivity and attributes for graphics primitives accessed from ESRI Shapefiles.

See “[IDLffShape](#)” in the *IDL Reference Guide* manual for further details.

VRML

Objects of the `IDLgrVRML` class allow you to save the contents of an Object Graphics hierarchy as a VRML 2.0 format file.

See “[IDLgrVRML](#)” in the *IDL Reference Guide* manual for further details.

XML

Objects of the `IDLffXMLSAX` class represent an XML SAX level 2 parser. The XML parser allows you to read an XML file and store arbitrary data from the file in IDL variables.

See “[IDLffXMLSAX](#)” in the *IDL Reference Guide* manual for further details.

Properties of Objects

IDL's graphics objects have a number of associated properties—things like color, line style, size, etc. Properties are set or changed via keywords to the object's `Init` method (specified when the object is created) or to the object's `SetProperty` method. If you are familiar with IDL Direct Graphics, many of the keywords used by IDL Object Graphics will be familiar to you. Note, however, that unlike IDL Direct Graphics, the IDL Object Graphics system allows you to change the value of an object's properties without re-creating the entire object. (Objects must be redrawn, however, with a call to the destination object's `Draw` method, for the changes to become visible.)

Setting Properties at Initialization

Often, you will set an object's properties when creating the object for the first time. Do this by specifying any keywords to the object's `Init` method directly in the call of `OBJ_NEW` that creates the object. For example, suppose you are creating a plot and wish to use a red line to draw the plot line. You could specify the `COLOR` keyword to the `IDLgrPlot::Init` method directly in the call to `OBJ_NEW`:

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata, COLOR=[255, 0, 0])
```

Remember that in most cases, an object's `Init` method cannot be called directly. Arguments to `OBJ_NEW` are passed directly to the `Init` method when the object is created.

For some graphics objects, you can specify a keyword that has the same meaning as an argument. In Object Graphics, the value of the keyword overrides the value set by the argument. For example,

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata, DATAX=newXData)
```

The `Plot` object uses the data in `newXData` for the plot's X data.

Setting Properties of Existing Objects

After you have created an object, you can set its properties using the object's `SetProperty` method. For example, the following two statements duplicate the single call to `OBJ_NEW` shown above:

```
myPlot = OBJ_NEW('IDLgrPlot', xdata, ydata)
myPlot->SetProperty, COLOR=[255, 0, 0]
```

Note

Not all keywords available when the object is being initialized are necessarily available via the SetProperty method. Keywords available when using an object's SetProperty method are noted with the word "Yes" in "Set" column of the property table. Keywords with the word "Yes" in the "Init" column can be set via the object's Init method. Available object properties of each object are detailed in the *IDL Reference Guide*.

Retrieving Property Settings

You can retrieve the value of a particular property using an object's GetProperty method. The GetProperty method accepts a list of keyword-variable pairs and returns the value of the specified properties in the variables specified. For example, to return the value of the COLOR property of the plot object in our example, use the statement:

```
myPlot->GetProperty, COLOR=plotcolor
```

This returns the value of the COLOR property in the IDL variable plotcolor.

You can retrieve the values of all of the properties associated with a graphics object by using the ALL keyword to the object's GetProperty method. The following statement:

```
myPlot->GetProperty, ALL=allprops
```

returns an anonymous structure in the variable allprops; the structure contains the values of all of the retrievable properties of the object.

Note

Not all keywords available when the object is being initialized are necessarily available via the GetProperty method. Keywords with the word "Yes" in the "Get" column can be retrieved via the object's GetProperty method. Available object properties of each object are detailed in the *IDL Reference Guide*.

When querying Object Graphics object properties with GetProperty, properties that are arrays return -1 when the property is in an empty or inactive state. For example, the VERT_COLORS property of IDLgrPolygon can be turned off or disabled by setting it to any scalar value. A subsequent query of this property returns a scalar value of -1 to indicate that the VERT_COLORS array is empty and not being used by the object.

Alpha Channel and Objects

IDL contains support for Alpha Channel in Object Graphics. Objects which support Alpha Channel in IDL are:

- IDLgrAxis
- IDLgrContour
- IDLgrPlot
- IDLgrPolygon
- IDLgrPolyline
- IDLgrROI
- IDLgrSurface
- IDLgrSymbol
- IDLgrText
- IDLgrVolume

The alpha channel has many uses. One of the most important is drawing primitives semi-transparently, which can be used to enhance your object graphics scene. An example might be a text label drawn semi-transparently to let other graphical details “show through” the text label. This would allow you to use a larger text font size, rather than using a small font size to squeeze text between details in a scene. Another use for alpha channel might be to draw polygons and surfaces semi-transparently, allowing you to see “inside” certain objects and structures.

Some of the most important uses for semi-transparent rendering are discussed in the following sections.

- [“Opacity and Transparency”](#) on page 522
- [“Blending Mathematics”](#) on page 522
- [“Rendering Order”](#) on page 523
- [“Viewing and Rotation”](#) on page 525
- [“Depth Buffer Updating”](#) on page 526

Opacity and Transparency

Opacity describes the degree to which an object blocks the appearance of other objects. In IDL, the value used for the ALPHA_CHANNEL properties in IDLgr* objects is a measure of the object's opacity. A value of 1.0 indicates complete opacity. The object completely blocks the appearance of other objects. Conversely, an opacity value of 0.0 indicates that the object does not block the appearance of objects at all. Intermediate values indicate varying degrees of visibility for covered objects.

Transparency is essentially the opposite of opacity. Transparency indicates the degree to which an object does not block the appearance of other objects. Complete or full transparency is indicated by an opacity value of 0.0, while an object that is not transparent at all has an opacity value of 1.0.

By default all IDLgr* graphic objects use an ALPHA_CHANNEL value of 1.0, indicating full opacity (zero transparency), matching the rendering behavior before the addition of the ALPHA_CHANNEL property. To change the opacity of the object, simply change the this property to a value between 0.0 (zero opacity or full transparency) and 1.0.

Blending Mathematics

Blending is the drawing of semi-transparent objects on a screen already containing objects. During rendering, the color of the pixels belonging to the primitive being rendered are blended with the color of the pixels that are already on the screen, producing the desired blending effect. This process is accomplished on a pixel-by-pixel basis.

IDL uses this well-established blending equation:

$$\text{newColor} = \text{oldColor} * (1 - \text{alpha}) + \text{primitiveColor} * \text{alpha}$$

An example might suppose that you want to draw a red square in an area of the screen that is completely green. By default, the alpha value is 1.0, so the result is:

$$[255, 0, 0] = [0, 255, 0] * (1.0 - 1.0) + [255, 0, 0] * 1.0$$

The green color is removed completely and replaced by red, the expected result of conventional non-blended rendering.

If the alpha value is changed to equal 0.5, the result is:

$$[127, 127, 0] = [0, 255, 0] * (1.0 - 0.5) + [255, 0, 0] * 0.5$$

The resulting color is the half of the red of the polygon combined with half of the green of the background, a pale yellow.

If you draw another red square in the same place with the same alpha, the red square is blended with the now current contents of the screen:

$$[190, 63, 0] = [127, 127, 0] * (1.0-0.5) + [255, 0, 0] * 0.5$$

Note

Large levels of semi-transparent rendering may reduce rendering performance. This is because the graphics blending operation that is performed involves reading the destination pixel from the frame buffer, combining it with the new color value and then writing the result back to the frame buffer. This is more expensive than simply overwriting the frame buffer contents with the new color value. The degree to which your performance will be impacted depends heavily on the hardware and software components of your graphics system.

Rendering Order

The colors of the pixels on the screen are important when drawing a blended primitive. Similarly, the order in which the primitives are drawn is also very important when drawing scenes with blended primitives.

In computer graphics, depth sorting presents a similar challenge. Without depth sorting, a scene would have to be drawn from back to front to obtain a correct result. IDL handles depth sorting by providing a "depth buffer" (also known as a "Z-buffer") allowing you to draw the primitives in any order while allowing the primitives closer to the viewer to still appear to be on top.

There is no similar feature for alpha-blended primitives. Be sure to draw the blended primitives carefully so that all primitives behind a blended primitive are drawn before the blended primitive.

If your scene consists of many primitives that are not blended and a few text labels that are drawn with blending, it is a good idea to defer the drawing of the labels until after everything else is drawn. This will allow users to see through all labels and to see the objects beneath. If a non-blended primitive is drawn on top of and after a blended primitive, it will cover the blended primitive. If any primitive is drawn behind but after a blended primitive, the primitive drawn later will not appear where the blended primitive covers it, due to depth buffering. In other words, it is not possible to blend primitives unless all objects behind the blended primitive which are to be blended are already drawn.

Note

If you have a complex scene where many primitives are blended, it may be difficult to determine the proper ordering.

Inter- and Intra-primitive Rendering Orders

Inter-primitive rendering order deals with the ordering of primitive objects within an `IDLgrModel`. For primitives which do not intersect each other, it is straightforward to order these in a back-to-front viewing order, particularly if your scene is fixed so it cannot be rotated by the user. This is done by arranging your primitives along the *Z* direction so that the objects farthest away appear first in the `IDLgrModels`, which makes them draw first.

If primitives intersect, it may be necessary to divide the object so that the back parts of each primitive are drawn first, and then the front parts. This can be a very difficult issue.

Intra-primitive rendering order deals with the ordering of graphical items within an `IDL` graphics primitive. Some primitives, such as `IDLgrSurface` and `IDLgrPolygon` actually consist of a large number of individual polygons. They are not all drawn at once, and the order in which they are drawn is also important when drawing with blending.

You can control the order in which the individual polygons are drawn in an `IDLgrPolygon` object by ordering the vertices or specifying the order in the `POLYGONS` property. Polygons specified first in the `POLYGONS` list are drawn first.

Viewing and Rotation

If you draw a typical height field with `IDLgrSurface` and invoke blending, the object might look right from some viewing orientations.

For example, try the following:

```
XOBJVIEW, OBJ_NEW('idlgrsurface', $
  BESELJ(shift(dist(40),20,20)/2,0) * 20, STYLE=2, $
  ALPHA_CHANNEL=0.5)
```

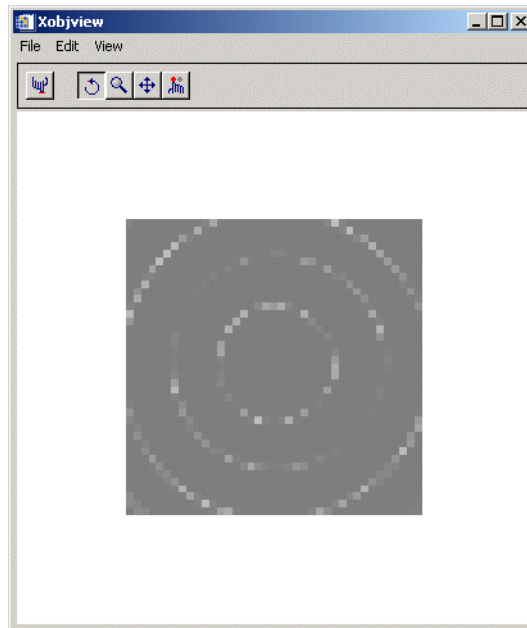


Figure 23-1: Viewing Alpha Channel in an Object

Notice in the previous figure that you can see-through the waves in the object to see other waves, but only when you view the object from certain directions. From other directions, all you see are the waves closer to you.

Solving this problem can be extremely difficult. A complete solution would generate a scene for every possible viewing angle, where the polygons are drawn back to front, splitting them if necessary. There are several techniques available for accomplishing this, one of them being the Binary Space Partition Tree, however this is not supported

directly in IDL. If the objects are simple, it might be possible to construct a few scenes that give correct or passable results.

For example, if you wanted to look at a semi-transparent sphere from all angles, creating eight models might suffice. Each of the eight models contains the polygons sorted in back-to-front order for a viewing direction corresponding to each of the eight octants formed by the half spaces of the three principle axes. As the user rotates the scene with a trackball, the program would select the appropriate model, based on the current viewing direction. More complex scenes may require more models.

Depth Buffer Updating

For any value of the `ALPHA_CHANNEL` property, IDL updates the depth buffer when the primitive is drawn, unless the `DEPTH_TEST_FUNCTION` property is set to a value that disables depth buffer updates. Thus, even if you draw a completely transparent primitive, the depth buffer is updated as if there were a visible primitive drawn there. This means that subsequent primitives drawn behind the transparent object are not visible. Though potentially confusing, this can also be a useful way to hide objects in certain situations.

After drawing a transparent object, that there may be gaps in objects drawn later. For example, supposed lines in a plot are drawn with `ALPHA_CHANNEL=0` (transparent), and then symbols are drawn. Where the symbols and lines intersect, there are gaps in the symbols. The gaps are caused by the invisible lines changing the depth buffer, thus masking out the symbols that are drawn later. At times, the ability to modify the depth buffer without changing the color buffer is a useful tool for clever clipping operations. In other contexts, you may consider using invisible polygons to mask out entire areas. However, if the partial or entire invisibility of objects drawn after a transparent object is unintended use one of the following options:

- Set the `DEPTH_TEST_FUNCTION=4`, or disable depth testing entirely.
- Set the `HIDE` property to 1 if `ALPHA_CHANNEL` becomes 0.

Either of these options would erase the gaps in the symbols caused by the transparent plot lines as described in the previous situation.

Undocumented Graphic Object Classes

Several of IDL's graphics objects are subclassed from more generic IDL objects. You may see references to the generic IDL objects when using IDL's HELP procedure to get information on an object, or when you use the OBJ_ISA or OBJ_CLASS functions. You may also notice that the generic objects are not documented in the ["Object Class and Method Reference"](#) in the *IDL Reference Guide* manual. This is not an oversight.

We have chosen not to document the workings of the more generic objects from which the IDL graphics objects are subclassed because we reserve the right to make changes to their operation. We strongly recommend that you do not use the undocumented object classes directly, or subclass your own object classes from them. Research Systems, Inc. does not guarantee that user-written code that uses undocumented features will continue to function in future releases of IDL.



Chapter 24: The Graphics Object Hierarchy

The following topics are covered in this chapter:

Overview	530	Atomic Graphic Objects	535
Scenes	531	Attribute and Helper Objects	536
Viewgroups	532	The Rendering Process	537
Views	533	Simple Plot Example	539
Models	534		

Overview

In this chapter we will discuss the organization of a group of graphics objects into a hierarchy or tree. A graphics tree may have any number of branches, each of which in turn may have any number of sub-branches, etc.

For example, a graphics object tree with four graphics atoms might be contained in three separate model objects, which are in turn contained in two distinct view objects, both of which are contained in one scene object. In this example (shown in the figure below), the scene object is the root of the graphics tree.

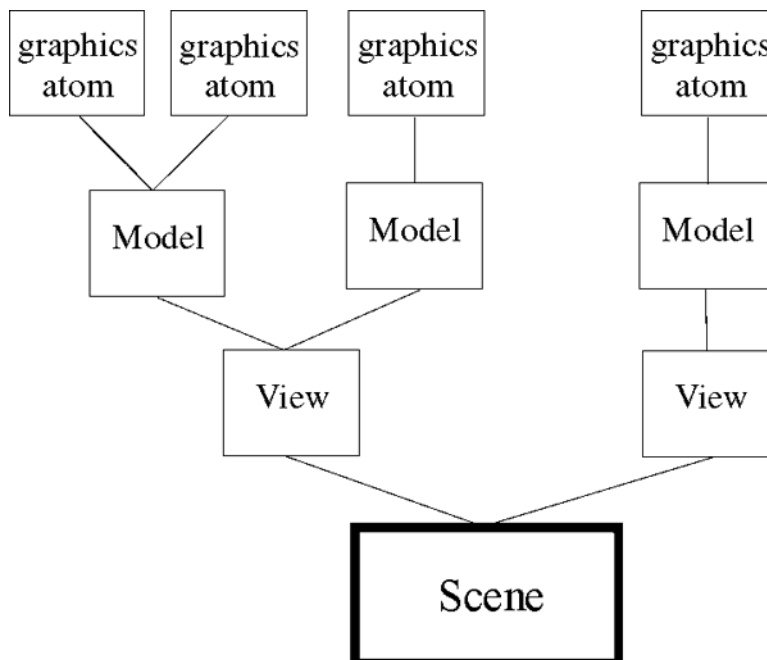


Figure 24-1: A graphics object tree.

The advantage of organizing graphic objects into a tree structure is that by manipulating any of the branches of the tree, all of the sub-branches of that branch can be altered simultaneously. In our example, changes to the spatial transformation associated with the model containing two graphics atoms will affect both of the atoms. Similarly, calling a window or printer object's Draw method on the scene object will render all of the objects in the tree to that window or printer.

Scenes

A scene, or instance of the [IDLgrScene](#) class, is the root-level object of most graphics trees. Instances of the `IDLgrScene` class have `Add` and `Remove` methods, which allow you to include or remove `IDLgrView` or `IDLgrViewgroup` objects in a scene. A scene object is one of the possible arguments for a destination object's `Draw` method.

It is not necessary to create a scene object if your graphics tree contains only one view object; in that case, the view can serve as the root of the tree.

Viewgroups

A viewgroup, or instance of the [IDLgrViewgroup](#) class, is a simple container object, similar to the Scene object. The Viewgroup differs from the Scene in two ways:

1. It will not cause an erase to occur on a destination when the destination object's Draw method is called.
2. It can contain objects which do not have Draw methods.

Viewgroups are designed to be placed within a scene, and therefore do not typically serve as the root-level object of a graphics tree. Instances of the [IDLgrViewgroup](#) class have Add and Remove methods, which allow you to include or remove objects in a viewgroup.

Views

A view, or instance of the [IDLgrView](#) class, can also serve as the root-level object of a graphics tree. Instances of the `IDLgrView` class have `Add` and `Remove` methods, which allow you to include or remove `IDLgrModel` objects in a view. A view object is one of the possible arguments for a destination object's `Draw` method.

Every graphics tree must contain at least one view object. Often, it is convenient to divide the objects being rendered into separate views, which are then contained by a `viewgroup` or `scene` object.

Models

A model, or instance of the `IDLgrModel` class, is a container for atomic graphic objects or for other model objects. The model object incorporates a transformation matrix (see [Chapter 25, “Transformations”](#) for an in-depth discussion of transformation matrices) that applies to all of the graphics atoms and model objects it contains. In addition to `Add` and `Remove` methods, the model object has methods to `Rotate`, `Scale`, and `Translate` the model and its contents.

Atomic Graphic Objects

An atomic graphic object, or graphic atom, is an instance of one of the following classes: [IDLgrAxis](#), [IDLgrContour](#), [IDLgrImage](#), [IDLgrLight](#), [IDLgrPlot](#), [IDLgrPolygon](#), [IDLgrPolyline](#), [IDLgrSurface](#), [IDLgrText](#), or [IDLgrVolume](#). Graphics atoms combined in a model object (using the model object's Add method) share the same transformation matrix and can be rotated, scaled, or translated together.

Attribute and Helper Objects

Attribute objects are used by atomic graphic objects to define how the graphics atom will be rendered; attribute objects themselves are not drawn, and thus do not need to be added to a model object. Attribute objects are instances of one of the following classes: [IDLgrFont](#), [IDLgrPalette](#), [IDLgrPattern](#), or [IDLgrSymbol](#). For example, a text object (a graphic atom) defines which type style it will be rendered in by setting its `FONT` property equal to an instance of the `IDLgrFont` object.

Helper objects are used to change or create data to make it suitable for a particular type of rendering. In IDL, there are several helper objects which are instances of the following classes: [IDLgrTessellator](#) and [TrackBall](#). The tessellator object changes a simple concave polygon (or a simple polygon with holes) into a number of simple convex polygons (general triangles) suitable for use by objects of the `IDLgrPolygon` class. The trackball object translates widget events from a draw widget (created with the `WIDGET_DRAW` function) into transformations that emulate a virtual trackball (for transforming object graphics in three dimensions).

For more information, see [Chapter 27, “Using Attributes and Helpers”](#).

The Rendering Process

In Object Graphics, rendering occurs when the Draw method of a destination object is called. A scene, viewgroup, or view is typically provided as the argument to this Draw method. This argument represents the root of a graphics hierarchy. When the destination's Draw method is called, the graphics hierarchy is traversed, starting at the root, then proceeding to children in the order in which they were added to their parent.

For example, suppose we have the following hierarchy:

```
oWindow = OBJ_NEW('IDLgrWindow')
oView = OBJ_NEW('IDLgrView')
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel
oXAxis = OBJ_NEW('IDLgrAxis', 0)
oModel->Add, oXAxis
oYAxis = OBJ_NEW('IDLgrAxis', 1)
oModel->Add, oYAxis
```

To draw the view (and its contents) to the window, the Draw method of the window is called with the view as its argument:

```
oWindow->Draw, oView
```

The window's Draw method will perform any window-specific drawing setup, then ask the view to draw itself. The view will then perform view-specific drawing (for example, clearing a rectangular area to a color), then calls the Draw method for each of its children (in this case, there is only one child, a model). The model's Draw method will push its transformation matrix on a stack, then step through each of its children (in the order in which they were added) and ask them to draw themselves. In this example, oXAxis will be asked to draw itself first; then oYAxis will be asked to draw itself. Once each of the model's children is drawn, the transformation matrix associated with the model is popped off of the stack.

Thus, for each object in the hierarchy, drawing essentially consists of three steps:

- Perform setup drawing for this object.
- Step through list of contained children and ask them to draw themselves.
- Perform follow-up drawing actions before returning control to parent.

The order in which objects are added to the hierarchy will have an impact on when the objects are drawn. Drawing order can be changed by using the Move method of a scene, viewgroup, view, or model to change the position of a specific object within the hierarchy.

The first time a graphic atom (such as an axis, plot line, or text) is drawn to a given destination, a device-specific encapsulation of its visual representation is created and stored as a cache. Subsequent draws of this graphic atom to the same destination can then be drawn very efficiently. The cache is destroyed only when necessary (for example, when the data associated with the graphic atom changes). Graphic attribute changes (such as color changes) typically do not cause cache destruction. To gain maximum benefit from the caches, modification of atomic graphic properties should be kept to bare minimum.

Simple Plot Example

The following section shows the IDL code used to create a simple object tree. While you are free to enter the commands shown at the IDL command line, remember that the IDL Object Graphics API is designed as a programmer's interface, and is not as well suited for interactive, ad hoc work at the IDL command prompt as are IDL Direct Graphics.

The following IDL commands construct a simple plot of an array versus the integer indices of the array. Note that no axes, title, or other annotations are included; the commands draw only the plot line itself. (This example is purposefully simple; it is meant to illustrate the skeleton of a graphics tree, not to produce a useful plot.)

```
; Create a view 2 units high by 100 units wide
; with its origin at (0,-1):
view = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,-1,100,2])
; Create a model:
model = OBJ_NEW('IDLgrModel')
; Create a plot line of a sine wave:
plot = OBJ_NEW('IDLgrPlot', SIN(FINDGEN(100)/10))
; Create a window into which the plot line will be drawn:
window = OBJ_NEW('IDLgrWindow')
; Add the plot line to the model object:
model->ADD, plot
; Add the model object to the view object:
view->ADD, model
; Render the contents of the view object in the window:
window->DRAW, view
```

To destroy the window and remove the objects created from memory, use the following commands:

```
OBJ_DESTROY, window
; Destroying the view object destroys all
; of the objects contained in the view:
OBJ_DESTROY, view
```




Chapter 25: Transformations

The following topics are covered in this chapter:

Overview	542	Model Transformations	552
Viewport	543	Coordinate Conversion	555
Projection	545	Example: Centering an Image	557
Eye Position	547	Example: Displaying a Surface	560
View Volume	549	Virtual Trackball and 3D Transformations	563

Overview

Unlike IDL Direct Graphics, the IDL Object Graphics system does not automatically position and size the objects to be rendered. It is up to you, as a programmer, to properly define how your graphic elements will be positioned when rendered.

There are three aspects to this transformation from a generic depiction of your data to a representation that can be rendered to an output device (a graphics destination object, such as a window or printer) with the perspective, size, and location you want.

Viewport

The first aspect is the view of the graphics objects to be rendered: the size of the viewing area (the viewport), the type of projection used, the position of the viewer's eye as it looks at the graphics objects, and the particular view volume in three-dimensional space that will be rendered to the viewing area. These elements of the view of your graphics objects are, appropriately, controlled by properties of the `IDLgrView` object being rendered.

Location

The second aspect of the transformation is the location and position of your graphics objects with respect to the viewing area. Graphics objects can be translated, rotated, or scaled by setting the appropriate properties of the `IDLgrModel` object that contains them.

Note

The viewport and location of an object are independent: It is possible, for example, to translate a graphic object so that it is no longer within the viewing area that is rendered in a window or on a printer.

Coordinate Systems and Scaling

The third aspect of the transformation is the conversion between data, device, and normalized coordinates. The IDL Object Graphics system gives you full control over which data values are used, which are displayed, and which coordinate systems are used. This means that you must explicitly ensure that the objects to be rendered and the view object to which they belong use the same coordinate system and are scaled appropriately.

This chapter discusses the properties and methods used to size and position both your viewing area and the graphics objects you wish to render.

Viewport

One of the first steps in determining how graphics objects will appear when rendered on a graphics destination object is to select the location and dimensions of the rectangular area—the viewport—on the destination in which the rendering will be displayed. Set the location and dimensions of the viewport using the `LOCATION` and `DIMENSIONS` keywords to the `IDLgrView::Init` method when creating the view object (or after creation using the `SetProperty` method). For example, the following statement creates a view object with a viewport that is 300 pixels by 200 pixels, with its lower left corner located 100 pixels up from the bottom and 100 pixels to the right of the left edge of the destination object:

```
myView = OBJ_NEW('IDLgrView', LOCATION=[100,100], $  
    DIMENSIONS=[300,200])
```

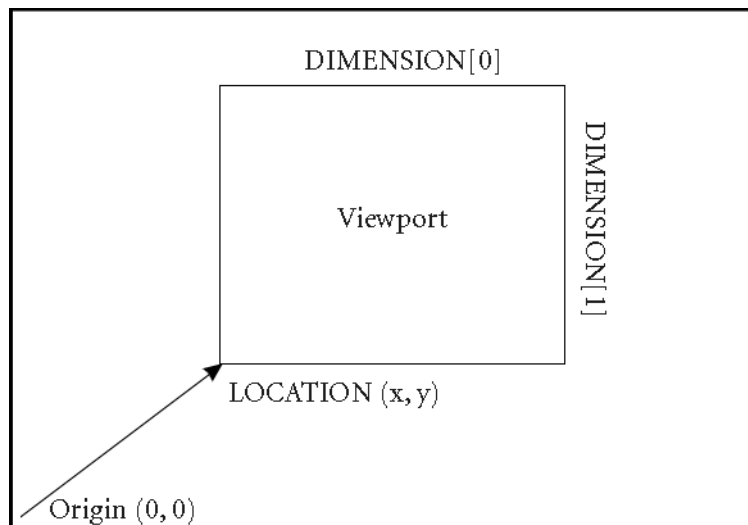


Figure 25-1: Positioning a view on the screen.

Both the `LOCATION` and `DIMENSIONS` properties of the view object honor the value of the `UNITS` property, which specifies the type of units in which measurements are made. (Pixels are the default units, so no specification of the `UNITS` keyword was necessary in the above example.)

The viewport of an existing view can be changed using the `SetProperty` method:

```
myView->SetProperty, LOCATION=[0,0], DIMENSIONS=[200,200]
```

changes the location of the viewport to have its lower left corner at (0, 0) and a size of 200 pixels by 200 pixels.

Note

The eye is positioned in only one dimension (along the z -axis) and always points in the $-z$ direction.

Projection

When three-dimensional graphics are displayed on a flat computer screen or printed on paper, they must be projected onto the viewing plane. A projection is a way of converting positions in 3D space into locations in the 2D viewing plane. IDL supports two types of projections—parallel and perspective—for each view.

Parallel Projections

A parallel projection projects objects in 3D space onto the 2D viewing plane along parallel rays. The figure below shows a parallel projection; note that two objects that are the same size but at different locations still appear to be the same size when projected onto the viewplane.

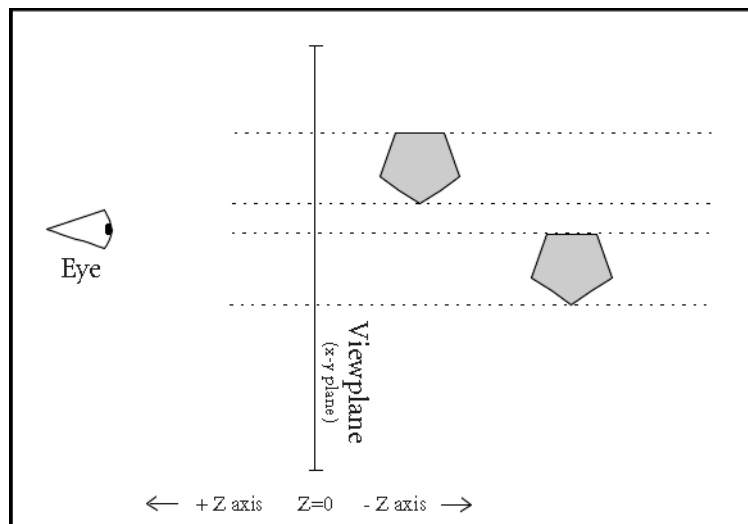


Figure 25-2: In a parallel projection, rays do not converge at the eye.

View objects use a parallel projection by default. To explicitly set a view object to use a parallel projection, set the `PROJECTION` keyword to the `IDLgrView::Init` method equal to 1 (or use the `SetProperty` method to set the projection for an existing view object):

```
myView->SetProperty, PROJECTION = 1
```

Perspective Projections

A perspective projection projects objects in 3D space onto the 2D viewing plane along rays that converge at the eye position. The figure below shows a perspective projection; note that objects that are farther from the eye appear smaller when projected onto the viewplane.

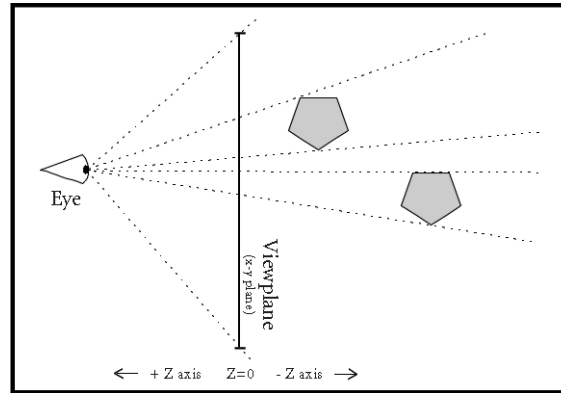


Figure 25-3: In a perspective projection, rays converge at the eye.

Set the PROJECTION keyword to the IDLgrView::Init method equal to 2 (or use the SetProperty method to set the projection for an existing view object) to use a perspective projection:

```
myView->SetProperty, PROJECTION = 2
```

Eye Position

The eye position is the position along the z -axis from which a set of objects contained in a view object are seen. Use the `EYE` keyword to the `IDLgrView::Init` method to specify the distance from the eye position to the viewing plane (or use the `SetProperty` method to alter the eye position of an existing view object). The eye position must be a z value larger than the z value of the near clipping plane (see “[Near and Far Clipping Planes](#)” on page 549) or zero, whichever is greater. That is, the eye must always be located at a positive z value, and must be outside the volume bounded by the near and far clipping planes.

For example, the following moves the eye position to $z = 5$:

```
myView->SetProperty, EYE=5
```

The eye is always positioned directly in front of the center of the viewplane rectangle. That is, if the `VIEWPLANE_RECT` property is set equal to `[-1, -1, 2, 2]`, the eye will be located at $X=0, Y=0$.

Changing the position of the eye has no effect when you are using a parallel projection. Changing the eye position when you are using a perspective projection has a somewhat counter-intuitive affect: moving the eye closer to the near clipping plane causes objects in the volume being rendered to appear smaller rather than larger. To understand why this should be true, consider the following diagram.

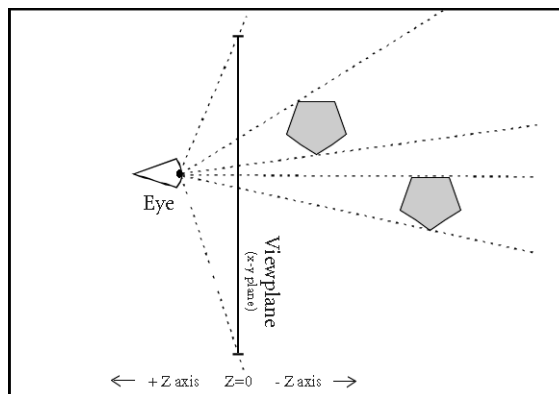


Figure 25-4: Moving the eye closer to the viewplane causes objects to appear smaller.

In a perspective projection, rays from the graphic objects in the view volume converge at the eye position. When the eye is close to the viewing plane, the projected rays cross the viewing plane (where rendering actually occurs) in a relatively small area. When the eye moves farther from the viewing plane, the projected rays become more nearly parallel and occupy a larger area on the viewing plane when rendered.

View Volume

The view volume defines the three-dimensional volume in space that, once projected, is to fit within the viewport. There are two parts to the view volume: the viewplane rectangle and the near and far clipping planes.

Viewplane Rectangle

The viewplane rectangle defines the bounds in the X and Y directions that will be mapped into the viewport. Objects (or portions of objects) that lie outside the viewplane rectangle will not be rendered. The viewplane rectangle is always located at $Z=0$.

Use the `VIEWPLANE_RECT` keyword to the `IDLgrView::Init` method (or use the `SetProperty` method if you have already created the view object) to set the location and extent of the viewplane rectangle. Set the keyword equal to a four-element floating-point vector; the first two elements specify the X and Y location of the lower left corner of the rectangle, and the second two elements specify the width and height. The default rectangle is located at $(-1.0, -1.0)$ and is two units wide and two units high $([-1.0, -1.0, 2.0, 2.0])$. For example, the following command changes the viewplane rectangle to be located at $(0.0, 0.0)$ and to be one unit square:

```
myView->SetProperty, VIEWPLANE_RECT = [0.0, 0.0, 1.0, 1.0]
```

Near and Far Clipping Planes

The near and far clipping planes define the bounds in the Z direction that will be mapped into the viewport. Objects (or portions of objects) that lie nearer to the eye than the near clipping plane or farther from the eye than the far clipping plane will not be rendered. The figure below shows near and far clipping planes.

Use the `ZCLIP` keyword to the `IDLgrView::Init` method (or use the `SetProperty` method if you have already created the view object) to set the near and far clipping planes. Set the keyword equal to a two-element floating-point vector that defines the positions of the two clipping planes: `[near, far]`. The default clipping planes are at $Z = 1.0$ and $Z = -1.0$ $([1.0, -1.0])$. For example, the following command changes the near and far clipping planes to be located at $Z = 2.0$ and $Z = -3.0$, respectively.

```
myView->SetProperty, ZCLIP = [2.0, -3.0]
```

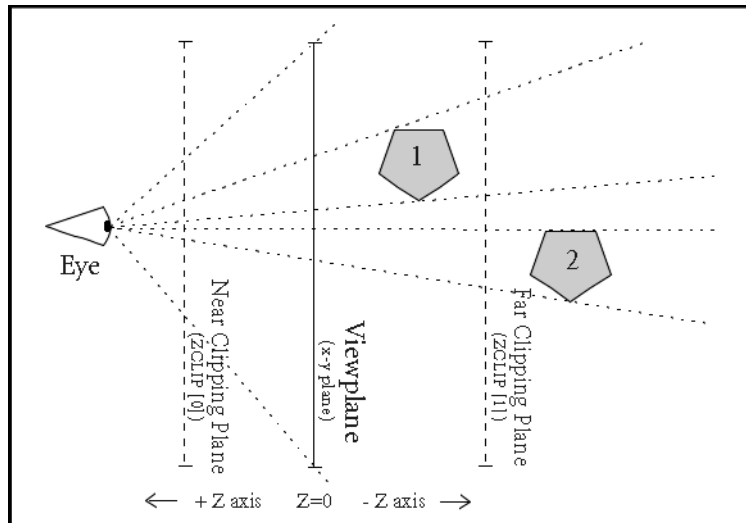


Figure 25-5: Near and Far Clipping Planes. Object 2 is not rendered, because it does not lie between the near and far clipping planes.

Finding an Appropriate View Volume

Finding an appropriate view volume for a given object tree is relatively simple in theory. To find the appropriate viewplane rectangle, you must find the overall X and Y range of the object (usually a model or scene object) that contains the items drawn in the object tree, accounting for any transformations of objects contained in the tree. Similarly, to find the appropriate near and far clipping planes, you can find the Z range of the object that contains the items drawn in the object tree. In practice, however, finding, adding, and transforming the ranges for a large object tree can be complicated.

Two routines contained in the IDL distribution provide an example of how the view volume can be computed in many cases. These routines are defined in the files `set_view.pro` and `get_bounds.pro`, located in the `examples/visual/utility` subdirectory of the IDL distribution. The `SET_VIEW` procedure accepts as arguments the object references of a view object and a destination object, computes an appropriate view volume for the view object, and sets the `VIEWPLANE_RECT` property of the view object accordingly. The

`SET_VIEW` procedure calls the `GET_BOUNDS` procedure to compute the *X*, *Y*, and *Z* ranges of the objects contained in the view object.

The `SET_VIEW` and `GET_BOUNDS` routines are used in the examples in this volume, and are available for your use when creating and displaying object hierarchies. They are, however, example code, and are not truly generic in the situations they address. When you encounter a situation for which these routines do not produce the desired result, we encourage you to copy and alter the code to suit your own needs.

Inspect the `SET_VIEW.PRO` and `GET_BOUNDS.PRO` files for further details.

Model Transformations

An `IDLgrModel` object is a container for any graphics atoms that are to be rotated, translated, or scaled. Each `IDLgrModel` object has a transformation property (set via the `TRANSFORM` keyword to the `IDLgrModel::Init` or `SetProperty` method), which is a 4 x 4 floating-point matrix. For a general discussion of transformation matrices and three-dimensional graphics, see “[Three-Dimensional Graphics](#)” in Chapter 18.

Note

A model object’s transformation matrix is akin to the transformation matrix used by IDL Direct Graphics and stored in the `!P.T` system variable field. Transformation matrices associated with a model object do not use the value of `!P.T`, however, and are not affected by the `T3D` procedure used in Direct Graphics.

By default, a model object’s transformation matrix is set equal to a 4-by-4 identity matrix:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

You can change the transformation matrix of a model object directly, using the `TRANSFORM` keyword to the `IDLgrModel::Init` or `SetProperty` method:

```
myModel = OBJ_NEW('IDLgrModel', TRANSFORM = tmatrix)
```

where `tmatrix` is a 4-by-4 transformation matrix. Alternatively, you can use the `Translate`, `Rotate`, and `Scale` methods to the `IDLgrModel` object to alter the model’s transformation matrix.

Translation

The `IDLgrModel::Translate` method takes three arguments specifying the amount to translate the model object and its contents in the *X*, *Y*, and *Z* directions. For example, to translate a model and its contents by 1 unit in the *X*-direction, you could use the following statements:

```
dx = 1 & dy = 0 & dz = 0
myModel->Translate, dx, dy, dz
```


How does this affect the transformation matrix? Notice that we could change the transformation matrix in an identical way using the following statements:

```

; Define translation values:
dx = 1 & dy = 0 & dz = 0
; Get existing transformation matrix:
myModel->GetProperty, TRANSFORM = oldT
; Provide a transformation matrix that performs the translation:
transT = [[1.0, 0.0, 0.0, dx], $
          [0.0, 1.0, 0.0, dy], $
          [0.0, 0.0, 1.0, dz], $
          [0.0, 0.0, 0.0, 1.0]]
; Multiply the existing transformation matrix by
; the matrix that performs the translation:
newT = oldT # transT
; Apply the new transformation matrix to the model object:
myModel->SetProperty, TRANSFORM = newT

```

Rotation

The `IDLgrModel::Rotate` method takes two arguments specifying the axis about which to rotate and the number of degrees to rotate the model object and its contents. For example, to rotate a model and its contents by 90 degrees around the y-axis, you could use the following statements:

```

axis = [0,1,0] & angle = 90
myModel->Rotate, axis, angle

```

How does this affect the transformation matrix? Notice that we could change the transformation matrix in an identical way using the following statements:

```

; Define rotation values:
axis = [0,1,0] & angle = 90
; Get existing transformation matrix:
myModel->GetProperty, TRANSFORM = oldT
; Define sine and cosine of angle:
cosa = COS(!DTOR*angle)
sina = SIN(!DTOR*angle)
; Provide a transformation matrix that performs the rotation:
rotT = [[cosa, 0.0, sina, 0.0], $
        [0.0, 1.0, 0.0, 0.0], $
        [-sina, 0.0, cosa, 0.0], $
        [0.0, 0.0, 0.0, 1.0]]
; Multiply the existing transformation matrix
; by the matrix that performs the rotation.
newT = oldT # rotT
; Apply the new transformation matrix to the model object:
myModel->SetProperty, TRANSFORM = newT

```

Scaling

The `IDLgrModel::Scale` method takes three arguments specifying the amount to scale the model object and its contents in the x , y , and z directions. For example, to scale a model and its contents by 2 units in the y direction, you could use the following statements:

```
sx = 1 & sy = 2 & sz = 1
myModel->Scale, sx, sy, sz
```

How does this affect the transformation matrix? Notice that we could change the transformation matrix in an identical way using the following statements:

```
; Define scaling values:
sx = 1 & sy = 2 & sz = 1
; Get existing transformation matrix:
myModel->GetProperty, TRANSFORM = oldT
; Provide a transformation matrix that performs the scaling:
scaleT = [[sx, 0.0, 0.0, 0.0], $
          [0.0, sy, 0.0, 0.0], $
          [0.0, 0.0, sz, 0.0], $
          [0.0, 0.0, 0.0, 1.0]]
; Multiply the existing transformation matrix
; by the matrix that performs the scaling.
newT = oldT # scaleT
; Apply the new transformation matrix to the model object:
myModel->SetProperty, TRANSFORM = newT
```

Combining Transformations

Note that model transformations are cumulative. That is, a model object contained in another model is subject to both its own transformation and to that of its container. All transformation matrices that apply to a given model object are multiplied together when the object is rendered. For example, consider a model that contains another model:

```
model1 = OBJ_NEW('IDLgrModel', TRANSFORM = trans1)
model2 = OBJ_NEW('IDLgrModel', TRANSFORM = trans2)
model2->Add, model1
```

The `model1` object is now subject to both its own transformation matrix (`trans1`) and to that of its container (`trans2`). The result is that when `model1` is rendered, it will be rendered with a transformation matrix = `trans1 # trans2`.

Coordinate Conversion

Most transformations are handled by the transformation matrix of a model object. For convenience, however, graphic atoms may also have a simplified transformation applied to them. Coordinate transformations applied to individual graphic atoms allow you to change only the translation (position) and scale; this is useful when converting from one coordinate system to another. For example, you may build your view object using normalized coordinates, so that values range between zero and one. If you create a graphic object—a surface object, say—based on the range of data values, you would need to convert your surface object (built with a data coordinate system) to match the view object (built with a normal coordinate system). To do this, use the [XYZ]COORD_CONV keywords to the graphic object in question. The [XYZ]COORD_CONV keywords take as their argument a two-element vector that specifies the translation and scale factor for each dimension.

For example, suppose you have a surface object whose data is specified in a range from $[0, 0, zMin]$ to $[xMax, yMax, zMax]$. If you wanted to work with this surface as if it were in a normalized $[-1, -1, -1]$ to $[1, 1, 1]$ space, you could use the following coordinate conversions:

```
; Create some data:
myZdata = DIST(60)
; Use SIZE to determine size of each dimension of myZdata:
sz = SIZE(myZdata)
; Create a scale factor for the X dimension:
xs = 2.0/(sz[1]-1)
; Create a scale factor for the Y dimension:
ys = 2.0/(sz[2]-1)
; Create a scale factor for the Z dimension:
zs = 2.0/MAX(myZdata)
```

Now, use the [XYZ]COORD_CONV keywords to the IDLgrSurface::Init method to translate the surface by minus one unit in each direction, and to scale the surface by the scale factors:

```
mySurface = OBJ_NEW('IDLgrSurface', myZdata, $
    XCOORD_CONV = [-1, xs], YCOORD_CONV = [-1, ys], $
    ZCOORD_CONV = [-1, zs])
```

Remember that using the [XYZ]COORD_CONV keywords is simply a convenience—the above example could also have been written as follows:

```
; Create some data:
myZdata = DIST(60)
; Use SIZE to determine the size of each dimension of myZdata:
sz = SIZE(myZdata)
```

```

; Create a scale factor for the X dimension:
xs = 2.0/(sz(1)-1)
; Create a scale factor for the Y dimension:
ys = 2.0/(sz(2)-1)
; Create a scale factor for the Z dimension:
zs = 2.0/(MAX(myZdata))
; Create a model object:
myModel = OBJ_NEW('IDLgrModel')
; Apply scale factors:
myModel->Scale, xs, ys, zs
; Translate:
myModel->Translate, -1, -1, -1
; Create surface object:
mySurface = OBJ_NEW('IDLgrSurface', myZdata)
; Add surface object to model object:
myModel->Add, mySurface

```

A Function for Coordinate Conversion

Often, it is convenient to convert minimum and maximum data values so that they fit in the range from 0.0 to 1.0 (that is, so they are normalized). Rather than adding the code to make this coordinate conversion to your code in each place it is required, you may wish to define a coordinate conversion function.

For example, the following function definition accepts a two-element array representing minimum and maximum values returned by the `XYZRANGE` keyword to the `GetProperty` method, and returns two-element array of scaling parameters suitable for the `XYZCOORD_CONV` keywords:

```

FUNCTION NORM_COORD, range
    scale = [-range[0]/(range[1]-range[0]), 1/(range[1]-range[0])]
    RETURN, scale
END

```

If you define a function like this in your code, you can then call it whenever you need to scale your data ranges into normalized coordinates. The following statements create a plot object from the variable `data`, retrieve the values of the `X` and `Y` ranges for the plot, and then use the `XYCOORD_CONV` keywords to the `SetProperty` method and the `NORM_COORD` function to set the coordinate conversion.

```

plot = OBJ_NEW('IDLgrPlot', data)
plot->GetProperty, XRANGE=xr, YRANGE=yr
plot->SetProperty, XCOORD_CONV=NORM_COORD(xr), $
    YCOORD_CONV=NORM_COORD(yr)

```

The function `NORM_COORD` is defined in the file `norm_coord.pro` in the `examples/visual/utility` subdirectory of the IDL distribution.

Example: Centering an Image

The following example steps through the process of creating an image object and provides two options for centering it within a 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.
windowSize = [400, 460]
windowMargin = (windowSize - worldelvSize)/2

; 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
```

Example: Displaying a Surface

The following example steps through the process of creating a surface object and all of the supporting objects necessary to display it.

Note

You do not need to enter the example code yourself. The example code shown here is duplicated in the procedure file `test_surface.pro`, located in the `examples/visual` subdirectory of the IDL distribution. You can run the example procedure by entering `TEST_SURFACE` at the IDL command prompt.

When creating this procedure, we allow the user to specify keywords that will return object references to the view, model, surface, and window objects. This allows us to manipulate the objects directly from the IDL command line after the procedure has been run.

```

PRO test_surface, VIEW=oView, MODEL=oModel, $
    SURFACE=oSurface, WINDOW=oWindow

; Create some data:
zData = DIST(60)

; Create a view object. We set the color of the view
; area to a dark grey using the COLOR keyword,
; and set the viewplane to a square area occupying one
; unit in each quadrant of the XY plane—a normalized
; coordinate system—using the VIEWPLANE_RECT keyword.
oView = OBJ_NEW('IDLgrView', COLOR=[60,60,60], $
    VIEWPLANE_RECT=[-1,-1,2,2])

; Create a model object:
oModel = OBJ_NEW('IDLgrModel')

; Add the model object to the view object:
oView->Add, oModel

; Create a surface object. We set the color of
; the surface to pure red, using the COLOR keyword:
oSurface = OBJ_NEW('IDLgrSurface', zData, color=[255,0,0])

; Add the surface object to the model object:
oModel->Add, oSurface

; Next, we use the GetProperty method of the surface
; object to retrieve the data range of the surface:
oSurface->GetProperty, XRANGE=xrange, YRANGE=yrange, ZRANGE=zrange

```



```

; Scale surface to normalized units and center using
; the SetProperty method of the surface object to change
; the [XYZ]COORD_CONV properties:
xs = [-0.5, 1/(xrange[1]-xrange[0])]
ys = [-0.5, 1/(yrange[1]-yrange[0])]
zs = [-0.5, 1/(zrange[1]-zrange[0])]
oSurface->SetProperty, XCOORD_CONV=xs, YCOORD_CONV=ys,$
    ZCOORD_CONV=zs

; Now we rotate the model object to display a standard view:
oModel->Rotate,[1,0,0], -90
oModel->Rotate,[0,1,0], 30
oModel->Rotate,[1,0,0], 30

; Finally, we create a window (destination) object
; and draw the contents of the view object to it:
oWindow = OBJ_NEW('IDLgrWindow')
oWindow->Draw, oView

```

END

Play with the example to learn how object transformations work and interact. Try the following commands at the IDL prompt to observe what they do:

First, compile `test_surface.pro`:

```
.RUN test_surface.pro
```

Now, execute the procedure. The variables you supply via the `SURFACE`, `MODEL`, `VIEW`, and `WINDOW` keyword will contain object references you can manipulate from the command line:

```
test_surface, VIEW=myview, MODEL=mymodel, $
    SURFACE=mysurf, WINDOW=mywin
```

This will create a window object and display the surface. Now try the following to translate the object to the right:

```
mymodel->Translate, 0.2, 0, 0
```

The model transformation changes as soon as you issue this command. The window object, however, will not be updated to reflect the new position until you issue a `Draw` command:

```
mywin->Draw, myview
```

Try a rotation in the *y* direction:

```
mymodel->Rotate, [0,1,0], 45
mywin->Draw, myview
```

Repeat the commands several times and observe what happens.

Try some of the following. Remember to issue a Draw command after each change in order to see what you have done.

```
mymodel->Scale, 0.5, 0.5, 0.5
mymodel->Scale, 1, 0.5, 1
mymodel->Scale, 1, 2, 1
mymodel->Rotate, [0,0,1], 45
mysurf->SetProperty, COLOR = [0, 255, 0]
myview->SetProperty, PROJECTION = 2, EYE = 2
myview->SetProperty, EYE = 1.1
myview->SetProperty, EYE = 6
```

Virtual Trackball and 3D Transformations

To create truly interactive object graphics, you must allow the user to transform the position or orientation of objects using the mouse. One way to do this is to provide a virtual trackball that lets the user manipulate objects interactively on the screen.

The procedure file `trackball__define.pro`, found in the `lib` directory of the IDL distribution, contains the object definition procedure for a virtual trackball object. This trackball object is used in several of the examples presented later in this volume, and is also used by other example and demonstration code included with IDL. The trackball object has three methods: `Init`, `Update`, and `Reset`. These methods allow you to retrieve mouse movement events and alter your model transformations accordingly.

The trackball object behaves as if there were an invisible trackball, centered at a position you specify, overlaid on a draw widget. The widget application's event handler uses the widget event information to update both the trackball's state and the model transformation of the objects displayed in the draw widget's window object. When the user clicks and drags in the draw widget, objects in the draw widget rotate as if the user were manipulating them with a physical trackball.

See “[TrackBall](#)” in the *IDL Reference Guide* manual for details on creating and using trackball objects. Several of the other example files located in the `examples/visual` subdirectory of the IDL distribution include trackball objects, and may be studied for further insight into the mechanics of transforming object hierarchies based on user input.



Chapter 26: Working with Color

The following topics are covered in this chapter:

Overview	566	Color and Destination Objects	570
Color and Digital Data	567	Palettes	571
Indexed Color Model	568	Using Color	572
RGB Color Model	569	How IDL Interprets Color Values	574

Overview

Color is often an integral part of the process of visualizing a dataset. The IDL Object Graphics system allows you to use color in a number of different ways; this chapter explains how to specify color when using Object Graphics and how IDL interacts with the destination devices on which graphics are finally displayed.

Color and Digital Data

The IDL Object Graphics system provides two color models for you to choose between when creating destination (window or printer) objects: an Indexed Color Model and an RGB Color Model. Indexed color allows you to map data values to color values using a color palette. RGB color allows you to specify color values explicitly, using an RGB triple. (See [“Specifying RGB Values” on page 572](#) for more information on RGB triples.) You choose one of these two color models to associate with each destination object.

Note

For some X11 display situations, IDL may not be able to support a color index model destination object in object graphics. We do, however, guarantee that an RGB color model destination will be available for all display situations.

The devices on which graphics are rendered—computer displays, printers, plotters, frame buffers, etc.—also support one or more color models. IDL performs any conversions necessary to support either the Indexed or RGB color model on any physical device. That is, the color model used by IDL is entirely independent of the color model used by the physical device. [“How IDL Interprets Color Values” on page 574](#) explains how IDL’s Object System color models interact with different device color models.

Note

You can specify the color of any graphic object using either a color index or red, green, and blue (RGB) value, regardless of the color model used by the destination object or the physical destination device. See [“Using Color” on page 572](#) for details.

Indexed Color Model

In the Indexed color model, you have control over how colors are loaded into a color lookup table. You do this by specifying a palette, which maps color index values into RGB values, for the destination object. When the contents of your destination object are rendered on the physical device (that is, when you call the Draw method for the destination object), the RGB values from the palette are either:

- passed directly through to the physical device (if it uses RGB values), or
- loaded into the physical device's lookup table (if it uses Indexed values).

Specify that a destination object should use the Indexed color model by setting the COLOR_MODEL property of the object equal to 1 (one):

```
myWindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL = 1)
```

Specify a palette object by setting the PALETTE property equal to an object of the IDLgrPalette class:

```
myWindow->SetProperty, PALETTE=myPalette
```

If you do not specify a palette object for a destination object that uses the Indexed color model, a grayscale ramp palette is loaded automatically.

When you assign a color index to an object that is drawn on the destination device, the color index is used to look up an RGB value in the specified palette. When you assign an RGB value to an object that is drawn on the destination device, the nearest match within the destination object's palette is found and used to represent that color.

RGB Color Model

In the RGB color model, IDL takes responsibility for filling the color lookup table on the destination device (if necessary). When the contents of your destination object are rendered on the physical device (that is, when you call the Draw method for the destination object), the RGB values are either:

- passed directly through to the physical device (if it uses RGB values), or
- matched as nearly as possible with colors loaded in the physical device's lookup table (if it uses Indexed values).

Specify that a destination object should use the RGB color model by setting the `COLOR_MODEL` property of the object equal to 0 (zero):

```
myWindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL = 0)
```

This is the default for newly-created destination objects.

Color and Destination Objects

Each destination object has one of the two color models described above associated with it. Destination objects use the Indexed color model if the `COLOR_MODEL` property is set equal to 1 (one) or the RGB color model if the `COLOR_MODEL` property is set equal to 0 (zero, the default). Once a destination object has been created, you cannot change the associated color model.

You can, however, create destination objects that use different color models in the same IDL session. That is, it is possible to have two window objects—one using the Indexed color model and one using the RGB color model—on your computer screen at the same time.

Remember also that you can specify the color of any graphic object using either a color index or an RGB value, regardless of the color model used by the destination object or the physical destination device. The main distinction between the two color models lies in how IDL manages the color lookup table (if any) of the physical destination device. See [“Using Color” on page 572](#) for details.

A Note about Draw Widgets

Drawable areas created with the `WIDGET_DRAW` function deserve a special mention. When a draw widget is created with the `GRAPHICS_LEVEL` keyword set equal to 2, the widget contains an instance of an `IDLgrWindow` object rather than an IDL Direct Graphics drawable window. By default, the window object uses the RGB color model; to use the indexed color model, set the `COLOR_MODEL` keyword to `WIDGET_DRAW` equal to 1 (one).

Palettes

Objects of the `IDLgrPalette` class are used to create color lookup tables. The following statements create a palette object that reverses a standard grayscale ramp palette:

```
rval = (gval = (bval = REVERSE(INDGEN(256))))  
myPalette = OBJ_NEW('IDLgrPalette', rval, gval, bval)
```

Palettes can be associated either with graphics destination objects (windows or printers) or with individual graphic atoms:

```
myWindow->SetProperty, PALETTE=myPalette
```

or

```
myImage->SetProperty, PALETTE=myPalette
```

Note

Palettes associated with graphic atoms are only used when the destination object uses an RGB color model; if the destination object uses an indexed color model, the destination object's palette is always used.

See “[IDLgrPalette](#)” in the *IDL Reference Guide* manual for details on creating palette objects.

Using Color

The color of a graphic object is specified by the `COLOR` property of that object. You can set the color of an object either when the object is created or afterwards. For example, the following statement creates a view object and sets its color value to the RGB triple `[60, 60, 60]` (a dark gray).

```
myView = OBJ_NEW('IDLgrView', COLOR = [60, 60, 60])
```

The following statement changes the color value of an existing axis object to the color value specified for entry 100 in the color palette associated with the axis object.

```
myAxis->SetProperty, COLOR=100
```

Remember that color palettes associated with individual graphic atoms are only used when the destination object uses an RGB color model.

Specifying RGB Values

RGB values are specified with RGB triples. An RGB triple is a three-element vector of integer values, $[r, g, b]$, generally ranging between 0 and 255. A value of zero is the darkest possible value for each of the three channels—thus an RGB triple of `[0, 0, 0]` represents black, `[0, 255, 0]` represents bright green, and `[255, 255, 255]` represents white.

For example, suppose we create a plot line with the following statements:

```
myWindow = OBJ_NEW('IDLgrWindow')
myView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0, 0, 10, 10])
myModel = OBJ_NEW('IDLgrModel')
myPlot = OBJ_NEW('IDLgrPlot', FINDGEN(10), THICK = 5, $
    COLOR=[255, 255, 255])
myModel->Add, myPlot
myView->Add, myModel
myWindow->Draw, myView
```

Notice the following aspects of the above example:

1. The newly-created window (destination) object uses an RGB color mode (the default).
2. The default color of the view object—the background against which the plot line is drawn—is white (`[255, 255, 255]`).
3. The default color of the plot object (and all objects, for that matter) is black. This means that it is necessary to specify a color other than black for the object if we wish it to show up against the black background.

Try changing the colors with the following statements:

```
myPlot->SetProperty, COLOR = [150, 0, 150]  
myView->SetProperty, COLOR = [75, 250, 75]  
myWindow->Draw, myView
```

How IDL Interprets Color Values

IDL determines colors to display differently based on whether the destination object uses an Indexed or RGB color model, and on whether the physical destination device supports an Indexed or RGB color model.

Indexed Color Model

If the destination object uses an Indexed color model, the color displayed is calculated from the value specified by the object's `COLOR` property as follows:

If a Color Index is Specified

- If the physical device uses an Indexed color model, the specified color index is used as an index into the physical device's lookup table. (Remember that the physical device's color lookup table is loaded via the `PALETTE` keyword to the destination object.)
- If the physical device uses an RGB color model, the specified color index is used as an index into the destination object's palette. The RGB triple stored at the index's location in the palette is used as the physical device's color value.

If an RGB Triple is Specified

- If the physical device uses an Indexed color model, the RGB triple is mapped to the index of the nearest match in the device's color lookup table.
- If the physical device uses an RGB color model, the RGB triple is passed directly to the device.

RGB Color Model

If the destination object uses an RGB color model, the color displayed is calculated from the value specified by the object's `COLOR` property as follows:

If a Color Index is Specified

If the graphic object for which the color is being determined has a palette associated with it, the RGB triple at that palette's color index is retrieved. Otherwise, the RGB triple at the specified index in the destination object's palette is retrieved.

- If the physical device uses an Indexed color model, the RGB triple retrieved is mapped to the index of the nearest match in the device's color lookup table.

- If the physical device uses an RGB color model, the RGB triple retrieved is passed directly to the device.

If an RGB Triple is Specified

- If the physical device uses an Indexed color model, the RGB triple is mapped to the index of the nearest match in the device's color lookup table.
- If the physical device uses an RGB color model, the RGB triple is passed directly to the device.



Chapter 27:

Using Attributes and Helpers

The following topics are covered in this chapter:

Overview	578	Pattern Objects	583
Font Objects	579	Symbol Objects	585
Palette Objects	582	Tessellator Objects	589

Overview

Attribute objects are not rendered directly, but are used to determine how graphic objects will be rendered. There are four attribute object classes: `IDLgrFont`, `IDLgrPalette`, `IDLgrPattern`, and `IDLgrSymbol`.

Helper objects perform operations on object instance data. There are two helper object classes: `IDLgrTessellator` and `Trackball`. For additional information the trackball object, see [“Virtual Trackball and 3D Transformations”](#) on page 563.

Font Objects

Font objects allow you to specify the type style and size used when rendering objects of the `IDLgrText` class. You can use either TrueType outline fonts or IDL's built-in Hershey vector fonts.

Fonts used by font objects are specified in a string constant constructed from a font name and one or more optional modifiers. The font name is the name by which your computer system knows the font (Times for the Times Roman font, for example). Modifiers specify the weight, angle, and other attributes of the font (Bold specifies a weight, italic an angle). The font name string looks like this:

```
'fontname*weight*angle*other_modifiers'
```

where `other_modifiers` can be any other font property supported by a given font, such as a slant. For example, the font name string for Helvetica bold italic is:

```
'helvetica*bold*italic'
```

The font name string for Times Roman Regular is:

```
'times'
```

While the font name must come first in the font name string, the order in which the modifiers are specified is not important.

IDL's default font is 12 point Helvetica regular.

See “[IDLgrFont](#)” in the *IDL Reference Guide* manual for details on creating font objects.

Determining Available Fonts

Each destination object includes a `GetFontnames` method, which returns the list of available fonts that can be used in `IDLgrFont` objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned as they are fixed—see [Appendix H, “Fonts”](#) in the *IDL Reference Guide* manual for more information.

Outline Fonts

IDL provides five TrueType outline fonts for use in font objects: Courier, Helvetica, Monospace Symbol, Symbol, and Times. Your system may support additional TrueType fonts—use them in the same way as those supplied by IDL.

The five TrueType fonts provided by IDL support the following modifiers:

Font	Modifier
Courier	bold, italic
Helvetica	bold, italic
Monospace Symbol	none
Symbol	none
Times	bold, italic

Table 27-1: TrueType Font Modifiers

Hershey Fonts

IDL supplies a set of vector fonts designed by Dr. A.J. Hershey. See [Appendix H, “Fonts”](#) in the *IDL Reference Guide* manual for information on Hershey fonts.

You can use Hershey fonts when creating font objects by specifying a fontname of the form `Hershey*fontnum` to the `IDLgrFont::Init` method.

Creating Font Objects

Specify a font name string when you create a font object. You can also specify a size, in points, for the font upon creation. For example, the following statement creates a font object using a bold version of the Times Roman font, with a size of 20 points:

```
myFont = OBJ_NEW('IDLgrFont', 'times*bold', SIZE=20)
```

To create a font object using a Hershey font, omit the font name string and specify the Hershey font’s index number with the `HERSHEY` keyword to the `IDLgrFont::Init` method. The following statement creates a font object using the Duplex Roman Hershey font, with a size of 14 points:

```
myHersheyFont = OBJ_NEW('IDLgrFont', 'hershey*5', SIZE=14)
```

Using Font Objects

To use a font object, use the `FONT` keyword to the `IDLgrText::Init` method (or change the text object’s font via the `SetProperty` method):

```
myText = OBJ_NEW('IDLgrText', 'Ay, Carumba', FONT = myFont)
```

or

```
myText->SetProperty, FONT=myHersheyFont
```

If no font object is specified, IDL uses the Helvetica font with a size of 12 points.

See [“Text Objects”](#) on page 611 for details on creating Text objects.

Font Objects and Resource Use

Because font objects are relatively complex, each font object uses a relatively large amount of system resources. As a result, it is better to re-use an existing font object than to create a second identical font object.

Palette Objects

Objects of the `IDLgrPalette` class are used to create color lookup tables. Color lookup tables assign individual numerical values to color values; this allows you to specify the color of a graphic object with a single number (a color index) rather than explicitly providing the red, green, and blue color values (an RGB triple). Palettes are most useful when you want data values to correspond to color values—that is, if you want a data value of 200, for example, to always correspond to a single color. This correspondence is one of the main uses of the Indexed Color Model. See [“Working with Color”](#) in Chapter 3 of the *Image Processing in IDL* manual for additional discussion of indexed color and its uses.

Creating Palette Objects

Specify three vectors representing the red, green, and blue values for the palette when you call the `IDLgrPalette::Init` method. The values in the red, green, and blue vectors must be integers between zero and 255, and the length of each vector must not exceed 256 elements. For example, the following statements create a palette object that reverses a standard grayscale ramp palette:

```
rval = (gval = (bval = REVERSE(INDGEN(256))))
myPalette = OBJ_NEW('IDLgrPalette', rval, gval, bval)
```

See [“IDLgrPalette”](#) in the *IDL Reference Guide* manual for details on creating palette objects.

Using Palette Objects

Palettes can be associated either with graphics destination objects (windows or printers) or with individual graphic atoms:

```
myWindow->SetProperty, PALETTE=myPalette
```

or

```
myImage->SetProperty, PALETTE=myPalette
```

Note

Palettes associated with graphic atoms are only used when the destination object uses an RGB color model; if the destination object uses an Indexed color model, the destination object’s palette is always used. See [“Working with Color”](#) in Chapter 3 of the *Image Processing in IDL* manual for details.

Pattern Objects

Objects of the `IDLgrPattern` class are used to fill objects of the `IDLgrPolygon` class. Pattern objects can create a solid fill (the default), a line fill (with control over the orientation, spacing, and thickness of the lines used), or a pattern fill (using a byte pattern you specify). Pattern objects do not have a color of their own; patterns take their color from the `COLOR` property of the polygon they fill.

Creating Pattern Objects

Specify a fill-pattern style when you call the `IDLgrPattern::Init` method. Set the argument to the `Init` method equal to zero to create a solid fill, equal to one to create a line pattern, or equal to two to use a bitmap byte array as the fill pattern. For example, the following statement creates a pattern object with a solid fill:

```
myPattern = OBJ_NEW('IDLgrPattern', 0)
```

The following statement creates a pattern object with lines ten pixels apart, 5 pixels wide, at an angle of 30 degrees:

```
myPattern = OBJ_NEW('IDLgrPattern', 1, SPACING=10, THICK=5, $
    ORIENTATION=30)
```

To create a pattern fill, specify a 32-by-4 byte array via the `PATTERN` property of the pattern object. The byte array you specify will be tiled over the area of the polygon to be filled. For example, the following statements create a pattern fill with a random speckle. The first statement creates a 32-by-4 byte array with random values ranging between 0 and 255. The second statement creates the pattern object.

```
pattern = BYTE(RANDOMN(seed, 32, 4)*255)
myPattern = OBJ_NEW('IDLgrPattern', 2, PATTERN=pattern)
```

See “[IDLgrPattern](#)” in the *IDL Reference Guide* manual for details on creating pattern objects.

Using Pattern Objects

To fill a polygon with the pattern specified by a pattern object, set the `FILL_PATTERN` property equal to the pattern object reference:

```
myPolygon->SetProperty, FILL_PATTERN = myPattern
```

The following statements create a triangle and fills it with the random speckle pattern:

```
pattern = BYTE(RANDOMN(seed, 32, 4)*255)
myPattern = OBJ_NEW('IDLgrPattern', 2, PATTERN=pattern)
```

```
myView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,10,10])
myModel = OBJ_NEW('IDLgrModel')
myPolygon = OBJ_NEW('IDLgrPolygon', [4, 7, 3], [8, 6, 3], $
    color=[255,0,255], fill_pattern=myPattern)
myView->Add, myModel
myModel->Add, myPolygon
myWindow = OBJ_NEW('IDLgrWindow')
myWindow->Draw, myView
```


Symbol Objects

Objects of the `IDLgrSymbol` class are used to display individual data points, either in an `IDLgrPlot` object or an `IDLgrPolyline` object. You can create symbol objects that display one of seven pre-defined symbols, any atomic graphic object, or any model object.

Creating Symbol Objects

Specify the type of symbol to use when you call the `IDLgrSymbol::Init` method.

To Use a Pre-defined Symbol

Specify one of the following values for the symbol type:

- 1 = Plus sign (the default)
- 2 = Asterisk
- 3 = Period
- 4 = Diamond
- 5 = Triangle
- 6 = Square
- 7 = X

For example, to create a symbol object using a red triangle for the symbol, use the following statement:

```
mySymbol = OBJ_NEW('IDLgrSymbol', 5, COLOR=[255,0,0])
```

To Use a Graphic Object as a Symbol

You can use an atomic graphic object or a model object as a symbol. For best results, create an object that fills the domain between -1 and 1 in all directions. For example, the following statements create a polygon object in the shape of a pentagon and define a symbol object to use the polygon:

```
pentagon=OBJ_NEW('IDLgrPolygon', [-0.8,0.0,0.8,0.4,-0.4], $  
    [0.2,0.8,0.2,-0.8,-0.8], COLOR=[0,0,255])  
mySymbol = OBJ_NEW('IDLgrSymbol', pentagon)
```

Note that we create the pentagon to fit in the plane between -1 and 1 in both the X and Y directions. We could also have created the pentagon to fit in a unit square and then scaled it to fit the domain between -1 and 1 .

For example:

```
pentagon=OBJ_NEW('IDLgrPolygon', [0.1,0.5,0.9,0.7,0.3], $
    [0.6,0.9,0.6,0.1,0.1], COLOR=[0,0,255])
symModel = OBJ_NEW('IDLgrModel')
symModel->Add, pentagon
symModel->Scale, 2, 2, 1
symModel->Translate, -1, -1, 0
mySymbol = OBJ_NEW('IDLgrSymbol', symModel)
```

Note

We create the symbol object to use the model object rather than the polygon object. Using a model object as a symbol allows you to apply transformations to the symbol even after it has been created.

Setting Size

By default, symbols extend one unit to each side of the data point they represent. Set the `SIZE` property of the symbol object to a two-element vector that describes the scaling factor in *X* and *Y* to apply to the symbol to change the size of the symbols that are rendered. For example, to scale a symbol so that it extends one tenth of a unit to each side of the data point, use the statement:

```
mySymbol->SetProperty, SIZE=[0.1, 0.1]
```

Setting Color

If you are using a pre-defined symbol, you can set its color using the `COLOR` property of the symbol object. If you are using a graphic object as a symbol, the symbol's color is determined by the color of the graphic object and the setting of the `COLOR` property of the symbol object itself is ignored. For example, the following statements create a symbol object that uses a red triangle:

```
mySymbol = OBJ_NEW('IDLgrSymbol', 5, COLOR=[255,0,0])
```

See [“IDLgrSymbol”](#) in the *IDL Reference Guide* manual for details on creating symbol objects.

Using Symbol Objects

To use a symbol, set the `SYMBOL` property of an `IDLgrPlot` or `IDLgrPolyline` object equal to the symbol object reference:

```
myPlot->SetProperty, SYMBOL=mySymbol
```

Suppose you wish to create a symbol object using the pentagon we created above. Suppose also that you wish to be able to use the pentagon code in more than one instance, and would like to be able to make changes to the pentagon object's color, size, and orientation. You might create a procedure like the following to define a pentagon object contained in a model object, and return the object references.

Note

You do not need to enter the example code yourself. The example code shown here is duplicated in the procedure file `penta.pro`, located in the `examples/visual` subdirectory of the IDL distribution.

```

;Allow user to set the color and retrieve the object
;references to the symbol, and model objects created.
PRO penta, COLOR=color, SYMBOL=symbol, MODEL=model
;If the color keyword is set, use the specified color.
;Otherwise, use blue.
IF KEYWORD_SET(color) THEN COLOR=color ELSE COLOR=[0,0,255]
;Create a model object.
model = OBJ_NEW('IDLgrModel')
;Create a polygon that takes up most of the domain
;between -1 and 1 in the X and Y directions. Set its color.
symbol = OBJ_NEW('IDLgrPolygon', [-0.8, 0.0, 0.8, 0.4, -0.4], $
    [0.2, 0.8, 0.2, -0.8, -0.8], COLOR=color)
;Add the polygon to the model.
model->ADD, symbol
END

```

Once you have compiled the `penta` procedure, call it with the `SYMBOL` and `MODEL` keywords set equal to named variables that will contain the object references of the model and polygon objects:

```
PENTA, SYMBOL=sym, MODEL=symmodel
```

Next, create a symbol object using the pentagon:

```
mySymbol = OBJ_NEW('IDLgrSymbol', symmodel)
```

Now, create a plot object using the pentagon as the plot symbol:

```
myPlot = OBJ_NEW('IDLgrPlot', FINDGEN(10), SYMBOL=mySymbol)
```

Next, display the plot:

```

myView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,10,10])
myModel = OBJ_NEW('IDLgrModel')
myView->Add, myModel
myModel->Add, myPlot
myWindow = OBJ_NEW('IDLgrWindow')
myWindow->Draw, myView

```

Note that the plotting symbols are larger than you might wish. Try making them smaller:

```
mySymbol->SetProperty, SIZE=[0.2,0.2]
myWindow->Draw, myView
```

Or, create the following procedure to spin the pentagons around the *z*-axis (enter `.RUN` at the command prompt, followed by these statements):

```
PRO SPIN, model, view, window, steps
FOR i = 0, steps do begin
    model->Rotate, [0,0,1], 10
    window->Draw, view
END
END
```

After compiling the `SPIN` procedure, call it from the command line and watch the pentagons spin:

```
SPIN, symmodel, myView, myWindow, 100
```

While it is unlikely that you will wish to create spinning plot symbols, this example demonstrates one of the key advantages of IDL Object Graphics over IDL Direct Graphics—once created, graphics objects can be easily manipulated in a variety of ways without the need to recreate the entire graph or image after each change.

Tessellator Objects

The `IDLgrTessellator` class is a helper class that converts a simple concave polygon (or a simple polygon with holes) into a number of simple convex polygons (general triangles). A polygon is simple if it includes no duplicate vertices, if the edges intersect only at vertices, and exactly two edges meet at any vertex.

Tessellation is useful because the `IDLgrPolygon` object accepts only convex polygons. Using the `IDLgrTessellator` object, you can convert a concave polygon into a group of convex polygons.

Creating Tessellator Objects

The `IDLgrTessellator::Init` method takes no arguments. Use the following statement to create a tessellator object:

```
myTess = OBJ_NEW('IDLgrTessellator')
```

See “[IDLgrTessellator](#)” in the *IDL Reference Guide* manual for details on creating tessellator objects.

Using Tessellator Objects

The procedure file `obj_tess.pro`, located in the `examples/visual` subdirectory of the IDL distribution, provides an example of the use of the `IDLgrTessellator` object. To run the example, enter `OBJ_TESS` at the IDL prompt. The procedure creates a concave polygon, attempts to draw it, and then tessellates the polygon and re-draws. Finally, the procedure demonstrates adding a hole to a polygon. (You will be prompted to press Return after each step is displayed.) You can also inspect the source code in the `obj_tess.pro` file for hints on using the tessellator object.



Chapter 28:

Working with Axes and Text

The following topics are covered in this chapter:

Overview	592	Text Objects	611
Axis Objects	593		

Overview

In IDL Object Graphics, axes and titles are not automatically included when plot or surface objects are created. Instead, you create axis and text objects and place them in the object hierarchy to annotate your plots and graphs.

Axis Objects

Axis objects provide a visual notation of data values in two- and three-dimensional plots and graphs. Each axis is represented by an individual axis object; that is, if you have a plot in X and Y , you will need to create an x -axis object and a y -axis object.

Note

Axis objects do not take their range values from data values or other objects, as you might expect if you are familiar with IDL Direct Graphics. Instead, axis objects have a default range of 0.0 to 1.0; you must explicitly set the range of values covered by the axis object using the RANGE property.

Creating Axis Objects

To create an axis object, specify an integer argument to the `IDLgrAxis::Init` method when calling `OBJ_NEW`. Specify 0 (zero) to create an x -axis object, 1 (one) to create a y -axis object, or 2 to create a z -axis object:

```
xaxis = OBJ_NEW('IDLgrAxis', 0)
yaxis = OBJ_NEW('IDLgrAxis', 1)
zaxis = OBJ_NEW('IDLgrAxis', 2)
```

The various keywords to the `Init` method allow you to control the number of major and minor ticks, the tick length and direction, the data range, and other attributes. For example, to create an x -axis object whose data range is between -5 and 5 , with the tick marks below the axis line, use the following command:

```
xaxis = OBJ_NEW('IDLgrAxis', 0, RANGE=[-5.0, 5.0], TICKDIR=1)
```

To suppress minor tick marks:

```
xaxis->SetProperty, MINOR=0
```

See “[IDLgrAxis](#)” in the *IDL Reference Guide* manual for details on creating axis objects.

Using Axis Objects

Suppose you wish to create an X - Y plot of some data and wish to include both x - and y -axes. First, we create some data to plot, the plot object, and the axis objects:

```
data = FINDGEN(100)
myplot = OBJ_NEW('IDLgrPlot', data)
xaxis = OBJ_NEW('IDLgrAxis', 0)
yaxis = OBJ_NEW('IDLgrAxis', 1)
```

Next, we retrieve the data range from the plot object and set the *x*- and *y*-axis objects' RANGE properly so that the axes will match the data when displayed:

```
myplot->GetProperty, XRANGE=xr, YRANGE=yr
xaxis->SetProperty, RANGE=xr
yaxis->SetProperty, RANGE=yr
```

By default, major tickmarks are 0.2 data units in length. Since the data range in this example is 0 to 99, we set the tick length to 2% of the data range instead:

```
xtl = 0.02 * (xr[1] - xr[0])
ytl = 0.02 * (yr[1] - yr[0])
xaxis->SetProperty, TICKLEN=xtl
yaxis->SetProperty, TICKLEN=ytl
```

Create model and view objects to contain the object tree, and a window object to display it:

```
mymodel = OBJ_NEW('IDLgrModel')
myview = OBJ_NEW('IDLgrView')
mywindow = OBJ_NEW('IDLgrWindow')
mymodel->Add, myplot
mymodel->Add, xaxis
mymodel->Add, yaxis
myview->Add, mymodel
```

Use the SET_VIEW procedure to add an appropriate viewplane rectangle to the view object. (See [“Finding an Appropriate View Volume”](#) on page 550 for information on SET_VIEW).

```
SET_VIEW, myview, mywindow
```

Now, display the plot:

```
mywindow->Draw, myview
```

The above example code is included in a procedure file named `obj_axis.pro`, located in the `examples/visual` subdirectory of the IDL distribution. You can run the example code by entering `obj_axis` at the IDL prompt. You can also examine the `.pro` file itself for examples of some of the topics discussed in this section.

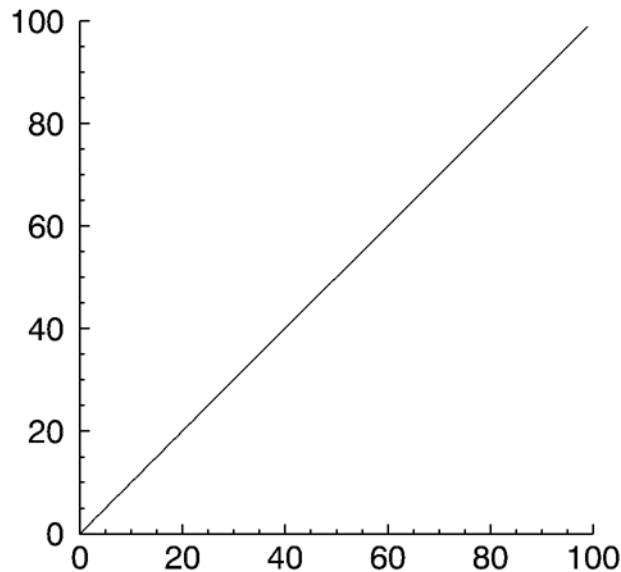


Figure 28-1: Axis Object

Logarithmic Axes

Creating a plot of logarithmic data requires that you create a logarithmic axis as well. The following example first creates a linear plot, then takes a logarithm of the same data and creates a log-linear plot.

The example code below is included in a procedure file named `obj_logaxis.pro`, located in the `examples/visual` subdirectory of the IDL distribution. You can run the example code by entering `obj_logaxis` at the IDL prompt. You can also examine the `.pro` file itself for examples of some of the topics discussed in this section.

```
;Create a window and a view.
PRO obj_logaxis
oWindow = OBJ_NEW('IDLgrWindow')

;Create a model for the graphics; add to the view.
oView = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[-0.2,-0.2,1.4,1.4])
oModel = OBJ_NEW('IDLgrModel')
```

```

oView->Add, oModel

;Create some simple data:
yData = FINDGEN(50)*20.

;Compute data range in X and Y:
yMin = MIN(yData, MAX=yMax)
yRange = yMax - yMin
xMin = 0
xMax = N_ELEMENTS(yData)-1
xRange = xMax - xMin

;Create an X-axis with a title:
oXTitle = OBJ_NEW('IDLgrText', 'Linear X Axis')
oXAxis = OBJ_NEW('IDLgrAxis', 0, RANGE=[xmin,xmax], $
    TICKLEN=(0.1*yRange), TITLE=oXTitle)
oModel->Add, oXAxis

;Create a Y-axis with a title:
oYTitle = OBJ_NEW('IDLgrText', 'Linear Y Axis')
oYAxis = OBJ_NEW('IDLgrAxis', 1, RANGE=[yMin,yMax], $
    TICKLEN=(0.1*xRange), TITLE=oYTitle)
oModel->Add, oYAxis

;Create a plot of the data:
oPlot = OBJ_NEW('IDLgrPlot', yData, COLOR=[255,0,0])
oModel->Add, oPlot

;Scale and translate the model so the plot fits within the view:
oModel->Scale, 1.0/xRange, 1.0/yRange, 1.0
oModel->Translate, -(xMin/xRange), -(yMin/yRange), 0.0

;Ensure that axis text recomputes its dimensions as needed:
oXAxis->GetProperty, TICKTEXT=oXTickText
oXTitle->SetProperty, RECOMPUTE_DIMENSIONS=2
oXTickText->SetProperty, RECOMPUTE_DIMENSIONS=2
oYAxis->GetProperty, TICKTEXT=oYTickText
oYTickText->SetProperty, RECOMPUTE_DIMENSIONS=2
oYTitle->SetProperty, RECOMPUTE_DIMENSIONS=2

;Draw the plot:
oWindow->Draw, oView

;Refresh the plot when ready:
val=''
READ, val, PROMPT='Press <Return> to refresh the window.'
oWindow->Draw, oView

;Now that the original plot has been displayed,

```

```

;switch to a logarithmic version of the plot when ready:
READ, val, $
    PROMPT='Press <Return> to draw with a logarithmic Y axis.'

;Only positive values are valid when computing
;the logarithmic data:
posElts = WHERE(yData GT 0, nPos)
IF (nPos GT 0) THEN BEGIN
    ;Compute new Y range:
    yValidData = yData(posElts)
    yValidMin = MIN(yValidData, MAX=yValidMax)

    ;Compute logarithmic data:
    yLogData = ALOG10(yValidData)

    ;Update the plot data:
    oPlot->SetProperty, DATAY=yLogData
ENDIF ELSE BEGIN
    MESSAGE, 'Original plot data is entirely non-positive.', $
        /INFORMATIONAL
    MESSAGE, 'Log plot will contain no data.', /NOPREFIX, $
        /INFORMATIONAL

    ;Create a fake log axis range:
    yValidMin = 1.0
    yValidMax = 10.0

    ;Simply hide the plot, since no valid log data exists:
    oPlot->SetProperty, /HIDE
ENDELSE

;Update the Y axis to be logarithmic, and modify the Y axis title:
oYAxis->SetProperty, /LOG, RANGE=[yValidMin, yValidMax]
oYTitle->SetProperty, STRING='Logarithmic Y Axis'

;Get the new Y axis logarithmic range:
oYAxis->GetProperty, CRANGE=crange
yLogMin = crange[0]
yLogMax = crange[1]
yLogRange = yLogMax - yLogMin

;Update the X axis ticklen:
oXAxis->SetProperty, TICKLEN=(0.1*yLogRange), $
    LOCATION=[0, yLogMin, 0]

;Update the model transform to match the new data ranges:
oModel->Reset
oModel->Scale, 1.0/xRange, 1.0/yLogRange, 1.0
oModel->Translate, -(xMin/xRange), -(yLogMin/yLogRange), 0.0

```

```
oWindow->Draw, oView  
READ, val, PROMPT='Press <Return> to quit.'  
OBJ_DESTROY, oView  
OBJ_DESTROY, oWindow  
OBJ_DESTROY, oXTitle  
OBJ_DESTROY, oYTitle  
END
```

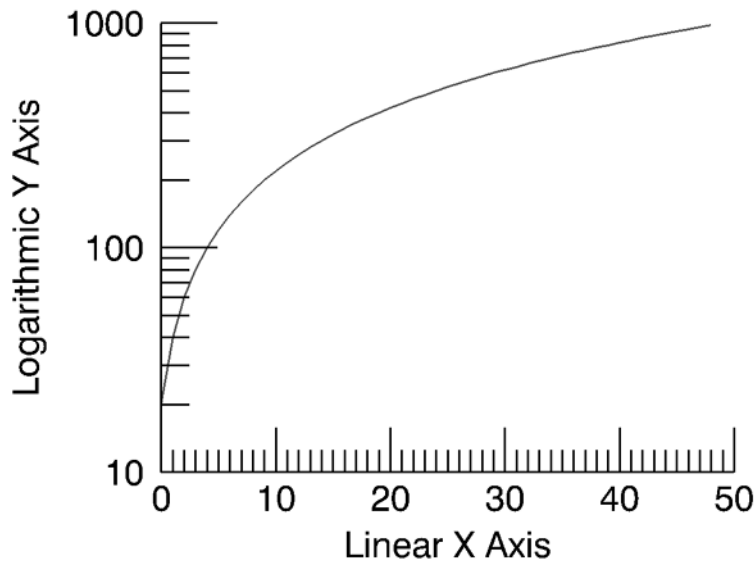


Figure 28-2: Logarithmic Axes

Date/Time Axes

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

Julian Dates and Times

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

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

Table 28-1: Example Julian Dates

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

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

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

Note

Julian values must be in the range -1095 to 1827933925, which corresponds to calendar dates 1 Jan 4716 B.C.E. and 31 Dec 5000000, respectively.

Precision of Date/Time Data

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

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

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

```

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

; Get machine characteristics
machine = MACHAR(/DOUBLE)

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

; Convert to seconds
PRINT, precision*86400d0

```

How to Generate Date/Time Data

The **TIMEGEN** function returns an array of double precision floating point values that represent date/time in terms of Julian dates. The first value of the returned array corresponds to a start date/time, and each subsequent value corresponds to the start

date/time plus that array element's one-dimensional subscript multiplied by a step size for a given date/time unit. Unlike the other array generation routines in IDL, TIMEGEN includes a START keyword, which is necessary if the starting date/time is originally provided in calendar (month, day, year) form.

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

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

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

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

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

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

2. Using the calendar format codes:

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

The resulting calendar dates are printed out as follows:

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

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

Displaying Date/Time Data on an Axis in Object Graphics

You can display date/time data on plots, contours, and surfaces through the tick settings of the date/time axis. Date/time data can be displayed on any axis (x, y or z). The date/time data is stored as Julian dates, but the LABEL_DATE routine and axis

keywords allow you to display this data as calendar dates. The following examples show how to display one-dimensional and two-dimensional date/time data.

Displaying Date/Time Data on a Plot Display

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

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

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

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

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

where %I represents minutes and %S represents seconds.

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

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

The oPlotModel object will contain the IDLgrPlot and IDLgrAxis objects. The oPlotView object contains the oPlotModel object with the DOUBLE keyword. The DOUBLE keyword is set for the oPlotView and oPlot objects because the date/time data is made up of double-precision floating-point values.

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

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

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

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

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

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

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

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

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

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

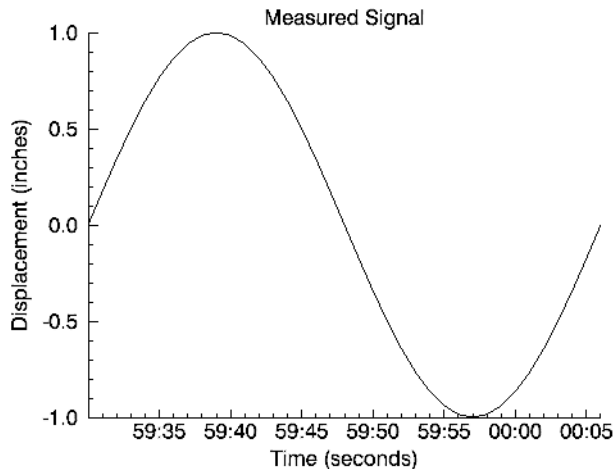


Figure 28-3: Displaying Date/Time data with IDLgrPlot

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

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

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

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

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

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

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

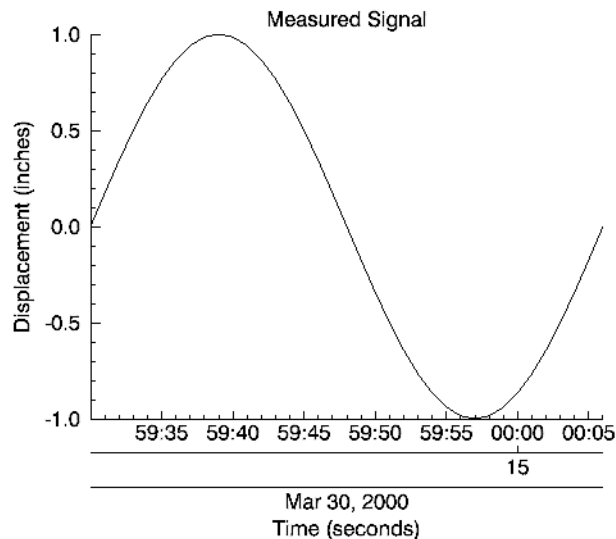


Figure 28-4: Displaying Three Levels of Date/Time data with IDLgrPlot

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

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

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

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

Displaying Date/Time Data on a Contour Display

Another possible example may be the surface temperature (in degrees Celsius) of each degree of a single circle on a sphere recorded at every second for 37 seconds

after the initial recording of 59 minutes and 30 seconds after 2 o'clock pm (14 hundred hours) on the 30th day of March in the year 2000:

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

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

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

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

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

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

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

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

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

The oContourModel object will contain the IDLgrContour and IDLgrAxis objects. The oContourView object contains the oContourModel with the DOUBLE keyword. The DOUBLE and DOUBLE_GEOM keywords are set for the oContourView and

`oContour` objects because date/time data is made up of double-precision floating-point values.

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

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

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

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

The `TICKFORMAT`, `TICKINTERVAL`, and `TICKUNITS` keywords specify the Y-axis as a date/time axis, which contains three levels related to the formats presented in the call to the `LABEL_DATE` routine. This example also contains the

TICKLAYOUT keyword. By default, this keyword is set to 0, which provides the date/time layout shown in the plot example. In this example, TICKLAYOUT is set to 2, which rotates and boxes the tick labels.

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

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

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

```
oContourWindow->Draw, oContourView
```

The Draw method to oContourWindow produces the following results:

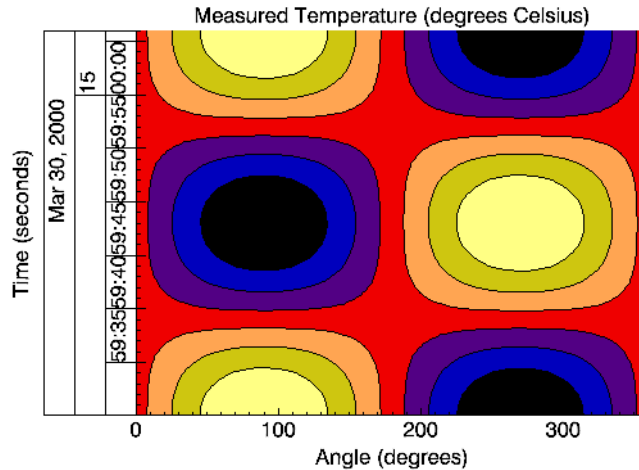


Figure 28-5: Displaying Date/Time data with IDLgrContour

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

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

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

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

Axis Titles and Tickmark Text

You can supply an axis title for an axis by setting the `TITLE` property equal to the object reference of an `IDLgrText` object. Text objects connected to axis objects via the `TITLE` property are automatically centered under or next to the axis they belong with.

Note

Titles and tickmark text inherit the color specified for the `IDLgrAxis` object itself, even if the `COLOR` property is specified for the `IDLgrText` object specified, unless the `USE_TEXT_COLOR` property for the axis is nonzero.

By default, major tick marks are labelled with the data values. You can supply a set of tickmark text values by setting the `TICKTEXT` property equal to either a single instance of an `IDLgrText` object containing a vector of text strings or to a vector of `IDLgrText` objects, each of which contains a single text string.

Note

Make sure that you have the same number of tick label strings as there are major tick marks for the axis.

Reverse Axis Plotting

IDL also allows you to plot data in Object Graphics by reversing the order of axis tick values. This is known as reverse axis plotting.

When using Object Graphics, each core object is a building block. Any number of building blocks may be combined together in a hierarchical tree to create an overall scene. An individual object is not aware of the other objects in the hierarchy; therefore, the designer of the hierarchy must control all interactions between the objects. For example, to properly display a reverse axis plot in Object Graphics, the designer must appropriately set the properties on the X axis, the Y axis, and the plot line, each of which contribute to the overall displayed results.

The following figure demonstrates how you can reverse the order of axis tick values using Object Graphics. You can run this example by entering `EX_REVERSE_PLOT` at the IDL command line. You can view the source for this example, `EX_REVERSE_PLOT.PRO`, in the `examples/visual` directory.

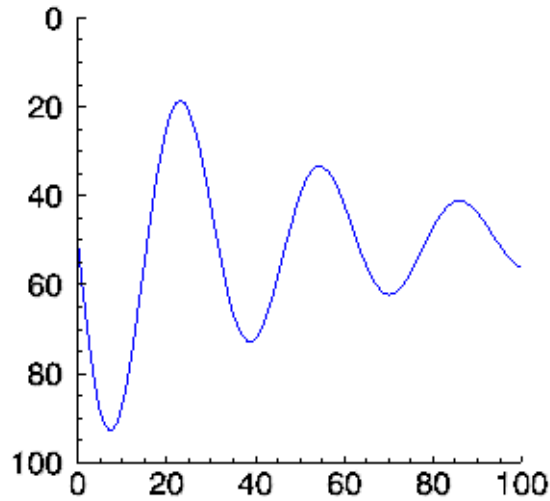


Figure 28-6: Reverse Axis Plotting Example

Text Objects

Text objects contain string values that are drawn to the destination object at a location you specify. You have control over the font used (via an `IDLgrFont` object), the angle of the text baseline, and the vertical direction of the text.

Creating Text Objects

To create a text object, specify a string or an array of strings to the `IDLgrText:Init` method when calling `OBJ_NEW`.

```
mytext = OBJ_NEW('IDLgrText', 'A Text String')
```

or

```
mytextarr = OBJ_NEW('IDLgrText', $  
  ['First String', 'Second String', 'Third String'])
```

See “[IDLgrText](#)” in the *IDL Reference Guide* manual for details on creating text objects.

Using Text Objects

Creating text annotations in their simplest form—two-dimensional text displayed at a given location—involves only specifying the text, and the location. For example, to display the words `Text String` in a window in the default font, the following statements suffice:

```
mywindow = OBJ_NEW('IDLgrWindow', DIMENSIONS=[400,400])  
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,10,10])  
mymodel = OBJ_NEW('IDLgrModel')  
mytext = OBJ_NEW('IDLgrText', 'Text String', LOCATION=[4,4], $  
  COLOR=[50,100,150])  
myview->Add, mymodel  
mymodel->Add, mytext  
mywindow->Draw, myview
```

The text is drawn at the specified location, with the baseline parallel to the x -axis.

Location and Alignment

Specifying a location via the `LOCATION` property picks a point in space where the text object will be placed. By default, text objects are aligned with their lower left edge located at the point specified by the `LOCATION` property.

You can change the horizontal position of the text object with respect to the point specified by `LOCATION` by changing the `ALIGNMENT` property to a floating-point value between 0.0 and 1.0. The default value (0.0) aligns and left-justifies text at the location specified. Setting `ALIGNMENT` to 1.0 right-justifies the text; setting it to 0.5 centers the text above the point specified. The vertical position with respect to location can also be set using the `VERTICAL_ALIGNMENT` property. The default value (0.0) bottom-justifies the text at the given location. A vertical alignment of 1.0 top-justifies the text.

3D Text and Text “On the Glass”

Text objects, like all graphics atoms, are located and oriented in three-dimensional space. (We often ignore the third dimension when making simple plots and graphs—in these cases we simply use the default z value of zero.) With text objects, however, there is an option to project text on the glass.

Projecting text on the glass ensures that it is displayed as if it were in flat, two-dimensional space no matter what its true orientation in three-dimensional space may be. In cases where text objects may be rotated at arbitrary angles, projecting on the glass ensures that the text will be readable.

To project text on the glass, set the `ONGLASS` property of the text object to a value other than zero.

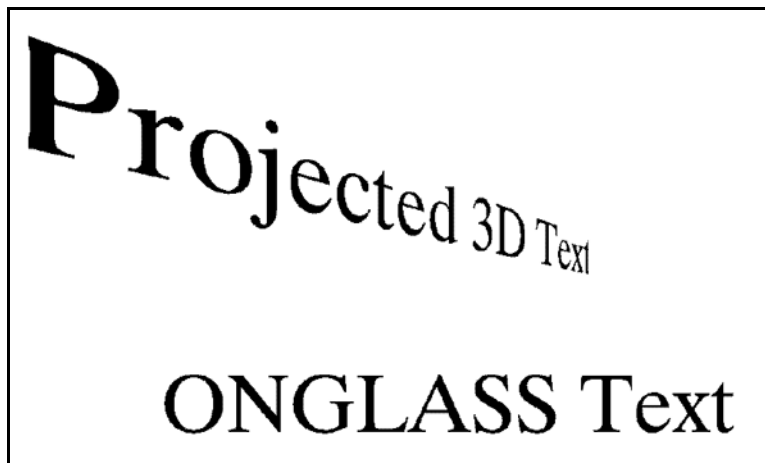
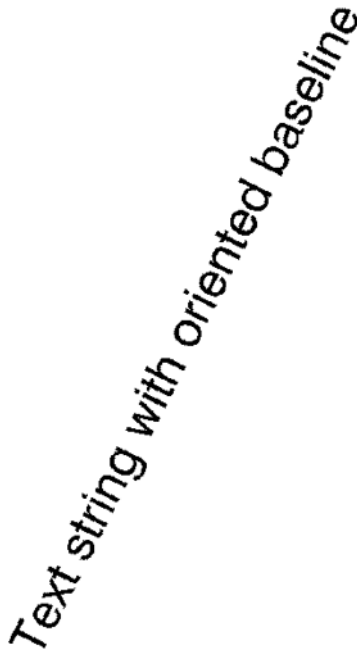


Figure 28-7: 3D Text and Text “On the Glass”

Baseline

The text baseline can be altered from its default orientation (parallel to the x -axis) by setting the text object's `BASELINE` property to a two- or three-element array. The new baseline will be oriented parallel to a line drawn between the origin and the coordinates specified. For example, the following statement makes the text baseline parallel to a line drawn between the points $[0, 0]$ and $[1, 2]$:

```
mytext->SetProperty, BASELINE=[1,2]
```



The image shows the text "Text string with oriented baseline" rotated 45 degrees counter-clockwise. The text is black and centered on a white background. Below the text is a horizontal line.

Figure 28-8: Baseline

The following statement makes the baseline parallel to a line drawn between the origin and a point located at $[2, 1, 3]$:

```
mytext->SetProperty, BASELINE=[2,1,3]
```

Notice that the orientation of the baseline is only an orientation; changing value of the `BASELINE` property does not change the location of the text object.

Upward Direction

In addition to the baseline orientation, you can control the upward direction of the text object. (The upward direction is the direction defined by a vector pointing from the origin to the point specified.) The upward direction defines the plane on which text is drawn; by specifying a baseline and an upward direction, you define the plane.

Note

The upward direction does not specify a slant angle. That is, even if you specify a direction that is not perpendicular to the baseline for the upward direction, the text will still be perpendicular to the baseline. All that matters is the plane defined by the baseline and upward direction.

For example, in the default situation, the baseline is oriented parallel to the x -axis, and the upward direction is parallel to the y -axis, pointing in the positive y direction.

Warning

If the baseline and upward direction are coincident—that is, if they do not define a plane on which to draw the text—IDL generates an error message.

Fonts

The type style and size of the characters displayed in a text object are controlled by the FONT property. Set the FONT property equal to the object reference of an IDLgrFont object to use that font's properties for the text object. If no font object is specified, IDL uses the default font (12 point Helvetica regular).

Font objects are discussed in “[Font Objects](#)” on page 579.

A Text Example

An example procedure named `rot_text.pro` is included in the `examples/visual` subdirectory of the IDL distribution. This file creates a simple text string, rotates it around the y - and z -axes using the `BASELINE` and `UPDIR` properties, and displays several different fonts.

You can run the example code by entering `rot_text` at the IDL prompt. You can also examine the `.pro` file itself for examples of some of the topics discussed in this section.



Chapter 29:

Working with Plots and Graphs

This chapter describes the use of contour, polygon, polyline, and plot objects to create plots and graphs. The following topics are covered in this chapter:

Contour Objects	616	Plot Objects	626
Polygon Objects	619	Legend Objects	630
Polyline Objects	625	A Plotting Routine	633

Contour Objects

Contour objects create a set of contour lines from data stored in a rectangular array or in a set of unstructured points. Contour objects can consist either of lines or of filled regions.

Creating Contour Objects

To create a contour object, provide a vector or two-dimensional array containing the values to be contoured to the `IDLgrContour::Init` method. For example, the following statement creates a contour from a two-dimensional array returned by the IDL `DIST` function:

```
mycontour = OBJ_NEW('IDLgrContour', DIST(20))
```

See “[IDLgrContour](#)” in the *IDL Reference Guide* manual for details on creating contour objects.

Using Contour Objects

Contour objects have a number of properties that determine how they are rendered. See “[IDLgrContour Properties](#)” in the *IDL Reference Guide* manual for a complete listing. The following code displays the contour object created above in the *X-Y* plane.

Note

In order to display the contour as on the plane (rather than as a three-dimensional image), you must set the `PLANAR` property of the contour object equal to one and explicitly set the `GEOMZ` property equal to zero.

```
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[0,0,19,19])
mymodel = OBJ_NEW('IDLgrModel')
data = DIST(20)
mycontour = OBJ_NEW('IDLgrContour', data, COLOR=[100,150,200], $
    C_LINestyle=[0,2,4], /PLANAR, GEOMZ=0, C_VALUE=INDGEN(20))

myview->Add, mymodel
mymodel->Add, mycontour
mywindow->Draw, myview
```


This results in the following figure.

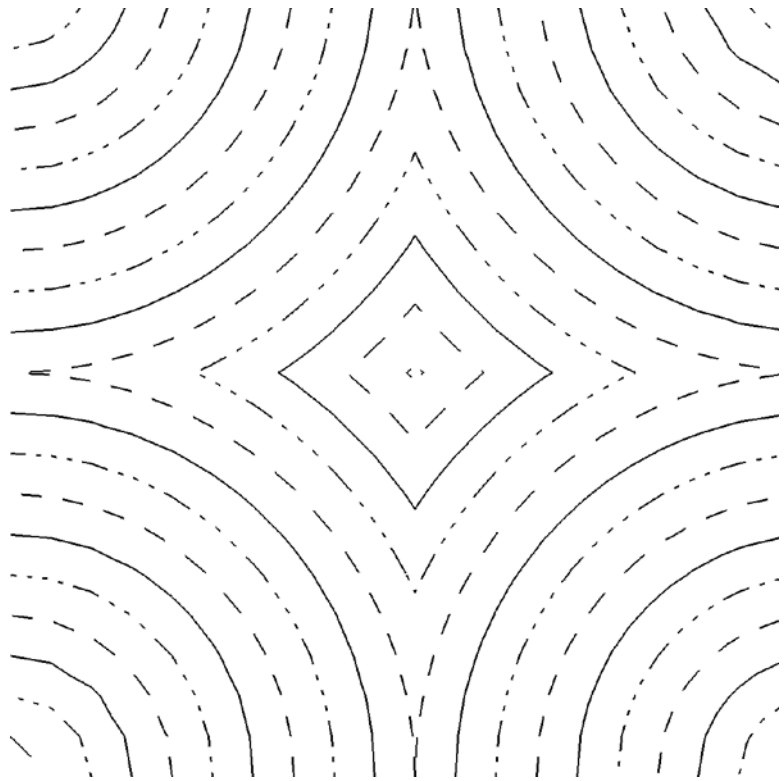


Figure 29-1: Contour Object

A more complex example using a contour object is shown in the contour demo. To start the demos, type `demo` at the IDL command prompt. Both the terrain elevation and vehicle tire data sets are displayed using the contour object.

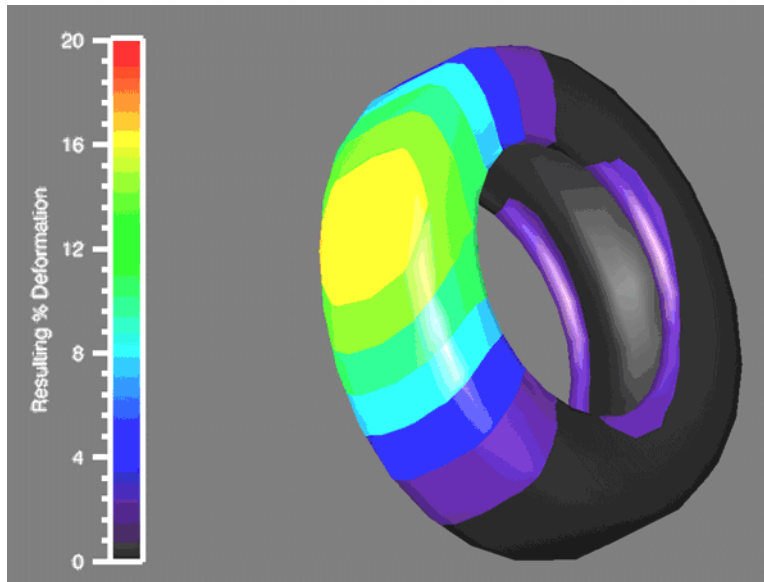


Figure 29-2: Complex Contour Object

Polygon Objects

Polygon objects represent one or more filled polygons that share a given set of vertices and rendering attributes. All polygons must be simple (the edges of the polygon should not intersect) and convex (the shape of the polygon should not have any indentations). Concave polygons can be converted into convex polygons using the helper object `IDLgrTessellator`. See “[Tessellator Objects](#)” on page 589 for more on tessellator objects.

Creating Polygon Objects

To create a polygon object, provide a two- or three-dimensional array (or two or three vectors) containing the locations of the polygon’s vertices to the `IDLgrPolygon::Init` method. For example, the following statement creates a square with sides one unit in length, with the lower left corner at the origin:

```
mypolygon = OBJ_NEW('IDLgrPolygon', [[0,0], [0,1], [1,1], [1,0]])
```

See “[IDLgrPolygon](#)” in the *IDL Reference Guide* manual for details on creating polygon objects.

Using Polygon Objects

Polygon objects have numerous properties controlling how they are rendered. You can set these properties when creating the polygon object, or use the `SetProperty` method to the polygon object to change these properties after creation.

Style

Set the `STYLE` property to an integer value that controls how the polygon is rendered. Set the `STYLE` property equal to 0 (zero) to render only the vertices. The following statement changes the polygon to display only the vertex points, in blue:

```
mypolygon->SetProperty, STYLE=0, COLOR=[0,0,255]
```

Set the `STYLE` property equal to 1 (one) to render the vertices and lines connecting them. The following statement draws the polygon’s outline in green:

```
mypolygon->SetProperty, STYLE=1, COLOR=[0,255,0,]
```

The default setting for the `STYLE` property is 2, which produces a filled polygon. The following statement draws the filled polygon in red:

```
mypolygon->SetProperty, STYLE=2, COLOR=[255,0,0]
```

Vertex Colors

You can supply a vector of vertex colors via the `VERT_COLORS` property. The colors in the vector will be applied to each vertex in turn. If there are more vertices than colors supplied for the `VERT_COLORS` property, IDL will cycle through the colors. For example, the following statements color each vertex and connecting line one of four colors:

```
vcolors = [[0,100,200],[200,150,200],[150,200,250],[250,0,100]]
mypolygon->SetProperty, STYLE=1, VERT_COLORS=vcolors
```

Fill Patterns

As demonstrated in “[Pattern Objects](#)” on page 583, you can fill a polygon with a pattern contained in an `IDLgrPattern` object. Set the `FILL_PATTERN` property equal to the object reference of the pattern object. If you have created a pattern object called `mypattern`, the following statement uses that pattern as the polygon’s fill pattern:

```
mypolygon->SetProperty, STYLE=2, FILL_PATTERN=mypattern
```

Shading

IDL provides two types of shading for filled objects. In Flat shading, the color of the first vertex in each polygon is used to define the color for the entire polygon. The polygon color has a constant intensity. In Gouraud shading, the colors along each line are interpolated between vertex colors, and then along scanlines from each of the edge intensities.

Set the `SHADING` property of the polygon object equal to 0 (zero) to use flat shading (this is the default), or equal to 1 (one) to use Gouraud shading. In the above example using vertex colors, adding the following statement:

```
mypolygon->SetProperty, STYLE=2, SHADING=1
```

creates a polygon fill in which the color values are interpolated between the vertex colors.

Texture Mapping

You can map an image onto a polygon object by specifying an `IDLgrImage` object to the `TEXTURE_MAP` property. The `TEXTURE_COORD` property defines how individual data points within the image data are mapped to the polygon’s vertices. Note that you must specify both `TEXTURE_MAP` and `TEXTURE_COORD` to enable texture mapping.

Polygon Mesh Optimization

IDLgrPolygon objects consist of a set of vertices and, optionally—via the POLYGON keyword—a connectivity array describing how those vertices are to be connected to form one or more polygons. Internally, IDL can identify three special types of polygonal meshes that may be represented very efficiently and therefore displayed substantially faster than individually described polygons. These special mesh types are characterized by repetitive patterns in the connectivity of the vertices. In performance terms, it is to the users advantage to utilize this optimization whenever possible by appropriately preparing the connectivity list according to the rules described for the corresponding type of mesh. The special mesh types are as follows:

Quad Strips

A quad strip is a connected set of four-sided polygons. To take advantage of accelerated quad strips, the connectivity should be set up so that the first and last vertex for one quad are the same as the second and third of the previous quad. See the figure below.

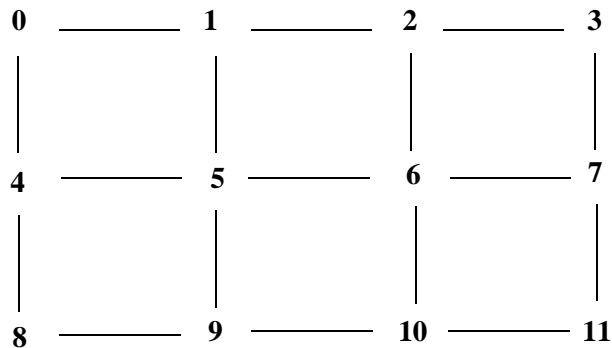


Figure 29-3: Quad Strip Mesh

For example, to use a quad strip optimization for the polygons shown above, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10 ,v11]
oPoly = OBJ_NEW(IDLgrPolygon, verts, $
  POLYGON=[4, 0, 1, 5, 4, $
            4, 1, 2 ,6, 5, $
            4, 2, 3, 7, 6, $
            4, 4, 5, 9, 8, $
            4, 5, 6, 10, 9, $
            4, 6, 7, 11, 10])
```

An alternate connectivity list that still uses quad strip optimization can also be used as in the following example, which orients each quad in the opposite direction of the first example.

```
oPoly = OBJ_NEW('IDLgrPolygon', verts, $
  POLYGON=[4, 4, 5, 1, 0, $
            4, 5, 6, 2, 1, $
            4, 6, 7, 3, 2, $
            4, 8, 9, 5, 4, $
            4, 9, 10, 6, 5,$
            4, 10, 11, 7, 6])
```

Triangle Fans

A triangle fan is a set of connected triangles that all share a common vertex. To take advantage of accelerated triangle fans, the connectivity should be set up so that the first vertex in every triangle is the common vertex, and the second vertex is the same as the last vertex of the previous triangle, as shown below.

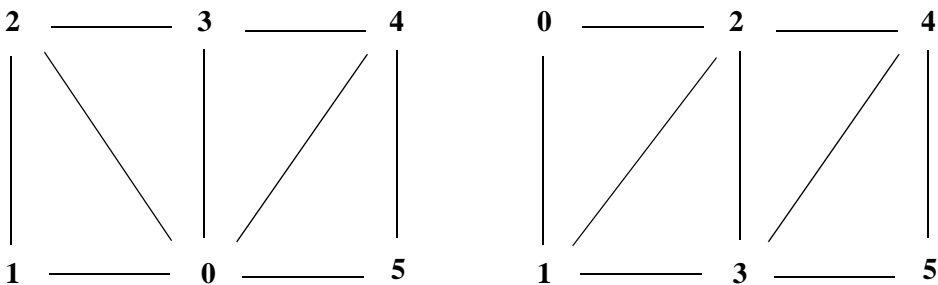


Figure 29-4: Triangle Fan Mesh (left) and Triangle Strip Mesh (right)

For example, to use a triangle fan optimization for the polygons shown in the left side of the figure, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5]
oPoly = OBJ_NEW(IDLgrPolygon, verts, $
    POLYGON=[3, 0, 1, 2, $
              3, 0, 2, 3, $
              3, 0, 3, 4, $
              3, 0, 4, 5])
```

Triangle Strips

A triangle strip is a set of connected triangles, each of which share two vertices with the previous triangle. To take advantage of accelerated triangle strips, the connectivity should be set up so that the first two vertices in every triangle must have been in the previous triangle and ordered in the same direction (counter-clockwise or clockwise) and the final vertex must be new, as shown in the right side of the previous figure.

For example, to use the triangle strip optimization for the polygons shown in the right-hand figure, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5]
oPoly = OBJ_NEW(IDLgrPolygon, verts, $
    POLYGON=[3, 0, 1, 2, $
              3, 2, 1, 3, $
              3, 2, 3, 4, $
              3, 4, 3, 5])
```

No limits are imposed on the number of meshes or types of meshes within any given polygon object. A single POLYGON keyword value might contain any combination of quad strips, triangle strips, triangle fans, or non-specialized polygons.

As the length of the strips or fans grows, and as the percentage of vertex connections that are optimized by the rules described above increases, the performance upgrade becomes more perceptible. The optimizations are a result of minimizing the time required to perform vertex transforms. If the drawing of the polygons are otherwise limited by fill-rate (as might occur on some systems if texture-mapping is being applied, for instance), these optimizations may not be of significant benefit. In any case, performance will not be hindered in any way by utilizing these specialized meshes, so it is suggested that they be applied whenever possible.

Note

The IDLgrSurface object always takes advantage of the quad mesh optimization automatically without programmer intervention.

Normal Computations

For `IDLgrPolygon` objects, normal vectors are computed by default at each vertex by averaging the normals of the polygons that share that vertex. These normals are then used to compute illumination intensities across the surface of the polygon.

Computing default normals is a computationally expensive operation. Each time the polygon is drawn, this computation will be repeated if the polygon has changed significantly enough to warrant a new internal cache (for example, if the connectivity, vertices, shading, or style have changed). In some cases, the normals do not actually change as other modifications are made. In these cases, the expense of default normal computation can be bypassed if the user provides the normals explicitly (via the `NORMALS` keyword). The provided normals will be reused every time the polygon is drawn (without further computation) until they are replaced explicitly by the user. See [COMPUTE_MESH_NORMALS](#) in the *IDL Reference Guide* for more information.

Polyline Objects

Polyline objects lines connect a series of points in two- or three-dimensional space.

Creating Polyline Objects

To create a polyline object, provide a 2-by- n or 3-by- n array (or two or three vectors) containing the locations of the polyline's constituent points to the `IDLgrPolyline::Init` method. For example, the following statement creates a line from the origin, to the point $X = 1, Y = 2$, then to the point $X = 4, Y = 3$:

```
mypolyline = OBJ_NEW('IDLgrPolyline', [[0,0], [1,2], [4,3]])
```

See [“IDLgrPolyline”](#) in the *IDL Reference Guide* manual for details on creating polyline objects.

Using Polyline Objects

Polyline objects have numerous properties controlling how they are rendered. You can set these properties when creating the polyline object, or use the `SetProperty` method to the polyline object to change these properties after creation.

Symbols

You can specify a symbol to render at each point in the polyline's path by setting the `SYMBOL` property to the object reference of an `IDLgrSymbol` object (or to an array of `IDLgrSymbol` objects). See [“Symbol Objects”](#) on page 585 for details.

Shading and Vertex Coloring

Polyline object can be shaded or their vertex points colored in the same manner as polygon objects. See [“Shading”](#) and [“Vertex Colors”](#) in [“Using Polygon Objects”](#) on page 619 for details.

Plot Objects

Plot objects maps a set of abscissa values to a set of ordinate values and creates a polyline connecting the points. Note that plot objects do not automatically create axes for the plot lines they create.

Creating Plot Objects

Create a plot line by providing a vector of *Y* values, and, optionally, a vector of *X* values. If no *X* values are provided, the *Y* values are plotted against the element indices of the *Y* vector.

The following statement creates a plot object plotting the values [2, 9, 4, 4, 6, 2, 8] against their own indices:

```
myplot = OBJ_NEW('IDLgrPlot', [2,9,4,4,6,2,8])
```

The following statements plot the same data versus a series of primes:

```
datay = [2,9,4,4,6,2,8]
datax = [0,1,2,5,7,11,13]
myplot = OBJ_NEW('IDLgrPlot', datax, datay)
```

See “[IDLgrPlot](#)” in the *IDL Reference Guide* manual for details on creating plot objects.

Using Plot Objects

Plot objects can be configured to draw regular *X* vs. *Y*, histogram, or polar plots. Set the HISTOGRAM property to create a histogram plot, or the POLAR property to create a polar plot. The following example uses the same data set to create a standard plot, a histogram plot, and a standard plot using a boxcar filter. All three plots are displayed in the same view.

```
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[-10,-10,20,20])
mymodel = OBJ_NEW('IDLgrModel')

x = (FINDGEN(21) / 10.0 - 1.0) * 10.0
y = [3.0, -2.0, 0.5, 4.5, 3.0, 9.5, 9.0, 4.0, 1.0, -8.0, $
     -6.5, -7.0, -2.0, 5.0, -1.0, -2.0, -6.0, 3.0, 5.5, 2.5, -3.0]
myplot1 = OBJ_NEW('IDLgrPlot', x, y, COLOR=[120, 120, 120])
myplot2 = OBJ_NEW('IDLgrPlot', x, y, /HISTOGRAM, LINESSTYLE=4)
y2 = SMOOTH(y, 5)
myplot3 = OBJ_NEW('IDLgrPlot', x, y2, LINESSTYLE=2)
```

```
myview->Add, mymodel  
mymodel->Add, myplot1  
mymodel->Add, myplot2  
mymodel->Add, myplot3  
  
mywindow->Draw, myview
```



Figure 29-5: Plot Object

Minimum and Maximum Values

You can control the minimum and maximum values of data plotted by a plot object. Set the `MAX_VALUE` property of the plot object to disregard data values higher than a specified value. Set the `MIN_VALUE` property to disregard data values lower than a specified value. Floating-point Not-a-Number (NaN) values are also treated as missing data and are not plotted.

For example, the following statement changes the minimum and maximum values of the histogram plot, and re-draws the view object:

```
myplot2->SetProperty, MAX_VALUE=8, MIN_VALUE=2  
mywindow->Draw, myview
```

Using Plotting Symbols

Set the SYMBOL property of a plot object equal to the object reference of a symbol object to display that symbol at each data point. For example, to use a triangle symbol at each data point, create the following symbol object, set the plot object's SYMBOL property, and re-draw:

```
mySymbol = OBJ_NEW('IDLgrSymbol', 5, SIZE=[.3,.3])
myplot1->SetProperty, SYMBOL=mySymbol
mywindow->Draw, myview
```

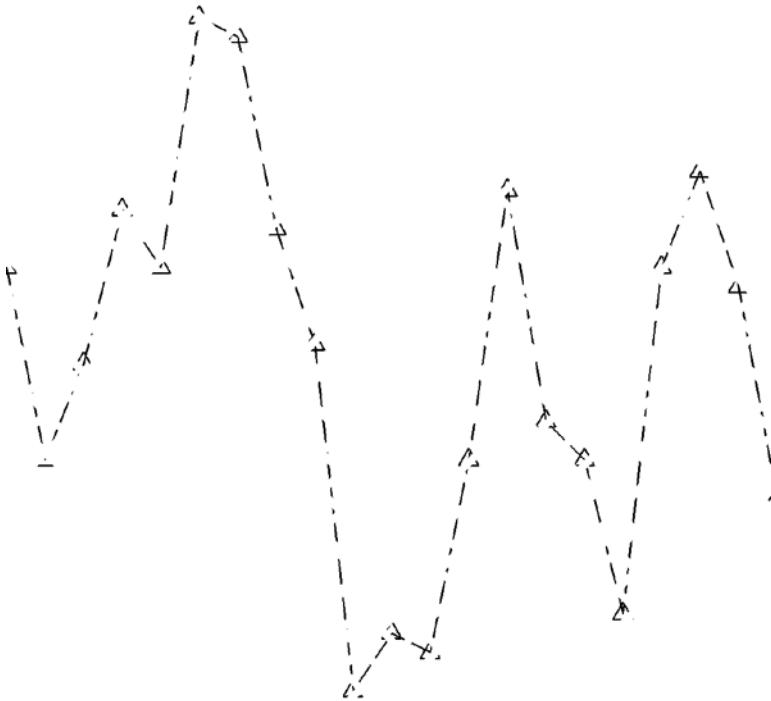


Figure 29-6: Plotting Symbols

Averaging Points

Use the NSUM property of the plot object to average the values of a group of data points before plotting. If there are m data points, m/NSUM data points are plotted. For example, the following statement causes IDL to average pairs of data points when plotting the line for the histogram plot:

```
myplot2->SetProperty, NSUM=2  
mywindow->Draw, myview
```

Polar Plots

To create a polar plot, provide a vector of radius values, a vector of theta values, and set the POLAR property to a nonzero value. The following example creates a simple polar plot:

```
mywindow = OBJ_NEW('IDLgrWindow')  
myview = OBJ_NEW('IDLgrView', VIEWPLANE_RECT=[-100,-100,200,200])  
mymodel = OBJ_NEW('IDLgrModel')  
r = FINDGEN(100)  
theta = r/5  
mypolarplot = OBJ_NEW('IDLgrPlot', r, theta, /POLAR)  
myview->Add, mymodel  
mymodel->Add, mypolarplot  
mywindow->Draw, myview
```

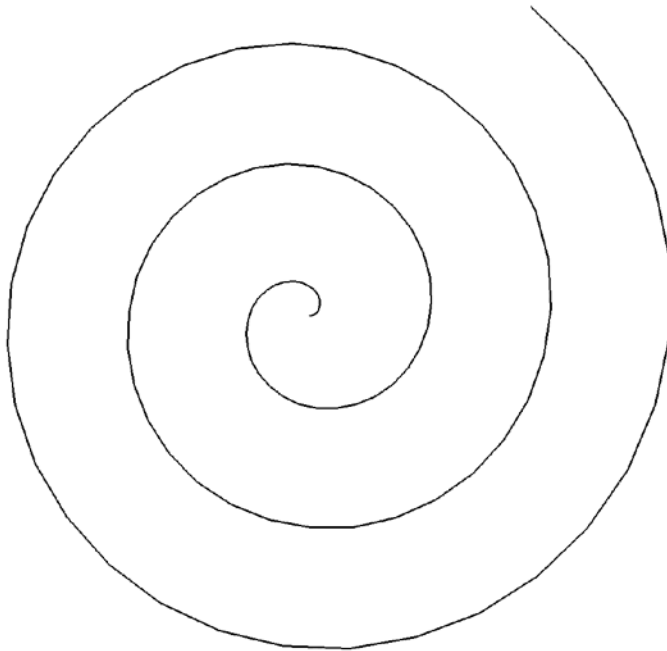


Figure 29-7: Polar Plot

Legend Objects

Legend objects provide a simple interface for displaying legends. The legend itself consists of a (filled and/or framed) box around one or more legend items (arranged in a single column) and an optional title string. Each legend item consists of a glyph patch positioned to the left of a text string. The glyph patch is drawn in a square which is a fraction of the legend label font height.

Creating Legend Objects

To create a legend object, you must provide an array of item names, along with arrays of symbols, line styles, or objects, along with arrays of attributes (such as color or thickness) for the items. The following simple example creates a legend object with two items. The first item (Cows) is represented by the predefined symbol number four (a diamond), and the second item (Weasels) is represented by a line-filled box.

```
itemNameArr = ['Cows', 'Weasels']
mytitle = OBJ_NEW('IDLgrText', 'My Legend')
mysymbol = OBJ_NEW('IDLgrSymbol', 4)
mypattern = OBJ_NEW('IDLgrPattern', 1)
myLegend = OBJ_NEW('IDLgrLegend', itemNameArr, TITLE=mytitle, $
    ITEM_TYPE=[0,1], ITEM_OBJECT=[mysymbol, mypattern], $
    /SHOW_OUTLINE)
```

See “[IDLgrLegend](#)” in the *IDL Reference Guide* manual for details on creating legend objects. See the next section for a more detailed explanation of the elements of the legend.

Using Legend Objects

The legend object allows you to define the annotations that correspond to the array of strings used as legend names in a variety of ways. The length of the argument string array is used to determine the number of items to be displayed. Each item is defined by taking one element from the `ITEM_NAME`, `ITEM_TYPE`, `ITEM_LINESTYLE`, `ITEM_THICK`, `ITEM_COLOR`, and `ITEM_OBJECT` vectors, if they are defined. If the number of items (as defined by the argument array or the `ITEM_NAME` array) exceeds any of the attribute vectors, the attribute defaults will be used for any additional items.

Specify a list of item names either via the argument to `IDLgrLegend::Init`, or via the `ITEM_NAME` property. The length of this array determines the size of the legend.

Use the `ITEM_TYPE` property to define whether an element in the legend is represented by a line (with an optional plotting symbol) or by a filled or unfilled box.

There should be one element of the `ITEM_TYPE` array per element in the input array or `ITEM_NAME` array.

Use the `ITEM_LINestyle` and `ITEM_THICK` properties to define the style and thickness of lines used as legend items. These arrays are ignored for elements that are not lines. Use the `ITEM_COLOR` property to specify the color of each legend element independently.

Use the `ITEM_OBJECT` property to specify that a graphic object be used as an annotation.

Dimensions

Until the legend is drawn to the destination object, the `[XYZ]RANGE` properties will be zero. Because you must know the size of the legend object in order to scale it properly for your window, you must use the `ComputeDimensions` method on the legend object to get the data dimensions of the legend prior to a draw operation.

The following example builds and displays a three-element legend.

```

; Create a window, view, and model:
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView')
mymodel = OBJ_NEW('IDLgrModel')
myview->Add, mymodel
; Create the legend with two items:
itemNameArr = ['Original Data', 'Histogram Plot', $
               'Boxcar-filtered (Width=5)']
mytitle = OBJ_NEW('IDLgrText', 'Plot Legend')
mysymbol = OBJ_NEW('IDLgrSymbol', 5, SIZE=[0.3, 0.3])
myLegend = OBJ_NEW('IDLgrLegend', itemNameArr, TITLE=mytitle, $
                  BORDER_GAP=0.8, GAP=0.5, $
                  ITEM_TYPE=[0,1], ITEM_LINestyle=[0,4,2], $
                  ITEM_OBJECT=[mysymbol, OBJ_NEW(), OBJ_NEW()], $
                  GLYPH_WIDTH=2.0, /SHOW_OUTLINE)
; Add the legend to the model:
mymodel->Add, myLegend
; Center the legend in the window.
; Note that you must use the ComputeDimensions method
; to get the dimensions of the legend.
dims = myLegend->ComputeDimensions(mywindow)
mymodel->Translate, -(dims[0]/2.), -(dims[1]/2.), 0
; Draw the legend:
mywindow->Draw, myview

```

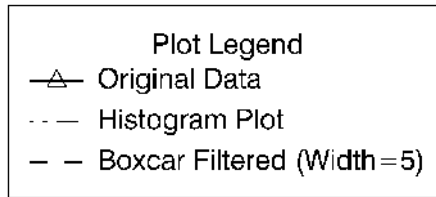


Figure 29-8: Legend Object

A Plotting Routine

This section develops a plotting routine that uses many of the object graphics features discussed here and in previous chapters. The code for this example is contained in the file `obj_plot.pro`, located in the `examples/visual` subdirectory of the IDL distribution.

The `OBJ_PLOT` routine will create a window object, and display within it a view of a single model object, which will contain a plot object, *x*- and *y*-axis objects, and an *x*-axis title object. It will use the Times Roman font for the axis title.

In creating the procedure, we allow the user to specify the data to be plotted, and we define keyword variables which can return the object references for the view, model, window, axis, and plot objects. This allows the user to manipulate the object tree after it has been created. We also specify the `_EXTRA` keyword, which allows the user to include other keyword parameters in the call. `OBJ_PLOT` itself passes any extra keyword parameters only to the plot object, but a more complex program could pass keyword parameters to any of the objects created. The following lines begin the procedure.

Note

See [“A Function for Coordinate Conversion”](#) on page 556 for a discussion of the `NORM_COORD` function used in this example. Also, `SET_VIEW` is discussed in [“Finding an Appropriate View Volume”](#) on page 550. (The files `set_view.pro` and `norm_coord.pro` are included in the `examples/visual/utility` subdirectory of the IDL distribution. `NORM_COORD` is also defined in the `obj_plot.pro` file.)

```
PRO obj_plot, data, VIEW=myview, MODEL=mymodel, WINDOW=mywindow, $
    CONTAINER=mycontainer, XAXIS=myxaxis, YAXIS=myyaxis, $
    PLOT=myplot, _EXTRA=e

; Next, create the window, view, and model objects:
mycontainer = OBJ_NEW('IDL_Container')
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView')
mymodel = OBJ_NEW('IDLgrModel')

; And a font object, specifying Times Roman for
; the font and the default size of 12 points:
myfont = OBJ_NEW('IDLgrFont', 'times')
```

```

; Next, create a plot object using data specified
; at the command line:
myplot = OBJ_NEW('IDLgrPlot', data, COLOR=[200,100,200])

; Now pass any extra keywords to OBJ_PLOT to the SetProperty
; method of the plot object. Keywords that do not apply to the
; SetProperty method are ignored.
myplot->SetProperty, _EXTRA=e

; Retrieve the data ranges from the plot object,
; and convert to normalized coordinates using the
; NORM_COORD function.
myplot->GetProperty, XRANGE=xr, YRANGE=yr
myplot->SetProperty, XCOORD_CONV=norm_coord(xr), $
    YCOORD_CONV=norm_coord(yr)

; Using the range from the plot object, create X- and
; Y-axis objects with appropriate ranges, and convert
; to normalized coordinates. Set the tick lengths to 5% of
; the data range (which is now normalized to 0.0-1.0).
myxaxis = OBJ_NEW('IDLgrAxis', 0, RANGE=[xr[0], xr[1]])
myxaxis->SetProperty, XCOORD_CONV=norm_coord(xr)
myyaxis = OBJ_NEW('IDLgrAxis', 1, RANGE=[yr[0], yr[1]])
myyaxis->SetProperty, YCOORD_CONV=norm_coord(yr)
myxaxis->SetProperty, TICKLEN=0.05
myyaxis->SetProperty, TICKLEN=0.05

; Add the model object to the view object,
; and the plot and axis objects to the model object.
myview->Add, mymodel
mymodel->Add, myplot
mymodel->Add, myxaxis
mymodel->Add, myyaxis

; Use the SET_VIEW routine to set an appropriate viewplane
; rectangle and zclip region for the view.
SET_VIEW, myview, mywindow

; Add a title to the X-axis, using the font object defined above:
xtext = OBJ_NEW('IDLgrText', 'X Title', FONT=myfont)
myxaxis->SetProperty, TITLE=xtext

; Add all objects to the container object.
; Destroying the container destroys all of its contents:
mycontainer->Add, mywindow
mycontainer->Add, myview
mycontainer->Add, myfont
mycontainer->Add, xtext

```

```

; Finally, draw the object tree:
mywindow->Draw, myview
END

```

Now, the OBJ_PLOT routine can be called with only the data parameter, if you choose. For example, the statement

```
OBJ_PLOT, FINDGEN(10)
```

creates and displays the object hierarchy with a simple plot line. However, if you do not retrieve the window, view, and other object references via the keywords, there is no way you can interactively modify the plot. A better way to call OBJ_PLOT would be:

```
OBJ_PLOT, FINDGEN(10), WINDOW=win, VIEW=view, PLOT=plot,
CONTAINER=cont
```

This statement creates the same object hierarchy, but returns the object references for the window, view, and plot objects in named variables. Having access the object references allows you to do things like change the color of the plot:

```
plot->SetProperty, COLOR=[255,255,255]
window->Draw, view
```

enlarge the viewplane rectangle by 10 percent:

```
view->GetProperty, VIEWPLANE_RECT=vr
vr2 = [vr[0]-(vr[0]*0.1), vr[1]-(vr[1]*0.1), $
      vr[2]+(vr[2]*0.1), vr[2]+(vr[2]*0.1)]
view->SetProperty, VIEWPLANE_RECT = vr2
window->Draw, view
```

or just clean it up:

```
OBJ_DESTROY, cont
```

Note that when using the OBJ_DESTROY procedure, any object added to the specified object (using the Add method) are also destroyed, recursively. We use a container object to collect all of the objects, including attribute objects and text object that are not explicitly added to the object tree, which allows you to destroy the entire collection with a single call to OBJ_DESTROY.

Improvements to the OBJ_PLOT Routine

A number of improvements to the OBJ_PLOT routine are left as exercises for the programmer:

- Provide error checking on the input arguments.
- Provide a way to set properties of the axis and text objects when calling `obj_plot`.
- Provide a graphical user interface to using IDL widgets.
- Do the object cleanup (destroying the objects created by `obj_plot`) when the user is finished with the routine. (This is easily accomplished if the routine has a widget interface.)
- Provide a way to retrieve data values once the data has been plotted, using the mouse to select data points.



Chapter 30: Working with Surfaces

This chapter describes the use of surface and light objects. The following topics are covered in this chapter:

Surface Objects	638	An Interactive Surface Example	646
Light Objects	643		

Surface Objects

Surface objects create a representation of functions of two variables. Surfaces are presented as three-dimensional objects in three-dimensional space, and thus are good candidates for interactive rotation, and scaling. Examples in this chapter discuss interactive manipulation of surface objects.

Creating Surface Objects

To create a surface object, provide a two-dimensional array of surface values (*Z* values) to the `IDLgrSurface::Init` method. Optionally, you can supply two vectors or arrays *X* and *Y* that specify the locations in the *XY* plane of the *Z* values provided. If *X* and *Y* are not provided, the surface is generated as a function of the array indices of each element of the *Z* array.

For example, the following statements create a surface object from the two-dimensional array created by the IDL command `DIST`, as a function of the *Z* data array indices:

```
zdata = DIST(40)
mysurf = OBJ_NEW('IDLgrSurface', zdata)
```

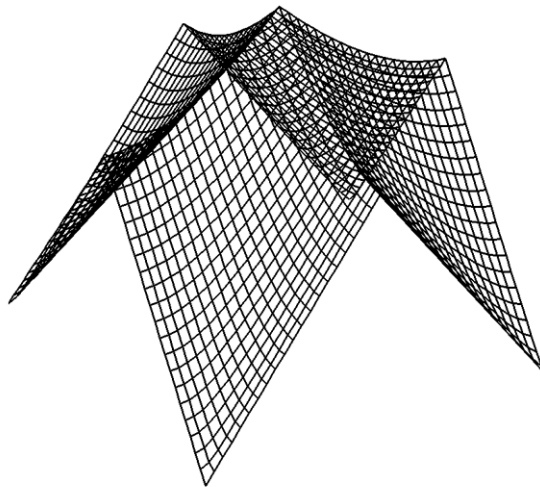


Figure 30-1: Surface Object

Similarly, if `xdata` and `ydata` are either 40-element vectors or 40x40 element arrays specifying the X and Y values which, when evaluated by some function, result in the `zdata` array, you would create the surface object with the following statement:

```
mysurf = OBJ_NEW('IDLgrSurface', zdata, xdata, ydata)
```

See “[IDLgrSurface](#)” in the *IDL Reference Guide* manual for details on creating surface objects.

Using Surface Objects

Surface objects have numerous properties controlling how they are rendered. You can set these properties when creating the surface object, or use the `SetProperty` method to the surface object to change these properties after creation.

Style

Set the `STYLE` property to an integer value that controls how the surface is rendered. Set the `STYLE` property equal to one of the following integer values:

- 0 = Display a single pixel for each data point.
- 1 = Display the surface as a wire mesh. (This is the default.)
- 2 = Display the surface as a solid.
- 3 = Display the surface using only lines drawn parallel to the *x*-axis.
- 4 = Display the surface using only lines drawn parallel to the *y*-axis.
- 5 = Display a wire mesh *lego*-type surface (similar to a histogram plot).
- 6 = Display a solid *lego*-type surface (similar to a histogram plot).

For example, the following statement changes the surface object to display the surface as a wire mesh, with the lines drawn in blue:

```
mysurf->SetProperty, STYLE=1, COLOR=[0,0,255]
```

The following statement draws the surface as a solid *lego*-type surface in green:

```
mysurf->SetProperty, STYLE=6, COLOR=[0,255,0]
```

Vertex Colors

You can supply a vector of vertex colors via the `VERT_COLORS` property. The colors in the vector will be applied to each vertex in turn. If there are more vertices than colors supplied for the `VERT_COLORS` property, IDL will cycle through the colors. For example, the following statements color each vertex and connecting line one of four colors:

```
vcolors = [[0,100,200],[200,150,200],[150,200,250],[250,0,100]]  
mysurf->SetProperty, STYLE=1, VERT_COLORS=vcolors
```

Shading

IDL provides two types of shading for surfaces. In Flat shading, the color of the first vertex in the surface is used to define the color for the entire surface. The color has a constant intensity. In Gouraud shading, the colors along each line are interpolated between vertex colors, and then along scanlines from each of the edge intensities.

Note

By default, only ambient lighting is provided for surfaces. If you do not supply a light source for your object hierarchy, solid surface objects will appear flat with either Flat or Gouraud shading. See “[Light Objects](#)” on page 643 for details on creating and using light objects.

Set the SHADING property of the surface object equal to 0 (zero) to use flat shading (this is the default), or equal to 1 (one) to use Gouraud shading. In the above example using vertex colors, adding the following statement:

```
mysurf->SetProperty, STYLE=2, SHADING=1
```

creates a surface in which the color values are interpolated between the vertex colors.

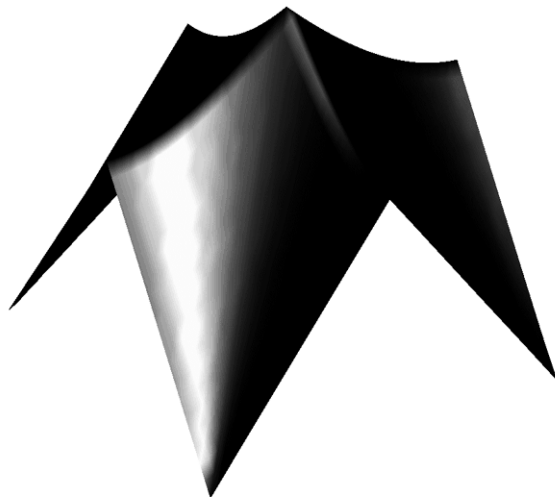


Figure 30-2: Surface Object Shading

Skirts

You can draw a skirt around the bottom edge of your surface object by setting the `SHOW_SKIRT` property of the surface object to 1. The skirt extends from the edge of the surface to a Z value specified by the `SKIRT` property. For example, the following statements draw the surface in wire mesh mode, with a skirt extending from the bottom of the surface to the value $z = 0.1$:

```
mysurf->SetProperty, STYLE=1, /SHOW_SKIRT, SKIRT=0.1
```

Hidden Line Removal

Set the `HIDDEN_LINES` property to the surface object equal to one to remove lines that are behind the visible parts of the surface from the rendering. By default, hidden lines are drawn. The following statement alters the surface to remove the hidden lines:

```
mysurf->SetProperty, /HIDDEN_LINES
```

Warning

Hidden line removal can be time-consuming.

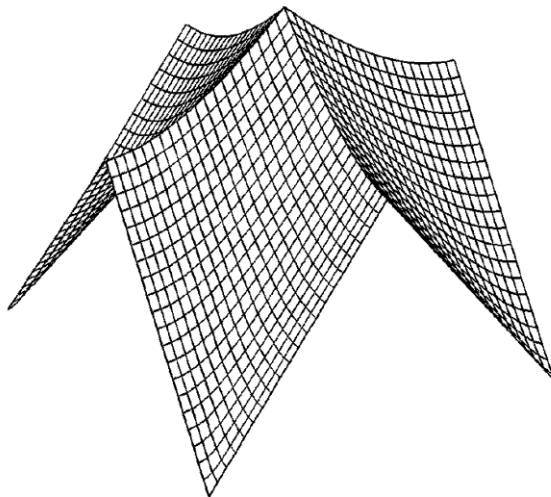


Figure 30-3: Surface Object Hidden Lines

Texture Mapping

You can map an image onto a surface object by specifying an `IDLgrImage` object to the `TEXTURE_MAP` property. The `TEXTURE_COORD` property defines how individual data points within the image data are mapped to the surface's vertices. If the `TEXTURE_COORD` property is not specified, the surface object will map the texture onto the entire data space (the region between 0.0 and 1.0 in normalized coordinates).

Light Objects

Objects of the `IDLgrLight` class represent sources of illumination for graphic objects. Although light objects are not rendered themselves, they are part of the model tree and thus can be transformed along with the graphic objects they illuminate.

If no light sources are specified for a given model, a default ambient light source is supplied. This allows you to display many objects without explicitly creating a light source. The use of only ambient light becomes problematic, however, when solid surfaces and other objects constructed from polygons are displayed. With only ambient lighting, all solid surfaces appear flat—in fact, they appear to be single two-dimensional polygons rather than objects in three-dimensional space.

Note

Graphic objects do not automatically cast shadows onto other objects.

Creating Light Objects

There are no arguments to the `IDLgrLight::Init` method. Keywords to the `Init` method allow you to control a number of properties of the light object, including the attenuation, color, cone angle (area of coverage), direction, focus, intensity, location, and type of light.

The following statement creates a default light object. The default light object is a white positional light, located at the origin.

```
mylight = OBJ_NEW('IDLgrLight')
```

There are four types of light objects available. Set the `TYPE` property of the light object to one of the following integer values:

- 0 = Ambient light. An ambient light is a universal light source, which has no direction or position. An ambient light illuminates every surface in the scene equally, which means that no edges are made visible by contrast. Ambient lights control the overall brightness and color of the entire scene. If no value is specified for the `TYPE` property, an ambient light is created.
- 1 = Positional light. A positional light supplies divergent light rays, and will make the edges of surfaces visible by contrast if properly positioned. A positional light source can be located anywhere in the scene.
- 2 = Directional light. A directional light supplies parallel light rays. The effect is that of a positional light source located at an infinite distance from scene.

- 3 = Spot light. A spot light illuminates only a specific area defined by the light's position, direction, and the cone angle, or angle which the spotlight covers.

See “[IDLgrLight](#)” in the *IDL Reference Guide* manual for details on creating light objects.

Using Light Objects

In addition to the type of light source, you can control several other properties of a light object. The following example creates a solid surface object and displays it first with only ambient lighting, then adds various light objects to the scene.

Note

The `SET_VIEW` function is discussed in “[Finding an Appropriate View Volume](#)” on page 550.

Begin by creating some data, the surface object, and supporting objects:

```
zdata = DIST(40)

mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView')
mymodel = OBJ_NEW('IDLgrMODEL')
mysurf = OBJ_NEW('IDLgrSurface', zdata, STYLE=2)

;Create the object hierarchy:
myview->Add, mymodel
mymodel->Add, mysurf

;Retrieve the X, Y, and Z ranges from the surface object:
mysurf->GetProperty, XRANGE=xr, YRANGE=yr, ZRANGE=zr

;Convert x, y, and z ranges to normalized coordinates.
xnorm = [-xr[0]/(xr[1]-xr[0]), 1/(xr[1]-xr[0])]
ynorm = [-yr[0]/(yr[1]-yr[0]), 1/(yr[1]-yr[0])]
znorm = [-zr[0]/(zr[1]-zr[0]), 1/(zr[1]-zr[0])]

mysurf->SETPROPERTY, XCOORD_CONV=xnorm, $
YCOORD_CONV=ynorm, ZCOORD_CONV=znorm

;Rotate the surface to a convenient orientation:
mymodel->Rotate, [1,0,0], -90
mymodel->Rotate, [0,1,0], 30
mymodel->Rotate, [1,0,0], 30

;Use the SET_VIEW routine to set an appropriate viewplane
```

```
;rectangle and zclip region for the view:  
SET_VIEW, myview, mywindow
```

```
;Draw the contents of the view:  
mywindow->Draw, myview
```

Once the surface object is drawn, we see that there is no definition or apparent three-dimensional shape to the surface. If we add a positional light one unit in the Z direction above the XY origin, however, details appear:

```
mylight = OBJ_NEW('IDLgrLight', TYPE=1, LOCATION=[0,0,1])  
mymodel->Add, mylight  
mywindow->Draw, myview
```

We can continue to alter the lighting characteristics by changing the properties of the existing light or by adding more light objects. (You can have up to eight lights in a given view object.) We can change the color:

```
mylight->SetProperty, COLOR=[200,0,200]  
mywindow->Draw, myview
```

We can change the intensity of the light:

```
mylight->SetProperty, INTENSITY=0.7  
mywindow->Draw, myview
```

An Interactive Surface Example

With a little programming, we can create an application that allows the user to display a surface object and transform its model tree interactively using the mouse. The file `surf_track.pro`, located in the `examples/visual` subdirectory of the IDL distribution, uses IDL widgets to create a graphical user interface to an object tree. The `SURF_TRACK` procedure creates a surface object from user-specified data (or from default data, if none is specified), and places the surface object in an IDL draw widget. The `SURF_TRACK` interface allows the user to specify several attributes of the object hierarchy via pull-down menus. Finally, the `SURF_TRACK` procedure uses the example trackball object (see “[Virtual Trackball and 3D Transformations](#)” on page 563 for details) to allow the user to rotate the surface in three dimensions.

Call the `SURF_TRACK` procedure without an argument to use the default surface (a Bessel function) or with a two-dimensional array as its argument:

```
;Make up some data:  
zdata = DIST(40)  
SURF_TRACK, zdata
```

We encourage you to inspect the code in `surf_track.pro` for hints on how to create a widget application around a draw widget that uses Object Graphics. Note especially that the `SURF_TRACK` procedure is well-behaved when it exits, destroying all of the objects it creates so as not to tie up memory with leftover objects for which object references are no longer available.

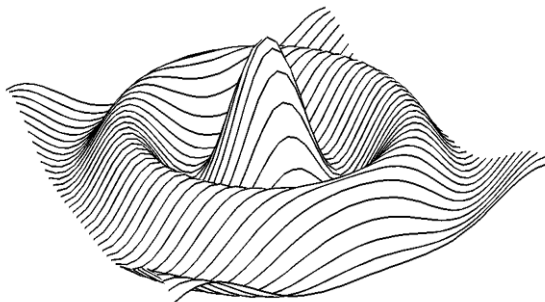


Figure 30-4: `STYLE=3` (Ruled xz), `HIDDEN_LINES=1` (hidden lines removed)

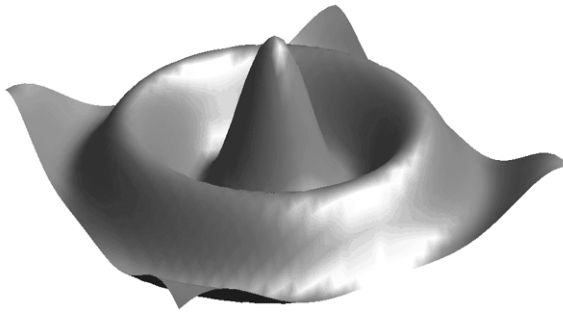


Figure 30-5: SHADING=1 (Gouraud), STYLE=2 (Solid)

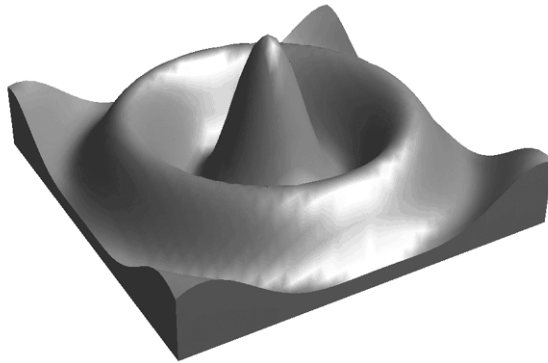


Figure 30-6: SKIRT=-0.402645



Chapter 31: Working with Images

The following topics are covered in this chapter:

Image Objects	650	Saving an Image to a File	656
Colorbar Objects	654		

Image Objects

An object of the `IDLgrImage` class (see “[IDLgrImage](#)” in the *IDL Reference Guide* manual) represents a two-dimensional array of pixel values, rendered on the plane $z = 0$. Image objects can have a single channel (one value per pixel—greyscale or color indexed), two channels (greyscale and Alpha), three channels (Red, Green, and Blue), or four channels (Red, Green, Blue, and Alpha). The Alpha channel, if present, determines the transparency of the pixel.

Image objects that have more than one channel can be interleaved either by pixel, by line, or by image. That is, if the image has three channels, width m and height n , the image array can be organized 3-by- m -by- n (pixel interleaving), m -by-3-by- n (line, or scanline interleaving), or m -by- n -by-3 (image, or planar interleaving).

Creating Image Objects

To create an image object, supply an array of pixel values to the `IDLgrImage::Init` method. If the image has more than one channel, be sure to set the `INTERLEAVE` property of the image object to the appropriate value. If image is an array of image data that is pixel interleaved, you would use the following statement to create an image object:

```
myimage = OBJ_NEW('IDLgrImage', image, INTERLEAVE=0)
```

For example, the following statements read a JPEG file located in the IDL distribution and create an image object from the RGB image. First, locate the `rose.jpg` file and read the image data into the variable `image`:

```
file = FILEPATH('rose.jpg', SUBDIR=['examples', 'data'])
READ_JPEG, file, image
;Create the objects:
mywindow = OBJ_NEW('IDLgrWindow', DIMENSIONS=[227,149])
myview = OBJ_NEW('IDLgrView', VIEW=[0,0,227,149])
mymodel = OBJ_NEW('IDLgrModel')
myimage = OBJ_NEW('IDLgrImage', image, INTERLEAVE=0)
;Organize the object hierarchy:
myview->Add, mymodel
mymodel->Add, myimage
;Draw to the window:
mywindow->Draw, myview
```



Figure 31-1: Image Object

See “[IDLgrImage](#)” in the *IDL Reference Guide* manual for details on creating image objects.

Note

IDLgrImage does not treat NaN data as missing. If the image data includes NaNs, it is recommended that the [BYTSCL](#) function (in the *IDL Reference Guide*) be used to appropriately handle those values. For example:

```
oImage->SetProperty, DATA = BYTSCL(myData, /NaN, MIN=0, MAX=255)
```

Using Image Objects

Several properties allow you to control the way image objects are rendered.

Alpha Blending

If your image data includes an alpha channel, use the `BLEND_FUNCTION` property of the image object to control how the alpha channel values will be interpreted. Set the `BLEND_FUNCTION` property equal to a two-element vector `[src, dst]` specifying one of the functions listed below for each of the source and destination

objects. The values of the blending function (V_{src} and V_{dst}) are used in the following equation

$$C_d' = (V_{src} \cdot C_i) + (V_{dst} \cdot C_d)$$

where C_d is the initial color of a pixel on the destination device (the background color), C_i is the color of the pixel in the image, and C_d' is the resulting color of the pixel.

Setting *src* and *dst* in the BLEND_FUNCTION vector to the following values determine how each term in the equation is calculated:

src or dst	V_{src} or V_{dst}	What the function does
0	n/a	Alpha blending is disabled. $C_d' = C_i$.
1	0	The value of V_{src} or V_{dst} in the equation is zero, thus the value of the term is zero.
2	1	The value of V_{src} or V_{dst} in the equation is one, thus the value of the term is the same as the color value.
3	Image_α	The value of V_{src} or V_{dst} in the equation is the value of the alpha channel of the image.
4	$1 - \text{Image}_\alpha$	The value of V_{src} or V_{dst} in the equation is one minus the value of the alpha channel of the image.

Table 31-1: BLEND_FUNCTION Vector Behavior

For example, setting BLEND_FUNCTION = [3, 4] creates an image in which you can see through the foreground image to the background to the extent defined by the alpha channel values of the foreground image.

Interleaving

Set the INTERLEAVE property of the image object to 0 (zero) to indicate that the image is interleaved by pixel, to 1 (one) to indicate that the image is interleaved by line, or to 2 to indicate that the image is interleaved by image, or plane. For example, the following statement changes the image object to use line interleaving:

```
myimage->SetProperty, INTERLEAVE=1
```

Palettes

If your image array contains indexed color data (that is, if it is an m -by- n array), you can specify a palette object to control the conversion between the image data and the palette used by an RGB-mode destination object. (See “Using Color” on page 572 for a discussion of the interaction between indexed color objects and RGB color destinations.) Set the PALETTE property of the image object equal to an instance of an IDLgrPalette object:

```
myimage->SetProperty, PALETTE = mypalette
```

To specify that an image be drawn in greyscale mode rather than through an existing color palette, set the GREYSCALE property equal to 1 (one). The GREYSCALE property is only used if the image data is a single channel (an m -by- n array).

Note

A 2-by- m -by- n array is considered to be a greyscale image with an Alpha channel. An image containing indexed color data cannot have an alpha channel.

Colorbar Objects

The `IDLgrColorbar` object consists of a color-ramp with an optional framing box and annotation axis. The object can be horizontal or vertical.

Creating Colorbar Objects

To create a colorbar object, you must provide a set of red, green, and blue values to be displayed in the bar. Axis values are determined from the number of elements in the color arrays unless otherwise specified via the `TICKVALUES` property. The following creates a colorbar one tenth of the window dimension wide by four-tenths of the window dimension high, with a red-green-blue color ramp:

```
mytitle = OBJ_NEW('IDLgrText', 'My Colorbar')
barDims = [0.1, 0.4]
redValues = BINDGEN(256)
greenValues = redValues
blueValues = REVERSE(redValues)
mycolorbar = OBJ_NEW('IDLgrColorbar', redValues, $
    greenValues, blueValues, TITLE=mytitle, $
    DIMENSIONS=barDims, /SHOW_AXIS, /SHOW_OUTLINE)
```

See “[IDLgrColorbar](#)” in the *IDL Reference Guide* manual for details on creating colorbar objects. See the next section for a more detailed explanation of the elements of the legend.

Using Colorbar Objects

The colorbar object allows you to define the size, colors, and various annotations.

Dimensions

Until the legend is drawn to the destination object, the `[XYZ]RANGE` properties will be zero. Because you must know the size of the legend object in order to scale it properly for your window, you must use the `ComputeDimensions` method on the legend object to get the data dimensions of the legend prior to a draw operation.

The following example builds and displays the colorbar described above:

```
; Create a window, view, and model:
mywindow = OBJ_NEW('IDLgrWindow')
myview = OBJ_NEW('IDLgrView')
mymodel = OBJ_NEW('IDLgrModel')
myview->Add, mymodel
; Create the colorbar. Make the bar one tenth of
```

```

; the window size horizontally and four tenths of
; the window size vertically. Show the axis values (using the
; default axis annotations) and draw an outline around the bar.
mytitle = OBJ_NEW('IDLgrText', 'My Colorbar')
barDims = [0.1, 0.4]
redValues = BINDGEN(256)
greenValues = redValues
blueValues = REVERSE(redValues)
mycolorbar = OBJ_NEW('IDLgrColorbar', redValues, $
    greenValues, blueValues, TITLE=mytitle, $
    DIMENSIONS=barDims, /SHOW_AXIS, /SHOW_OUTLINE)
mymodel->Add, mycolorbar
; Center the colorbar in the window.
; Note that you must use the ComputeDimensions method to
; get the dimensions of the colorbar.
barPlusTextDims = mycolorbar->ComputeDimensions(mywindow)
mymodel->Translate, -barDims[0]+(barPlusTextDims[0]/2.), $
    -barDims[1]+(barPlusTextDims[1]/2.), 0
; Draw the colorbar:
mywindow->Draw, myview

```

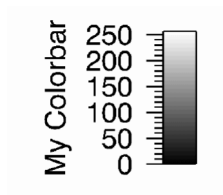


Figure 31-2: Colorbar Object

Saving an Image to a File

If you have created a scene or view containing graphical objects and wish to save the rendering to a file, you will first need to create an image object from which to retrieve the image data. The following steps render an object to a window, create an image object from the window, and save the image data as a TIFF file.

First, create the view to be rendered. Use an indexed color model for the window object, setting the background color to white and the foreground color of the plot object to black.

```
mywindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL=1)
myview = OBJ_NEW('IDLgrView', $
    VIEWPLANE_RECT=[0,-4,10,8], COLOR=255)
mymodel = OBJ_NEW('IDLgrModel')
myplot = OBJ_NEW('IDLgrPlot', RANDOMN(seed, 10), COLOR=0, $
    THICK=3)
; Organize the object hierarchy:
myview->Add, mymodel
mymodel->Add, myplot
; Draw to the window:
mywindow->Draw, myview
; Next, use the window object's Read method to create
; an image object with the rendered scene as its image data:
myimage = mywindow->Read()
; Retrieve the image data using the GetProperty method
; of the image object:
myimage->GetProperty, DATA=image
; Display the image data using Direct Graphics:
TV, image
; Write the image to a TIFF file named myfile.tif:
WRITE_TIFF, 'myfile.tif', image
```

Create an MPEG File

If you have a series of image objects (or simple image arrays), you can combine them into a single MPEG file using the IDLgrMPEG helper object. Suppose you have an array `imagearray` containing IDLgrImage objects that represent a time-series. You could use the following commands to create an MPEG file from the images.

First, create an MPEG object, and populate the file with frames from the *imagearray* array:

```
myMPEG = OBJ_NEW('IDLgrMPEG', FILENAME='mympeg.mpg')
FOR image = 0, N_ELEMENTS(imagearray) DO BEGIN
    myMPEG->Put, imagearray[image], image
ENDFOR
; Save the MPEG file:
myMPEG->Save
```

Note

Note that *imagearray* can contain either *IDLgrImage* objects or simple two-dimensional image arrays. All of the arrays or image objects must have the same dimensions.

See “[IDLgrMPEG](#)” in the *IDL Reference Guide* manual for details on creating MPEG objects.



Chapter 32: Working with Volumes

This chapter describes the process of creating and displaying volume objects. The following topics are covered in this chapter:

Volume Objects	660	Volume Object Attributes	662
----------------------	-----	--------------------------------	-----

Volume Objects

A volume object contains a three dimensional data array of voxel values and a set of rendering attributes. The voxel array is mapped to colors and opacity values through a set of lookup tables in the volume object. Several rendering methods are provided to draw the volume to a destination.

Creating Volume Objects

To create a volume object, create a three dimensional array of voxels and pass them to the `IDLgrVolume::Init` method. Voxel arrays must be of `BYTE` type. For example, the following will create a simple volume data set and create a volume object which uses it:

```
data = BYTARR(64,64,64)
FOR i=0,63 DO data[* ,i,0:i] = i*2
data[5:15, 5:15, 5:55] = 128
data[45:55, 45:55, 5:15] = 255
myvolume = OBJ_NEW('IDLgrVolume', data)
```

The volume contains a shaded prism along with two brighter cubes (one located within the prism).

See “[IDLgrVolume](#)” in the *IDL Reference Guide* manual for details on creating volume objects.

Note

You do not need to enter the example code in this chapter yourself. The example code shown here is duplicated in the procedure file `obj_vol.pro`, located in the `examples/visual` subdirectory of the IDL distribution. You can run the example procedure by entering `OBJ_VOL` at the IDL command prompt. The procedure file stops after each operation (roughly corresponding to each section below) and requests that you press return before continuing.

Using Volume Objects

A volume object has spatial dimensions equal to the size of the data in the volume. In the example, the volume object occupies the range 0-63 in the *x*-, *y*-, and *z*-axes. To make the volume easier to manipulate, we use the `XCOORD_CONV`, `YCOORD_CONV`, and `ZCOORD_CONV` properties of the volume object to center the volume at 0,0,0 and scale it to fit in a unit cube:

```
cc = [-0.5, 1.0/64.0]
myvolume->SetProperty, XCOORD_CONV=cc, YCOORD_CONV=cc, $
    ZCOORD_CONV=cc
; Create a window and view tree:
mywindow = OBJ_NEW('IDLgrWindow', DIMENSIONS=[200,200])
myview = OBJ_NEW('IDLgrView',VIEWPLANE_RECT=[-1,-1,2,2], $
    ZCLIP=[2.0,-2.0], COLOR=[50,50,50])
mymodel = OBJ_NEW('IDLgrModel')
myview->Add, mymodel
mymodel->Add, myvolume
; Rotate the volume a little and draw it:
mymodel->rotate, [1,1,1], 45
mywindow->Draw, myview
```

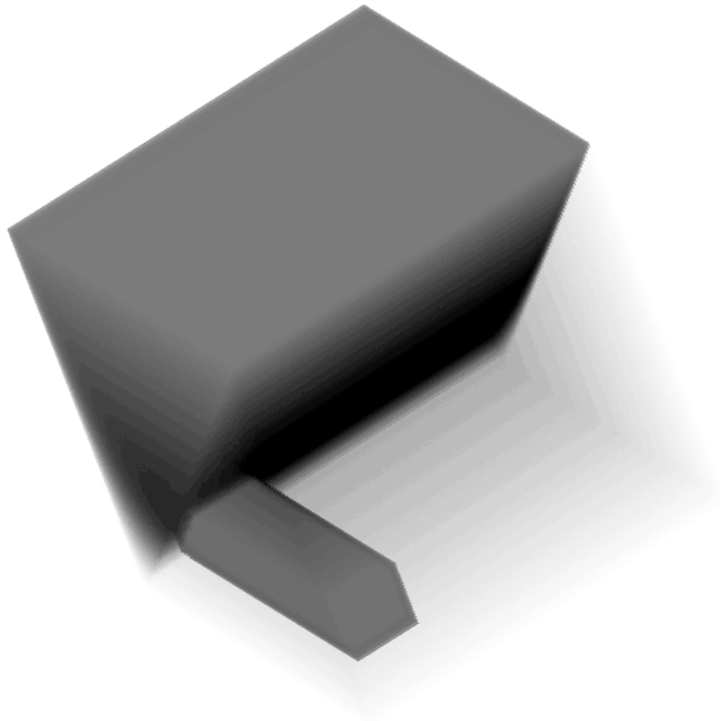


Figure 32-1: Volume Object

Volume Object Attributes

Volume objects have numerous properties controlling how they are rendered. These properties can be set when the object is created or set using the SetProperty method.

Opacity

The opacity table controls the transparency of a given voxel value. Manipulation of the opacity table is critical to improving the quality of a rendering. The following example makes the prism transparent and the cubes opaque, allowing the cube within the prism to be seen, by setting the OPACITY_TABLE0 array to low values for the prism and high values for the cubes.

```
opac = BYTARR(256)
opac[0:127] = BINDGEN(128)/8
;Voxel value of one cube:
opac[255] = 255
;Voxel value of the other cube:
opac[128] = 255
myvolume->SetProperty, OPACITY_TABLE0=opac
mywindow->Draw, myview
```

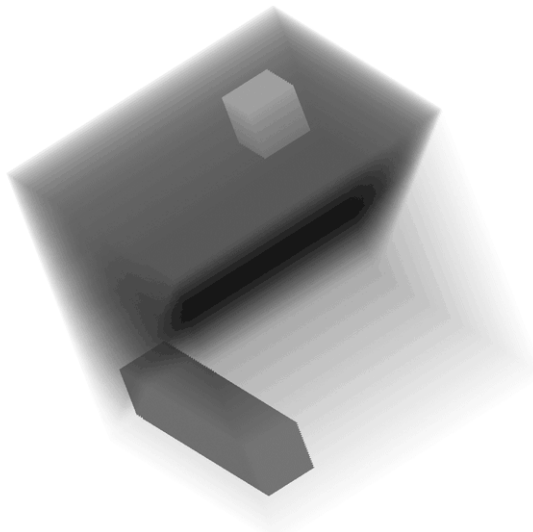


Figure 32-2: Volume Object Opacity

Color

Each voxel value can be assigned an individual color as well. This color mapping can be changed by changing the `RGB_TABLE0` property. To further highlight the cubes, we change their colors to blue and red, using the following statements:

```
rgb = bytarr(256,3)
;Grayscale ramp for the prism:
rgb[0:127,0] = bindgen(128)
rgb[0:127,1] = bindgen(128)
rgb[0:127,2] = bindgen(128)
;One cube is red:
rgb[128,*] = [255,0,0]
;One cube is blue:
rgb[255,*] = [0,0,255]
myvolume->SetProperty, RGB_TABLE0=rgb
mywindow->Draw, myview
```

Lighting

Adding lights enhances the edges of volumes. Gradients within the volume are used to approximate a surface normal for each voxel, and the lights in the current view are then applied. The gradient shading is enabled by setting the `LIGHTING_MODEL` property equal to one. The ambient volume color is selected by setting the `AMBIENT` property of the volume object to a color value. Setting the `TWO_SIDED` property allows both sides of a voxel to be lighted. An example of this using a light source follows:

```
myvolume->SetProperty, AMBIENT=[100,100,100], LIGHTING_MODEL=1, $
    TWO_SIDED=1
lmodel = OBJ_NEW('IDLgrModel')
myview->Add, lmodel
light = OBJ_NEW('IDLgrLight', TYPE=2, LOCATION=[0,0,1], $
    COLOR=[255,255,255])
lmodel->Add, light
mywindow->Draw, myview
; Disable lighting:
myvolume->SetProperty, LIGHTING_MODEL=0
```

Note

Only **DIRECTIONAL** light sources are honored by the volume object. Because normals must be computed for all voxels in a lighted view, enabling light sources increases the rendering time.

Compositing

The volume object supports a number of methods for blending the projected voxels together to form an image. By default, Alpha blending is used. (In Alpha blending, each voxel occludes voxels behind it according to the opacity of the voxel in front). Another common compositing technique is the maximum intensity projection (MIP). Set the volume object to use MIP compositing by setting the `COMPOSITE_FUNCTION` property equal to one. See “[IDLgr Volume Properties](#)” in the *IDL Reference Guide* manual for other options.

```
myvolume->SetProperty, COMPOSITE_FUNCTION=1
mywindow->Draw, myview
```

ZBuffering

When combining a volume with other geometry in the Object Graphics system, volume objects should in general be drawn last to ensure they intersect the other (solid) objects properly. To increase rendering speed, the intersection operation is disabled by default. To enable the intersection calculations, set the `ZBUFFER` property of the volume object equal to one.

```
myvolume->SetProperty, ZBUFFER=1
```

Additionally, volume objects allow for control over the rendering of invisible (opacity equals zero) voxels. By default, the zbuffer will be updated for such voxels (even though no change is made in the image color). This writing to the zbuffer by transparent voxels be disabled by setting the `ZERO_OPACITY_SKIP` property.

```
myvolume->SetProperty, ZERO_OPACITY_SKIP=1
```

Note

In volumes with large numbers of voxels with their opacity set to zero, enabling `ZERO_OPACITY_SKIP` can improve rendering performance.

Interpolation

By default, when rendering a volume object, values between the voxels are estimated using nearest neighbor sampling. When higher quality rendering is desired, trilinear interpolation can be selected instead by setting the `INTERPOLATE` property equal to one.

```
myvolume->SetProperty, INTERPOLATE=1
```


Note

Trilinear interpolation will cause the rendering to take considerably longer than nearest neighbor interpolation.

Rendering speed

Rendering speed can be improved by reducing the quality of the rendering. Use the `RENDER_STEP` property to control this speed/quality trade-off. The value of the `RENDER_STEP` property specifies a step size in the screen dimensions which is used to skip voxels during the rendering process. Larger values yield faster rendering times, but lower final image quality. For example, to render only half as many voxels in the screen Z dimension, use the following statement:

```
myvolume->SetProperty, RENDER_STEP=[1,1,2]
```

A more complex example using a volume object is shown in the volume visualization demo. To start the demos, type `demo` at the IDL command prompt.

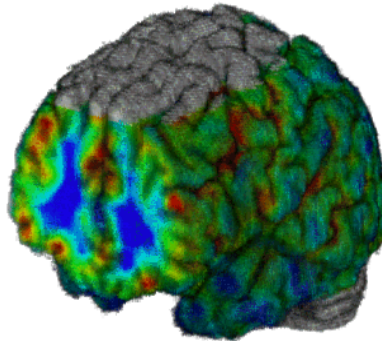


Figure 32-3: Volume Object Rendering



Chapter 33: Selecting Objects

This chapter will describe the IDL Object Graphics selection and direct manipulation features. The following topics are covered in this chapter:

Selection and Data Picking	668	Data Picking	672
Selection	669	A Data Picking Example	673
A Selection Example	671		

Selection and Data Picking

When graphical items are drawn to a window, it is often useful to be able to click the mouse on a certain location and request a list of the items that are displayed at that particular location. In IDL, this is called selection. Because IDL object graphics are retained in memory, they can be uniquely identified by their individual object references, and therefore can be reported as having been selected.

In many cases, it is also useful to be able to request the data value of the object at the user-selected location. In IDL, this is called data picking.

Selection

With object graphics, the process of selection is very similar to drawing, except that nothing is displayed on the screen, and information about which objects were selected is returned to the user. Selection is performed via the `Select` method of an `IDLgrWindow` object.

Three types of objects may be selected: view objects, model objects, and graphic atoms. For a given scene that contains more than one view, you can use the `Select` method to determine which view is selected at a given location. Likewise, for a given view, you can use the `Select` method to determine which models and/or graphical atoms within that view are selected.

An object is considered to be selected if its graphical rendering falls within a box centered on a given location. The dimensions of the box are set via the `DIMENSIONS` keyword to the `Select` method. Both the location argument and dimensions keyword values are measured in units specified via the `UNITS` keyword.

The `Select` method returns a vector of objects, sorted in depth order (nearest to the eye is first), that meet the criteria of having been selected at the given location. If no objects are selected at the given location, the `Select` method returns `-1`.

See “[IDLgrWindow::Select](#)” in the *IDL Reference Guide* manual for a detailed description of the `Select` method.

Selecting Views

To determine which of a set of views within a given scene are selected at a given location, call the `Select` method on an `IDLgrWindow` object with an instance of an `IDLgrScene` object as its first argument, and the location at which the selection is to occur as its second argument:

```
myLoc = [myMouseEvent.x, myMouseEvent.y]
mySelectedViews = myWindow->Select(myScene, myLoc)
```

Selecting Graphic Atoms

To determine which graphic items within a given view are selected at a given location, call the `Select` method on an `IDLgrWindow` object should be called with an instance of an `IDLgrView` object as its first argument, and the location at which the selection is to occur as the second argument:

```
myLoc = [myMouseEvent.x, myMouseEvent.y]
mySelectedGraphics = myWindow->Select(myView, myLoc)
```

Note

If a model within the view is set as a selection target, the model object, rather than its contained graphic atoms, is returned in the vector of selected objects.

Selecting Models

In some cases, a group of graphic atoms may be considered subcomponents of the model in which they are contained. As a result, you may want to know when a model object (rather than one or more of its atomic parts) has been selected. To enable selection of a model (rather than its graphic atoms), the model object must be marked as a selection target.

To mark a model as being a selection target, set the `SELECT_TARGET` property of the model object to a nonzero value.

```
myWindow = OBJ_NEW('IDLgrWindow')
myView = OBJ_NEW('IDLgrView')
myModel = OBJ_NEW('IDLgrModel')
myView->Add, myModel
myModel->SetProperty, /SELECT_TARGET
myAxis = OBJ_NEW('IDLgrAxis', 0)
myModel->Add, myAxis
myWindow->Draw, myView
```

In the above example, if a selection at location $[myX, myY]$ would normally select the axis object, the returned value of the `Select` method will be the object reference to `myModel` rather than the object reference to `myAxis`.

A Selection Example

An example procedure named `sel_obj.pro` is included in the `examples/visual` subdirectory of the IDL distribution. This file creates two views, places models within the views, and provides an interface to let you choose between selecting models or graphic atoms. A mouse click in one of the views will update a label that identifies the current selections.

Data Picking

To get the data value that corresponds to a particular window location, use the `PickData` method of an `IDLgrWindow` object. Note that you must draw the view to the window before calling the `PickData` method.

```
myLoc = [myMouseEvent.x, myMouseEvent.y]
result = myWindow->PickData(myView, myModel, myLoc, returnedXYZ)
```

The `PickData` method returns a value that is 0 (zero) if the pick hit the background of the view, 1 (one) if the pick hit the one of the graphic atoms in the view, or -1 if an error occurred (for instance, if the pick location lies outside of the given view).

The data value at the pick is returned in the *returnedXYZ* argument. This value represents the mapping of the window location to the data space of the model.

A Data Picking Example

The example procedure `surf_track.pro` includes code using the `PickData` method to retrieve data values from a surface object. `surf_track.pro` is located in the `examples/visual` subdirectory of the IDL distribution, and is described in [“An Interactive Surface Example”](#) on page 646.



Chapter 34: Using Destination Objects

The following topics are covered in this chapter:

Overview	676	Clipboard Objects	685
Window Objects	677	Printer Objects	687
Using Window Objects	680	VRML Objects	694
Instancing	682	Bitmap and Vector Graphic Output	695
Buffer Objects	684		

Overview

Once a graphic object tree has been created, it can be displayed, or drawn, to a physical destination device (such as a computer screen or printer), to a memory location (such as a buffer or the operating system clipboard), or to a particular file format (such as a VRML file). Destination objects represent the final locations to which object graphics are drawn, and provide methods that allow you to control the properties of the physical device, memory buffer, or file format.

Each destination object includes a `GetFontnames` method, which returns the list of available fonts that can be used in `IDLgrFont` objects. This method will only return the names of the available TrueType fonts. Hershey fonts will not be returned as they are fixed—see [Appendix H, “Fonts”](#) in the *IDL Reference Guide* manual for more information.

There are five destination objects:

1. buffers (`IDLgrBuffer` objects),
2. clipboards (`IDLgrClipboard` objects),
3. printers (`IDLgrPrinter` objects)
4. VRML files (`IDLgrVRML` objects), and
5. windows (`IDLgrWindow` objects).

Of the five destination objects, Window objects are the most common and most often used, and will be addressed first.

Note

Output to `IDLgrClipboard` and `IDLgrPrinter` objects can be in bitmap or vector format. See [“Bitmap and Vector Graphic Output”](#) on page 695 for information on choosing a suitable graphics output type based on scene content.

Window Objects

Objects of the `IDLgrWindow` class represent a rectangular area on a computer screen into which graphics hierarchies can be rendered. Window objects can be either stand-alone windows on the screen or drawable areas in an IDL draw widget.

Creating Window Objects

There are two ways to create window objects: directly via the window object's `Init` method and indirectly by creating a draw widget that uses a window object as its drawable area.

Using the `Init` Method

The `IDLgrWindow::Init` method takes no arguments. Use the following statement to create a window object:

```
myWindow = OBJ_NEW('IDLgrWindow')
```

The window is displayed on the screen as soon as it has been created.

Creating a Draw Widget that Uses a Window Object

To create a draw widget that uses an Object Graphics window object rather than a Direct Graphics window for its drawable area, set the `GRAPHICS_LEVEL` keyword to the `WIDGET_DRAW` function equal to 2:

```
drawwid = WIDGET_DRAW(base, GRAPHICS_LEVEL=2)
```

Once the draw widget has been realized, you can then retrieve the object reference to the draw widget's window object using the `WIDGET_CONTROL` procedure:

```
WIDGET_CONTROL, drawwid, GET_VALUE=myWindow
```

Color Model

By default, window objects use the RGB color model. To create a window that uses the Indexed color model, set the `COLOR_MODEL` property of the window object equal to 1 (one) when creating the window:

```
myWindow = OBJ_NEW('IDLgrWindow', COLOR_MODEL=1)
```

You cannot change the color model used by a window after it has been created.

See [“Working with Color”](#) in Chapter 3 of the *Image Processing in IDL* manual for a discussion of the two color models.

Hardware vs. Software Rendering

The `RENDERER` property to the `IDLgrWindow` object (and the preference of the same name in the IDL Development Environment) allows you to select between the operating system's native (hardware) rendering system and a platform independent (software) rendering system for IDL Object Graphics displays.

Hardware rendering allows IDL to make use of 3D graphics accelerators that support OpenGL, if any are installed in the system. In general, such accelerators will provide better rendering performance for many object graphics displays

The software rendering system will generally run more slowly than the hardware rendering system. However, use of the software rendering system has a few important advantages:

- Software rendering is available in situations where hardware rendering is not (remote display to non-OpenGL capable X servers, for example).
- The number of expose events an IDL application will have to respond to is much smaller when software rendering is used.
- The software rendering system is generally much faster than the hardware rendering system for Instancing.
- Software rendering can be used to avoid bugs in hardware rendering system driver software (over which RSI has no control).
- Finally, on some displays (most notably SGI systems with 24 or fewer bitplanes), the quality of the screen display will be better when using the software rendering system because its design allows more bitplanes to be used for graphics display.

Note

By default, IDL uses the renderer specified in the Graphics tab of the IDLDE Preferences dialog. If the preference is not explicitly set in the IDLDE and your platform has a native OpenGL implementation, IDL uses the native OpenGL as the default. If your platform does not have a native OpenGL implementation, IDL uses its own software implementation regardless of the IDLDE preference or the value of the `RENDERER` property.

Note on Window Size Limits

The OpenGL libraries IDL uses impose limits on the maximum size of a drawable area. The limits are device-dependent — they depend both on your graphics hardware and the setting of the `RENDERER` property. Currently, the smallest maximum drawable area on any IDL platform is 1280-by-1024 pixels; the limit on your system may be larger.

Using Window Objects

To render a graphics tree to a window, call the `IDLgrWindow::Draw` method. The argument must be either an `IDLgrView` object or an `IDLgrScene` object.

```
myWindow->Draw, myView
```

or

```
myWindow->Draw, myScene
```

All objects contained within the view or scene object will be drawn to the window.

Erasing a Window

To erase the contents of a window, call the `IDLgrWindow::Erase` method. You can optionally supply a color to use to clear the window. By default, the window is erased to white.

For example, to erase the window to black:

```
myWindow->Erase, COLOR=[0,0,0]
```

Exposing or Hiding a Window

To expose a window so that it is the front-most window on the screen, call the `IDLgrWindow::Show` method with a nonzero value as the argument:

```
myWindow->Show, 1
```

To hide a window, call the `IDLgrWindow::Show` method with a zero value as the argument:

```
myWindow->Show, 0
```

Iconifying a Window

To iconify a window, call the `IDLgrWindow::Iconify` method with a nonzero value as its argument:

```
myWindow->Iconify, 1
```

To restore an iconified window, call the `IDLgrWindow::Iconify` method with a zero value as its argument:

```
myWindow->iconify, 0
```


Setting the Window Cursor

To set the appearance of the mouse cursor in an `IDLgrWindow` object, call the `IDLgrWindow::SetCurrentCursor` method with a string argument representing the name of the cursor. Valid string values for the cursor name argument are:

ARROW	CROSSHAIR
ICON	IBEAM
MOVE	ORIGINAL
SIZE_NE	SIZE_NW
SIZE_SE	SIZE_SW
SIZE_NS	SIZE_EW
UP_ARROW	

The following statement sets the cursor to an up arrow:

```
myWindow->SetCurrentCursor, 'UP_ARROW'
```

The `ORIGINAL` cursor sets the cursor to the window system's default cursor.

See “[IDLgrWindow::SetCurrentCursor](#)” in the *IDL Reference Guide* manual for details on cursor values.

Saving/Restoring Windows

When an instance of an `IDLgrWindow` object is restored via the `RESTORE` procedure), it is not immediately displayed on the screen. It will be displayed as soon as one of its methods (`Draw`, `Erase`, `Iconify`, etc.) is called.

Instancing

For interactive graphics, where views are drawn repeatedly over time, it is often the case that one small part of the view is changing continuously, but the other objects in the view remain static. In such a case, it may be more efficient to take a snapshot of the unchanged portion of the view and display the snapshot for each draw instead of re-rendering each of the unchanging objects from scratch. The objects that are changing are rendered as usual. This process is called instancing. It is to your advantage to use instancing only in cases where displaying the snapshot image is faster than rendering each of the objects that remain unchanged.

The following example shows how a typical instancing loop would be set up. First, hide the objects in the view that will be changing. In this example, we assume that the objects that change continuously are contained by a single model object, with the object reference `myChangingModel`. We set the `HIDE` property for this model to remove it from the rendered view.

```
myChangingModel->SetProperty, HIDE=1

;Next, create an instance of the remaining portion
;of the view by setting the CREATE_INSTANCE keyword to
;the window's Draw method:
myWindow->Draw, myScene, /CREATE_INSTANCE

;Next, hide the unchanging objects.
;Assume that the unchanging portion of the
;scene is contained in a single model object.
myUnchangingModel->SetProperty, HIDE=1

;Set the HIDE property for the changing model
;object equal to zero, revealing the object:
myChangingModel->SetProperty, HIDE=0

;Set the view object's TRANSPARENT property.
;This ensures that we will not erase the
;instance data (the unchanging part of the scene)
;when drawing the changing model.
myView->SetProperty, /TRANSPARENT

;Next, we set up a drawing loop that will render
;the changing model. For example, this loop might
;rotate the changing model in 1 degree increments.
ROT = 0
FOR i=0,359 DO BEGIN
    ROT=ROT+1
    myChangingModel->Rotate, [0,1,0], ROT
```

```
    myWindow->Draw, myView, /DRAW_INSTANCE  
ENDFOR
```

```
    ;After the drawing loop is done, ensure nothing is hidden,  
    ;and that the view erases as it did before:
```

```
myUnchangingModel->SetProperty, HIDE=0
```

```
myView->SetProperty, TRANSPARENT=0
```

Buffer Objects

Objects of the `IDLgrBuffer` class represent a memory buffer into which graphics hierarchies can be rendered. Object trees can be drawn to instances of the `IDLgrBuffer` object and the resulting image can be retrieved from the buffer using the `Read()` method. The off-screen representation avoids dithering artifacts by providing a full-resolution buffer for objects using either the RGB or Color Index color models.

Creating Buffer Objects

The `IDLgrBuffer::Init` method takes no arguments. Use the following statement to create a buffer object:

```
myBuffer = OBJ_NEW('IDLgrBuffer')
```

This creates an object that is available as a destination device to be rendered into or copied from.

See [“IDLgrBuffer”](#) in the *IDL Reference Guide* manual for details on creating and using buffer objects.

Clipboard Objects

Objects of the `IDLgrClipboard` class send Object Graphics output to the operating system native clipboard or to a file in bitmap or vector format. The file type and destination is dependent upon the platform and the values of Draw method keywords.

Note

What appears when producing bitmap or vector output is dependent upon several factors. See [“Bitmap and Vector Graphic Output”](#) on page 695 for details.

Writing to a File from `IDLgrClipboard`

The file type produced when the `IDLgrClipboard::Draw` method is passed an `IDLgrView`, `IDLgrViewgroup`, or `IDLgrScene` object varies depending upon keyword settings and the platform on which the call is issued. If the `FILENAME` keyword is set to a non-empty string, the name of the file IDL creates is specified by the string. If the `FILENAME` keyword is a non-zero, numeric value, IDL creates a file named `idl.ext` where `ext` is replaced with the appropriate extension shown in parentheses in the following table.

Keyword Settings	Windows File Type	UNIX File Type
VECTOR = 1, POSTSCRIPT = 1	Encapsulated PostScript (EPS)	Encapsulated PostScript (EPS)
VECTOR = 1, POSTSCRIPT = 0	Enhanced MetaFile (EMF)	Encapsulated PostScript (EPS)
VECTOR = 0, POSTSCRIPT = 1	Encapsulated PostScript (EPS)	Encapsulated PostScript (EPS)
VECTOR = 0, POSTSCRIPT = 0	Bitmap (BMP)	Encapsulated PostScript (EPS)

Table 34-1: File Types Produced by `IDLgrClipboard` Draw Method

Note

PostScript clipboard output can be generated using the CMYK color model. See the `IDLgrClipboard::Draw` method in the *IDL Reference Guide* for details.

Writing to the Clipboard from IDLgrClipboard

Objects can be written to the operating system clipboard using `IDLgrClipboard::Draw`. When the `FILENAME` keyword equals an empty string (" "), equals 0 (zero), or is not specified, the output is written to the clipboard.

Note

The `IDLgrClipboard` object empties the Windows clipboard before writing to it.

Creating Clipboard Objects

The `IDLgrClipboard::Init` method takes no arguments. Use the following statement to create a clipboard object that represents the system-native clipboard buffer:

```
myClipboard = OBJ_NEW('IDLgrClipboard')
```

The following code creates an `IDLgrClipboard` object and outputs the contents of an `IDLgrView`, `IDLgrViewgroup`, or `IDLgrScene` to various files based on the platform. This is useful to determine exactly how the contents of the window are translated into bitmap or vector graphics. In the following code, `myview` denotes the name of the object (view, viewgroup, or scene) to be output. Vector postscript output is also generated using the CMYK color model.

```
oClip = OBJ_NEW('IDLgrClipboard')

; Create Windows-only output file types.
if !VERSION.OS_FAMILY eq 'Windows' then begin
    oClip->Draw, myview, VECTOR=0, POSTSCRIPT=0, $
        FILENAME="clipboard.bmp"
    oClip->Draw, myview, VECTOR=1, POSTSCRIPT=0, $
        FILENAME="clipboard.emf"
endif

; Create bitmap and vector PostScript files.
oClip->Draw, myview, VECTOR=0, POSTSCRIPT=1, $
    FILENAME="clipboard_bitmap.eps"
oClip->Draw, myview, VECTOR=1, POSTSCRIPT=1, $
    FILENAME="clipboard_vector.eps"

oClip->Draw, myview, VECTOR=1, POSTSCRIPT=1, $
    /CMYK, FILENAME="clipboard_cmyk.eps"

obj_destroy, oClip
```

See “[IDLgrClipboard](#)” in the *IDL Reference Guide* manual for details.

Printer Objects

Objects of the `IDLgrPrinter` class represent a physical printer onto which graphics hierarchies can be rendered in either bitmap or vector mode. What appears when producing bitmap or vector output depends upon several factors. See [“Bitmap and Vector Graphic Output”](#) on page 695 for details.

Creating Printer Objects

The `IDLgrPrinter::Init` method takes no arguments. Use the following statement to create a printer object:

```
myPrinter = OBJ_NEW('IDLgrPrinter')
```

This creates an object that maintains information about the printer. By default, this information pertains to the default printer installed for your system. To select a different printer or setup attributes of the printer, use the printer dialogs described in the next section.

See [“IDLgrPrinter”](#) in the *IDL Reference Guide* manual for details on creating printer objects. Also see [Chapter 7, “Printing in IDL”](#) for information on defining a printer.

Color Model

By default, printer objects use the RGB color model. To create a printer that uses the Indexed color model, set the `COLOR_MODEL` property of the printer object equal to 1 (one) when creating the printer:

```
myWindow = OBJ_NEW('IDLgrPrinter', COLOR_MODEL=1)
```

You cannot change the color model used by a printer after it has been created.

See [“Working with Color”](#) in Chapter 3 of the *Image Processing in IDL* manual for a discussion of the two color models.

Printer Dialogs

IDL includes two functions useful for controlling printers and print jobs.

DIALOG_PRINTERSETUP

Call the `DIALOG_PRINTERSETUP` function with the object reference of a printer object as its argument to open an operating system native dialog for setting the applicable properties of a particular printer. `DIALOG_PRINTERSETUP` returns a nonzero value if you pressed the **OK** button in the dialog, or zero otherwise.

```
result = DIALOG_PRINTERSETUP(myPrinter)
```

See [DIALOG_PRINTERSETUP](#) in the *IDL Reference Guide* for details.

DIALOG_PRINTJOB

Call the `DIALOG_PRINTJOB` function with the object reference of a printer object as its argument to open an operating system native dialog to initiate a printing job. `DIALOG_PRINTJOB` returns a nonzero value if you pressed the **OK** button in the dialog, or zero otherwise.

```
result = DIALOG_PRINTJOB(myPrinter)
```

See [DIALOG_PRINTJOB](#) in the *IDL Reference Guide* for details.

Drawing to a Printer

To draw a graphics tree to a printer, call the `IDLgrPrinter::Draw` method. The argument must be either an `IDLgrView` object, an `IDLgrViewGroup` object, or an `IDLgrScene` object.

```
myPrinter->Draw, myView
```

or

```
myPrinter->Draw, myScene
```

All objects contained within the scene, viewgroup, or view will be drawn to the printer.

Note

The scene or view to be drawn may be the same as the scene or view being displayed in one or more windows.

Printing in Bitmap or Vector Graphic Mode

The `IDLgrPrinter::Draw` method `VECTOR` keyword specifies whether the output is in bitmap or vector format. The following table shows the keyword options and results for each platform.

Keyword Settings	Windows Printer Output	UNIX File Type
<code>VECTOR = 0</code>	Bitmap (BMP)	Encapsulated PostScript (EPS) file (e.g. <code>xprinter.eps</code>)
<code>VECTOR = 1</code>	Enhanced MetaFile (EMF)	Encapsulated PostScript (EPS) file (e.g. <code>xprinter.eps</code>)

Table 34-2: File Types Produced by `IDLgrPrinter Draw` Method

`VECTOR=0` is the default. Because Windows printer output is usually sent directly to the printer, EMF and BMP files are not viewable. On UNIX, the printer output is directed to a file named `xprinter.eps` by default. For more information on printing views, scenes, or viewgroups, see “[IDLgrPrinter::Draw](#)” in the *IDL Reference Guide* manual.

Positioning Objects Within a Page

Objects can be positioned in a printed page by first determining the size of the page. Use the `IDLgrPrinter` object `DIMENSIONS` property to return the size of the “drawable” area of the page. You can then use these dimensions to draw a view of specified dimensions in the center of the printed page. The following two examples show positioning objects within the printed page:

- The first example scales an `orb` object based on the page size and draws the view containing the `orb` to the center of the hardcopy page. See “[Example: Centering an Orb](#)” on page 690.
- The second example creates two `IDLgrAxis` objects and an `orb` object, each with a `UNITS` property value set to centimeters. The view is positioned in the center of the page, but the other object locations are specified in centimeters and drawn to the view in precise positions. See “[Example: Precisely Positioning Vector and Bitmap Output](#)” on page 691.

Example: Centering an Orb

The following example positions a view containing an orb object in the center of a page when it is printed. Centering the view is a common task. Using this example as a guideline, you can easily adapt it to meet your own needs.

```

PRO center_doc

; Define dimensions in centimeters (cm).
dims = [5.0, 5.0]

; Create a view with centimeters as the units. Add the view to a
; model.
oView = OBJ_NEW('IDLgrView', $
    UNITS=2, $
    VIEWPLANE_RECT=[-dims[0]/2, -dims[1]/2, dims[0], dims[1]], $
    ZCLIP=[MAX(dims), -MAX(dims)], EYE=MAX(dims)+1, $
    COLOR=[200,200,200])
oModel = OBJ_NEW('IDLgrModel')
oView->Add, oModel

; Create an orb object and add it to the model.
oOrb1 = OBJ_NEW('orb', COLOR=[0,255,0], SHADING=1, $
    STYLE=2, HIDDEN=0)
oModel->Add, oOrb1

; Make radius 40% of window width.
oModel->Scale, dims[0]*0.4, dims[0]*0.4, dims[0]*0.4
oModel->Rotate, [1,1,0], 10

; Create a light and add it to the model.
oLight = OBJ_NEW('IDLgrLight', TYPE=1, LOCATION=[1.5,1.5,2])
oModel->Add, oLight

; Create a printer object, setting centimeters as the units.
oPrinter=OBJ_NEW('IDLgrPrinter', UNITS=2)

; Retrieve the drawable area of the page in the pagesize
; variable and use this to position the view.
oPrinter->GetProperty, DIMENSIONS=pageSize
centering = ((pageSize - dims)/2.)
oView->SetProperty, LOCATION=centering, DIMENSIONS=dims

; Print the view.
oPrinter->Draw, oView, VECTOR=1

OBJ_DESTROY,[oPrinter]
OBJ_DESTROY,[oView]

```

END

The following figure shows a subset of the output. The orb is positioned in the center of a printed page when you run this example.

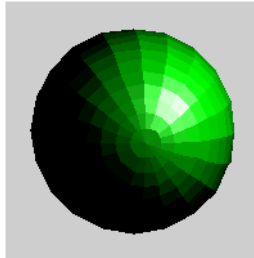


Figure 34-1: Output Centered in Printed Page

Example: Precisely Positioning Vector and Bitmap Output

The following example creates a model and draws some IDLgrAxis objects to the printer in vector mode. It then creates a second model for an orb object and plots the orb, drawing it to the printer in bitmap mode. The entire view is centered in the page, as shown in the previous example. However, this example precisely positions the orb and axes within the view using data units (defined as centimeters).

```
PRO center2_doc

; Set the view dimensions in units of centimeters (cm).
viewDims = [10.0, 10.0]

; Set the orb origin in cm, relative to the lower left
; corner of the view.
orbLoc = [3.0, 4.0]

; Set the Orb radius in cm.
orbRadius = 2.2

; Create the Orb object.
; The Orb object creates a unit orb with a default radius of 1.
oOrbModel = OBJ_NEW('IDLgrModel')
oOrb = OBJ_NEW('orb', COLOR=[0,255,0], SHADING=1, STYLE=2)
oOrbModel->Add, oOrb

; Create axes model. Create and position the axis objects.
oAxesModel = OBJ_NEW('IDLgrModel')
oX = OBJ_NEW('IDLgrAxis', 0, RANGE=[1,viewDims[0]-1], $
```

```

    /EXACT, LOCATION=[orbLoc[0]-orbRadius, 1])
oAxesModel->Add, oX
oY = OBJ_NEW('IDLgrAxis', 1, RANGE=[1, viewDims[1]-1], $
    /EXACT, LOCATION=[1, orbLoc[1]-orbRadius])
oAxesModel->Add, oY

; Add a box to show view extent.
oAxesModel->Add, OBJ_NEW('IDLgrPolygon', $
    [0, viewDims[0], viewDims[0], 0], $
    [0, 0, viewDims[1], viewDims[1]], STYLE=1)

; Create the view using the previously defined dimensions.
oView = OBJ_NEW('IDLgrView', $
    UNITS=2, VIEWPLANE_RECT=[0, 0, viewDims[0], viewDims[1]], $
    ZCLIP=[MAX(viewDims), -MAX(viewDims)], EYE=MAX(viewDims)+1, $
    COLOR=[255,255,255])
oTopModel = OBJ_NEW('IDLgrModel')
oView->Add, oTopModel

; Add a light.
oLight = OBJ_NEW('IDLgrLight', TYPE=1, LOCATION=[1.5,1.5,2])
oTopModel->Add, oLight

; Set up printer to print user-requested view. Center
; entire printer output in the page.
oPrinter=OBJ_NEW('IDLgrPrinter', UNITS=2)
oPrinter->GetProperty, DIMENSIONS=pageSize
centering = ((pageSize - viewDims)/2.)
oView->SetProperty, LOCATION=centering, DIMENSIONS=viewDims

; Print view containing axes in vector mode then remove model.
oTopModel->Add, oAxesModel
oPrinter->Draw, oView, VECTOR=1
oTopModel->Remove, oAxesModel

; Now float the orb into the view and print it in bitmap mode.
oTopModel->Add, oOrbModel
oView->SetProperty, VIEWPLANE_RECT = $
    [-orbRadius, -orbRadius, 2 * orbRadius, 2 * orbRadius], $
    LOCATION=[orbLoc[0]-orbRadius,orbLoc[1]-orbRadius]+centering, $
    DIMENSIONS=[2*orbRadius, 2*orbRadius]
oPrinter->Draw, oView, VECTOR=0

; oPrinter->NewDocument
OBJ_DESTROY,[oPrinter]
OBJ_DESTROY,[oView]

END

```

The following figure shows a subset of the output. The entire plot area is positioned in the center of a printed page when you run this example.

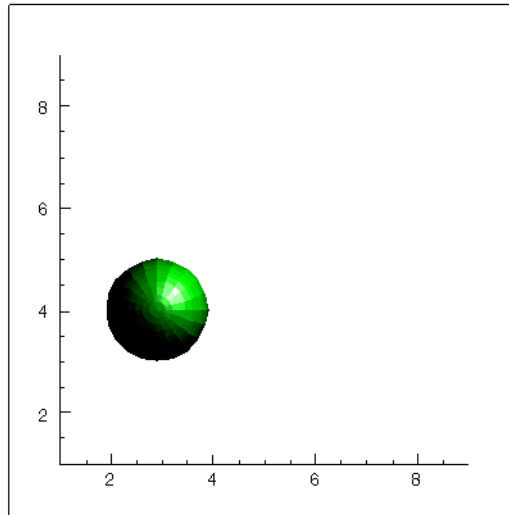


Figure 34-2: Positioning Objects Within a Printed Page

Starting a New Page on a Printer

To ensure that any subsequent calls to the `IDLgrPrinter::Draw` method occur on a new page, call the `IDLgrPrinter::NewPage` method:

```
myPrinter->NewPage
```

Submitting a Printer Job

To submit a printer job, call the `IDLgrPrinter::NewDocument` method. This method submits the printing job (consisting of all previous calls to `IDLgrPrinter::Draw` and `IDLgrPrinter::NewPage`) to the printer.

After this method has been called, the printer is prepared to accept a new batch of graphics calls (via `IDLgrPrinter::Draw`).

```
myPrinter->NewDocument
```

VRML Objects

Objects of the `IDLgrVRML` class allow you to save the contents of an Object Graphics hierarchy into a VRML 2.0 format file. The graphics tree can only contain a single view due to limitations in the VRML specification. The resulting VRML file is interactive and allows you to explore the geometry interactively using a VRML browser.

Creating VRML Objects

The `IDLgrVRML::Init` method takes no arguments. Use the following statement to create a VRML object:

```
myVRML = OBJ_NEW( 'IDLgrVRML' )
```

This creates an object that will convert object hierarchies rendered to it into VRML format files.

See [“IDLgrVRML”](#) in the *IDL Reference Guide* manual for details on creating and using VRML objects.

Bitmap and Vector Graphic Output

The IDLgrClipboard and IDLgrPrinter destination objects allow objects in a scene, viewgroup, or view to be output as vector or bitmap graphics. Which output is suitable depends upon the contents of the scene being sent to the output destination object. Understanding the difference between bitmap and vector graphics will help clarify why there is a difference in how the final output is displayed, and how the output can be edited.

Bitmap Graphics

Bitmaps are a collection of bits that describe the individual pixels within an image. Each pixel is a specific color, and the matrix of these pixels compose the image. In bitmap graphics, the contents of a view, viewgroup, or scene are captured as an image and are drawn with pixels in the bitmap. They can be edited only by altering individual pixels. The following figure shows the individual pixels that are visible when a small segment of an image is greatly enlarged.

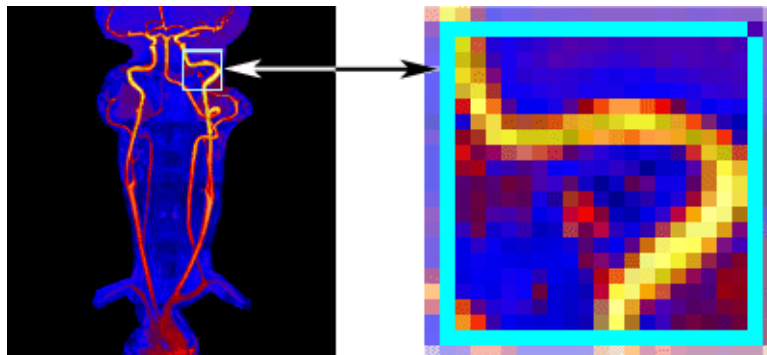


Figure 34-3: Sample Bitmap Image

IDLgrClipboard bitmap graphic output can be edited by any pixel-based paint program. In IDL, bitmap graphics can be stored as Bitmap (BMP) or PostScript (EPS) files under Windows, and as PostScript files under UNIX. Characteristically, bitmaps are large files, and image quality degrades when the image is substantially enlarged or reduced.

Vector Graphics

Vector graphics are described by simple graphic primitives. In the following figure, the vector output of the plot, shown on the left, is composed of multiple individual line segments that are defined mathematically. The IDLgrText objects are rendered as text primitives. All these primitives can be edited in vector graphic files. For example, in the following figure the final line segments in the plot have been repositioned in the right-hand image.

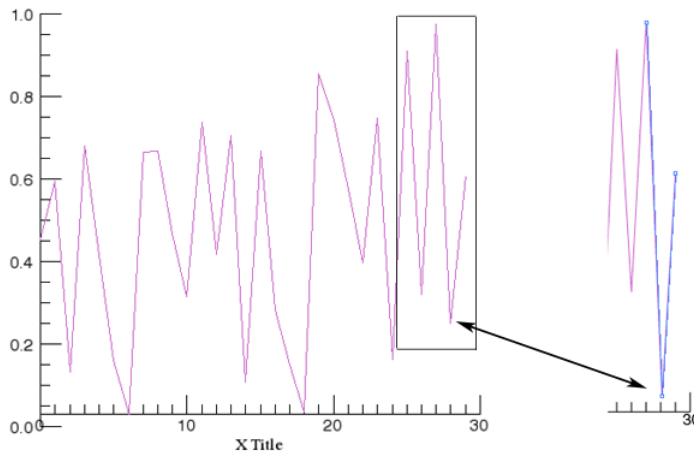


Figure 34-4: Sample Vector Image

IDLgrClipboard vector graphic output can be edited by an object-based graphical editor. In IDL, vector graphics can be stored as Enhanced MetaFile (EMF) or Encapsulated PostScript (EPS) files under Windows, and as Encapsulated PostScript (EPS) files under UNIX.

The main advantages of vector graphics are excellent scalability, and the ability to easily edit text and graphic features of the objects in the display. The graphic quality is maintained regardless of whether the graphic size is increased or decreased. The capabilities of the graphic editor determines what can be successfully edited. Simple lines and horizontal text can be easily edited in an EMF file inserted into a Microsoft Word document. However, more sophisticated graphic editors provide support for editing intricate graphic features and non-horizontal text. See [“Text Rendering in Vector Graphics”](#) on page 699 for more information. Vector graphics file sizes are generally smaller compared to bitmap graphics.

Guidelines for Choosing Bitmap or Vector Graphics

Advanced 3-D graphics rendering system output does not always map perfectly to a 2-D vector graphics system. The vector output is an approximation of what is displayed on the screen. How closely the vector output matches what is displayed depends upon the scene contents. Vector output may differ dramatically from bitmap output, and may also differ between the vector file formats (Encapsulated PostScript, Xprinter, and Enhanced MetaFile).

In general, scenes containing multiple, intersecting surfaces with various shading, transparency and lighting definitions are displayed with greater accuracy in a bitmap format than a vector format. However, simple 2-D plots are perfectly suited to vector output. Views containing the following items should not be output to vector graphic files:

- Transparent or semi-transparent objects — transparent objects in a view are not rendered in vector graphic files. Semi-transparent objects are rendered fully opaque.
- Textured or patterned objects — surfaces and polygons with textures or patterns are rendered without their textures or patterns.
- Hidden lines – polygon and surface objects drawn with the `HIDDEN_LINES` property set may experience missing lines.
- Volumes — volumes, other than those drawn in low quality wire frame mode (where the destination device `QUALITY=0`), are not rendered.
- Clipped objects — text strings and image objects do not appear clipped by clipping planes in vector graphic files. These objects only appear clipped by view boundaries.
- Smoothly shaded polygons and surfaces — Gouraud (smooth) shaded `IDLgrPolygon` and `IDLgrSurface` objects are displayed with smooth shading only in vector PostScript files generated by `IDLgrClipboard`, not in Enhanced MetaFile (EMF) vector format files, or in `IDLgrPrinter` vector EPS files. Polygons and surfaces appear with flat shading in EMF files and when printed.
- Lines and text in Xprinter — line style dash length is limited, and line style patterns cannot start and end with a '1' bit when vector output is generated by Xprinter under UNIX. Also, text is always drawn as a set of triangles in Xprinter vector output, and cannot be edited.
- Objects dependent on depth buffering — depth buffering controls are not respected in vector graphic files. See [“Primitive Object Sorting in Vector Graphics”](#) on page 700 for more information.

Controlling What is Displayed in Vector Graphics

Several factors beyond the differences between bitmap and vector graphics (described in the previous section) affect a vector graphics file in terms of content and the ability to edit text. Keywords provide control over factors such as object sorting, polygon shading, and text rendering when using the Draw method of the IDLgrClipboard or IDLgrPrinter destination objects. See the following sections for more information:

- [“Smooth Shading in Vector Graphics”](#) in the following section
- [“Text Rendering in Vector Graphics”](#) on page 699
- [“Primitive Object Sorting in Vector Graphics”](#) on page 700

Smooth Shading in Vector Graphics

The IDLgrClipboard Draw method supports the VECT_SHADING keyword, which affects the appearance of the surfaces and polygons when the VECTOR and POSTSCRIPT keywords have also been set. When SHADING=1 (Gouraud shading) for IDLgrSurface or IDLgrPolygon, use this keyword to control the rendering quality. Set the VECT_SHADING keyword to one of the following:

- 0 = disable smooth shading. Setting this keyword causes all polygons and surfaces to be rendered with flat shading. This will override the SHADING value assigned to a surface or polygon object. This may be valuable when using slower PostScript interpreters.
- 1 = enable smooth shading. Setting this keyword renders smoothly shaded polygons in the Encapsulated PostScript file. This is the default.

Note

Polygons and surfaces in Enhanced MetaFiles (EMF) will be rendered using flat shading. Only the output in Encapsulated PostScript (EPS) files is affected by this keyword, and only when the VECTOR keyword has been set.

Text Rendering in Vector Graphics

Text can be easily edited in vector graphic files when the text is output as text primitives. In bitmap files, text glyphs cannot be edited except by modifying individual pixels. In a vector graphic file, IDLgrText objects are rendered as graphic primitives that can be edited. The IDLgrClipboard or IDLgrPrinter VECT_TEXT_RENDER_METHOD keyword controls whether text appears as filled triangles or text primitives when the VECTOR keyword is also set. Set the VECT_TEXT_RENDER_METHOD keyword to one of the following:

- 0 = render text as text primitives. This uses the output device's text primitives when rendering text. This allows the text to be edited by object-based graphics programs. This is the default.
- 1 = render text as triangles. This produces text glyphs that closely match the text on the display device. The output file size is larger and contains filled triangles to represent text. This can preserve backward compatibility with the display of text objects prior to IDL 6.1, which introduced text primitives.

Note

When using the IDLgrPrinter object under UNIX, the Xprinter output is regarded as write-only. As there is no support for 3-D text, IDL always generates filled triangles when rendering text in the Xprinter output. See “[IDL Printer Setup in UNIX or Mac OS X](#)” on page 131 for more information on Xprinter.

Setting VECT_TEXT_RENDER_METHOD=0 creates a vector graphics file with text rendered as primitives. The text associated with the graphic can be scaled, transformed or repositioned when edited in an object-oriented graphics application.

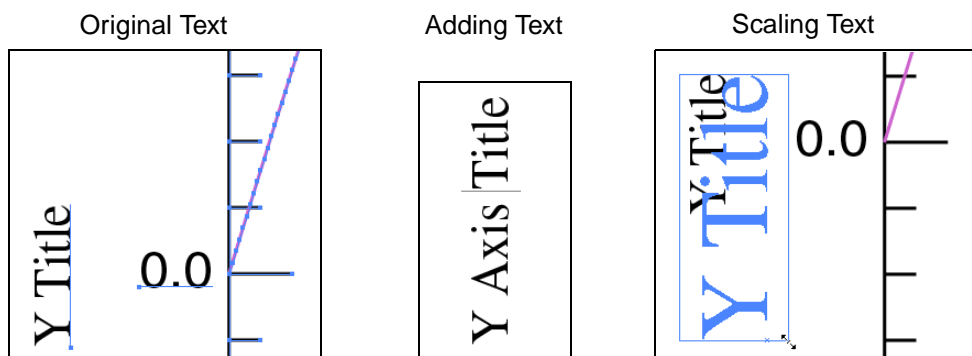


Figure 34-5: Editing Text Objects Output as Vector Graphics

An Enhanced MetaFile (EMF) inserted into a Microsoft Word document can be edited. However, not all versions of Microsoft Word support advanced 3-D graphic primitives such as those associated with obliquely or vertically aligned text. Choosing to edit a file with non-horizontally aligned text may result in the text being flattened into two dimensions. Typically, each letter becomes its own string and alignment is altered. To edit non-horizontal text and preserve the original quality, create an Encapsulated PostScript (EPS) file that can be modified in a more sophisticated object-oriented image editing program.

Setting `VECT_TEXT_RENDER_METHOD=1` creates text that is rendered as filled triangles. Elements of the plot in the following figure are composed of line segments that can be edited, but the text characters cannot be individually edited. The triangles composing the letters of the text object are visible in the right-hand image.

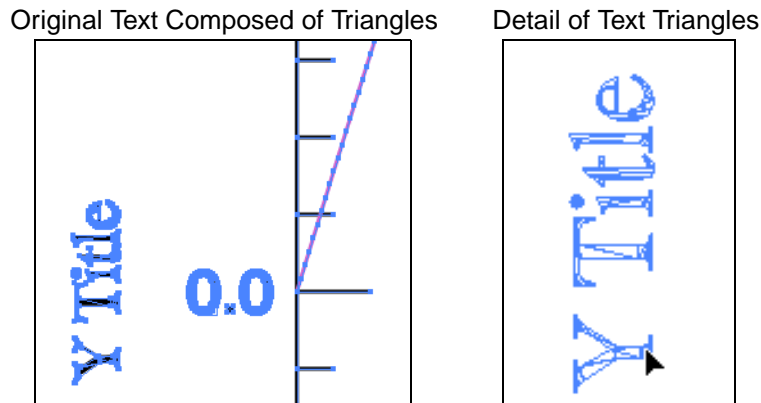


Figure 34-6: Text Objects Output as Triangles

Primitive Object Sorting in Vector Graphics

The `IDLgrPrinter` and `IDLgrClipboard` Draw methods support the `VECT_SORTING` keyword, which affects the appearance of the output when the `VECTOR` keyword has also been set. Use this keyword to simulate the depth buffer in Object Graphics in the output vector graphics file. Set the `VECT_SORTING` keyword to one of the following:

- 0 = disable sorting. The object primitives appear in the vector output file in the same order they are drawn on the display device. This is the order in which they appear in the graphics tree.
- 1 = enable sorting. Objects are ordered from back to front based on each primitive object's average depth value. This is the default.

The following figure shows the results of changing the `VECT_SORTING` keyword. When sorting is disabled (`VECT_SORTING=0`) as in the left image, the first object added to the model is drawn first in the display and in the destination device. In the code used to create the left image, the text is added to the model before the surface. Therefore it appears behind the surface in the vector graphics file. When the order is reversed, the text is drawn on top of the surface. When sorting is enabled (`VECT_SORTING=1`) as in the right image, primitive objects are sorted according to their depth in the view. Most distant objects are drawn first. When two objects have the same average depth, the object added to the model first is drawn first and will appear behind subsequent objects.

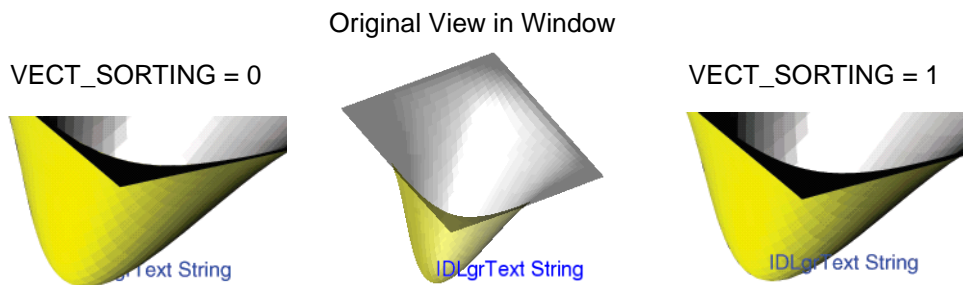


Figure 34-7: Controlling the Sorting of Object Primitives

Note

Vector output does not support depth test functions. Vector output resolves Z (depth) ties by using the `DEPTH_TEST_FUNCTION` default `LESS` depth test.

There are two instances in which the above sorting model is not applicable:

- In a window containing overlapping, transparent views
- In a window containing `IDLgrImage` objects

See the following sections for details.

Sorting Issues with Transparent Views

When a window contains multiple views, the objects in each view are sorted as a separate group. This simulates the default clear operation that IDL performs when drawing each view to a destination, clearing the depth buffer and repainting the view with the view color. Depending upon the ordering and transparency of the views, the vector output might not match what is displayed, regardless of the value of `VECT_SORTING`. Consider objects in a transparent view that are positioned behind

an object in a non-transparent view. In the display, objects in the transparent view are occluded by the object that appears closer to the viewer. However, in the vector output, the objects in the transparent view interact with and are visible in the output. This occurs because IDL does not clear the depth buffer or repaint the view when it is transparent.

In the simple example shown in the following figure, the `IDLgrText` object is added to a transparent view and is positioned behind the surface. The view associated with the `IDLgrSurface` is not transparent. The view containing the surface and the transparent view containing the text are added to an `IDLgrViewgroup` and displayed in the window. The left image shows the vector file output, and the right image shows the bitmap file output. In the vector output, all of the text is visible because the views are sorted independently. This behavior occurs because the transparent view containing the text is added to the viewgroup after the view containing the surface. If the view containing the text is added first, then only the surface (whose view is not transparent) is drawn.

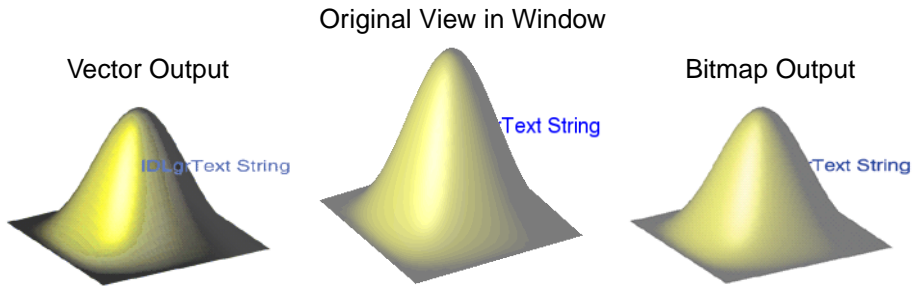


Figure 34-8: Interaction of Object Primitives with Transparent Views

Transparent Images

When IDL draws a semi-transparent image with vector output, it must draw it completely opaque, as it does with other primitives. Therefore, if you use image layers, where one image is semi-transparent in order to let you see another image drawn before it, the output will not be correct with vector output since the semi-transparent image will be drawn opaquely, completely hiding the image drawn before it. You should use bitmap output to get the desired results because semi-transparent rendering is not available with vector output.

Note

As described in [“Guidelines for Choosing Bitmap or Vector Graphics”](#) on page 697, all transparent objects (not just image objects) are rendered opaque in vector output.

Sorting Issues Among Image and Non-Image Objects

On a display device, IDLgrImage objects are drawn as “pixel primitives,” which means that they do not update the depth buffer when they are written to the screen and also are not tested against the depth buffer to determine if they should be drawn or not. Images are rendered at $Z=0$ in viewing coordinates. This means:

- Images always overwrite any graphical data on the screen in the area in which they are drawn, regardless of their relative depth in the scene. Even objects that are rendered closer to the viewer than the image are overwritten.
- Objects that are drawn after an image is on the screen are drawn as if the image was not there. Since rendering the image did not update the depth buffer in the region where the image was rendered, the objects drawn after the image are not depth-tested against the image. This means that if you render an object, after rendering an image, so that it appears deeper than the image ($Z < 0$ in viewing coordinates), the object will render “on top” of the image, even though it is physically behind it in the scene.

For these reasons, IDL applications often place image objects in the graphics tree so that they render first, unless the application wishes to make use of the behaviors described in the above two points. IDL emulates this behavior with vector graphics when `VECT_SORTING` is on as follows:

- Image objects are drawn in the order that they are positioned in the graphics tree.
- Non-image objects positioned before, after, or between image objects in the graphics tree are sorted amongst themselves. That is, non-image objects that are positioned in the tree before the first image are sorted and drawn first. Then the image is drawn. Then the next group of non-image objects are sorted and drawn, etc.

These steps assure consistency between bitmap and vector output for overlapping image and non-image primitives. However, some sorting differences may occur between non-image primitives that overlap each other but do not overlap images. For example, consider two non-image primitives drawn on the screen so that they do not overlap an image, and one of these primitives is positioned in the graphics tree before (drawn before) the image, and the other is positioned in the graphics tree after (drawn after) the image. These two primitives are not sorted with respect to each other and are always drawn so that the second primitive is drawn after the first, regardless of their relative depth in the scene. If these primitives overlap, the result may not be correct if the first primitive is closer to the viewer than the second. Again, in this case, consider using bitmap output for more accurate output.



Chapter 35:

Subclassing from Object Graphics

This chapter describes the creation of composite classes or subclasses in Object Graphics. The following topic is covered in this chapter:

[Creating Composite Classes or Subclasses](#) 706

Creating Composite Classes or Subclasses

RSI has provided a rich set of basic objects that can be used for creating visualizations. You may find that you are using a certain combination of these objects again and again within your applications for a particular purpose. If this is the case, you might want to consider defining a composite object class that encapsulates the combination of those subcomponents.

In fact, RSI has already defined a few composite classes on your behalf. These include the [IDLgrColorbar](#) object and the [IDLgrLegend](#) object found in the *IDL Reference Guide*. You will find the IDL code for these objects in the `lib` directory of your IDL distribution.

Another example can be found in the `idlexshow3__define.pro` in the `examples/visual/utility` subdirectory. In this case, an image, surface, and contour representation are combined into a single object called the `IDLexShow3` object. To see this object being used in an application, run the `show3_track` routine, defined in the file `show3_track.pro` in the `examples/visual` directory. This example creates the following visualization:

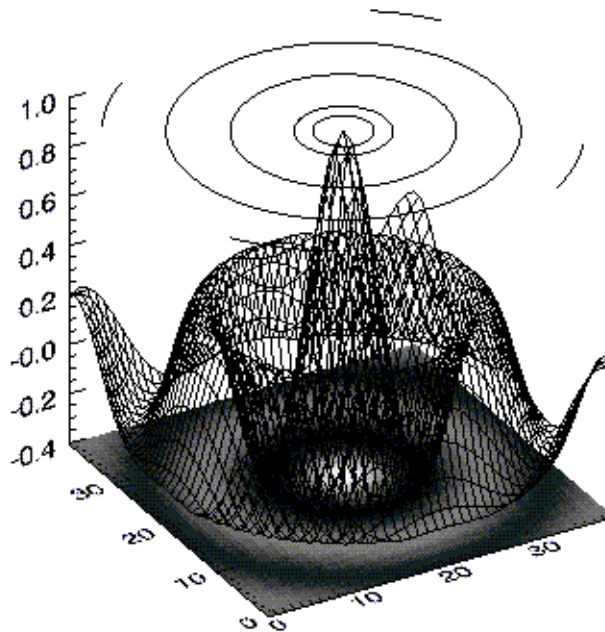
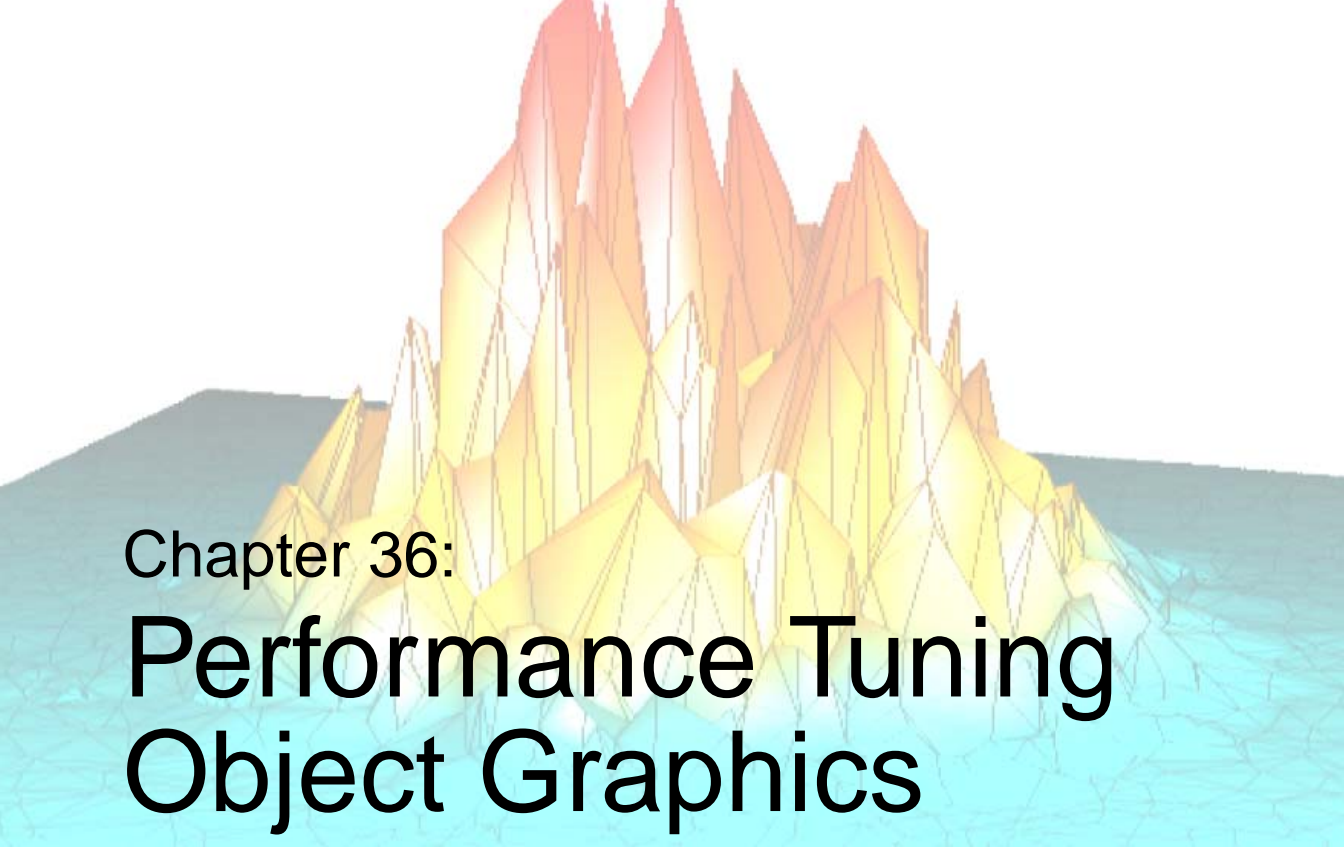


Figure 35-1: `Show3_track` example

You may also find that you want to customize one or more of the classes available in Object Graphics. For instance, you may want to create a specialized image object that can handle 16-bit palettes. An example of this is provided in `idlexpalimage__define.pro` in the `examples/visual/utility` subdirectory of the IDL distribution.



Chapter 36: Performance Tuning Object Graphics

The following topics are covered in this chapter:

Overview	710	Improving Redraw Performance	716
Polygon Mesh Optimization	711	Back-face Culling	717
Normal Computations	714	Lighting	718
Retained Graphics and Expose Events ...	715		

Overview

The Object Graphics subsystem is designed to provide a rich set of graphical functionality that can be displayed in reasonable time. This section offers suggestions on how to utilize the object graphics in such a way as to take full advantage of performance enhancement benefits.

Polygon Mesh Optimization

IDLgrPolygon objects consist of a set of vertices and, optionally, a connectivity array describing how those vertices are to be connected to form one or more polygons. Internally, IDL can identify three special types of polygonal meshes that may be represented very efficiently and therefore displayed substantially faster than individually described polygons. These special mesh types are characterized by repetitive patterns in the connectivity of the vertices. In performance terms, it is to the user's advantage to utilize this optimization whenever possible by appropriately preparing the connectivity list according to the rules described for the corresponding type of mesh. The special mesh types are as follows:

Quad Strips

A quad strip is a connected set of four-sided polygons (see [“Polygon Mesh Optimization”](#) in Chapter 29). To take advantage of accelerated quad strips, the connectivity should be set up so that the first and last vertex for one quad are the same as the second and third of the previous quad.

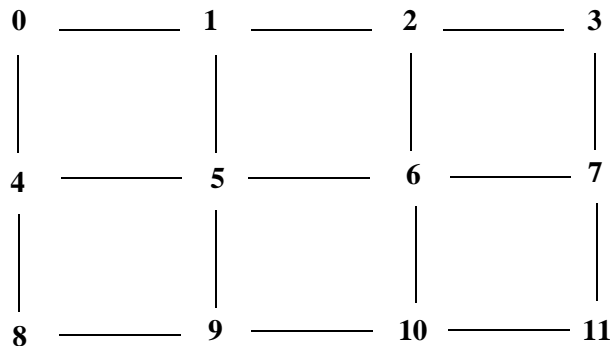


Figure 36-1: Quad Strip Mesh

For example, to use a quad strip optimization for the polygons in the figure above, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11]
oPoly = OBJ_NEW('IDLgrPolygon', verts, $
    POLYGON=[4, 0, 1, 5, 4, $
              4, 1, 2, 6, 5, $
              4, 2, 3, 7, 6, $
              4, 4, 5, 9, 8, $
              4, 5, 6, 10, 9, $
              4, 6, 7, 11, 10])
```

Triangle Fans

A triangle fan mesh is a set of connected triangles that all share a common vertex. To take advantage of accelerated triangle fans, the connectivity should be set up so that the first vertex in every triangle is the common vertex, and the second vertex is the same as the last vertex of the previous triangle.

For example, to use a triangle fan optimization for the polygons in the left-hand figure below, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5]
oPoly = OBJ_NEW('IDLgrPolygon', verts, $
    POLYGON=[3, 0, 1, 2, $
              3, 0, 2, 3, $
              3, 0, 3, 4, $
              3, 0, 4, 5])
```

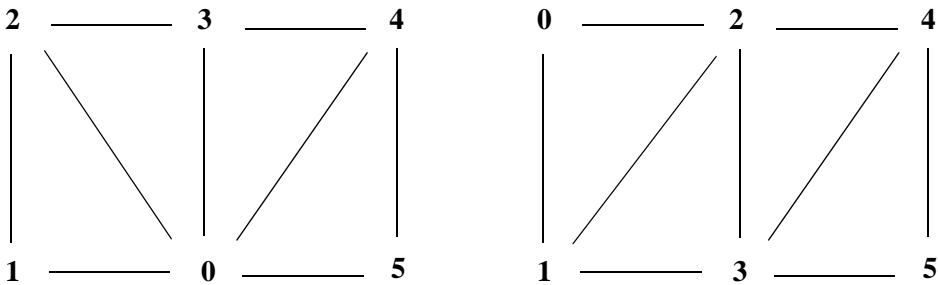


Figure 36-2: Triangle Fan Mesh (left) and Triangle Strip Mesh (right)

Triangle Strips

A triangle strip mesh is a set of connected triangles, each of which share two vertices with the previous triangle. To take advantage of accelerated triangle strips, the connectivity should be set up so that the first two vertices in every triangle must have been in the previous triangle and ordered in the same direction (counter-clockwise or clockwise) and the final vertex must be new.

For example, to use the triangle strip optimization for the polygons in the right-hand figure above, the connectivity for the vertices should be as follows:

```
verts = [v0, v1, v2, v3, v4, v5]
oPoly = OBJ_NEW('IDLgrPolygon', verts, $
    POLYGON=[3, 0, 1, 2, $
              3, 2, 1, 3, $
              3, 2, 3, 4, $
              3, 4, 3, 5])
```

No limits are imposed on the number of meshes or types of meshes within any given polygon object. A single POLYGON keyword value might contain any combination of quad strips, triangle strips, triangle fans, or non-specialized polygons.

As the length of the strips or fans grows, and as the percentage of vertex connections that are optimized by the rules described above increases, the performance upgrade becomes more perceptible. The optimizations are a result of minimizing the time required to perform vertex transforms. If the drawing of the polygons are otherwise limited by fill-rate (as might occur on some systems if texture-mapping is being applied, for instance), these optimizations may not be of significant benefit. In any case, performance will not be hindered in any way by utilizing these specialized meshes, so it is suggested that they be applied whenever possible.

Note

The IDLgrSurface object always takes advantage of the quad mesh optimization automatically without programmer intervention.

Normal Computations

For IDLgrPolygon objects, normal vectors are computed by default at each vertex by averaging the normals of the polygons that share that vertex. These normals are then used to compute illumination intensities across the surface of the polygon.

Computing default normals is a computationally expensive operation. Each time the polygon is drawn, this computation will be repeated if the polygon has changed significantly enough to warrant a new internal cache (for example, if the connectivity, vertices, shading, or style have changed). In some cases, the normals do not actually change as other modifications are made. In these cases, the expense of default normal computation can be bypassed if the user provides the normals explicitly (via the `NORMALS` keyword). These normals can be computed by using the `COMPUTE_MESH_NORMALS` routine in the *IDL Reference Guide*. The resulting normals, if passed in via the `NORMALS` keyword of the IDLgrPolygon object, will be reused every time the polygon is drawn (without further computation) until they are replaced explicitly by the user.

Retained Graphics and Expose Events

During the course of an IDL session, it is possible that an IDL window will be obscured by another window. When the hidden window is brought to the front, its contents need to be regenerated. The user interface toolkit portions of the window are repaired automatically. However, the drawable portion of the window (in which graphics are rendered) requires special attention. The user can choose between two methods to handle this situation. The first option is to set the `RETAIN` property on the `IDLgrWindow` object to 2, which suggests that IDL is required to retain a backing store of the entire contents of the window. When the window is exposed, the backing store will be copied to the screen. The second option is to set the `RETAIN` property to 0 (no retention), and to request that expose events are to be reported for draw widgets. Whenever a portion of the window becomes exposed, an event is generated. The event handler for the drawable can then re-issue a draw of the appropriate contents for that window.

While the second option may seem a bit more complicated, it is to the users advantage to take this approach for performance reasons. When `RETAIN` is 0, the window device drivers are able to utilize a double-buffered rendering scheme that can capitalize on hardware acceleration. For interactive applications, this hardware acceleration can have a crucial impact on the perceived manipulation capabilities of the interface. When `RETAIN` is 2, on the other hand, IDL will render to an offscreen pixmap, which often relies on a software implementation. If several drawing calls are issued in a row, the performance may be noticeably slower.

Improving Redraw Performance

Within interactive graphics applications, it is often necessary to redraw a given view over and over again (for example, as the user clicks and drags within the view to manipulate one or more objects). During those redraws, it may be that only a small subset of the objects within the view is changing, while the remaining objects are static. In such a case, it may be more efficient to take a snapshot of the unchanged portion of the view. This snapshot can be reused for each draw, and only the changing portion of the view needs to be re-rendered. This process is called instancing. For more information on instancing, see [“Instancing”](#) on page 682.

Back-face Culling

For polygonal meshes that describe a closed shape (for example, a sphere), it is often wasteful to spend any time rendering the polygons whose normal vector faces away from the eye because it is known that the polygons whose normals face toward the eye will obscure those back-facing polygons. Therefore, for efficiency, it may be beneficial to employ back-face culling, which is simply the process of choosing to skip the rasterization of any polygons whose normal vector faces away from the eye.

On an `IDLgrPolygon` object, set the `REJECT` property to a value of 1 to enable back-face culling.

Lighting

Lighting computations are generally set up to compute the light intensity based on the normal vector for the polygon. If the polygon normal faces away from the eye, the lighting model will likely determine that the light intensity for that polygon is zero. When the polygonal mesh being rendered is a closed surface, this is not a problem because the back-facing polygons will always be obscured. However, when the polygon mesh represents an open shape (for which back-facing polygons may be visible), the dark appearance of these polygons may hinder the user's perception of the overall shape. In such a case, two-sided lighting can be useful. Two-sided lighting is the process of reversing the normals for all back-facing polygons before computing the light intensities for that polygon.

In IDL's Object Graphics, two-sided lighting is enabled by default. When the additional lighting calculation is not required, one-sided lighting can be used to improve rendering performance. On an `IDLgrModel` object, set the `LIGHTING` property to a value of 1 to enable one-sided lighting.



Index

Symbols

- !EDIT_INPUT system variable, command recall, [88](#)
- !MAP system variable, [340](#)
- !ORDER system variable, [376](#)
- !P.MULTI system variable, [271](#)
- # operator, [442](#)
- ## operator, [443](#)
- \$ character, operating system commands, [92](#)
- .sav file, saving data and variables, [171](#)
- @ character, [176](#)

Numerics

- 3D
 - text objects, [612](#)
 - transformations, [323](#)

A

- aborting IDL, [170](#)
- accuracy
 - floating-point operations, [438](#)
 - numerical algorithms, [436](#)
- action routines, [153](#)
- Aitoff map projection, [352](#)
- Albers equal-area conic projection, [362](#)
- aliasing, [413](#)
- aligning text objects, [612](#)
- alpha blending, [651](#), [664](#)
- alpha channel, [650](#)
- analytic signal, [415](#)
- animation, [321](#)
- annotating text objects, [611](#)
- annotations, plots, [255](#)
- ARMA filter, [420](#)

arrays
 multiplying, 442
 rotating, 315
 sparse, 482
 stored in structure form, 482
 symmetric, 442

ASCII files
 importing using macros, 191
 reading, 223

atomic graphic objects, 510, 535

attribute objects, 514, 536, 578

autoregressive moving average filters, 420

axes
 logarithmic, 269
 plotting, 287
 range, 251
 scaling, 249

axis object, 510, 593

axis tick labels, 609

axis title, 609

azimuthal equidistant map projection, 351

azimuthal map projections, 348

B

back-face culling, 717

backing store
 bitmap buffered, 104
 graphics, 104
 system buffered, 104

bandpass filters, 421

bandstop filters, 421

bar charts, 262

baseline changes to text objects, 613

batch file, defined, 162

batch files
 creating, 176
 interpretation of, 177
 startup preference, 109

bilinear, transform, 427

binary data
 importing using macros, 197
 template interface, 231

bitmap buffered backing store, 104

bitmap graphics
 defined, 695
 IDLgrClipboard, 685
 IDLgrPrinter, 687
 text rendering, 695
 versus vector, 695
 when to use, 697

boundaries, maps, 341

box charts, 262

boxcar filter, 425

Bristol Technology
 printer manager, 131
 printing graphics, 129

buffer object, 516

buffer objects
 creating, 684
 overview, 684

byte ordering
 big endian, 198, 232
 binary data, 198, 231
 little endian, 198, 232
 native method, 198, 232

C

caching, 25

CALDAT, 278, 601

calendar dates, stored as Julian, 276, 599

CDE File Manager, 160

cell drawing using contour method, 296

central map projection, 350

channels
 alpha, 650
 image objects, 650

chromacoded editor (Windows), 80

CIA World Map database, 368

clipboard object, 516

- clipboard objects, creating, 686
- clipboard support, graphics windows, 46
- clipping planes, 549
- cluster analysis, routines, 493
- code, using the IDL editor, 74
- color, mapping voxel values, 663
- color indices, 390
- color maps
 - flashing, 146
 - sharing (Motif), 146
- color model
 - destination objects, 570
 - digital data, 567
 - indexed, 568
 - palette objects, 571
 - printers, 687
 - RGB, 567, 569
 - window objects, 677
- color property, specifying color values, 572
- color systems
 - HLS, 381
 - HSV, 381
 - RGB, 381
- color tables
 - common block, 387
 - loading, 382
 - obtaining, 387
 - overview, 381
 - predefined, 384
 - switching between devices, 387
- colorbar object, 513
- colorbar objects
 - creating, 654
 - overview, 654
 - using, 654
- coloring, vertices, 625
- colors
 - indices, 390
 - reserving for IDL, 146
 - tables, *See* color tables
- COLORS common block, 387
- combining, contour and surface plots, 323
- combining transformations, 554
- Command Input Line, 47
- command line options, Motif platform, 147
- command line switches, 27
- command recall
 - setting the buffer, 89
 - use of, 86
- Common Desktop Environment (CDE) File Manager, 160
- common methods in object classes, 507
- compiling
 - automatically, 165
 - from memory preference, 106
 - preferences, 106, 106
- composite classes, 706
- concave polygons, 589
- conformal conic map projection, 361
- container object, 508
- continents, 344
- contour object, 510, 616
- contour plots
 - direction of grade, 309
 - filling, 308
 - labeling, 306
 - overlying images, 301, 301
 - overview, 295
 - smoothing, 307
- Control Panel Buttons
 - modifying in Motif, 150
 - Motif platform, 48
- controlling the device cursor, 393
- convex polygons, 589
- Cooley-Tukey algorithm, 414
- coordinate conversion, 555, 557
- coordinate systems
 - scaling coordinates, 542
 - transformation, 542
- coordinate transformations, 555
- coordinates
 - converting, 247, 318

coordinates (*continued*)
 data, 246
 device, 246
 homogeneous, 313
 normal, 246
 correlation
 analysis, 446
 coefficient, 446, 447
 routines, 449
 creating
 buffer objects, 684
 clipboard objects, 686
 colorbar objects, 654
 contour objects, 616
 image objects, 650
 legend objects, 630
 light objects, 643
 plot objects, 626
 polygon objects, 619
 polyline objects, 625
 printer objects, 687
 surface objects, 638
 volume objects, 660
 VRML objects, 694
 window objects, 677
 creating axis objects, 593
 creating text objects, 611
 culling to improve performance, 717
 cursor
 controlling position, 393
 determining position, 291
 curve fitting
 discussion, 450
 routines, 451
 customizing IDL, preferences, 94
 cyclical fluctuation, 485
 cylindrical equidistant map projection, 360
 cylindrical map projections, 358

D

data coordinates, 246
 data picking, 668, 672
 date/time, precision, 277, 600
 date/time data
 displaying, 279, 601
 generation, 277, 600
 using system variables to display, 284
 debugging, executive commands, 90
 default font, 579, 614
 Delaunay triangulation, 459
 deleting, lines in Output Log, 98
 derived variables, 489
 destination device, 676
 destination objects, 516
 color models, 570
 drawing, 676
 device
 controlling the cursor, 393
 coordinates, 246
 graphics, independent, 242
 independent graphics, 242
 device independent graphics, 503
 DFT, 401
 dialogs for printing, 687
 dicer, 334
 DICOM object, 517
 differentiation, routines, 469
 digital filters, 420
 digital signal processing, 397
 DIGITAL_FILTER function, 421
 Direct Graphics, 502
 clipboard support, 46
 overview, 240
 printing, 129
 direction of grade, 309
 discrete Fourier transform, 401
 discrete wavelet transform, 417
 DISPLAY environment variable, 26
 displaying images, overview, 375
 displaying isosurfaces, 332

- dot product, 445
- draw widgets, 570, 677
- drawing
 - continents on maps, 344
 - destination device, 676
 - meridians, 341
 - parallels, 341
- drawing to a printer object, 688
- DWT, 417
- DXF object, 517

E

- editing, command line, 86
- editing resource files, 144
- Editor window
 - compiling and saving, 106
 - preferences, 106
- Editor windows, defined, 45
- editors, external (Motif), 83
- eigenvalues
 - complex, 454
 - real, 453
 - repeated, 456, 457
 - routines for computing, 458
- eigenvectors
 - complex, 454
 - real, 453
 - repeated, 457
 - routines for computing, 458
- EMF file, 696
- endian
 - big, 198, 232
 - byte ordering, 198, 232
 - little, 198, 232
- environment variables
 - DISPLAY, 26
 - IDL_DEVICE, 24
 - IDL_DIR, 24
 - IDL_DLM_PATH, 24
 - IDL_HELP_PATH, 25
- environment variables (*continued*)
 - IDL_PATH, 25
 - IDL_STARTUP, 25
 - IDL_TMPDIR, 26
 - LM_LICENSE_FILE, 26
 - PATH, 21
 - TERM, 26
 - UNIX platform, 26
 - used by IDL, 22
- equal-area map projection, 362
- erasing a window object, 680
- error
 - floating-point, 438
 - mathematical error assessment, 439
 - rounding, 438
 - truncation, 439
- executive commands, 90
- exiting IDL
 - confirm exit, 97
 - options, 35
 - preferences, 97
- expose events, 715
- exposing window objects, 680
- external editors (Motif), 83
- eye position, 547

F

- far clipping plane, 549
- Fast Fourier transform, 401
 - implementation, 414
- FFT, 401, 409
 - Cooley-Tukey algorithm, 414
 - implementation, 414
- file selection
 - using compound widgets, 219
 - using dialogs, 217
- files
 - MPEG, 656
 - saving images, 656
 - specifying search path, 112

filling contours, 308
 filling polygons, 261, 620
 filters
 autoregressive moving average, 420
 bandpass, 421
 bandstop, 421
 boxcar, 425
 digital, 420
 FIR, 420
 highpass, 421
 IIR filter, 420
 Kaiser's window, 421
 lowpass, 421
 moving average, 420
 notch, 427
 rectangular, 425
 finding, text, IDLDE search features, 55
 finite, impulse response filters, 420
 FIR filter, 420
 fixed pixels, 303
 flashing color maps, 146
 floating-point accuracy, 438
 font object, 514, 579
 FONT property of text objects, 614
 fonts
 default, 579, 614
 hardware, 256
 Hershey, 580
 preferences, 110
 selecting, 256
 specifying
 Motif platform, 110
 Windows platform, 110
 TrueType, 579
 type size, 614
 type style, 614
 Fourier transform, discrete, 401
 Fourier transforms, fast, 401
 frequency plot leakage, 407
 frequency plot smearing, 407
 frequency response function, 428

G

Gaussian, iterated quadrature, 465
 Gauss-Krueger map projection, 359
 general perspective map projection, 354
 glyph. *See* TrueType fonts
 gnomonic map projection, 350
 gnomic map projection, 350
 Gouraud shading, 329
 grade, 309
 graphics
 atoms, 510, 535
 bitmap versus vector, 695
 clipboard support, 46
 coordinate systems, 246, 313
 data picking, 668
 device independent graphics, 242, 503
 devices, 241
 devices for Direct Graphics, 502
 IDLDE windows, OS clipboard support, 46
 modes, 240, 502
 object-oriented, 242, 503
 selecting, 668
 two-dimensional arrays, 294
 graphics hierarchy, 530
 graphics object tree, 530
 graphics windows
 backing store, 104
 layout preferences, 103
 sizing, 103
 graphs, 615
 graticule, 343
 gridding
 data extrapolation, 459
 Delaunay triangulation, 459
 drawing meridians, 341
 routines, 460
 uniformly-spaced planar data, 459
 gridding plots, 265

H

- Hammer-Aitoff map projection, [353](#)
 - Hamming window, [411](#)
 - Hanning window, [409](#)
 - hardware rendering, setting preference for, [104](#)
 - HDF files, importing using macros, [203](#)
 - HDF-EOS files, importing using macros, [203](#)
 - helper objects, [515](#), [536](#)
 - Hershey fonts, [580](#)
 - hidden line removal, [641](#)
 - hidden object classes, [527](#)
 - hiding the toolbars, [102](#)
 - hiding window objects, [680](#)
 - hierarchy of objects, [530](#)
 - highpass filters, [421](#)
 - high-resolution continent outlines, [368](#)
 - Hilbert transform, [415](#)
 - histogram
 - plot, [398](#)
 - plotting mode, [260](#)
 - HLS color system, [383](#)
 - home directory, changing, [108](#)
 - HOME environment variable, [23](#)
 - homogeneous coordinates, [313](#)
 - HSV color system, [383](#)
 - hue, [383](#)
 - hypothesis testing
 - routines, [464](#)
 - statistics, [462](#)
- ## I
- iconifying windows, [680](#)
 - IDL
 - Direct Graphics, [240](#), [502](#)
 - Object Graphics, [240](#), [502](#)
 - IDL GUIBuilder
 - generating files, menu option, [51](#)
 - IDLDE, Windows platform, [45](#)
 - IDL Printing, [128](#)
 - IDL_Container class, [508](#)
 - IDL_CPU_TPOOL_NTHREADS, [24](#)
 - IDL_DEVICE environment variable, [24](#)
 - IDL_DIR environment variable, [24](#)
 - IDL_DLM_PATH environment variable, [24](#)
 - IDL_HELP_PATH environment variable, [25](#)
 - IDL_PATH environment variable, [25](#)
 - IDL_PATH_CACHE_DISABLE environment variable, [25](#)
 - IDL_STARTUP environment variable, [25](#)
 - IDL_TMPDIR environment variable, [26](#)
 - IDLDE preferences, [93](#)
 - IDLffDICOM class, [517](#)
 - IDLffDXF class, [517](#)
 - IDLffJPEG2000 object, [517](#)
 - IDLffLangCat class, [517](#), [517](#)
 - IDLffMrSID class, [517](#)
 - IDLffShape class, [518](#)
 - IDLffXMLSAX class, [518](#)
 - IDLgrAxis class, [510](#)
 - IDLgrBuffer class, [516](#)
 - IDLgrClipboard class, [516](#)
 - IDLgrColorbar class, [513](#)
 - IDLgrContour class, [510](#)
 - IDLgrFont class, [514](#)
 - IDLgrImage class, [510](#)
 - IDLgrLegend class, [513](#)
 - IDLgrLight class, [510](#)
 - IDLgrModel class, [509](#)
 - IDLgrMPEG class, [517](#)
 - IDLgrPalette class, [514](#)
 - IDLgrPattern class, [514](#)
 - IDLgrPlot class, [510](#)
 - IDLgrPolygon class, [511](#)
 - IDLgrPolyline class, [511](#)
 - IDLgrPrinter class, [516](#)
 - IDLgrScene class, [509](#)
 - IDLgrSurface class, [511](#)
 - IDLgrSymbol class, [514](#)
 - IDLgrTessellator class, [515](#)

- IDLgrText
 - bitmap graphics, 695
 - vector graphics, 696
- IDLgrText class, 511
- IDLgrView class, 509
- IDLgrViewgroup class, 509
- IDLgrVolume class, 512
- IDLgrVRML class, 516, 518
- IDLgrWindow class, 516
- IIR, 420
- IIR filter, 427
- image interleaving, 650
- image object, 510
- image objects
 - alpha blending, 651
 - channels, 650
 - creating, 650
 - interleave property, 653
 - palette, 653
 - pixel interleaving, 650
 - saving as MPEG file, 656
 - saving to a file, 656
 - using, 651
- images
 - combining with 3D graphics, 327
 - determining file type, 214
 - dialog for reading, 211
 - dialog for saving, 212
 - displaying, 375
 - file selection
 - compound widget, 219
 - using a dialog, 217
 - macros for importing, 187
 - orientation, 375
 - overlying with contour plots, 301
 - overview, 373
 - position in display, 376
 - processing, 372, 374
 - query file type, 214, 215
 - querying, 214
 - raster, 373
- images (*continued*)
 - read routines, 215
 - reading from display, 379, 391
 - routines, 374
 - scaling, 377
 - size of display, 376
 - writing routines, 216
- import macro
 - ASCII files, 191
 - binary files, 197
 - image files, 187
 - scientific data formats, 203
- include files *see* batch files
- indexed color model, 567, 568
- infinite impulse response filters, 420
- initialization of objects, 519
- instancing
 - back-face culling, 717
 - interactive graphics, 682
 - lighting, 718
 - redraw performance, 716
 - window objects, 682
- integration
 - bivariate functions, 466
 - discussion, 465
 - numerical, 465
 - routines, 469
 - trivariate functions, 467
- interactive, object instancing, 682
- interleaving, 391
 - image, 650
 - image objects, 650
 - line, 650
 - pixel, 650
 - planar, 650
 - scanline, 650
- interpolation
 - routines, 460
 - tabulated data points, 459
- interpolation of voxel values, 664

interrupting
 program execution, 168
 variable context, 169
 isosurfaces, overview, 332

J

journaling, 180
 Julian dates and times, 276, 599
 Julian dates/time, calendar conversion, 276,
 599

K

Kaiser filter, 421
 keyboard
 interrupt, 168
 shortcuts in IDLDE, 69

L

Lambert's conformal conic map projection,
 361
 Lambert's equal area map projection, 352
 LangCat object, 517
 launching IDL, 20
 layout
 graphics preferences, 103
 multiple windows, 101
 leakage, 407
 legend object, 513, 630
 life-cycle methods, 507
 light object, 510
 light objects
 creating, 643
 overview, 643
 types of lights, 643
 using, 644
 light source
 changing parameters, 330

light source (*continued*)
 shading, 329
 lighting, 663, 718
 lightness, 383
 line
 editing, IDL command line, 86
 line interleaving, 650
 linear
 algebra, 446
 correlation, 446
 systems
 condition number, 470
 overdetermined, 471
 solving simultaneous equations, 470
 underdetermined, 473
 linear equations, simultaneous, 476
 linear systems, routines, 476
 lines
 limit in Output Log, 98
 Output Log, 98
 saved in recall buffer, 98
 LM_LICENSE_FILE variable, 26
 location, object graphics to view area, 542
 location of text, 611
 logarithmic
 plots, 595
 scaling, 269
 lowpass filters, 421

M

MA filter, 420
 Macintosh mouse, differences, 32
 macro
 importing
 ASCII files, 191
 binary data, 197
 HDF files, 203
 HDF-EOS files, 203
 image files, 187
 NETCDF files, 203

- macros
 - creating in UNIX, 118
 - creating in Windows, 121
 - working with, 115
 - macros, sharing, 97
 - magnitude, signal spectra, 405
 - main IDL directory, changing, 108
 - map projections
 - Aitoff, 352
 - Albers equal-area conic, 362
 - azimuthal, 348
 - azimuthal equidistant, 351
 - boundaries, specifying, 341
 - central gnomonic, 350
 - cylindrical, 358
 - cylindrical equidistant, 360
 - drawing continent boundaries, 341
 - general perspective, 354
 - gnomonic, 350
 - Hammer-Aitoff, 353
 - high-resolution outlines, 368
 - Lambert's conformal conic, 361
 - Lambert's equal area, 352
 - Mercator, 358
 - Miller cylindrical, 361
 - Mollweide, 364
 - orthographic, 348
 - overview, 338
 - pseudocylindrical, 363
 - Robinson, 363
 - satellite, 354
 - sinusoidal, 363
 - stereographic, 349
 - Transverse Mercator, 359
 - warping images to maps, 366
 - mathematics
 - error assessment routines, 439
 - routines, 436
 - maximum intensity projection, 664
 - maximum size, drawable window, 679
 - maximum values in plots, 627
 - memory
 - graphics system use, 503
 - object graphics system, 242
 - optimizing Windows performance, 98
 - Menu Editor, opening, 54
 - Mercator map projection, 358
 - meridians, 341
 - graticules, 343
 - gridding, 343
 - Microsoft Windows, mouse differences, 32
 - Miller cylindrical map projection, 361
 - minimization, 480
 - See also* optimization
 - minimum values in plots, 627
 - missing data, 275
 - model class, methods, 552
 - model object, 509, 534
 - model objects, selecting models, 670
 - Mollweide map projection, 364
 - Motif widgets, 153
 - mouse
 - determining position, 291
 - emulating three-button, 32
 - mouse cursor, 681
 - moving average filter, 420
 - MPEG object, 517
 - MPEG objects, overview, 656
 - MrSID object, 517
 - multiple correlation coefficient, 447
 - Multiple Document Panel, 45
 - multiple plots on a page, 271
 - multiplying arrays, 442
 - multi-threading
 - environment variable, 24
 - setting default threads, 24
 - multivariate analysis, routines, 493
- N**
- near and far clipping planes, 549
 - NETCDF files, importing using macros, 203

new page, 693
 Newton's method, 478
 non-interactive mode, 176
 nonlinear equations
 discussion, 478
 routines, 479
 nonparametric hypothesis tests, 462
 NORM_COORD function, 556
 normal, coordinates, 246
 normal computations, 624, 714
 notch filter, 427
 numerical integration, 465
 Numerical Recipes in C, 437
 Nyquist frequency, 413

O

object classes
 attribute objects, 536
 attributes, 514, 579
 axis, 510, 593
 buffer, 516, 684
 clipboard, 516
 colorbar, 513
 common methods
 Cleanup, 507
 GetProperty, 507
 Init, 507
 SetProperty, 507
 composite objects, 513
 container, 508
 contour, 510, 616
 destination objects, 516
 DICOM, 517
 DXF, 517
 file format objects, 517, 517, 517, 517, 517,
 517, 518, 518
 font, 514, 579
 helper objects, 515, 536
 IDLffJPEG2000, 517
 image, 510
 object classes (*continued*)
 LangCat, 517
 legend, 513, 630
 light, 510, 643
 model, 509, 534
 Rotate method, 553
 Scale method, 554
 Translate method, 552
 MPEG, 517
 MrSID, 517
 naming conventions, 506
 palette, 514, 582
 pattern, 514, 583
 plot, 510, 626
 polygon, 511, 619, 711
 polyline, 511, 625
 printer, 516, 687
 scene, 509, 531
 Shape, 518
 structure, 509
 surface, 511, 638
 symbol, 514, 585
 tessellator, 515, 589
 text, 511, 611
 TrackBall, 515
 undocumented, 527
 view, 509, 533
 viewgroup, 509, 532
 volume, 512
 VRML, 516, 518, 694
 window, 516
 XMLSAX, 518
 Object Graphics, 502
 choosing a renderer, 104
 classes, 499
 clipboard support, 46
 composite classes, 706
 device independent, 503
 hierarchy, 509
 instancing, 716
 printing, 129

Object Graphics (*continued*)
 typographical conventions used, 506
 object graphics
 expose events, 715
 indexed color model, 567
 performance tuning, 711
 setting properties, 519
 object tree, 509, 530
 object-oriented graphics, 503
 objects
 Object Graphics, 240
 clipboard support, 46
 object-oriented, graphics, 242
 Oetli, Thomas, 368
 one-tailed hypothesis tests, 462
 on-the-glass text, 612
 opacity table, 662
 OpenGL rendering, setting, 104
 operating system, issuing commands, 92
 optimization
 discussion, 480
 routines, 481
 orientation of text objects, 614
 orthographic map projection, 348
 outer product, 444
 Output Log
 platform specifics, 46
 preferences, 98
 overlaying images, 301

P

palette, indexed color data, 653
 palette object, 514, 582
 palette objects, 571
 parallel projection, 545
 parallels, 341
 parametric hypothesis tests, 462
 partial correlation coefficient, 448
 path, IDLDE, 112
 path caching, 25

PATH environment variable, 21
 pattern filling of polygon objects, 620
 pattern object, 514, 583
 performance
 improvement, 104
 optimizing memory, 98
 performance tuning, 711
 perspective projection, 546
 phase, signal spectra, 405
 pixel interleaving, 650
 pixels
 fixed, 303
 scalable, 301
 scaling, 377
 two-dimensional image arrays, 373
 pixmap objects, using, 680
 planar interleaving, 650
 plot
 maximum data values, 627
 minimum data values, 627
 plot object, 510, 626
 plot objects
 averaging points, 628
 minimum and maximum values, 627
 plotting symbols, 628
 plots, 615
 contour, 295
 filled contour, 308
 frequency smearing, 407
 logarithmic, 269, 595
 overplotting, 253
 shaded surface, 329
 step, 398
 X versus Y, 248
 plotting
 annotation, 255
 axes, 287
 combining images with graphics, 327
 date/time data, 276
 font selection, 256
 histogram style, 260

- plotting (*continued*)
 - Julian date/time, 276, 599
 - keyword parameters, 245
 - location on page, 273
 - missing data, 275
 - multi-dimensional arrays, 294
 - multiple plots on a page, 271
 - object graphics example, 633
 - overplotting, 253
 - overview, 244
 - polar coordinates, 289
 - reverse axis, Object Graphics, 609
 - scaling axes, 249
 - surfaces, 310
 - symbols, 257, 258
 - titles, 250
 - two-dimensional arrays, 294
- polar plots, 289
- polygon filling, 261
- polygon mesh optimization, 711
- polygon object, 511, 619, 711
- polygon objects
 - back-face culling, 717
 - normal computations, 714
 - optimization methods, 711, 712, 713
 - quad strip mesh, 711
 - triangle fan mesh, 712
 - triangle strip mesh, 713
- polygons, 589
- polyline object, 511, 625
- position of graphics, 542
- positioning objects, 542
- positioning text objects, 611
- power spectrum, 408
- preferences
 - change directories, 98
 - changing, 93
 - exit, 97
 - read-only files, 98
 - sharing, 97
 - startup, 97
- principal components analysis, 489
- print manager, 129, 131
- printer object, 516, 687
- printer objects
 - color model, 687
 - creating, 687
 - dialogs, 687
 - drawing, 688
 - starting a new page, 693
 - submitting print jobs, 693
- Printing
 - IDL for Windows, 130
 - IDL UNIX, 131
- printing
 - from IDLDE, 51
 - graphics, 129
- Printing in IDL, 128
- printing, direct graphics, 129
- pro files
 - double-clicking, 160
 - dragging and dropping on IDLDE, 160
- program files
 - executing, 165
 - interrupting execution, 168
- programs
 - creating .sav files, 171
 - restoring, 171
- Project Window, overview, 45
- projection
 - overview, 545
 - parallel, 545
 - perspective, 546
- projection matrix, 471
- projections
 - Aitoff, 352
 - Albers equal-area conic, 362
 - azimuthal, 348
 - azimuthal equidistant, 351
 - central gnomonic, 350
 - cylindrical, 358
 - cylindrical equidistant, 360

projections (*continued*)
 general perspective, 354
 gnomonic, 350
 Hammer-Aitoff, 353
 high-resolution continent outlines, 368
 Lambert's conformal conic, 361
 Lambert's equal area, 352
 Mercator, 358
 Miller cylindrical, 361
 Mollweide, 364
 orthographic, 348
 pseudocylindrical, 363
 Robinson, 363
 satellite, 354
 sinusoidal, 363
 stereographic, 349
 Transverse Mercator, 359

properties
 retrieving, 520
 setting, 519

Properties dialog, opening, 54

properties of objects, 519

pseudocylindrical map projections, 363

Q

quad strip optimization, 711

quadrature function, 415

query, image file type, 215

querying, images, 214

R

raster images, 373

reading
 ASCII data, 191, 223
 binary data, 197, 231
 data using macros, 187, 191, 197
 HDF files, 203
 HDF-EOS files, 203

reading (*continued*)
 image file types, 215
 image files, 187, 211
 images from the display, 379
 NETCDF files, 203
 scientific format data, 203

recall buffer
 changing, 89
 preferences, 98

recent files list, 52

recent projects, 52

rectangular, filter, 425

rendering
 hardware versus software, 104, 678
 polygon objects, 619
 polyline objects, 625
 speed of volumes, 665
 surface objects, 639

rendering process, rendering objects, 537

rendering style, polygon objects, 619

replacing text, 56

reserving colors, 146

resolution of map databases, 368

resource files, 144

resources for an X Window, 144

restoring .sav files, 171

restoring windows, 681

retained graphics, 715

retrieving object properties, 520

revealing window objects, 680

reverse axis, Object Graphics, 609

RGB, interleaving, 391

RGB color model, 567, 569

RGB color system, 381

right-handed coordinate system, 313

Robinson map projection, 363

Rotate method, 553

rotating
 arrays, 315
 images, 315
 views, 321

rotation, 552, 553

routines

- cluster analysis, 493
- correlation, 449
- curve and surface fitting, 451
- differentiation/integration, 469
- eigenvalues/eigenvectors, 458
- gridding/interpolation, 460
- hypothesis testing, 464
- linear systems, 476
- mathematical error assessment, 439
- multivariate analysis, 493
- nonlinear equations, 479
- optimization, 481
- signal processing, 431
- sparse arrays, 484
- time-series analysis, 487

row-indexed sparse storage method, 482

S

sampled data analysis, 413

sampled images, 373

sampling, aliasing data, 413

satellite map projection, 354

saturation, 383

SAVE and RESTORE, 171

saving, image files, 212

saving files, 50

saving windows, 681

scalable pixels, 301

Scale method, 554

scaling, 554, 554

- axes, 249
- coordinate systems, 542
- graphics atoms, 552
- images, 377
- maps, 341
- matrices, 314
- pixels, 377

scanline interleaving, 650

scene object, 509

scene objects, 531

scientific data format, importing using macros, 203

search path, specifying with preferences, 112

seasonal effect, 485

selecting

- graphic atoms, 669
- views in a window object, 669

selection

- model objects, 670
- window objects, 669

setting, window object cursor, 681

setting properties

- existing objects, 519
- initialization, 519
- objects, 519

shaded surfaces, 329

- plotting, 329

shading

- Gouraud interpolation, 329
- light source, 329
- parameters, 330
- polylines, 625

shading polygon objects, 620

Shape object, 518

shared color maps (Motif), 146

shortcuts, keyboard, 69

signal

- analysis transforms, 400
- processing, 397

signal processing routines, 431

simple polygons, 589

simultaneous linear equations, 470

singular value decomposition, 470

sinusoidal map projection, 363

sizing graphics windows, 103

skirts, 641

slicing volumes, 334

smearing frequency plots, 407

SMOOTH function, 425

- smoothing contours, [307](#)
 - software rendering, setting preference for, [104](#)
 - sparse arrays, [482](#)
 - routines, [484](#)
 - splash screen preference, [97](#)
 - standardized variables, [490](#)
 - starting IDL, [20](#)
 - startup
 - directory, [108](#)
 - working directory, [108](#)
 - startup file
 - batch file execution, [109](#)
 - overview, [30](#)
 - startup preferences
 - options, [108](#)
 - specifying, [97](#)
 - stationary series, [485](#)
 - statistics
 - hypothesis testing, [462](#)
 - routines, [436](#)
 - Status Bar, IDLDE, [47](#)
 - step plot, [398](#)
 - stereographic map projection, [349](#)
 - stopping program execution, overview, [168](#)
 - structure objects, [509](#)
 - structures, arrays stored in structure form, [482](#)
 - submitting a print job, [693](#)
 - surface fitting
 - discussion, [450](#)
 - routines, [451](#)
 - surface object, [511](#)
 - overview, [638](#)
 - surface objects
 - creating, [638](#)
 - hidden line removal, [641](#)
 - interactive example, [646](#)
 - rendering style, [639](#)
 - shading, [640](#)
 - skirts, [641](#)
 - texture mapping, [642](#)
 - using, [639](#)
 - surfaces, plotting, [310](#)
 - switches, command line, [27](#)
 - symbol object, [514](#), [585](#)
 - symbol use for polylines, [625](#)
 - symbols, plots, [257](#), [258](#)
 - symmetric arrays, [442](#)
 - system buffered backing store, [104](#)
 - system variables
 - !EDIT_INPUT, [88](#)
 - !MAP, [340](#)
 - !ORDER, [376](#)
- ## T
- TERM environment variable, [26](#)
 - tessellator object, [515](#), [589](#)
 - text
 - replacing, [56](#)
 - searching in IDLDE, [55](#)
 - selecting in IDL editor (Windows), [78](#)
 - text object
 - atomic object, [511](#)
 - creating and using, [611](#)
 - editing output, [699](#)
 - texture mapping, [642](#)
 - texture mapping polygon objects, [620](#)
 - three-dimensional
 - coordinate conversion, [318](#)
 - graphics, [313](#)
 - transformation, [323](#)
 - matrices, [313](#)
 - tick labels, [609](#)
 - tick marks, [265](#)
 - TIMEGEN, [277](#), [600](#)
 - time-series analysis, [485](#)
 - routines, [487](#)
 - titles, multiline on plots, [250](#)
 - toolbars
 - IDLDE, [44](#)
 - Motif platform, [48](#), [150](#)
 - show/hide preference, [102](#)

toolbars (*continued*)
 specifying layout, 102
 TrackBall object, 515
 transformation matrices, 313
 transformations, 554
 combining, 554
 coordinate, 555
 model class example, 557, 560
 model objects, 552
 rotation, 552, 553
 scaling, 552
 translation, 552, 552
 transforms
 Fourier, 401
 Hilbert, 415
 Tustin bilinear, 427
 wavelet, 417
 Translate method, 552
 translation, 314, 552, 552
 transparency
 alpha channel, 650
 in vector graphics, 697
 transparency of voxels, 662
 Transverse Mercator map projection, 359
 trend analysis, 485
 triangle fan mesh, 712
 triangle strip mesh, 713
 triangulation
 drawing fonts, 104
 TrueType fonts, 104
 true map scale, 341
 true-color displays, 389
 TrueType fonts, 579
 graphic preferences, 104
 Tustin transform, 427
 two-tailed hypothesis tests, 462
 typographical conventions, 40, 506

U

unconstrained minimizer, 480
 undocumented object classes, 527
 unsharp masking, 378
 upward direction of text objects, 614
 using
 colorbar objects, 654
 image objects, 651
 pixmap objects, 680
 volume objects, 660
 window objects, 677, 680
 UTM map projection, 359

V

Variable Watch Window, IDLDE, 47
 variables
 derived, 489
 standardized, 489
 vector graphic, smooth shading, 698
 vector graphics
 defined, 696
 display results, 698
 IDLgrClipboard, 685
 IDLgrPrinter, 687
 inserting EMF file, 696
 object sorting, 700
 object sorting issues
 IDLgrImage objects, 703
 transparent views, 701
 text rendering, 696, 699
 transparency, 697
 versus bitmap, 695
 when to use, 697
 vectors, multiplying, 444
 view area, 542
 view object, 509, 533
 view volume
 example programs, 550
 finding, 550
 overview, 549
 viewplane rectangle, 549
 viewgroup object, 509, 532

viewplane rectangle, [549](#), [557](#)
 viewport, [542](#), [543](#)
 volume, visualization, [332](#), [334](#)
 volume object, [512](#)
 volume objects

- attributes, [662](#)
- color values, [663](#)
- compositing, [664](#)
- creating, [660](#)
- interpolating values, [664](#)
- lighting, [663](#)
- opacity table, [662](#)
- overview, [660](#)
- rendering speed, [665](#)
- using, [660](#)
- zbuffering, [664](#)

voxel transparency, [662](#)
 voxel values, [660](#)
 VRML object, [516](#), [518](#)
 VRML objects

- creating, [694](#)
- overview, [694](#)

W

wavelet transform, [417](#)
 window object, [516](#)
 window objects

- color model, [677](#)
- creating, [677](#)
- draw widgets, [677](#)
- erasing, [680](#)
- exposing, [680](#)
- hiding, [680](#)
- iconifying, [680](#)
- instancing, [682](#)
- maximum size, [679](#)
- restoring, [681](#)
- saving, [681](#)
- selection, [669](#)
- setting the cursor, [681](#)

window objects (*continued*)

- using, [677](#), [680](#)

windowing, Hamming windowed signal, [411](#)
 windowing algorithm, HANNING function, [409](#)
 windows

- arranging layout, [103](#)
- clipboard support for graphics, [46](#)
- separating the IDLDE, [101](#)
- show/hide preference
 - Microsoft Windows platform, [100](#)
 - Motif platform, [101](#)

working directory, changing on startup, [108](#)
 writing

- image files, [212](#)
- routines for image files, [216](#)

X

X - Y Plots, [248](#)
 X resources, using, [144](#)
 XMLSAX object, [518](#)
 Xprinter

- defined, [131](#)
- printing graphics, [129](#)
- vector graphics, [697](#)

Z

zbuffering, [664](#)