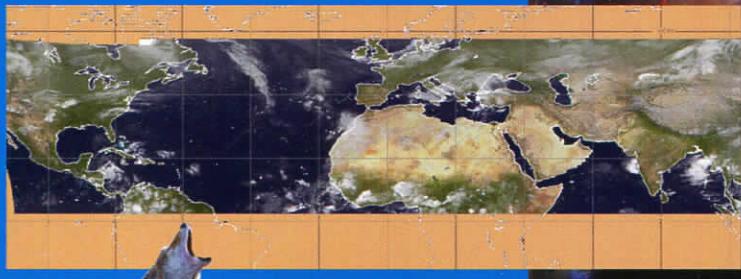
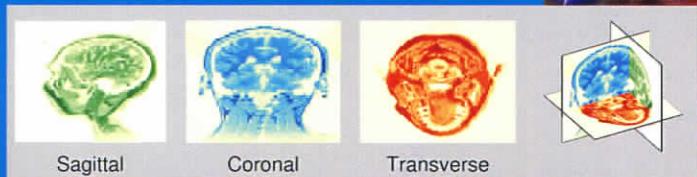
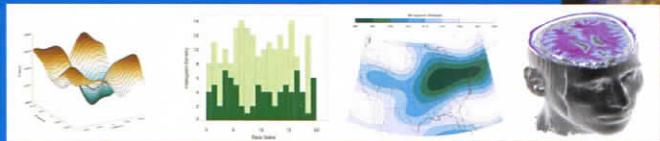
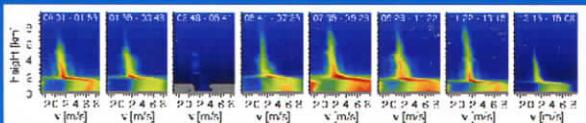


Coyote's Guide to:

TRADITIONAL IDL GRAPHICS

Using Familiar Tools Creatively



David W. Fanning

COYOTE'S GUIDE TO:

TRADITIONAL
IDL GRAPHICS

DAVID W. FANNING

Coyote's Guide To:

Traditional IDL Graphics: Using Familar Tools Creatively

David W. Fanning

Copyright © 2011 Coyote Book Publishing

Coyote Book Publishing
1645 Sheely Drive
Fort Collins, CO 80526
Phone: 970-221-0438
Fax: 970-221-4762
Toll-Free: 1-888-461-0155
E-Mail: books@idlcoyote.com
Web Page: www.idlcoyote.com

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Notice of Liability: The information in this book is distributed on an "as is" basis, without warranty. While every precation has been taken in the preparation of this book, neither the author nor the publisher shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this book or by the computer software described in it.

IDL is a registered trademark of ITT Visual Solutions, Inc.
Macintosh is a registered trademark of Apple Computer, Inc.
Windows is a registered trademark of Microsoft Corporation.
UNIX is a registered trademark of UNIX Systems Laboratories.

ISBN 13: 978-0-966-23835-8
ISBN 10: 0-966-23835-4
March 2011

Printed in the United States of America.

*To Tony Kehoe, who attended the first IDL course
I ever taught, and whose Coyote spirit kept me
laughing forever after. Vaya con Dios, amigo.*

Contents



Table of Contents	v
Foreword	xv
Preface	xvii
Cover Photo and Image Credits	xix
Acknowledgements	xxi
Chapter 1: Traditional Graphics	1
What Is the Traditional Graphics System? 1	
Philosophy Behind this Book 3	
Using this Book 3	
Version of IDL Required 4	
IDL Programs Required 4	
Data Files Required 7	
Style Conventions 7	
Obtaining Help 9	
Working with IDL Commands 9	
Anatomy of an IDL Command 10	
Help with IDL Commands 13	
Error Messages 14	
Creating Command Journals 15	

Chapter 2: Working With Graphics Windows17

Creating Graphics Windows	17
The Current Graphics Window	18
Setting the Current Graphics Window	19
Deleting Graphics Windows	19
Positioning and Sizing Graphics Windows	20
Viewing a Hidden Graphics Window	22
Putting a Title on a Graphics Window	23
Erasing a Graphics Window	23
Creating a “Maximum Size” Window	24
Determining the Size of the Current Graphics Window	24
Determining the Size of the Display Monitor	25
Size Tools Are Not Platform Independent	26
Device Independent Graphics Windows	28
Resizeable Graphics Windows	29

Chapter 3: Working With Color33

Color History and Transition	33
Vital Information for UNIX Users	34
Use a TrueColor Visual Class	35
Understanding IDL Color Models	36
Colors Are Represented as Color Triples	36
Specifying Colors by Name	41
Working Around a Color Bug on UNIX Machines	46
Color Models Affect Image Display	47
Setting the Color Model	51
Obtaining the Color Model	52
Loading Color Tables	53
Loading Your Own Color Tables	55
Creating Your Own Color Tables	56
Saving a Color Table	61

Using Other Color Systems	63
HLS Color System	64
HSV Color System	65
Chapter 4: Creating Line Plots	67
A Basic Line Plot	67
Line Plot Colors	67
How IDL Chooses Drawing Colors	68
Color Keyword Equivalents to System Variables	72
Specifying Data Colors	73
Plotting Data with Line Plots	74
Annotating Line Plots	77
Line Plot Titles	77
Creating Your Own Axis Labels	79
Formatting Axis Labels	80
Suppressing Axis Labels	81
Using Calendar Dates as Axis Labels	83
Using Irregular Tick Spacing and Values	83
Using Symbols and Line Styles	83
Creating Your Own Plot Symbols	86
Overplotting Additional Data Vectors	89
Positioning Line Plots	91
Creating Multiple Line Plots	93
Customizing Line Plot Axes	99
Axis Range Scaling	99
Suppressing Axes	102
Using Logarithmic Axes	103
Adding Additional Axes to a Line Plot	105
Adding Other Annotation to Line Plots	106
Understanding Coordinate Systems	108
Selecting Label Points Interactively	109
Creating a Histogram Line Plot	110

Handling Missing Data in a Line Plot	113
Using a Refurbished Line Plot Command	116
Line Plots in Resizeable Graphics Windows	120
Chapter 5: Creating Contour Plots	121
Basic Contour Plots	121
Advantages of Keywords versus System Variables	125
Customizing Contour Plots	127
Selecting Contour Levels	129
Modifying Contour Lines	132
Adding Color to Contour Plots	134
Using Colors in Contour Plots	136
Creating Color Filled Contour Plots	138
Choosing a Different Fill Algorithm	145
Contouring Irregularly Sampled Data	147
Gridding Irregular Data	155
Contouring a Real World Example	156
Using a Refurbished Contour Command	162
Labeling Contour Intervals	166
Using Colors in Contour Plots	166
Contour Plots in Resizeable Graphics Windows	167
Chapter 6: Creating Surface Plots	171
Basic Surface Plots	171
The Surface Command Is Not True 3D	174
Customizing Surface Plots	177
Adding Color to Surfaces	180
Creating Shaded Surface Plots	187
Shading With Another Data Set	192
Setting Up a 3D Scatter Plot	194
Surfaces in PostScript Output	197

Using a Refurbished Surface Command	198
Surface Plots in Resizeable Graphics Windows	201
A True 3D Surface Command	203
Adding Surface Texture Maps	205
Chapter 7: Creating Image Plots	209
Reading Image Data	209
Information About Image Files	213
Reading Images In Other File Formats	214
Using Traditional Image Display Commands	218
Preparing Image Data for Display	219
Using the BytScl Command	220
Is the Image Upside-Down?	226
Displaying Images in Color	228
Resizing and Positioning Images	232
Precise Resizing of Images	238
Resizing Images in PostScript	240
Alternative Image Display Commands	241
Positioning and Resizing Images	243
Zooming Images	252
Displaying Multiple Images	254
Transparent Images	257
Creating Blended Images	260
Creating Transparent Images	261
Images in Resizeable Graphics Windows	265
Chapter 8: Image Processing	267
Basic Image Processing	267
Contrast Enhancement	267
Window Leveling	269
Histogram Equalization	270
Histogram Matching	273

Additional Methods of Contrast Stretching	276
Differential Image Scaling	278
Image Partitioning	280
Image Filtering	285
Image Smoothing	285
Filtering with Image Convolution	289
High-Pass Filtering	290
Image Sharpening	291
Image Filtering in the Frequency Domain	296
Image Edge Enhancement	303
Image Noise Reduction	304
Noise Reduction in the Frequency Domain	307
Regions of Interest in Images	310
Image Preparation	312
Image Thresholding	312
Using Morphological Operators	313
Identifying Image Features	317
Processing the Image ROI	319
Creating an Image Mask	323
Using the Blob Analyzer	323
Principal Component Analysis of Images	326
Chapter 9: Creating PostScript Output	333
Why PostScript Output is Essential	333
How Does the PostScript Device Work?	334
Configuring the PostScript Device	337
Producing Color (or Grayscale) Output	340
Setting Up the PostScript “Window”	345
Producing Encapsulated PostScript Output	350
Setting the Language Level	352
Using PostScript Fonts	352
Configuring the PostScript Device Interactively	365
Displaying Images in PostScript	371
Sizing PostScript Images Correctly	373

Displaying Color Images in PostScript	375
Writing Device Independent Programs	377
Protecting Window Commands in PostScript	378
Work in Decomposed Color Space	381
Position Graphics in Normalized or Data Coordinates	382
Printing PostScript Files	382
Printing from UNIX Computers	383
Printing from Windows Computers	383
Printing from Macintosh Computers	384
Converting PostScript for Non-PostScript Printers	384
Viewing PostScript Files	384
Chapter 10: The Z-Graphics Buffer	387
Purpose of the Z-Graphics Buffer	387
Configuring the Z-Graphics Device	388
The Depth Buffer Versus the Frame Buffer	390
Character Size in the Z-Graphics Device	396
Warping Images in the Z-Buffer	398
Transparency Effects in the Z-Buffer	401
Chapter 11: Creating Raster Output	405
Presentation Quality by Leveraging PostScript	405
Creating Raster Output in IDL	406
Creating Raster Files	407
Color Vectors in Raster Output	407
All-in-One Raster Output Command	409
Using the Z-Buffer for Raster Output	409
Using ImageMagick for Raster Output	411
The Role of PS_Start	412
The Role of PS_End	414
Chapter 12: The Coyote Graphics System	417
Do We Need Another Graphics System?	417

Using Resizeable Graphics Windows	418
How Does a Resizeable Graphics Window Work?	419
Working with Multiple Resizeable Windows	427
Changing Colors in Resizeable Windows	431
Using Command Delay	433
Saving and Restoring Window Visualizations	433
Configuring Resizeable Window Properties	435
Window Properties	435
PostScript Properties	438
ImageMagick Properties	440
Using Coyote Graphics Commands	441
Preparing Commands for Resizeable Windows	442
Multiple Plot Layouts with Coyote Commands	445
Controlling Window Output Programmatically	449
Index	451

Foreword



Foreword

Most IDL programmers are probably familiar with David Fanning in one form or another. They might know him as the author of *IDL Programming Techniques*, which remains a valuable resource 10 years after the publication of the second edition in late 2000. His first edition in 1997 was the first book ever published describing the IDL language. Or, they might know him as the curator of the *Coyote's Guide to IDL Programming* web site, with its vast collection of tools, tips, and programs for the IDL developer. They might recognize his name as the most frequent contributor to the Usenet newsgroup *comp.lang.idl-pwave*, where he typically leads the discussions to help both new and experienced users. Or, they might know David more personally, either as a student in one of his IDL programming classes, or through his consulting work in fields as diverse as medical imaging, astronomy, and meteorology.

This current book is both more and less ambitious than *IDL Programming Techniques*. It is less ambitious in that its scope is limited to traditional IDL graphics. But it is more ambitious in that this is not just another guide to IDL. The current book also shows how to use IDL programs from David's powerful Coyote Library to enhance the power and usability of traditional IDL graphics commands. The result is a modern graphics system that retains the simplicity familiar to scientists and programmers who use IDL.

Let's hope David doesn't wait another 10 years for the publication of his next IDL book!

Wayne Landsman
Curator, IDL Astronomy Library

Preface



Preface

About a year ago, I got it into my head to return to my roots. I thought it would be a good idea to take a month off this past summer and walk the length of Oregon on the Pacific Crest Trail, a 400 mile hike I first did in the summer of 1970. The way I imagined it, the hike would give me time to reflect on my life and give me a sense of direction for where I might go next. I told friends I was hoping for a life-changing epiphany of the sort I had experienced the first time I did the hike.

I spent a lot of time preparing for the walk, but too much of it was spent listening to music from the 60s and 70s and too little of it was spent getting into shape. So, in the end, the hike was much harder than I expected it to be. About the only thing on my mind most days were murderous thoughts about the weight of my pack. It was all I could do to put one foot in front of the other to get my 20 miles a day covered. In the end, an illness brought the hike to a premature end and I never did get to have my epiphany. I thought the hike was a bust.

But while I was recovering from my illness, I got it into my head that what I *really* wanted to do (how odd!) was write a book about these old traditional IDL graphics routines. A sort of field guide of the kind I was using on the hike to find my way. It seemed then, and does now, a strange idea. Who would be interested? Who would pay to read such a book? Weren't the new function graphics routines in IDL 8 the cat's meow? Who would want to use anything else?

What I didn't plan on was how much I would learn returning to these graphics roots by writing this book, and how excited I would be by the discoveries I made. I guess a 25-year relationship with IDL is like any long marriage. It has its ups and downs, its routines, its bright moments, its

days of anguish and despair. But, I am as much in love with IDL right now, as excited about it, as I have been in the past 25 years. And that's saying something.

I believe the Coyote Graphics System that is outlined on these pages has the potential to change the way a lot of us write IDL programs. I won't say it is better than the IDL function graphics system. But I will say it is orders of magnitude simpler than that system, and I believe that fact alone will be a key to its success. Programmers, even novice programmers, will be able to build new graphics displays with this system that are impossible to build with the new IDL 8 function graphics system.

Maybe this book will accomplish nothing more than provide a bit of competition among graphics systems. That also would be a good thing for IDL users, I think. "The enemy, like the friend, makes you strong," as Barry Lopez says in his wonderful story, "Drought." In any case, the graphics system outlined here is available to any IDL user, using any version of IDL. You don't need the very latest gewgaw to do neat things in IDL. That alone, I think, makes this system interesting and worthwhile.

But, mostly, I think there is something wonderful about the feel of an old tool in a skillful hand. It makes you feel like you are building something worthwhile. Something lasting. Something that may, in fact, outlive even the *next* new graphics system. I hope you enjoy this journey, both backward and forward, as much as I have.

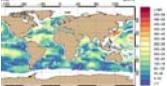
February 2011

Credits



Photo and Image Credits

	A NASA Hubble Space Telescope image of the star V838 Moncerotis, courtesy of NASA (www.nasa.gov) and the Space Telescope Science Institute (www.stsci.edu).
	A spectral image from radar analysis data courtesy of Bernat Puigdomènech Treserras, a software engineer from Montreal, Quebec, who offered this image in response to an appeal for images on the IDL newsgroup.
	Global mosaic of the Earth, courtesy of the Cooperative Institute for Meteorological Satellite Studies (CIMSS), University of Wisconsin – Madison, USA
	A visualization prepared by the author from data supplied by Stephanie Jenovrier, while a visiting scientist at the National Snow and Ice Data Center, Boulder, Colorado.
	A visualization prepared by the author from lidar data supplied by Shane Mayor and the Atmospheric Lidar Research Group, Dept. of Physics, California State University - Chico, Chico, California.

	A visualization prepared by the author from data provided by the Goddard Earth Sciences Data and Information Services Center, NASA Goddard Space Flight Center, Greenbelt, Maryland.
	Image of Coyote taken and supplied by Tom Davenport, owner of Prairie Photography, and outdoor photographer extraordinaire, from Hayden, Idaho.
	Image of the author taken by Susan Parker, Hilo, Hawaii, while the author was teaching IDL classes for astronomers on the Big Island.

Thanks



Acknowledgements

There are a great many people who help in writing a book. From the reviewers and readers who struggle to understand your unorthodox spelling and punctuation, to your family who bears the brunt of this strange obsession, to your tennis partners who put up with erratic play and lack of attention after a long day of writing. Everyone contributes. And, in this book, more people have contributed than ever before. Many won't get thanked personally here, but that is not because I value your contributions any less.

I am especially indebted to a long list of chapter reviewers: Ben Tupper, Roman Schreiber, Volker Tolls, Yang Wang, John Correira, Helmar Adler, Fabian Pfeifer, Louis Giglio, Hrafn Gudmundsson, Anne Martel, Brian Daniel, Robin Wilson, Matt Savoie, Glenn Campbell, Wayne Landsman, Karsten Rodenacker, and Kevin Jim all contributed by reading and commenting on various chapters, as well as testing code for me. One reviewer stands out for service above and beyond. Robert Dahni has the best eye for typos I've ever seen. He read the entire book and I have no doubt would do it again, if I asked him. He is tireless and a true friend with his red pencil. The remaining errors are mine alone.

I wish to thank Matt Savoie and Mary Jo Brodzik, who taught me more about the right way to write IDL programs than I would have imagined possible. It was a distinct pleasure to work with both of you.

Wayne Landsman not only agreed to write the Forward to the book, but he gave me many valuable suggestions and ideas as I was developing the Coyote Graphics programs. They are much better than they would have been without his help.

Robert Howard designed the book cover and became a friend who got me more and more excited about what I was trying to do with this book every time I talked to him over coffee at The Wild Boar.

My son, Brian, put some of that expensive MBA education to use advising me on the economic realities of the book business. Alas, too late, since I had most of the book written before I realized what a potentially disastrous career move I was making.

I am always indebted to the readers and contributors of the IDL newsgroup. They provide a sounding board for my ideas, both good and bad, and their good humor and respect as they ask and answer questions in that wonderful community is always a source of inspiration to me. I had the readers of the newsgroup in mind as I wrote this book.

Finally, I can't say enough about my wife, Carol, whose willingness to let me walk out the door on a 400 mile hike, quit my job, and spend five months pretending she didn't exist while I was writing this book brings tears to my eyes every time I think about it. This book, truly, would not exist without her love and support. I will be forever grateful the Universe saw fit to introduce me to that friendly, cheerful young woman in the raggedy bell-bottom jeans, while The Doors' "Light My Fire" rumbled in the very air around us.

Chapter 1



Traditional Graphics

What Is the Traditional Graphics System?

The traditional graphics system in IDL¹ goes by several other (sometimes disparaging) names: direct graphics, raster graphics, old graphics. It doesn't matter what you call it, the commands that make up the traditional graphics system have been an integral part of IDL from the beginning of time. They are not modern in any sense of the word. But, they are also used by nearly every IDL programmer, and it would be hard to find an IDL graphics program that didn't use them.

Traditional graphics commands are the “standard” commands of the IDL language: *Plot*, *Contour*, *Surface*, *TV*, and *Window*. They are not sophisticated, and have shortcomings galore, but in the right hands (and by this I mean almost anyone's hands) they can still—somehow—get the job done.

And this, more than anything else, is why they are not going away any time soon. People use them, and there is an enormous body of IDL code in the world that uses them, code that continues to serve the original purposes of its authors to explore and understand scientific data.

A traditional graphics command is an example of what we call a *raster graphic* command. That is to say, when the *Plot* command is used to create a line plot, the software algorithm underlying the *Plot* command calculates which pixels in the output display window must be turned a particular

1. IDL is short for Interactive Data Language and is a registered trademark and software product sold by ITT Visual Information Solutions of Boulder, Colorado, USA.

color and simply sets those pixels to the proper color. Such raster graphic output has no persistence or memory.

In other words, once an IDL traditional graphics command has been issued, neither IDL nor the graphics window has any knowledge of what has just happened. This means that IDL has no way to respond to common user interactions like resizing the graphics window. If you want to re-display a traditional graphics command when a graphics window is resized, the only way to do so is to execute the graphics command again so that it goes into the newly resized window.

Such non-persistent behavior was considered state-of-the-art in 1975, but is not really consistent with modern visualization software. Nevertheless, raster graphics commands are widely used in IDL because they are fast and simple, and because they are easy to use in computer programs written by scientists and engineers, rather than by computer professionals. They are not going to remind you of Fred Astaire, but in the right hands, even traditional graphics commands have been known to sing and dance a little. My goal for this book is to give you enough information to make them do so in *your* hands. I think you will be surprised at how “modern” these old, traditional commands can look with a little refurbishment.

I’m not a Luddite. After several false starts (e.g., Live Tools), I like the direction the creators of IDL are going in making their more modern and complete graphics system, called *function graphics*, easy for scientists and engineers to use. I sometimes use these new graphics routines myself, and I am happy to do so.

But the new function graphics commands are both complicated and powerful. They do what they do quite well, but it seems completely impossible to make them do something new, so they are less useful in some programming situations than traditional graphics commands. Also, in version 8 of IDL (the current version as I write this) they are not completely bug free, although they have become remarkably easier to use than any other object graphics incarnation so far. (Object graphics is an graphics system completely separate from the traditional graphics system, and is what the new function graphics system is built on.) They have a future, although their great promise has not been completely realized yet, in my opinion.

In the meantime, people are going to want to know how to write simple, effective IDL programs without taking weeks to learn the ins and outs of a new graphics system. They are going to want to know how to extend and maintain older software they inherit from colleagues. They are going to want to create graphics displays that are just not possible with the new

graphics routines. And some people are just slow to change. (Poll your older colleagues to see how many are still using the UNIX shell and programs they were introduced to as graduate students!) This, then, is a book written for those people. And, for people who just want to get on with their jobs without a lot of fuss and bother.

I've been working with IDL almost daily since 1987. For most of that time I have used nothing but traditional graphics commands, and still do to this day. What you can expect to find here, then, is what I think are the essential ideas for how to create IDL graphics programs from traditional graphics commands that work like they are suppose to work in today's modern computer environment.

Philosophy Behind this Book

This book grew out of nearly 25 years of teaching scientists and engineers to use and write programs in the IDL language. As I answered thousands of IDL questions, I realized most questions fall into a handful of broad categories. The truth is, most of us want to do pretty much the same things with IDL. We want to analyze and display our data, write efficient programs to solve our scientific problems, and—most of all—get our work done quickly. What most of us do not want to do is read computer software manuals.

IDL is a big software program that is getting bigger every day. It comes with an impressive amount of documentation, which no one I know wants to read. IDL can be intimidating even to experienced users, let alone to someone just starting out to learn its mysteries.

This book is meant to teach you what you absolutely need to know to work with IDL traditional graphics commands on a daily basis. And, more importantly, it is meant to do that with examples that are easy to understand and follow. More than anything, it is a book that shows you *how* to use traditional graphics commands in IDL. Essentially, this is the book I wish someone had written for me when I was starting to learn to use IDL graphics commands.

Using this Book

I've tried to make each chapter in the book self-contained so that you can pick the book up and turn to any chapter to learn what is most pressing for you. But I have also arranged the chapters in the book in more or less the

order I teach the material in IDL programming courses. If you are just starting to learn IDL, it probably makes sense to start at the beginning and work your way through the material in the order it is presented. The material in later chapters will build on concepts and techniques that were developed in earlier chapters.

Version of IDL Required

I used the latest version of IDL to develop the examples in this book (IDL 8.0 at the time of this writing), but I can't think of an example in the book that wouldn't work in any version of IDL going back to at least IDL 6.0 and maybe earlier. Don't be afraid to try the examples in any version of IDL. I assume you have a 24-bit graphics card in your computer, but this is the only concession to modernity I have made.

IDL Programs Required

My purpose is to teach you how to use IDL's traditional graphics commands. But that said, these traditional graphics commands are often inadequate for producing high quality graphical output, both on the display and in other formats. In nearly 25 years of working with IDL and teaching people how to use it, I have developed a library of IDL routines that often make life considerably easier for the IDL programmer.

This library is known as the [Coyote Library](#) and I describe and use programs from the library in this book unapologetically. When I do so, those programs will be indicated in the text by a blue typeface to distinguish them from the normal built-in or user-library commands that come with IDL. The graphics routines in the [Coyote Library](#) have come to be known as the Coyote Graphics System. We will use these routines throughout the course of the book, as appropriate, but the System itself, and the resizeable graphics window at the heart of the System, will be thoroughly explained in the chapter entitled "The Coyote Graphics System" on page 417.

I am going to assume that you have downloaded and installed the library on your machine, typically in a directory named *coyote*, and that the *coyote* directory has been placed somewhere on your IDL path. Occasionally, there are name conflicts with other programming libraries you may have installed. When you are working the examples in the book, you may find it useful to place the *coyote* directory first on your IDL path, so the [Coyote Library](#) programs are found before programs with the same names in other directories. (See "Help with IDL Commands" on page 13 for information on how to discover and resolve name conflicts.)

A copy of the [Coyote Library](#) as it existed when this book was published will be found on my web page. Directions for installing the library and adding it to your IDL path will be found in a README file in the same directory. Just open the file with your browser.

<http://www.idlcoyote.com/books/tg/lib/README.html>
http://www.idlcoyote.com/books/tg/lib/coyote_tg.zip

A program library, however, is a dynamic and changing thing. You may want to have the most current version of the [Coyote Library](#), with the latest bug fixes and new ideas. It can be obtained over the Internet in several ways. You can obtain it directly from my web page, [Coyote's Guide to IDL Programming](#).

<http://www.idlcoyote.com/programs/coyoteprograms.zip>

Or you can obtain it via a Subversion code repository as an open source project named *idl-coyote*. You will find directions for downloading the files there.

<http://idl-coyote.googlecode.com/>

You will need to have a Subversion version control system installed on your machine to use the repository. You can find more information about Subversion on this web page.

<http://subversion.apache.org/>

Install the library in a *coyote* sub-directory somewhere on your computer. Typical locations might be in your IDL workspace, in your home directory, or in any location that is convenient for you. It is probably not a good idea to install the *coyote* directory in the directories of the IDL distribution, as you will find this problematic when you update to a new version of IDL.

Add the *coyote* directory to your IDL path (i.e., the IDL system variable, *!Path*). This can be done in several ways. Typically it is done via the IDL Preferences in the IDL Workbench (see Figure 1) or it is done by setting up the IDL path in an IDL start-up file. If you are not sure how to do this for your particular IDL set-up, please read the article “Coyote Library Installation,” which you can find on my web page.

http://www.idlcoyote.com/code_tips/installcoyote.html.

To confirm that the *coyote* directory has been added to your IDL path, run this command at your IDL prompt.

```
IDL> Print, !Path
```

You should see the *coyote* directory listed among the directories in your IDL path.

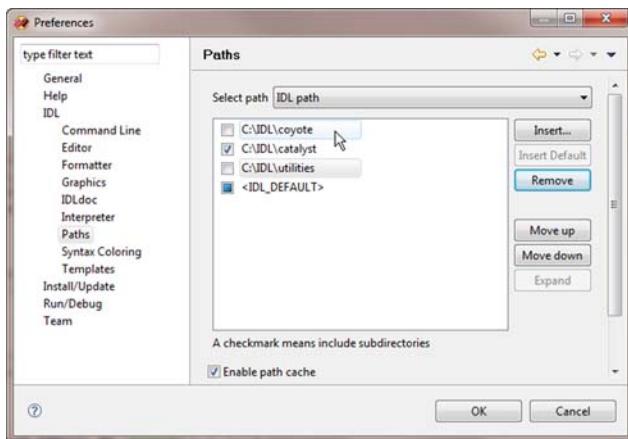


Figure 1: The *coyote* directory can be added to the IDL !Path variable using the Preferences Dialog in the IDL Workbench. It is best to add the *coyote* directory to the front of the IDL path to avoid potential name conflicts with other IDL libraries.

If it is not there, there is another method you can use to add it (or any directory) to your IDL path. From within IDL, change to the *coyote* directory (or whatever directory you want to add to the IDL path) with the *CD* command. Suppose, for example, you installed the *coyote* files in the */usr/idl/coyote* directory. You would type this command.

```
IDL> CD, '/usr/idl/coyote', Current=thisDir
```

Once in the *coyote* directory, execute the [AddToPath](#) command, which is a command from the [Coyote Library](#). The *CD* command after the [AddToPath](#) command will return you to the directory you were in before you switched to the *coyote* directory.

```
IDL> AddToPath
IDL> CD, thisDir
```

Now the *coyote* directory is the first directory on your IDL path. See all the directories on your IDL path by typing the [PrintPath](#) command.

```
IDL> PrintPath
```

You should see each directory on your IDL path printed to your command log, each on a separate line. The *coyote* directory should be the first directory in the list. If you like, you can put the *CD* and [AddToPath](#) commands in your IDL start-up file to execute every time you start IDL. The [AddTo-](#)

[Path](#) method adds the *coyote* directory to your IDL path for just this one particular IDL session, not permanently.

Data Files Required

Unless I specifically mention it, the data files used in this book are found in the normal IDL distribution, usually in the */examples/data* sub-directory of the main IDL directory. I will call this the “IDL examples directory” in the book. Normally, you will access these data files via a [Coyote Library](#) program named [cgDemoData](#) that has been written specifically for this purpose.

Other data files used in the book can be found on my web page in the book’s *data* directory.

<http://www.idlcoyote.com/books/tg/data/>

Style Conventions

I use a consistent style throughout the book so that I don’t confuse you about the function or purpose of the text. First, IDL commands that you should type are always set in Courier type face. Commands that you should type at the IDL command line are preceded by the IDL command prompt, IDL>, as in this example.

```
IDL> Surface, data
```

Some IDL commands are more conveniently typed in editor windows as main-level IDL programs. You can use the editor of your choice or the editor that is supplied with the IDL Development Environment (IDLDE) or Workbench. It is completely up to you how you choose to type and execute the commands. Nearly all the commands in this book can be typed at the IDL command line.

Capitalization

I use a particular style of capitalization for IDL commands in this book. This style is completely arbitrary. IDL is case insensitive, except for commands that interact with the operating system (e.g., the file names used in commands that open files will be case sensitive on UNIX machines) and when IDL is performing string comparisons. I capitalize the first letter of all IDL commands and keywords. In addition, any letter that may serve as a mnemonic is also capitalized. Here are some examples.

```
Surface, data, CharSize=2.0, Color=180
XLoadCT
Widget_Control, tlb, Set_UValue=info, /No_Copy
```

The only exception to this rule are Coyote Graphics System commands, which always start with the lowercase letters “cg” and are written in a blue typeface.

I do not capitalize the first letter of variable names, although I may capitalize subsequent letters in variable names that may be compound words. Here are some examples.

```
data = FindGen(11)
buttonValue = thisValue
ptrToData = Ptr_New()
```

I completely capitalize IDL reserved words. (Reserved words are words that IDL uses in various language constructs such as FOR loops, CASE statements, Boolean operations, etc. It is illegal in IDL to have a variable with the same name as a reserved word.) Here are some examples.

```
REPEAT test UNTIL
FOR j=0,10 DO BEGIN
ENDWHILE
```

You may use any capitalization you like when you type the commands at the IDL command line or into a text editor.

Comments

Anything to the right of a semi-colon in an IDL command is treated as a comment by IDL and is ignored by the IDL interpreter. In general, I try to write comments on their own line in an IDL program. Here is an example.

```
; This is the loop part of the program.
FOR j=0,10 DO BEGIN
    data = j*2
    count = count + j
ENDFOR
```

Occasionally, you will see a comment on the end of a command line. I do this especially when I am documenting the fields of an IDL structure variable, for example.

```
info = {r:r, $ ; The red color vector
        g:g, $ ; The green color vector
        b:b } ; The blue color vector
```

Line Continuation Characters

The line continuation character in IDL is the dollar sign, \$. This indicates to the command interpreter that the IDL command is continued on the next text line. (See the example above.) You will see a lot of line continuation characters in the IDL commands in this book. My advice is to ignore

the line continuation characters (leave them out) and just keeping typing the IDL command on the same command line. (Naturally, if the line gets too long, you can add a line continuation character whenever you like.) For example, you might type the command above like this.

```
IDL> info = { r:r, g:g, b:b }
```

This will make it much easier for you to re-type the command if you make a typing mistake or if you need to modify the command later.

Obtaining Help

If you have difficulty installing the program files or if you need help with some other aspect of IDL programming, you will find the *Coyote's Guide to IDL Programming* web page helpful. There you will find information about this book and about IDL programming in general. If worst comes to worst, you will also find a form there that will allow you to contact me directly. *Coyote's Guide to IDL Programming* can be found here.

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

If you have difficulty getting an example to work you can probably find the author or another IDL expert to help you on the IDL newsgroup (comp.lang.idl-pwave). This informal and fun newsgroup is an excellent source of help and information whenever you need it. You can access the IDL newsgroup with a variety of open-source or free software. Here is an article that explains what other IDL programmers use to access the newsgroup.

http://www.idlcoyote.com/misc_tips/readnews.html

Working with IDL Commands

This book is a doing book. I want you typing commands, making mistakes, seeing what happens. For this reason, the vast majority of the commands in this book are to be typed at the IDL command line. (If you want to keep a record of your commands as you type them, you can create a journal file to record them. See “Creating Command Journals” on page 15 for additional information.)

IDL has evolved tremendously as a programming language in the 20 plus years I’ve been working with it. But there is still a lot to be gained from learning how to use IDL from the command line. In particular, you learn to figure things out, to try things, to experiment with your data. I call it “learning by noodling around.” I think it is one of the best ways to work with IDL and one of its biggest advantages as a programming language.

Anatomy of an IDL Command

Here is what you need to know to get started. First, you will see a lot of commands like this in the book.

```
Contour, peak, lon, lat, XStyle=1, YStyle=1, /Follow, $  
    Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]
```

It helps if you know what you are looking at.

The word *Contour* in the command above is the name of the IDL command or program you want to run. It must be spelled out completely. Some command names can be quite long, but no shortcuts are allowed. The words *peak*, *lon*, and *lat* in this command are variables. They are also sometimes called command arguments or command parameters. They are used to pass information into or out of the command or program. The words *XStyle*, *YStyle*, *Follow*, *Levels*, and *C_Labels* are keywords or keyword parameters. Keywords are by convention optional parameters to the command. Like variables, they are used to pass information into and out of the command or program.

Positional Parameters

The three variables *peak*, *lon*, and *lat* in the command above are also called positional parameters. In this particular instance, these positional parameters are input variables (i.e., they are bringing data into the command), but you cannot tell by looking at them. They could just as easily be output variables. Or, they could be both input and output variables. The command line syntax is exactly the same. You will only be able to tell by context and by reading the published documentation for the command or program. All the IDL commands used in this book, other than the built-in IDL commands, contain documentation headers with information about the various parameters and how they are used in the command. Don't be afraid to open an IDL program in a text editor to learn more about the command.

A positional parameter has a defined sequence or order to the right of the command name. (Note that keyword parameters, discussed below, do not affect positional parameter order.) In this case, the variable *peak* must be to the right of the command *Contour* and to the left of the variable *lon*, for example. The variable *lon* must be to the right of *peak* and to the left of *lat*, and so on. You cannot leave out, say, the second positional parameter and specify the first and third.

For example, these two commands are incorrectly formatted and will cause errors. The first because the order of the positional parameters is

changed, and the second because the second positional parameter is not present.

```
Contour, lon, peak, lat, XStyle=1, YStyle=1, /Follow, $  
      Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]  
Contour, peak, , lat, XStyle=1, YStyle=1, /Follow, $  
      Levels=vals, C_Labels=[1,0,1,0,0,1,1,0]
```

Positional parameters are often required parameters to the command, but they do not have to be. For example, in the *Contour* command *peak* is a required parameter, but *lon* and *lat* are optional positional parameters. Again, you will know this by reading the published documentation for the command.

Keyword Parameters

XStyle, *YStyle*, *Follow*, *Levels*, and *C_Labels* are keyword parameters. Unlike positional parameters, keyword parameters are permitted in any order to the right of the command name. They can even come in the midst of positional parameters without affecting the relative positions of those parameters. In other words, keyword parameters are not counted like positional parameters. This is a valid construction of the *Contour* command above.

```
Contour, peak, Levels=vals, lon, XStyle=1, YStyle=1, $  
      /Follow, lat, C_Labels=[1,0,1,0,0,1,1,0]
```

By convention keyword parameters are optional parameters. Like positional parameters they can be input parameters or output parameters to the command. You will know by context and by reading the documentation for the command.

Notice the way in which the keywords are used in the command above. Keywords can be set to a particular value (e.g., *XStyle=1*), to a variable (e.g., *Levels=vals*), set with a slash character (e.g., */Follow*), or set to a vector of values (e.g., *C_Labels=[1,0,1,0,0,1,1,0]*).

Consider the */Keyword* syntax more closely. Some keywords have a binary quality. That is, they are either on/off, yes/no, true/false, 1/0, etc. These keywords are often “set” or “turned on” by the syntax */Keyword*. The syntax */Keyword* is identical to (means the same as) the syntax *Keyword=1*. No more and no less.

In fact, the *Contour* command above could have been written like this.

```
Contour, peak, Levels=vals, lon, /XStyle, /YStyle, $  
      /Follow, lat, C_Labels=[1,0,1,0,0,1,1,0]
```

This command means the same thing as the command above. The reason the command was not written like this, is that it might falsely imply that the *XStyle* and *YStyle* keywords have a binary quality (i.e., they are either on or off), which they don't. They can be set to other values besides 0 and 1.

IDL Procedures and Functions

This particular command, the *Contour* command, is an IDL procedure. IDL commands will be either procedures, like this one, or functions. Here is an example of an IDL command, the *BytScl* command, that is a function. Its purpose is to scale its arguments into byte type data.

```
scaled = BytScl(image, Top=199, Min=0, Max=maxValue)
```

Notice the difference between the *Contour* procedure and the *BytScl* function. First of all, in the function command the positional parameters and keywords are enclosed in parentheses. In the procedure command the parameters and keywords are simply listed on the command line. But the most important difference is that the function command explicitly returns a value, which is placed in a variable on the left-hand side of the equal sign. This is the fundamental difference between a function command and a procedure command in IDL.

Function commands always explicitly return a value that must be assigned to a variable (even if just a temporary variable). The return value of a function may be any kind of IDL variable, including scalars, vectors, and structures. In this case, the return value, *scaled*, is a byte array of the same dimensions as the *image* positional parameter.

Sometimes you will see a function command and a procedure command written together. For example, consider these two commands.

```
scaled = BytScl(image, Top=199, Min=0, Max=maxValue)
TV, scaled
```

The first command is a function command and the second command is a procedure command that uses as its positional parameter the return value of the function. It would not be unusual in IDL to see these two commands written like this.

```
TV, BytScl(image, Top=199, Min=0, Max=maxValue)
```

In this case, the *BytScl* command must be evaluated first and a value returned in a temporary variable. That return value (temporary variable) is then used as the positional parameter of the *TV* command.

It will probably take a while to become familiar enough with the various IDL commands so that you know immediately which is a procedure and

which is a function, but try to remember this: if you are looking for one value from a command, the command is probably a function. Most of the traditional graphics commands we will encounter in this book will be procedures, but not always.

Help with IDL Commands

IDL comes with an extensive on-line help system that can give you extremely helpful information about IDL commands and their parameters. On-line help is accessed by simply typing a question mark at the IDL command prompt or by selecting the Help menu item from the IDL Development Environment pull-down menu. Almost all the information in the IDL documentation set is available on-line. Check the *docs* sub-directory of the main IDL directory if you are having trouble locating what you need. To access the IDL on-line help system, simply type a question mark at the IDL prompt, like this.

```
IDL> ?
```

The IDL *Help* command is also available to obtain information about variables, your current graphics device, the amount of memory you have used, etc.

```
IDL> Help
IDL> Help, !D.Window
IDL> Help, !P, /Structure
IDL> Help, /Device
IDL> Help, /Memory
```

Another important use of the *Help* command, especially if the command you are using is not working properly, is to use it to identify which user-written command you are using. Sometimes you are not using the command you think you are using because some other command with the same name has been found first on your IDL path. You can use the *Source* keyword to list the source directory for all the compiled routines in your current IDL session.

```
IDL> Help, /Source
Compiled Procedures:
CGDEMODATA_CANCEL      C:\IDL\coyote\cgdeemodata.pro
CGDEMODATA_CENTERTLB   C:\IDL\coyote\cgdemodata.pro
CGDEMODATA_EVENT        C:\IDL\coyote\cgdemodata.pro
MAXMIN                  C:\IDL\coyote\maxmin.pro

Compiled Functions:
FILEPATH                 C:\Programs\IDL8\lib\filepath.pro
```

CGDEMODATA	C:\IDL\coyote\cgdemodata.pro
CGDEMODATA_READDATA	C:\IDL\coyote\cgdemodata.pro
MAKEY	C:\IDL\coyote\cgdemodata.pro
MAKEZ	C:\IDL\coyote\cgdemodata.pro
SMOOTH2	C:\IDL\coyote\cgdemodata.pro

By examining the list, you can determine if you are using the command you think you are using.

If you suspect a particular function is being found in the wrong place, you can use the *File_Which* command to locate the command file in your IDL path.

```
IDL> Print, File_Which('cgcolor.pro')
C:\IDL\coyote\cgcolor.pro
```

Error Messages

It seems self-evident, but error messages also provide a great deal of help to the IDL programmer. You ignore them at your peril, and yet I encounter a great many new IDL programmers who don't even read them! They not only tell you what the error is, but they tell you in what IDL program, and on what line in that program, the error occurred. They often show you the path your program took to arrive at the error. This is called a *traceback*. If you don't use this information, you can waste considerable time spinning your wheels and feeling frustrated.

Here is a typical error message.

```
% Variable is undefined: STARTDDIRECTORY.
% Execution halted at: SELECTIMAGE      1553
C:\IDL\catalyst\source\applications\selectimage.pro
%                                $MAIN$
```

The error message tells you the error occurred in the file *selectimage.pro*, and it gives you the absolute path to the file. The error occurred on line 1553 of that file. The error indicates that the variable *startDirectory* on that line is undefined. The error message doesn't tell you how to solve the problem. Rather, it gives you clues to solve the problem. (Oddly enough, there is a certain kind of person who finds these kinds of puzzles as soothing as Sudoku.)

In this example, you might ask yourself, "Why in the world would *that* variable be undefined *there!*" There can be any number of reasons, but clearly something you didn't anticipate has happened. Most errors fall into two broad categories: you either made an assumption about your program that simply isn't true, or you made a typing error. There may be other reasons

for programming errors, but I can't think what they are right now. The answer to this particular question might be found many lines away from where the error occurred, but at least you have a place to start looking for the source of the error, and you have a better idea of what it is you are looking for, if you read the error message.

Creating Command Journals

You may want to save a journal or record of the commands you type at the IDL command line. If so, you can create a journal file. A journal file is an IDL batch file. A batch file is a file that is executed in IDL as if you were typing the commands at the IDL command line. Typically, FOR loops and other multiple-line commands are not found in batch files. A journal file is opened in IDL by using the *Journal* command and specifying the name of the file you want to open. The file will be a new file, open for writing. There is no way to append to a journal file from the IDL command line. To open a journal file named, for example, *book_commands.pro*, type this.

```
IDL> Journal, 'book_commands.pro'
```

You do not have to include a file extension on Windows machines, where IDL will attach a *.pro* extension automatically. All subsequent commands that you enter at the IDL command line will go into this journal file. If you do not provide a file name, the default name of the journal file will be *idlsave.pro*.

```
IDL> a = [3, 5, 7, 3, 6, 9]
IDL> Help, a
      A  INT = Array[6]
IDL> Plot, a
```

When you want to close the journal file, just type the *Journal* command again, all by itself, at the IDL command line, like this.

```
IDL> Journal
```

The journal file is a simple ASCII text file that you can edit, if you like, with any text editor, including the editor built into the IDL Workbench. When you want to replay the commands in the journal file, use the @ sign as the first character on the IDL command line. For example, to replay the commands in the *book_commands.pro* file above, type this.

```
IDL> @book_commands
```

You do not have to provide a file extension (a **.pro* file extension will be assumed. Nor do you have to put the file name in quotes to indicate this is a string. (This is the *only* place in IDL where this is true!)

Note: Be sure to give each journal file you create a unique name. You cannot append to journal files, so if you open a second file with the same name as the first most operating systems will simply overwrite the first journal file without warning.

If you would like to have a unique journal file name each time you wanted a journal file, you could use the [Coyote Library](#) program [TimeStamp](#) to create a journal file with a unique name based on the current time of day. (The [TimeStamp](#) program can produce 12 different styles of time stamps, by the way. This is just the default version.) Notice how the file name is constructed by concatenating strings together with the plus sign.

```
IDL> time = TimeStamp()
IDL> filename = 'journal' + time + '.pro'
IDL> Print, filename
      journal_tue_sep_28_12_46_19_2010.pro
IDL> Journal, filename
IDL> a = [3, 5, 7, 3, 6, 9]
IDL> Help, a
      A  INT = Array[6]
IDL> Plot, a
IDL> Journal
```

To run the journal file and play the commands over again, you simply type this.

```
IDL> @journal_tue_sep_28_12_46_19_2010
```

Chapter 2



Working With Graphics Windows

Creating Graphics Windows

You will learn more about IDL traditional graphics windows as you work through the examples in this book, but here are some commands you should be familiar with before you get started.

First, a graphics window can be created directly with the *Window* command or indirectly by issuing a traditional graphics display command when no graphics window is open. For example, you can create and open a window by typing this.

```
IDL> Window
```

Note: Only IDL traditional graphics commands can be displayed in a traditional graphics window. The object graphics system in IDL is a completely separate graphics system, and object graphics output can only be displayed in object graphics windows. It is impossible to mix and match graphics systems in IDL for this reason. Each requires a window of its own type.

Notice that the title bar of the window you just created has the name *IDL 0* in the upper left corner of the window. The 0 is this window's *graphics window index number*. Each graphics window has a unique graphics window index number associated with it when the window is created on the display.

The *Window* command without any positional parameters always creates a window with window index number 0. We say this is "Window 0". You can have up to 256 graphics windows open at any one time in an IDL ses-

sion. You can assign a graphics window index number for windows 0 through 31. IDL will assign graphics window index numbers for windows 32 through 255 by creating windows with the *Free* keyword (discussed below). For example, if you want to create a window with graphics window index number 10, you can type this.

```
IDL> Window, 10
```

If a window with the same graphics window index number already exists on the display, a *Window* command like this will first destroy the old one and then create a new one with this index number.

If you prefer (this is *always* a good idea when you are creating windows in IDL programs), you can open a window with a graphics window index number that is “free” or unused. The *Free* keyword is used for that purpose, like this.

```
IDL> Window, /Free
```

A window that is created with a *Free* keyword will have a graphics window index number greater than 31. The *Free* keyword is the only way to create a graphics window with an index number greater than 31.

The Current Graphics Window

If you have been typing these commands, you now have at least three graphics windows open on the display. Only one of those windows is the *current graphics window*. The current graphics window is the window that will receive the output of an IDL traditional graphics command. There is one, and only one, current graphics window. The graphics window index number of the current graphics window is always stored in the *!D.Window* system variable. (A system variable is the same as a global variable in IDL.) The value of *!D.Window* will be -1 if there are no traditional graphics windows created or open to draw into.

This means, for example, that if you want to create a window and store its graphics window index number so you can later delete it or make it the active window (see details below), you can store it in an IDL variable, like this.

```
IDL> Window, /Free
IDL> thisWindowIndex = !D.Window
```

Setting the Current Graphics Window

To make a window the current graphics window (so you can display graphics in it), you use the `WSet` command and pass it the window's graphics window index number. For example, suppose you want window 10 to be the current graphics window. You would type this.

```
IDL> WSet, 10
```

All subsequent graphics commands will be displayed in window 10.

Note: A window becomes the current graphics window when it is created. This is true whether the window is created with the `Window` command or whether it is created as a result of issuing an IDL graphics command. But, IDL traditional graphics windows that are created as draw widgets (using `Widget_Draw`) do not automatically become the current graphics window. Those windows must be made the current graphics window by issuing the `WSet` command, as described above.

The current graphics window is not necessarily the window that is in front of others on the display screen. You will learn more about this in a moment. But for now, realize that the current graphics window might be obscured by other windows on your display. That doesn't prevent graphics output from being displayed in the current graphics window when a graphics command is executed.

Deleting Graphics Windows

Graphics windows are deleted with the `WDelete` command and the window's graphics index number. A window does not have to be the current graphics window to be deleted. For example, to delete window 10, you can type this.

```
IDL> WDelete, 10
```

If a window is the current graphics window and it is deleted, the `!D.Window` system variable will default to some other open graphics window number, if there is one, or to -1 to indicate there is no currently open and available graphics window.

You do not have control over, nor do you necessarily know, which open window will become the next current graphics window. You should always deliberately set the current graphics window with the `WSet` command if you want to guarantee your graphics output will show up in a particular

graphics window. This is absolutely essential when writing widget programs.

Here is a trick to delete all the graphics windows that exist on the display currently. Use a *While* loop to delete each current graphics window in turn, until the *!D.Window* system variable is set to -1.

```
IDL> WHILE !D.Window NE -1 DO WDelete, !D.Window
```

Such a command can easily be put into an IDL program. For example, here is a program named *CleanUp* that, when run, will delete all the currently open IDL traditional graphics windows on the display.

```
PRO CleanUp  
    WHILE !D.Window NE -1 DO WDelete, !D.Window  
END
```

Positioning and Sizing Graphics Windows

Windows are positioned and sized according to an internal algorithm when they are created. You can influence how this internal algorithm works by setting window properties in, for example, the IDL Workbench Preferences dialog, as shown in Figure 1.

Window size and layout can also be changed by setting window preferences in versions of IDL starting with IDL 6.2 and later. The preferences *IDL_GR_WIN_HEIGHT* and *IDL_GR_WIN_WIDTH* can be set so that the default window size is 500 by 500, like this.

```
IDL> Pref_Set, 'IDL_GR_WIN_HEIGHT', 500, /Commit  
IDL> Pref_Set, 'IDL_GR_WIN_WIDTH', 500, /Commit
```

To see a complete list of the IDL preferences that can be set this way, type this.

```
IDL> Help, /Preferences
```

Or, to see more detail,

```
IDL> Help, /Preferences, /Full
```

You can position and size windows when you create them with keywords to the *Window* command. For example, to create a window that is 200 pixels wide and 300 pixels high, use the *XSize* and *YSize* keywords, like this.

```
IDL> Window, 1, XSize=200, YSize=300
```

Windows are positioned on the display with respect to the upper left-hand corner of the display in pixel or device coordinates (the terms are used

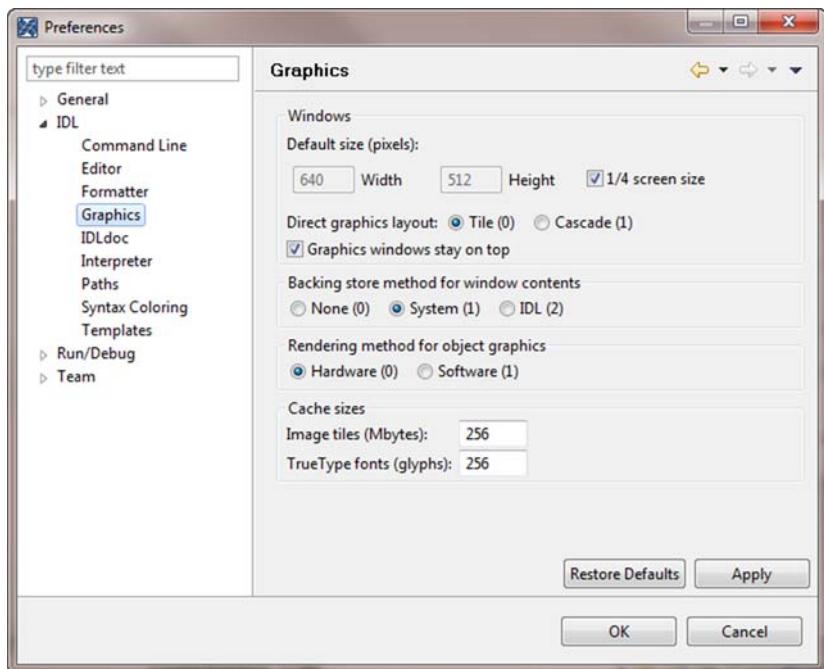


Figure 1: The default window size and layout can be controlled by setting preferences in the Graphics Preferences dialog of the IDL Workbench.

interchangeably). To position a window with its upper left-hand corner at location (75,150) on the display, use the `XPos` and `YPos` keywords, like this.

```
IDL> Window, 2, XPos=75, YPos=150
```

Traditional graphics windows cannot be programmatically sized or positioned after they appear on the display. In other words, there are no IDL commands you can use to resize a traditional graphics window once it has appeared on the display. Users can move graphics windows once they are on the display with the mouse, of course. And they can resize graphics windows with the mouse, although this does not generally result in anything positive, since traditional graphics windows do not redraw their contents after being resized.

To create resizeable traditional graphics windows that redraw their contents in the resized window, you have to write IDL widget programs.

(Programs that allow user interactivity via a graphical user interface.) The program [cgWindow](#) from the [Coyote Library](#) is a general purpose traditional graphics widget program that can provide a resizeable, re-drawable IDL traditional graphics window for most IDL graphics display commands. To see how [cgWindow](#) works, type this command.

```
IDL> cgWindow, 'Plot', Findgen(11)
```

We will make further use of [cgWindow](#) throughout the book. (See, in particular, “Using Resizeable Graphics Windows” on page 418.)

Viewing a Hidden Graphics Window

Creating a graphics window with the *Window* command, for example, gives that window the window focus and also makes it the current graphics window. That is, the graphics window is now the active window with respect to the Window Manager (the operating system program that manages windows on your computer). Just because a graphics window has the window focus does not mean, in general, that it is the current graphics window. To type a command at the IDL command line, you have to move the window focus back to the command window. On some platforms, especially Windows computers, this causes the graphics window to pop back behind other windows.

Or, sometimes a graphics window just gets behind other windows on the display and you would like to bring it forward so you can see it. To bring a window forward on the display, without changing the window focus, use the *WShow* command and the window’s graphics index number, like this.

```
IDL> WShow, 1
```

Notice that your cursor and the window focus are still in the command window or window where you are typing IDL commands.

***Note:** Starting with IDL 7, the *WShow* command no longer keeps the window focus in the IDL command window, making this command quite a bit less useful. It does still bring the graphics window forward on the display. But, in IDL 8, the command appears to no longer work at all and has no effect on graphics windows. Oddly, if you open a widget application after opening an IDL graphics window in IDL 8, the *WShow* command occasionally works to bring the graphics window forward. This implies this might be a bug that will be fixed in future versions of IDL.*

Bringing a window forward with the *WShow* command does *not* make the window the current graphics window. If you want to move a window forward and make it the current graphics window (assuming it is not the current window already), then you have to type both a *WShow* and a *WSet* command, like this.

```
IDL> WShow, 2  
IDL> WSet, 2
```

Typing *WShow* without a parameter will bring the current graphics window forward on the display.

```
IDL> WShow
```

Note: On Windows and Macintosh computers, you can use the ALT-TAB or OPTION-TAB keys, respectively, to cycle through and select windows that are open but not currently visible on the display and make them the window with the window focus.

Putting a Title on a Graphics Window

Sometimes you would like your graphics window to have a more descriptive title than just its graphics window index number. You can use the *Title* keyword to put a title on the window, like this.

```
IDL> Window, Title='Example IDL Graphics Window', $  
      XSize=400, YSize=50
```

You see the result in Figure 2.

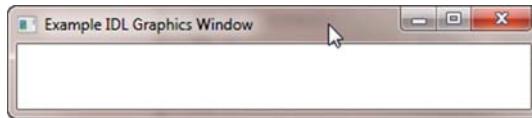


Figure 2: Windows can be created with titles in IDL.

Erasing a Graphics Window

To erase the current graphics window, you can use the *Erase* command, like this.

```
IDL> Erase
```

To erase graphics windows that are *not* the current graphics window, you must first make the window the current graphics window with the `WSet` command and then issue the `Erase` command.

If you want to erase the current graphics display with a particular color index (or 24-bit color value, if you are using the decomposed color model), you can use the `Color` keyword. For example, you can erase the current graphics display with a charcoal gray color when you are using the indexed color model by typing this.

```
IDL> Device, Decomposed=0  
IDL> TVLCT, 70, 70, 70, 100  
IDL> Erase, Color=100
```

Or, if you use the decomposed color model, you could type this.

```
IDL> Device, Decomposed=1  
IDL> Erase, Color='464646'xL
```

Or, if you want to erase the display with a color without regard for the color model you are using, you can use the `cgColor` command and type this.

```
IDL> Erase, Color=cgColor('charcoal')
```

Creating a “Maximum Size” Window

Surprisingly often, especially when creating large research applications, IDL users need to make very large graphics windows. In fact, many times these windows are created to be, essentially, the size of the display monitor. But, they are not *exactly* the size of the display monitor, because if they were they would be obscured by window decorations like window task bars, application bars, tool bars, etc. In other words, there is more happening on a display monitor than just a graphics window.

Unfortunately, knowing exactly how big to create a “maximum size” graphics window is not as easy as you might like it to be. In fact, it points out some of the difficulties software developers have in creating software that works in a platform independent manner. It’s worth considering some of the difficulties and some of the tools you have at your disposal.

Determining the Size of the Current Graphics Window

If you create a window, or if you make a window the current graphics window, you can programmatically determine the size of that window by examining the `!D.X_Size` and `!D.Y_Size` system variables.

```
IDL> Window, 1, XSize=450, YSize=350
IDL> Print, !D.X_Size, !D.Y_Size
      450      350
```

Determining the Size of the Display Monitor

It is also possible to determine the size of the display monitor. One way to do this, for example, is to use the *Device* command with the *Get_Screen_Size* keyword.

```
IDL> Device, Get_Screen_Size=screenSize
IDL> Print, screenSize
      1280      1040
```

Another method to obtain the size of the display monitor is to use the IDL library routine *Get_Screen_Size*.

```
IDL> Print, Get_Screen_Size()
      1280      1040
```

It turns out that *Get_Screen_Size* is actually a wrapper for an object in IDL that can obtain information about the display monitor. The object is the *IDLsysMonitorInfo* object. To obtain the display monitor size with this object, you write code like this.

```
IDL> monitorInfo = Obj_New('IDLsysMonitorInfo')
IDL> rects = monitorInfo -> GetRectangles()
IDL> pIndex = monitorInfo -> GetPrimaryMonitorIndex()
IDL> Print, rects[[2,3], pindex]
      1280      1040
```

Normally, these values are the size of the display monitor and do not take into account the usual task bar at the bottom (or sometimes the top) of the window. Fortunately, there is a keyword, named *Exclude_Taskbar*, to the *GetRectangles* method that can account for the task bar.

```
IDL> mInfo = Obj_New('IDLsysMonitorInfo')
IDL> rects = mInfo -> GetRectangles(/Exclude_Taskbar)
IDL> pIndex = mInfo -> GetPrimaryMonitorIndex()
IDL> Print, rects[[2,3], pindex]
      1280      994
```

Unfortunately, this keyword is not accessible via the *Get_Screen_Size* function. And even more unfortunately, the keyword only works for Windows machines. The *Exclude_Taskbar* keyword is ignored on all UNIX machines, including Macintosh.

Size Tools Are Not Platform Independent

But this is not the worst news. It turns out that *all* these results are platform dependent! In other words, you might well get different results from printing the *!D.X_Size* and *!D.Y_Size* system variables, or by trying to determine the size of the display monitor, depending upon what kind of machine you are running IDL on.

Here is the situation as of IDL 8.0 for a 1440x900 display monitor.

The size of the display monitor can be reliably determined for Windows and UNIX by using the *Get_Screen_Size* keyword to the *Device* command.

```
; Windows and UNIX computers.  
IDL> Device, Get_Screen_Size=screenSize  
IDL> Print, screenSize  
1440 900
```

Macintosh computers subtract the size of the ever-present menu bar from these values. For example, on a Macintosh with a 1440x900 display monitor, you find these results.

```
; Macintosh computers.  
IDL> Device, Get_Screen_Size=screenSize  
IDL> Print, screenSize  
1440 878
```

The IDL library function *Get_Screen_Size* can also be used to return accurate display monitor sizes for UNIX and Windows computers, but not for Macintosh computers.

```
; Windows and UNIX computers.  
IDL> Print, Get_Screen_Size()  
1440 900  
;  
; Macintosh computers.  
IDL> Print, Get_Screen_Size()  
1440 878
```

Note: The values for the screen size window shown here are representative. The numbers you will see as you type the commands depend on many factors, such as the window manager you are using, which version of the operating system you are using, and so forth.

Normally, the *!D.X_Size* and *!D.Y_Size* system variables return the size of the current graphics window on all platforms. But there is one exception on UNIX computers, excluding Macintosh computers. If a window is created on the display that is, say, the size of the display, then on UNIX

computers, excluding Macintosh computers, the *!D.X_Size* and *!D.Y_Size* system variables will contain the size of the unobstructed graphics window.

```
; UNIX computers, excluding Macintosh.  
IDL> Device, Get_Screen_Size=screenSize  
IDL> Print, screenSize  
1440 900  
IDL> Window, XSize=screenSize[0], YSize=screenSize[1]  
IDL> Print, !D.X_Size, !D.Y_Size  
1440 854
```

Macintosh computers, like Windows computers, will continue to display the actual size of the created window.

```
; Macintosh computers.  
IDL> Device, Get_Screen_Size=screenSize  
IDL> Print, screenSize  
1440 878  
IDL> Window, XSize=screenSize[0], YSize=screenSize[1]  
IDL> Print, !D.X_Size, !D.Y_Size  
1440 878
```

So, here is where we are. If we want to learn what size window to create for a “maximum size” graphics window, we can programmatically find the values for Windows computers by using the *Exclude_Taskbar* keyword in the *GetRectangles* method of the *IDLsysMonitorInfo* object. We have no programmatic way to find this information for UNIX computers, but for non-Macintosh UNIX computers, we can open a larger window and obtain the values we need by examining the *!D.X_Size* and *!D.Y_Size* system variables. We have no method to obtain this information from Macintosh computers, but we can program a “fudge factor” of, say, 22 pixels to account for the Macintosh “dock.”

There is an example of a function that can return values of a maximum size window for all platforms, using these methods, in the [MaxWindowSize](#) command from the [Coyote Library](#).

```
IDL> maxsize = MaxWindowSize()  
IDL> Window, XSize=maxsize[0], YSize=maxsize[1], /Free
```

Note: Because a graphics window has to be created and destroyed to obtain this information for UNIX computers, excluding Macintosh, there is a momentary window “flash” when this function is called on UNIX computers. This is unfortunate, but unavoidable.

Device Independent Graphics Windows

One of the goals of this book is to write device-independent IDL programs. “Device-independent”, for the purposes of this book, means works the same on your computer display, in the Z-graphics device, and in a PostScript file. Each of these display devices uses a “window” of some sort, which is where the result of graphics commands ends up. (Only your computer has an area that is conventionally thought of as a window.)

To this end, the [Coyote Library](#) routine `cgDisplay` allows the user to specify a particular window “size” in a device-independent manner. To create a 500x400 pixel window on your computer display, for example, you would type this.

```
IDL> cgDisplay, 500, 400
```

The very same command will set the resolution of the Z-graphics device to 500x400 or will create and center a PostScript “window” with this aspect ratio on the PostScript page. If you get into the habit of opening “windows” with `cgDisplay` it will be much easier to write IDL graphics programs that work identically on every platform and on every device.

The `cgDisplay` command can open windows on the display with different graphics window index numbers. Use the `WID` keyword to assign an index number, or use the `Free` keyword to get a window index number that is currently not being used. Keywords are ignored on devices that don’t support graphics windows.

```
IDL> cgDisplay, 500, 400, WID=3
IDL> cgDisplay, 500, 400, /Free
```

One of the advantages of `cgDisplay` is that it creates a graphics window with a white background by default. This makes it much easier to write IDL programs that look the same on all devices. It is especially important to create an `cgDisplay` window if you are going to use the `Layout` keyword on [Coyote Library](#) routines to position graphics in the window. Since graphics routines using the `Layout` keyword cannot erase what is currently in the window, getting the background color right is important. Here are some commands that illustrate the point.

```
IDL> cgDisplay
IDL> cgLoadCT, 33, RGB_Table=palette
IDL> cgPlot, cgDemoData(1), Layout=[2,2,3], Color='red'
IDL> cgContour, cgDemoData(2), NLevels=12, $
      Layout=[2,2,1], Color='dodger blue'
```

```
IDL> cgSurf, cgDemoData(2), /Elevation, Layout=[2,2,4], $  
      Palette=palette  
IDL> cgImage, cgDemoData(19), Multimargin=4, /Axes, $  
      Layout=[2,2,2]
```

You see the result in Figure 3.

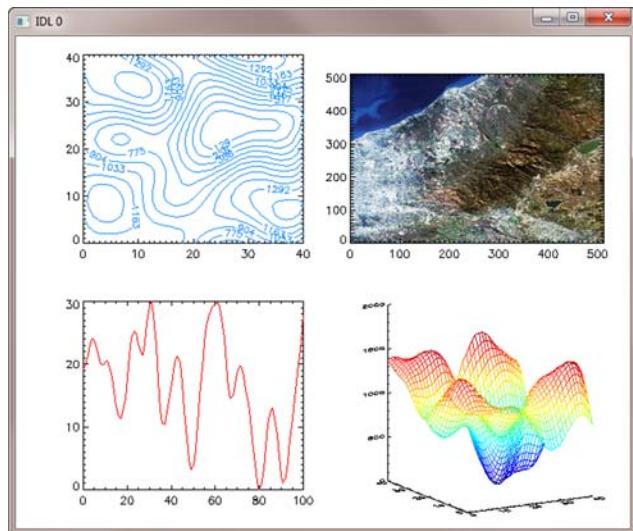


Figure 3: Use `cgDisplay` to create a window with a white background if you are going to use the `Layout` keyword to position Coyote Graphics routines.

You can use `cgErase` to erase a graphics window. If called without an argument, the current graphics window is erased.

```
IDL> cgErase
```

Resizeable Graphics Windows

A traditional IDL graphics window (i.e., one created with the `Window` command or by issuing a traditional graphics command) can be resized by the user because this is an allowed window system operation. But it does no good to do so because the graphic displayed in the window has no connection to this kind of window operation. In other words, the graphic displayed inside a resized window is not resized or re-displayed.

To create a true resizeable graphics window in IDL, you have to build a widget program (a program with a graphical user interface) which can receive widget “events.” If the event is a “resize window” event, then the program can resize the window and re-display the graphic in the newly-resized window. Presuming, of course, that the program knows how to recreate the graphic display and has all the data and information it needs to do so.

IDL leaves you pretty much on your own to create such a widget program if you want to. Most people don’t bother.

There is a powerful resizeable graphics window program in the [Coyote Library](#), however, which can be used to create a resizeable graphics window for any traditional IDL graphics command. The program is `cgWindow` and it is a fundamental part of the graphic routines in the [Coyote Library](#), which have collectively come to be known as the Coyote Graphics System. You will learn more about this system in “The Coyote Graphics System” on page 417 of this book.

For now, it is enough to know that `cgWindow` maintains a list of one or more commands (there is no limit) that can be “executed” when the graphics window is resized. Commands can be added and deleted from the command list as needed.

Here, for example, is how you would display a simple line plot in this resizeable graphics window.

```
IDL> cgWindow, 'Plot', FIndGen(11)
```

If you now want to see a surface plot in the window, you would do this.

```
IDL> cgWindow, 'Surface', Dist(40), /ReplaceCmd
```

While `cgWindow` is designed to work with any traditional graphics command or commands, it is especially effective when used with other graphics routines in the Coyote Graphics System. These routines all have keywords (*Window* and *AddCmd*) that will allow them to automatically send their contents, or add their contents, to a `cgWindow` display. Here, for example, is the same code we used to create Figure 3, but now the code is displayed in a resizeable graphics window.

```
IDL> cgLoadCT, 33, RGB_Table=palette
IDL> cgPlot, cgDemoData(1), Layout=[2,2,3], $
      Color='red', /Window
IDL> cgContour, cgDemoData(2), NLevels=12, $
      Layout=[2,2,1], Color='dodger blue', /AddCmd
```

```
IDL> cgSurf, cgDemoData(2), /Elevation, $  
      Layout=[2,2,4], Palette=palette, /AddCmd  
IDL> cgImage, cgDemoData(19), Multimargin=4, /Axes, $  
      Layout=[2,2,2], /AddCmd
```

You see the result in Figure 4. Not only is this window resizeable, but it has controls to automatically send the window commands directly to a PostScript file or to save the window contents in any of five raster image file formats. The graphics output can also be saved to disk, so you can send it to a colleague to view, or so you can open it and see the very same display sometime later.

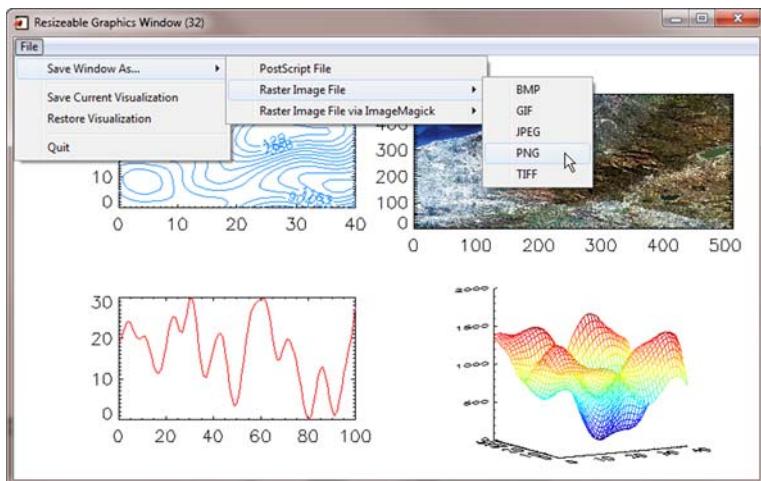


Figure 4: The `cgWindow` program creates a *resizeable graphics window* for traditional graphics commands. Window commands can be sent directly to a PostScript file, or saved in any of five different raster image file formats. The graphics display can also be saved to a file, where it can be restored and viewed again at a later time.

You can have as many `cgWindow` programs open on the display as you like. In fact, the point is to be able to work with these resizeable graphics windows in exactly the same way you currently work with traditional graphics windows. To that end, you can select the current `cgWindow` with the `cgSet` command and you can delete a `cgWindow` with the `cgDelete` command. There is also an `cgControl` command that can set various properties of the graphics window (e.g., background color, its settings for

displaying multiple plots in the window, the color palette associated with the window, etc.).

The details of how `cgWindow` works will be postponed until the last chapter in the book, when you will have an appreciation for how easy it is to write programs that can take advantage of its capabilities.

Chapter 3



Working With Color

Color History and Transition

The purpose of this chapter is to present a brief introduction to color in the traditional graphics system in IDL. (Color is handled differently and—some would say—more consistently in the object graphics system. But that is not a topic for this book.) According to participants in IDL programming classes, nothing is as consistently frustrating as trying to get color output to work correctly on the different graphics output devices supported by IDL (e.g., your Windows display, your colleague's UNIX display, in the Z-graphics buffer, in PostScript output, etc.). This chapter will explain how color works in IDL and will introduce you to several color tools from the [Coyote Library](#) that will make it easier for you to write IDL programs that work in a device independent way.

Starting about the time of IDL 5.5, color handling was in a state of flux. Very few versions of IDL were consistent with earlier versions of the software. We were in a constant state of confusion. Programs that would run and display correctly on this computer would not display correctly on a colleague's computer, and vice versa. Sometimes colors wouldn't display properly even if you were using the same computer operating system and version of IDL!

Some of the confusion was caused by the fact that we were in a period of transition from 8-bit to 24-bit graphics cards. It was extremely important then to know how to write IDL programs that could coexist in both of these environments. But I haven't run into an 8-bit graphics display in a very long time, so I presume nearly everyone has made the transition to 24-bit graphics cards successfully.

Unfortunately, we haven't been able to escape the 8-bit environment or mind-set completely. We still have older graphics devices (e.g., older versions of the PostScript and Z-graphics devices) that are 8-bit devices. And, there is an enormous amount of legacy software written from an 8-bit point of view we have to deal with. (Even more alarming, there is a great deal of IDL software that *continues* to be written from an 8-bit point of view!) This is, to put it politely, a handicap when we want to use this software on 24-bit graphics displays.

So we still have problems to overcome. This chapter is designed to help you meet these problems head on, and to help you write more flexible traditional graphics programs in IDL. We will use what we learn here in examples throughout this book.

Vital Information for UNIX Users

If you are running IDL on a Windows computer, you can skip this section and ponder this question instead. How is it that an operating system so reviled by UNIX users can actually do some things right?

More on that (and on what Windows does wrong!) later, but this section is addressed to UNIX users in particular. If you were a UNIX user running versions of IDL prior to IDL 6.2, you were put, by default, into the X windows color environment, *DirectColor*, from which it was next to impossible to recover. Nothing you could do in that environment made much sense. Graphics windows were always flashing colors at you when you switched from one window to another. And the red on black color scheme in most IDL graphics windows, which made it almost impossible to see what was in the window, was thought to be just the (strange) way IDL's designers had planned it.

Most of the people using IDL this way did not realize they had missed the color memo until they observed other IDL programmers working in completely different ways. Unfortunately, the memo (if there *was* a memo!) was easy to miss. Even if they got the memo, getting their machines set up correctly was more like fooling around with mystical incantations, with its strange vocabulary ("backing store," "X window visual class"), then it was like using a piece of modern software with (supposedly) helpful defaults.

You might be one of these unfortunate users. How would you know? Here is how. Start an IDL session and type the following commands. You use the *Device* command to get information about your current graphics device.

```
IDL> Window  
IDL> Device, Get_Visual_Name=theVisual, $  
      Get_Visual_Depth=theDepth  
IDL> Print, theVisual, theDepth
```

If the visual name is *DirectColor*, then you can be completely excused for not understanding how color works in IDL. No one else does either in that visual environment.

I think almost everyone these days will see the depth as 24, meaning a 24-bit graphics display, which is typical. If you have an 8-bit depth or a *PseudoColor* visual name, then you are probably stuck with an ancient computer and I feel your pain, but you are in reasonable shape for colors. If your depth is 16, you are probably okay, but you will be confused when I talk about specific, vibrant colors. You will probably experience them as dull, lifeless things. If your depth is 24 and you have a *PseudoColor* visual name, and you are reading this book, it is probably because a few of your colleagues are complaining about your programs and would like you to know how to write programs that can coexist easily with 24-bit devices. The discussion that follows will certainly help.

Use a *TrueColor* Visual Class

If you have a 24-bit graphics card (the depth was 24 in the commands above), then you want to be using a *TrueColor* visual class, *not* a *DirectColor* visual class. (If you have an 8-bit depth, then you should be using the *PseudoColor* visual class.) Unfortunately, the visual class is selected at the moment when IDL opens its first graphics window, and cannot be changed in that IDL session.

Selection of an X windows visual class can be done in one of two places. You can modify your *.Xdefaults* file (if you have one) to include the *idl.gr_visual* and *idl.gr_depth* resources, like this.

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

Or, you can modify your IDL startup file to select a 24-bit *TrueColor* visual by adding the following command to your IDL startup file. Be sure to add it in front of any commands that might open an IDL graphics window.

```
Device, True_Color=24
```

You will have to exit IDL and restart it for these changes to take effect.

Now you will at least have put yourself in a position from which colors *can* be understood, although it will probably still require diligent study. As a

warning, I should point out that even those of us who consider ourselves reasonably knowledgeable in the color arena find ourselves scratching our heads a great deal more frequently than seems absolutely necessary. Your mileage may vary, too. But, read on.

Understanding IDL Color Models

The central problem to be overcome in trying to understand color in IDL is this: two completely different color models can be used to specify colors on a 24-bit graphics display, and IDL traditional graphics commands work differently depending upon which model you chose to use in the IDL session. This problem is only compounded by the fact that some IDL programs (those written by many of your colleagues, no doubt, but some commonly used ones in IDL's own library, too) will only work in one color model and not the other. And then, of course, colors work differently on Windows and UNIX machines, depending upon which version of IDL you are using. Sigh... It does take some time and experience to sort it all out.

The two color models are called *decomposed color* and *indexed color*. We sometimes refer to these models as “decomposition on” and “decomposition off,” respectively, because of the way each model is selected with the *Device* command. The *Decomposed* keyword is set to 1 to indicate the *decomposed color model* is to be used, and is set to 0 to indicate the *indexed color model* is to be used.

```
Device, Decomposed=1 ; Selects Decomposed Color Model.  
Device, Decomposed=0 ; Selects Indexed Color Model.
```

By default, when IDL starts up in a 24-bit environment (a *TrueColor* environment on UNIX, hopefully!), it will be using the decomposed color model. This is also true of Windows machines with a 24-bit graphics card. Or, another way to say this, color decomposition is *on*. But, what does this mean?

Colors Are Represented as Color Triples

Every color in IDL is represented, ultimately, as a three-element byte vector of red, green, and blue values, in which each value can vary between 0 and 255. We call this vector a *color triple*. If we think of the values between 0 and 255 as color intensities, so that 0 represents none of the color, and 255 represents as much of the color as we can provide, then we have the possibility of specifying 256 times 256 times 256, or approximately 16.7 million, different colors in IDL. We say we have a palette of 16.7 million

colors to choose from. This is also known as *true color*, because it is similar enough to what we see with our eyes to be a representation of the visible world.

The two color models arise from how we select a color from this color palette. We might choose to select one of the 16.7 million colors directly, by specifying its color triple. We call this the decomposed color model (Figure 1). Or, we might choose to load a 256-element subset of these 16.7 million colors into a color table, and use the index number of the color table to select a specific color from the 256 choices we made available. If we do this, we specify a color triple by indicating which index in the color table is associated with the color we want to use. We call this the indexed color model (Figure 2).

If we decide to select our color directly, we must specify a color triple. But, rather than using a three-element vector, as the object graphics system does, in the traditional graphics system we create a 24-bit value that can be *decomposed* into three 8-bit values. This is what is meant by *color decomposition*. You see an example of the decomposed color model in Figure 1.

Consider a green color, which can be represented by the color triple [53, 156, 83]. The first element is the red value, the second is the green value, and the third is the blue value. To construct a 24-bit value that can be decomposed into this color triple, we write code like this.

```
IDL> clr = [53, 156, 83]
IDL> greenColor = clr[0] + clr[1]*2L^8 + clr[2]*2L^16
IDL> Print, greenColor
5479477
```

The highest 8 bits in this 32-bit long integer value are not set and are all zeros. Displayed as a binary value, with the highest 8 bits removed, the number looks like this, with the lowest 8 bits on the right. The lowest 8 bits represent the red color, the middle 8 bits represent the green color, and the highest 8 bits represent the blue color.

```
0 1 0 1 0 0 1 1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1
```

If we want to express this green color as a color index, we will have to load the color into the color table at a particular color index. Suppose we load it at color index 100 with the *TVLCT* command, like this.

```
IDL> TVLCT, 53, 156, 83, 100
IDL> greenColor = 100
```

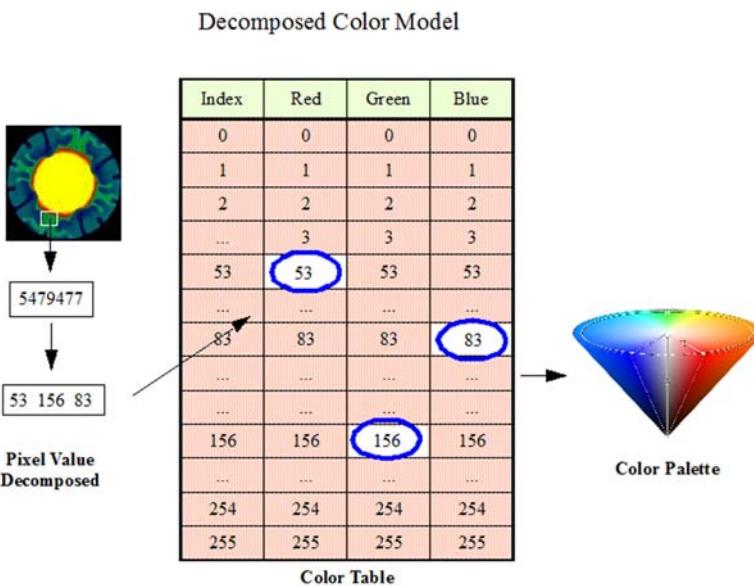


Figure 1: In the decomposed color model a color is expressed as a number that can be decomposed into a red, green, and blue color value. By independently being able to chose the color values, all 16.7 million colors are available to the user. In a 24-bit color image, the three independent color values are selected by the pixel values in the three color planes.

The last argument to *TVLCT* is the color index where we are loading the color triple [53, 156, 83]. This is how we will access this color when using the indexed color model. Note that a single index (0 to 255) is used to select three separate values: the red, green, and blue color values associated with that index in the color table. You see this illustrated in Figure 2.

We see now that the same green color can be represented as a 24-bit number (*greenColor* = 5479477L) or as a color index (*greenColor* = 100) in IDL. In the vast majority of IDL graphics output commands, colors are input as a value to a *Color* keyword (or to an equivalent keyword like *Background*, etc.). Whether that value is interpreted as a value to be decomposed or as an index number into a color table depends on what color model is currently selected in the IDL session. We say it depends on the *color model* (or, sometimes, the *color decomposition state* or *color mode*) of the IDL session.

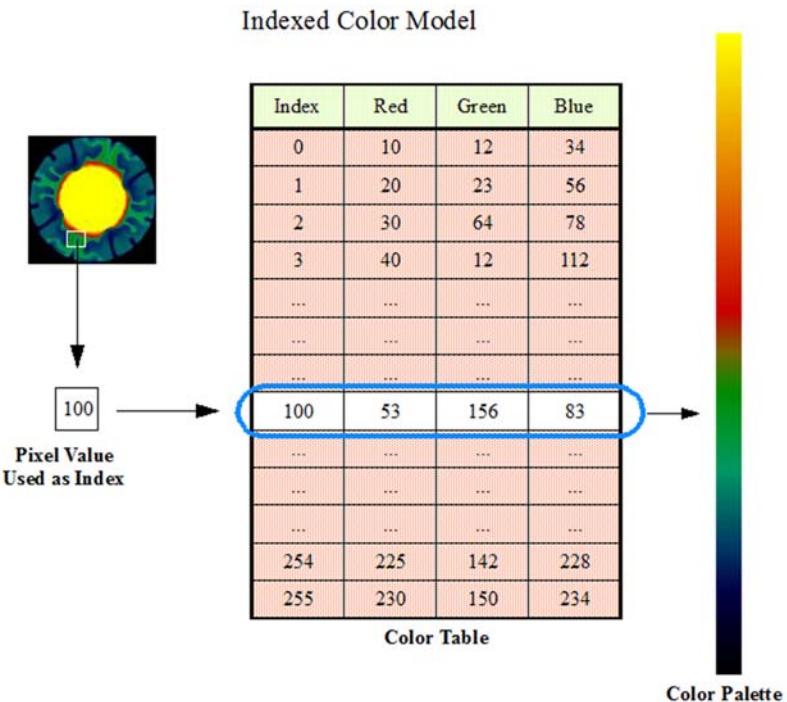


Figure 2: In the indexed color model 256 colors out of the palette of 16.7 million colors are loaded into a color table. One number, the index, is used to look up the particular red, green, and blue values to use in creating the color triple with which pixels having the index value will be displayed.

Naturally, you can get strange results if the color value you supply is mismatched with the color model that can interpret the value appropriately. Most IDL users run into problems when they use color values that represent color index numbers in their code, but IDL is set to use the decomposed color model (which, remember, is the default color model), which interprets such color values as numbers to be decomposed. If you decompose any number from 0 to 255 (which are valid color table index numbers) into a 24-bit value, the only bits you can possibly set are those bits used to represent red colors. For example, the binary value 200 is represented like this.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0
```

Now, do you understand why you might be seeing red plots on black backgrounds in IDL? Here is an example of exactly this sort of mismatch between a color value and a color model.

```
IDL> Device, Decomposed=1  
IDL> Plot, cgDemoData(1), Color=100
```

You see a red plot on a black background, as in Figure 3, where I have actually made the background color charcoal, so you can see the red plot a little better. Even so, this plot is *extremely* difficult to see. I have talked to users who believed this was the normal color scheme in IDL!

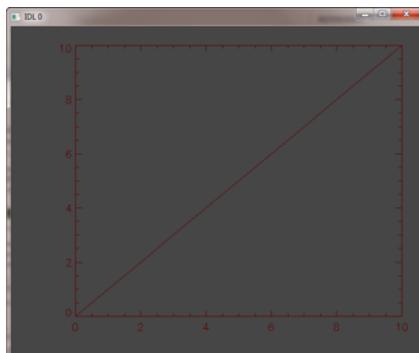


Figure 3: A mismatch between the color model in effect when a graphics command is issued and the way colors are specified can often end up with red plots on black backgrounds.

The solution, of course, is to match your color model with the color representation of your number (an indexed color model, for example, when you use the color index number 100 to represent a green color).

```
IDL> Device, Decomposed=0  
IDL> Plot, Findgen(11), Color=100
```

Or, done the other way around.

```
IDL> Erase  
IDL> Device, Decomposed=1  
IDL> Plot, Findgen(11), Color=5479477L
```

Because so much IDL software has been written in the past with an 8-bit world view, many users find it advantageous to make sure they use the indexed color model when working with this software. This necessarily limits them to only 256 colors out of a palette of 16.7 million colors, but

often this is enough. So you will see the following line in many IDL startup files.

```
Device, Decomposed=0 ; Start in indexed color mode.
```

Although not *exactly* the same as having to dress in mustard yellow shirts with wide, paisley ties and bell-bottomed pants to go to work, it does tend to date you, nonetheless. (I'm looking at a mid-1970s wedding photo for a reference here.)

So I have taken it on as my special mission to the IDL community to teach people to take advantage of that 24-bit graphics card they paid so much money for and learn to use the 16.7 million colors available to them in a better way.

Specifying Colors by Name

Here is the problem, as I see it. The kind of code we are talking about writing doesn't exactly give me the warm, cozy feeling of "green."

```
Plot, Findgen(11), Color=100  
Plot, Findgen(11), Color=5479477L
```

The whole "color as number" scenario doesn't make much sense to me. Especially when I am busy trying to figure out what color model or "decomposition state" I happen to be using when I get around to displaying some graphics. It would make a lot more sense to be able to write code like this.

```
Plot, Findgen(11), Color='green'
```

And expect to find a green plot on my display no matter what decomposition state I am in or what color model I am using when I type the command.

Of course, IDL doesn't work this way. But we can't have IDL dictating how we work, or we will all go paranoid and schizophrenic, sure enough. So I have written an "independent color" program, named `cgColor`. With `cgColor` I can write code like the following and I can always expect a green plot to appear on my display, no matter what color model is currently selected.

```
IDL> Plot, Findgen(11), Color=cgColor('green')
```

How does it work, and how many colors does it know about?

The program currently "knows" the names of about 200 colors. I chose these from a spectrum of colors to represent various drawing colors I would like to use in my own IDL programs, including the popular Brewer

colors. But if you don't like my colors, you can load your own colors from a text file that you can create and pass to `cgColor`. You can list the 200 colors in alphabetical order, like this.

```
IDL> colornames = cgColor(/Names)
IDL> Print, (colornames) [Sort(colornames)], $
Format='(6A18)'
```

Most of the colors you will want to use are probably in the list!

If you don't know the name of a color to use, `cgColor` allows you to select a color interactively from a palette of colors. Use the `SelectColor` keyword like this.

```
IDL> Plot, Findgen(11), Color=cgColor(/SelectColor)
```

Or, you can simple use the `PickColorName` program, like this.

```
IDL> color = PickColorName()
```

You will see something that looks like the example in Figure 4.



Figure 4: You can learn which color names are available to you by selecting them interactively with programs like `PickColorName`.

The program works very simply. It has four vectors internally. One vector is filled with color names, the other three vectors are filled with the red,

green, and blue values of the colors associated with those names. Here is a simplified representation of the four vectors.

```
names = ['teal', 'khaki', 'salmon']
r =      [     0,      240,      250 ]
g =      [   128,      230,      128 ]
b =      [   128,      140,      114 ]
```

When you ask for a color name, the name is found in the *names* vector with the *Where* function. The location where the name is found is called the color index. The same index is used to find the corresponding red, green, and blue values in the color vectors to create the color triple.

```
theIndex = Where(StrUpCase(names) EQ 'KHAKI')
colorTriple = [r[theIndex], g[theIndex], b[theIndex]]
```

Next, the program determines what color decomposition state is currently in effect for this IDL session. The *Get_Decomposed* keyword is used for this purpose.

```
Device, Get_Decomposed=currentState
```

If color decomposition is turned on, the program creates a 24-bit integer value from the color triple, and returns that as the result of the function. The [Coyote Library](#) program [Color24](#) is used to create the 24-bit value.

```
IF currentState EQ 1 THEN Return, Color24(colorTriple)
```

If, however, color decomposition is turned off, then the program loads the color at a particular color index number, and returns the color index number where the color was loaded to the user.

```
IF currentState EQ 0 THEN BEGIN
    TVLCT, Reform(colorTriple, 1, 3), 255-(theIndex)-1
    RETURN, 255-(theIndex)-1
ENDFOR
```

The 200 colors are designed to load themselves at unique indices at the top of the color table. This makes it possible to use various drawing colors on your display and in PostScript files, for example, without having to think much about where those colors should be loaded in a color table. That is to say, this method of specifying drawing colors *almost always* does the right thing.

But there are times when you want a color to be loaded at a particular color index number. You can do that with [cgColor](#) by simply specifying what that index number should be. For example, if you want to load a yellow color at color index 240, you can call [cgColor](#) like this.

```
color = cgColor('yellow', 240)
```

Note that the value of the variable *color* will depend on the decomposition state in effect when this command is issued. Colors are actually loaded into the color table *only* if color decomposition is turned off (i.e., indexed color is turned on). Otherwise, colors are turned into 24-bit values that can be decomposed into the proper color values.

```
IDL> Device, Decomposed=1
IDL> Print, cgColor('yellow')
      65535
IDL> Print, cgColor('yellow', 240)
      65535
IDL> Device, Decomposed=0
IDL> Print, cgColor('yellow')
      205
IDL> Print, cgColor('yellow', 240)
      240
```

While the `cgColor` program was originally designed to select colors at the moment graphics commands are being executed, there are times when colors have to be pre-loaded into the color table (e.g., when drawing graphics to a PRINTER device, or sometimes when drawing a filled contour plot). The `cgColor` program has been modified to help with that. Setting the *Triple* keyword will result in a color triple being returned instead of the usual output. The triple is returned as a column vector, which will allow it to be used as input to the `TVLCT` command that loads colors in the current color table. So, for example, if you are pre-loading a color table and you want to have yellow at color index 200 (regardless of the current color model), you can type code like this.

```
IDL> LoadCT, 0, /Silent
IDL> TVLCT, cgColor('yellow', /Triple), 200
```

In fact, multiple colors can be loaded by specifying a vector of color names, rather than a single color name.

```
IDL> TVLCT, cgColor(['teal', 'khaki', 'salmon'], $
      /Triple), 201
```

You can see what colors you currently have loaded in your color table by using the [Coyote Library](#) program `CIndex`. You will see the yellow, teal, khaki, and salmon colors loaded at color indices 200 to 203.

```
IDL> CIndex
```

To see all the `cgColor` colors loaded in the color table, starting at color index 32, type the following command, then click your cursor inside the `CIndex` window to update its display.



Figure 5: The *CIndex* program shows you the colors loaded in the current color table. Clicking anywhere inside the graphics window will refresh the window with the latest color table vectors.

```
IDL> TVLCT, cgColor(/All, /Triple), 32
```

The `cgColor` colors are loaded in indices 32 through 231.

Using `cgColor` With Color Table Indices

One of the major hurdles to overcome in making the transition to using decomposed color and having access to 16.7 million colors simultaneously is that so much software has been written assuming an indexed color model. This software loads a color table and then uses indices into that color table to specify the drawing colors.

For example, here is code that uses the three colors we just loaded into the color table.

```
IDL> TVLCT, cgColor(['teal', 'khaki', 'salmon'], $  
    /Triple), 201  
IDL> Plot, cgDemoData(1), Background=202, Color=201  
IDL> Oplot, cgDemoData(1), Color=203, PSym=2, $  
    Symsize=1.5
```

This kind of code only makes sense if we execute it using the indexed color model. If we execute it using the decomposed color model we get a window filled with a red color.

But, we can use `cgColor` to convert color table indices to any appropriate value by simply passing the color index as a string parameter. In other words, instead of passing the string “khaki” as the argument to `cgColor`, pass the string “202”. If `cgColor` reads the string as a “number” and needs a 24-bit value, it will create the value from the color triple loaded in the current color table at index 202.

In other words, these same commands written this way will work properly no matter which color model you are using.

```
IDL> TVLCT, cgColor(['teal', 'khaki', 'salmon'], $  
   /Triple), 201  
IDL> Plot, cgDemoData(1), Background=cgColor('202'), $  
   Color=cgColor('201')  
IDL> Oplot, cgDemoData(1), Color=cgColor('203'), $  
   PSym=2, Symsize=1.5
```

You will learn a great deal more about the benefits of using a program like `cgColor` to specify your graphics drawing colors in the chapters that follow, as we will make extensive use of it to write color model and graphics device independent IDL programs.

Working Around a Color Bug on UNIX Machines

There is a very, well, interesting color bug that manifests itself on UNIX machines, including Macintosh computers, in IDL versions 7 and 8, and perhaps earlier. I mention it here because there is a possibility you might run into it in the course of typing the commands in this book. You will notice you have a problem if you start an IDL session and run a graphics command that opens a graphics window with a white background color. The window will appear to be completely blank.

In fact, what is happening is that the system color variables, `!P.Background` and `!P.Color`, which determine the background and foreground drawing colors for plots, both get set to the color white. This happens on UNIX machines, when using the decomposed color model, and it only occurs for the first graphics window that opens in the IDL session.

You can solve the problem in one of two ways. First, simply closing the window and running the program that created the window again will solve the problem, because the problem only effects the first graphics window in the IDL session. Second, you can solve the problem in a more general

way, by simply opening a graphics window and then deleting it in your IDL start-up file. The graphics window can be a pixmap window (a window that exists only in the video RAM of your computer, not on the display). I recommend all UNIX users insert the following two lines of code into their IDL start-up files.

```
Window, XSize=10, YSize=10, /Pixmap, /Free  
WDelete, !D.Window
```

This will keep you from ever seeing this particular bug.

Color Models Affect Image Display

Probably the number one reason so many IDL users limit themselves to 256 colors by selecting the indexed color model in their IDL startup files is because the choice of color model also affects the display of 2D images with the IDL image display commands *TV* and *TVScl*. In particular, if you have the decomposed color model selected (it is the default color model, remember) and you load a color table, and display a 2D image, the image is *not* displayed in color. This is enormously frustrating to users!

Here are some commands you can type to see what I mean.

```
IDL> Device, Decomposed=1  
IDL> LoadCT, 22  
IDL> image = cgDemoData(7)  
IDL> TV, image
```

You see the result in Figure 6.

The image, which is supposed to be seen in nice pastel colors, is displayed instead in grayscale colors. And it doesn't matter what color table we load, all we can get out of this situation is grayscale colors.

To display the image correctly, we have to switch to the indexed color model.

```
IDL> Device, Decomposed=0  
IDL> TV, image
```

You see the result in Figure 7.

What accounts for this? I'm not sure. I've always thought IDL was "building" a 24-bit (also called a *true-color image*) image out of the 8-bit image by replicating the 8-bit image three times. Any 24-bit image of this type, in which all pixels have the same value in the red, green, and blue channel, will necessarily be displayed in grayscale.

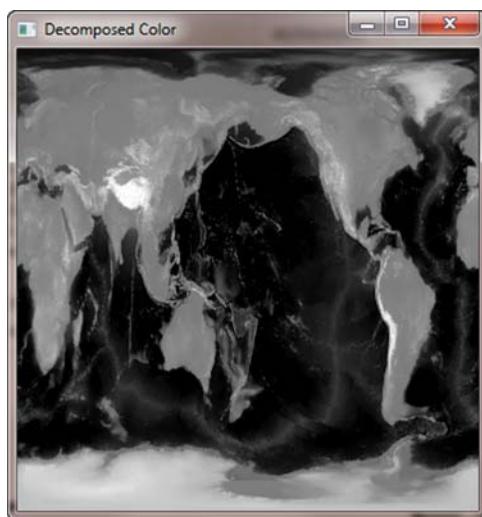


Figure 6: Even with color tables loaded, 2D images are displayed in grayscale unless the indexed color model is used.

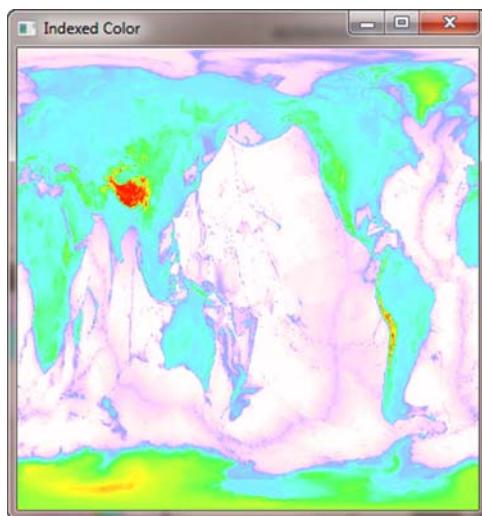


Figure 7: To correctly display 2D images, you must use the indexed color model.

Displaying 24-bit Images

But there is also a problem in how 24-bit images are displayed, at least on machines running Microsoft Windows operating systems. Consider this 24-bit rose image.

```
IDL> rose = cgDemoData(16)
IDL> Help, rose
ROSE           BYTE      = Array [3, 227, 149]
```

A 24-bit image (this one is pixel interleaved) has color information built into the image itself. It displays normally with a decomposed color model as shown in Figure 8.

```
IDL> Device, Decomposed=1
IDL> TV, rose, True=1
```

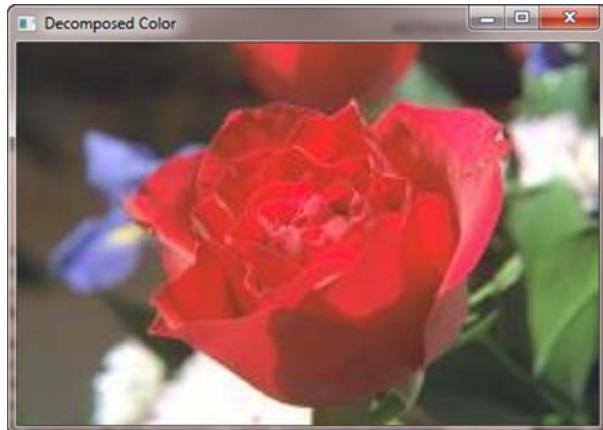


Figure 8: You have to use the color decomposition model to display 24-bit images correctly on Windows machines with color tables loaded, up until IDL 7. In IDL 7 and later, 24-bit images are always displayed correctly, no matter which color model is used.

But if we use the indexed color model, the image is displayed correctly on Windows machines (using versions of IDL prior to IDL 7) only if the gray-scale color table is loaded. (Since IDL 7, a 24-bit image is displayed correctly in either color model.) It displays incorrectly if any other color table is loaded, as shown in Figure 9. The image is always displayed correctly on UNIX machines. But, of course, UNIX users have to be aware of this to write machine portable IDL code. And, in any case, the same prob-

lem exists if you are displaying images in a PostScript file, as you will see later.

```
IDL> Device, Decomposed=0
IDL> Window, XSize=227*2, YSize=149, Title='Indexed'
IDL> LoadCT, 0, /Silent
IDL> TV, rose, True=1, 0
IDL> LoadCT, 22, /Silent
IDL> TV, rose, True=1, 1
```

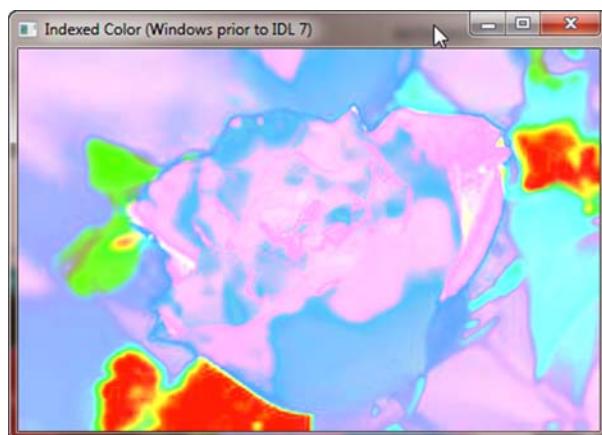


Figure 9: On Windows machines, in IDL versions prior to IDL 7, 24-bit images were passed, incorrectly, through the color table vectors when in indexed color mode, resulting in incorrect color output. This is still true today for the PostScript device.

What happens in this case is that the RGB values in the 24-bit image, which in fact represent the colors the user wants to display, are routed through the color table vectors to look up different RGB values for the display of the image. Yikes!

For this reason, and many others (which you will learn about in more detail in “Creating Image Plots” on page 209), a great many IDL users no longer use the built-in *TV* or *TVScl* commands to display images. Instead, they use one of several smart *TV* substitute commands that can be found in IDL libraries on the Internet. These commands determine which color model is in use at the time the command is used, switch to the proper model to display the image correctly, then switch back to the starting color model after the image is displayed. Thus, both 8-bit and 24-bit images are

always displayed in the proper color, regardless of the color model currently in effect.

Another advantage of these commands is that they can automatically determine how the three color planes of a 24-bit image are interleaved, so you don't have to worry about setting the *True* keyword to the correct interleave value with the *TV* command.

Two of the most popular of these substitute commands are [cgImage](#), a [Coyote Library](#) program, and *ImDisp*, a *TV* substitute command written by Liam Gumley and available from his web page.

<http://cimss.ssec.wisc.edu/~gumley/index.html>

I use [cgImage](#) almost exclusively in the code examples in this book to display images so I don't have to worry about which color model or IDL version you are using, and so I can know we are both seeing the same colors when an image is displayed.

Setting the Color Model

One of the goals of this book is to help you write device independent graphics programs. Another goal is to encourage you to use decomposed color whenever possible (i.e., pretty much always). These two goals are mutually exclusive for IDL programmers using versions of IDL older than IDL 7.1.1. (At least, if you define "device independent" as meaning the program produces identical results on the display, in a PostScript file, and in the Z-graphics buffer.)

The reason, of course, is that some devices are not compatible with the decomposed color model. The Z-graphics device didn't become compatible until IDL 6.4. The PostScript device didn't become compatible until IDL 7.1, and compatibility wasn't fully implemented until IDL 7.1.1 when the *Get_Decomposed* keyword worked correctly. The Printer device, as well as older graphics devices (e.g. CGM, HP, etc.), are still not compatible with the decomposed color model.

This means that you can't just issue the following command and expect it to work.

```
IDL> Device, Decomposed=1
```

The *Decomposed* keyword presumes the device is capable of using either the color decomposed model or the indexed color model. It is not an allowed keyword for those devices that do not yet recognize decomposed color.

This makes it difficult to write programs that are going to run on computers other than your own, because you don't always know what version of IDL the users of your programs have installed. In practice, it means writing a very long list of case statements to account for all the different devices and when they became compatible with 24-bit decomposed color.

As you can imagine, no one does this. Or, rather, if you are going to do it, you only want to do it one time and be done with it. I have done this for you in the [Coyote Library](#) command `SetDecomposedState`. This command allows you to set the color decomposition model of your choice (0 indicates indexed color, and 1 indicates decomposed color). The program takes into account all the device and version dependencies and sets the device to use decomposed color if it is possible to do so. For example, to use decomposed color, you type the following command. The *Current-State* keyword returns the current color decomposition state before it is changed, so the color decomposition state can be returned to its previous state later.

```
IDL> SetDecomposedState, 1, CurrentState=theState
```

After turning color decomposition on, we typically issue graphics commands, and, when we are finished, return to the previous state so we don't change the user's normal manner of working in IDL.

```
IDL> SetDecomposedState, theState
```

Note that just because you tried to turn color decomposition on, doesn't mean that it actually happened. We still have to write our programs to be color model independent, but at least we can take advantage of the color decomposed model if it is available. The primary advantages being, of course, that we have access to 16.7 million colors simultaneously and we don't have to load drawing colors in the color table, contaminating the very colors we want to use for other purposes. As time moves on, there will be fewer and fewer IDL users who cannot take advantage of decomposed color.

Obtaining the Color Model

There is also a problem obtaining the current color model state in a device independent way. That is to say, just as the device has to be 24-bit color compatible to be able to use the *Decomposed* keyword to set the color decomposition state, the device has to be 24-bit color compatible to use the *Get_Decomposed* keyword. But the situation is even more complicated than that.

For one thing, the PostScript device in IDL version 7.1 is 24-bit color compatible and accepts the *Decomposed* keyword, but the complementary *Get_Decomposed* keyword didn't work until IDL 7.1.1. For another, the following command works differently on Windows machines than it does on UNIX machines.

```
IDL> Device, Get_Decomposed=currentState, Decomposed=1
```

On Windows machines, the current color decomposition state is returned, and then color decomposition is turned on for the graphics device. On UNIX machines, the color decomposition state is turned on for the graphics device, and then it is *this* state that is returned in the *currentState* variable. Yikes!

This means the only way to get the current state of the graphics device and then set the state to something else, in a device independent way, is to use two commands to do so.

```
IDL> Device, Get_Decomposed=currentState  
IDL> Device, Decomposed=1
```

All of these dependencies have been encapsulated in the [Coyote Library](#) routine [GetDecomposedState](#).

```
IDL> currentState = GetDecomposedState()
```

It is this routine that is called from within [SetDecomposedState](#) to obtain the current state of the graphics device and return it in the output keyword *CurrentState*.

Loading Color Tables

IDL comes with a standard set of 41 color tables, found in the file *colors1.tbl*, which is located in the */resource/colors/* sub-directory of the IDL distribution. The files are normally accessed and loaded by either the *LoadCT* or *XLoadCT* command. The *LoadCT* command, which you have already used in this chapter, is normally used when you know exactly which color table you want to load. For example, to load the standard gamma II color table, which is color table index 5 in the *colors1.tbl* file, you would issue a command like this.

```
LoadCT, 5
```

Using the *LoadCT* command masks, to some extent, what is really happening in IDL. Since the *LoadCT* command is an IDL library file, you could open the file in a text editor and read the IDL code to find out what it does.

You would find that it reads three vectors from the color table file. We call these the red, green, and blue color vectors, and each vector contains 256 elements. Those vectors are loaded into the color table with the *TVLCT* command, which is the fundamental command for loading colors in IDL. The *TVLCT* command loads color vectors of any length from 1 to 256.

```
TVLCT, red, green, blue
```

Depending upon the size of the color table, which is always stored in the system variable *!D.Table_Size* (and is always 256 if you have a 24-bit graphics card), these color vectors are sometimes resampled before they are loaded. That is to say, if you had a color table with only 96 entries, these color vectors would be resampled to 96 colors, and those colors loaded with *TVLCT*. The resampling is done with the *Congrid* command. The resampling is a statistical process in which the end points are kept fixed, and colors (values, really) are dropped out of the larger vector in a more or less uniform manner, so that the reduced number of colors more or less represents the color range in the larger vector. So, for example, if you had 96 colors in your color table, or if you only want to use 96 colors in a particular color table, you can resample and load your color vectors, like this.

```
r = Congrid(red, 96)
g = Congrid(green, 96)
b = Congrid(blue, 96)
TVLCT, r, g, b
```

If you want to load those 96 colors, but you want to start loading them at color index 64, rather than zero, so that they were loaded at color indices 64 through 159, then you will use a fourth positional parameter to *TVLCT*, which is the starting color index number.

```
TVLCT, r, g, b, 64
```

As it happens, you can do the exact same thing, with the *LoadCT* command, by using the *NColors* and *Bottom* keywords. To see what I mean, start with the default grayscale color table (color index 0), and load the Hue-Sat-Value-2 color table (color table index 22) into the 96 color indices, starting at color index 64. View the results by using the *CIndex* command. You see the result in Figure 10.

```
IDL> LoadCT, 0
IDL> LoadCT, 22, NColors=96, Bottom=64
IDL> CIndex
```

Note: The *LoadCT* command will issue a message in the command log window whenever a new color table is loaded. I have always found this

more of an annoyance than a help, especially in widget programming. If you want to turn this message off, use the `Silent` keyword to the `LoadCT` command.

```
IDL> LoadCT, 0, /Silent
```



Figure 10: Colors in IDL color tables can be restricted to portions of the color table by using the `NColors` and `Bottom` keywords to `LoadCT`.

Loading Your Own Color Tables

There are times when the color tables supplied with IDL are not adequate for your purposes, or you want to use other color tables. Both `LoadCT` and `XLoadCT` have a `File` keyword that you can use to load your own color table file.

For example, the [Coyote Library](#) that you downloaded to use with this book contains a file with 27 Brewer color tables. Brewer colors were designed by Cynthia Brewer (<http://colorbrewer2.org/>) to work particularly well on maps. This color table file is named `fsc_brewer.tbl`. Both diverging and sequential color tables are available. You can load the Brewer color tables with a command like this, which also uses the

`Find_Resource_File` function from the [Coyote Library](#) to locate the file wherever it might be in your IDL path.

```
IDL> brewerFile = Find\_Resource\_File('fsc_brewer.tbl')
IDL> XLoadCT, File=brewerFile
IDL> LoadCT, 4, File=brewerFile
```

The IDL programs *XLoadCT* and *LoadCT* can work with just a single color table file at a time. Sometimes it is nice to be able to display either the IDL or Brewer (or some other!) color table file. The [Coyote Library](#) routines *XColors* and *cgLoadCT* are drop-in replacements for *XLoadCT* and *LoadCT* and have the Brewer color tables always available, in addition to the normal IDL color tables. We will often use these program in the examples in this book because of their additional functionality. To use Brewer color tables, set the *Brewer* keyword.

```
IDL> cgLoadCT, 4, /Brewer
IDL> XColors, /Brewer, Index=4
```

For example, if you want to reverse a color table using *LoadCT*, you would do it like this.

```
IDL> LoadCT, 5
IDL> TVLCT, r, g, b, /Get
IDL> TVLCT, Reverse(r), Reverse(g), Reverse(b)
```

With *cgLoadCT*, this can be done in a single statement.

```
IDL> cgLoadCT, 5, /Reverse
```

With *XColors*, you can flip between IDL and Brewer color tables, reverse color tables on the fly, and use the program to communicate with other IDL programs in a variety of ways, all impossible to do with *XLoadCT*. You see in Figure 11 an example of how *XColors* appears on the display.

Creating Your Own Color Tables

It is easy in IDL to create your own color tables. First, I'll show you how to construct a simple color table. Then I'll show you how to extend the ideas behind the simple color table to construct any kind of color table you like.

Suppose we want a color table that runs from a yellow color in the first index to a red color in the last index. In terms of color triples, we want a color table that goes from [255, 255, 0] to [255, 0, 0]. You already know that a color table is made up of three vectors, containing the values for the red, green, and blue portion of a specific color. And, in most color tables, we would like a smooth progression from one value to the next, until we reach the final value.

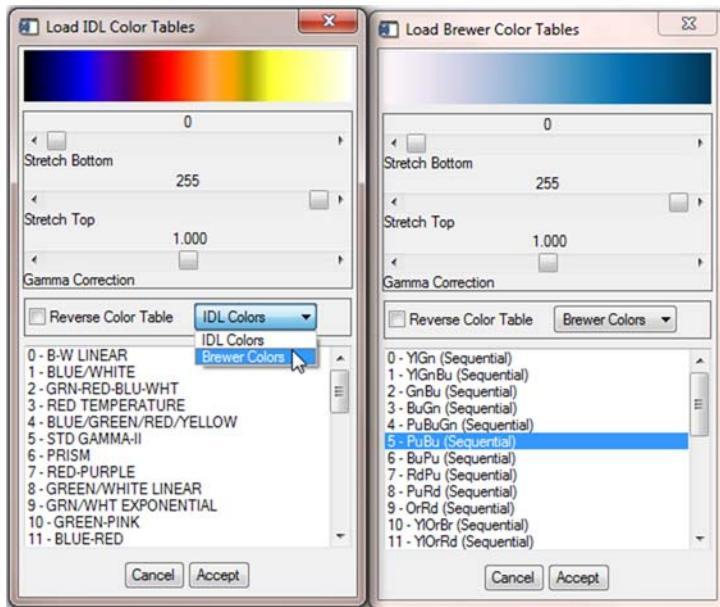


Figure 11: XColors has more features than XLoadCT and allows the user to access both the IDL and Brewer color tables simultaneously.

What would constitute a smooth progression of colors? We see that for each of the red, green, and blue vectors we must go from the starting value in that vector to the ending value in that vector. And we must do it in some arbitrary number of steps that will be the size of our color vector. We can write a general expression for the vector that looks like this.

```
vector = beginNum + ((endNum - beginNum) * scaleFactor)
```

where we define the beginning number, the ending number, and the scale factor, which will depend upon the number of steps we want to take in getting from the beginning to the ending number.

Suppose we define these quantities like this.

```
IDL> beginNum = 10.0
IDL> endNum = 20.0
IDL> steps = 5
IDL> scaleFactor = FIndGen(steps) / (steps - 1)
```

Then using the equation above, we print the vector values.

```
IDL> Print, beginNum + ((endNum - beginNum) * $  
scaleFactor)  
10.0000 12.5000 15.0000 17.5000 20.0000
```

This looks right, so let's apply it to our color table problem. The red vector must go from 255 (the red value in the yellow color) to 255 (the red value in the red color). The green vector must go from 255 to 0. And the blue value must go from 0 to 0.

The red and blue vectors are extremely simple, since their values don't change. We can use the *Replicate* command to create those vectors. We will have to use our formula for the green vector, however. Here is the code to create a color table 256 elements in length.

```
IDL> steps = 256  
IDL> rVec = Replicate(255, steps)  
IDL> bVec = Replicate(0, steps)  
IDL> scaleFactor = FIndGen(steps) / (steps - 1)  
IDL> beginNum = 255 & endNum = 0  
IDL> gVec = beginNum + ((endNum - beginNum) * $  
scaleFactor)
```

Finally, load the color table vectors you created with *TVLCT*, and display an image that uses these 256 colors.

```
IDL> TVLCT, rVec, gVec, bVec  
IDL> Window, XSize=256, YSize=40, Title='Color Table'  
IDL> cgImage, BIndGen(steps) # Replicate(1B, 40)
```

You see the result in Figure 12.



Figure 12: A yellow-red color table constructed with IDL commands.

Using these principles you can construct as complicated a color table as you like. For example, suppose you want a 256 element color table that goes from yellow to red, as before, but you want it to go through a series of blue colors in the middle of the table. You simply break this down into two problems, each with 128 steps, that are similar to the first example. In

other words, in 128 steps go from yellow [255, 255, 0] to blue [0, 0, 255], and then in 128 more steps from blue to red [255, 0, 0]. The code looks like this.

```
IDL> steps = 128  
IDL> scaleFactor = FIndGen(steps) / (steps - 1)
```

Set up the first 100 steps, going from yellow to blue.

```
IDL> rVec = 255 + (0 - 255) * scaleFactor  
IDL> gVec = 255 + (0 - 255) * scaleFactor  
IDL> bVec = 0 + (255 - 0) * scaleFactor
```

Now do the second 128 steps, going from blue to red.

```
IDL> rVec = [rVec, 0 + (255 - 0) * scaleFactor]  
IDL> gVec = [gVec, Replicate(0, steps)]  
IDL> bVec = [bVec, 255 + (0 - 255) * scaleFactor]
```

Load the color vectors into the color table, and display an image using the colors.

```
IDL> TVLCT, rVec, gVec, bVec  
IDL> Window, XSize=256, YSize=40, Title='Color Table'  
IDL> cgImage, BIndGen(steps*2) # Replicate(1B,40)
```

You see the result in Figure 13.



Figure 13: A more complicated color table, running through blue colors in the center of the table.

Note: The IDL command `XPalette` allows you to create color tables by doing exactly this kind of interpolation between color values interactively. But I think it always helps to know what it is doing.

It is even possible to do a piecewise interpolated color table. Consider for example, that you would like a color table that shows particular colors at

particular color indices and you would like smooth transitions between them. Suppose you want the colors shown in Table 1.

Color Index	Red	Green	Blue
0	0	0	0
28	100	30	150
46	120	20	40
89	20	20	255
153	0	200	230
255	50	220	80

Table 1: A more complicated color table in which colors should flow smoothly from one specified index and color to the next.

In this case, we use the IDL command *Interpol* to perform the piecewise interpolation from one value to the next. The code looks like this.

```
IDL> r = Interpol([0, 100, 120, 20, 0, 50], $  
[0, 28, 46, 89, 153, 255], Findgen(256))  
IDL> g = Interpol([0, 30, 20, 20, 200, 220], $  
[0, 28, 46, 89, 153, 255], Findgen(256))  
IDL> b = Interpol([0, 150, 40, 255, 230, 80], $  
[0, 28, 46, 89, 153, 255], Findgen(256))  
IDL> TVLCT, r, g, b  
IDL> Window, XSize=256, YSize=40, Title='Color Table'  
IDL> cgImage, BIndGen(steps*2) # Replicate(1B,40)
```

You see the result in Figure 14.



Figure 14: An even more complicated, piecewise interpolated color table built with simple IDL commands.

Saving a Color Table

Before you can save a color table, you have to be able to obtain the RGB vectors that represent the color table. You may have created the vectors yourself, as above, or you may have created the color table by manipulating the color vectors interactively. (For example, you might have used [XColors](#), [XLoadCT](#), [XPalette](#), or other tools to manipulate the color table vectors by stretching the top or bottom of the color table, or by manipulating the gamma correction.)

If you manipulated the color table interactively, you can obtain the RGB vectors currently loaded in the color table by using the *Get* keyword to *TVLCT*. The vectors will be returned in the first three positional parameters. That is to say, the first three positional parameters will be output variables, rather than the input variables they are normally when you use the *TVLCT* command.

```
IDL> TVLCT, rVec, gVec, bVec, /Get
IDL> Help, rVec, gVec, bVec
      RVEC           BYTE      = Array[256]
      GVEC           BYTE      = Array[256]
      BVEC           BYTE      = Array[256]
```

These vectors contain as many elements as your color table (see [!D.Table_Size](#)), and will typically be 256 elements in length if you are using a 24-bit graphics card.

The simplest way to save these RGB vectors so they can be recalled later is to use the *Save* command. The vectors, including their names (*rVec*, *gVec*, and *bVec*), are saved in a machine-portable binary format (XDR) so they can be restored on any machine or platform running IDL.

```
IDL> Save, rVec, gVec, bVec, $
      Filename='mycolortable.sav', $
      Description='Yellow-Blue-Red Color Table'
```

When you want to use the color table, restore the variables and load them into the color table.

```
IDL> Restore, Filename='mycolortable.sav', $
      Description=desc
IDL> IF desc NE '' THEN Print, desc
      Yellow-Blue-Red Color Table
IDL> TVLCT, rVec, gVec, bVec
```

Another way to save the vectors is to simply write them to a file. I recommend that you use the XDR binary format and that you write the size of the

vectors into the file first, so you can recreate the vectors in the correct size when you read them back out.

```
IDL> OpenW, lun, 'mycolortable.tbl', /Get_Lun, /XDR
IDL> WriteU, lun, N_Elements(rVec), rVec, gVec, bVec
IDL> Free_Lun, lun
```

To read the vectors out of the file, you write code similar to this.

```
IDL> OpenR, lun, 'mycolortable.tbl', /Get_Lun, /XDR
IDL> theSize = 0L
IDL> ReadU, lun, theSize
IDL> rVec = BytArr(theSize)
IDL> gVec = (bVec = rVec)
IDL> ReadU, lun, rVec, gVec, bVec
IDL> Free_Lun, lun
IDL> TVLCT, rVec, gVec, bVec
```

A third way to save a color table is to use the *ModifyCT* command to either add a color table to the color table file (there is an upper limit of 256 color tables) or substitute your color table for one of the 41 color tables in the *colors1.tbl* file distributed with IDL. You will need administrator privileges to modify this file, but if you don't have them you can always copy this file to another file name and change the modified file. Load the modified file instead of the one distributed with IDL by using the *File* keyword with *LoadCT*, *XLoadCT*, *XColors*, *cgLoadCT*, etc.

Suppose, for some reason, we wanted to have a 256 element color table in which the first 128 colors were grayscale colors, and the next 128 colors were an orange color table, going from orange [255, 165, 0] to white [255, 255, 255]. We could construct such a color table like this.

```
IDL> LoadCT, 0, NColors=128 ; Indices 0 to 127
IDL> steps = 128
IDL> scaleFactor = FIndGen(steps) / (steps - 1)
IDL> rVec = Replicate(255, steps)
IDL> gVec = 165 + ((255 - 165) * scaleFactor)
IDL> bVec = 0 + ((255 - 0) * scaleFactor)
IDL> TVLCT, rVec, gVec, bVec, 128
```

And we could exchange this for the Prism color table (a vile, nasty color table, at least for teaching purposes!) in the normal IDL distribution. The Prism color table is index number 6. (Have you made a backup copy of *color1.tbl* in case something goes drastically wrong in the next few minutes? I'd recommend it.) First, be sure you get the current color table vectors you just loaded into the color table.

```
IDL> TVLCT, r, g, b, /Get  
IDL> ModifyCT, 6, 'GREY-ORANGE', r, g, b  
IDL> XColors
```

You see what the color table looks like in Figure 15.

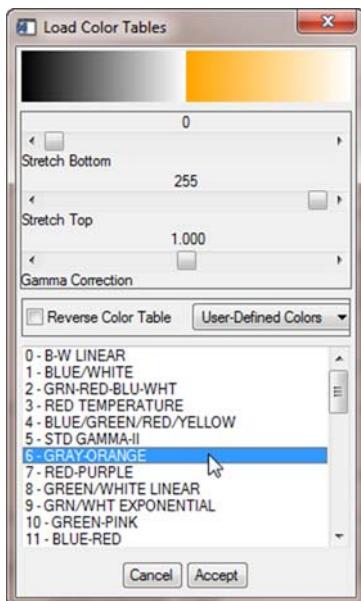


Figure 15: The Prism color table is replaced by one of our own making using *ModifyCT*.

If you prefer to add a color table to the file, rather than replacing a current color table, just give *ModifyCT* the next highest number in the color table. In this case, number 41.

```
IDL> ModifyCT, 41, 'GREY-ORANGE', r, g, b
```

Using Other Color Systems

While colors in IDL are always expressed as RGB values, and we must load RGB vectors into the color table, it is sometimes useful to express colors in other color systems. IDL also supports the HLS (hue, lightness, saturation) and HSV (hue, saturation, value) color systems. Colors in these systems are created with the *HLS* and *HSV* commands, respectively. Both

of these commands load the color table that results from calling them into the current color table. And both return, in an optional parameter, the color system values converted to RGB values so these can be saved, reused, and so forth.

HLS Color System

The HLS color system is sometimes also referred to as the HSL (hue, saturation, lightness or luminosity) or HSI (hue, saturation, intensity) color system. The system is typically drawn as a double cone or spiral, and is (like the HSV system) a non-linear deformation of the normal RGB color cube. You see an example of an HLS color cone or spiral in Figure 16.

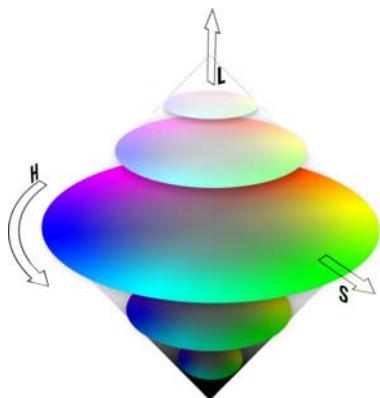


Figure 16: A model of the hue, lightness, saturation (HLS) color system displayed as a spiral.

In IDL we specify the starting hue, which is a number from 0 to 360 (red equals 0, green equals 120, and blue equals 240) and indicate how many times we want to loop through the color spiral. In addition, we specify the starting and ending lightness (a number from 0 to 100) and saturation (also a number from 0 to 100) values. The command looks like this.

```
HLS, light1, light2, sat1, sat2, hue, numloops, rgb
```

The *rgb* parameter is an output parameter that will contain a 256x3 array of RGB values that was loaded in the color table.

Here is code for a typical color table using the HSL color system. The output is shown in Figure 17.

```
IDL> HLS, 0, 100, 50, 100, 0, 1, rgb
```

Color Index Numbers																									
Change Colors																									
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Figure 17: A color table built from specifying hue, saturation, and lightness.

HSV Color System

The HSV color system is often preferred by artists because of its similarities to the way humans perceive color. It is often visualized as a conical object in which the value is a number from the tip of the cone to the flat base, saturation is the distance from the center axis of the cone, and hue is the rotation about the cone. You see a representation of the HSV color system in Figure 18.¹

To produce a green temperature scale color table in the HSV color system, you would type a command like this.

```
IDL> HSV, 0, 100, 0, 100, 120, 0, rgb
```

If you receive HLS or HSV values from elsewhere, and you want to load them into an IDL color table, you can use the *Color_Convert* command to convert values in these systems into RGB colors, and vice versa. Your code will look something like this.

1. Image downloaded from Wikipedia and used under the terms of the GNU Free Documentation License.

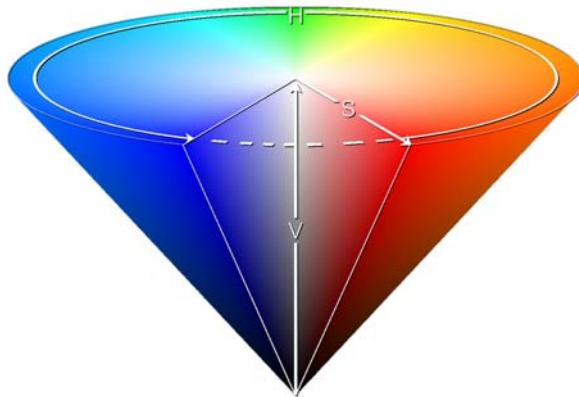


Figure 18: A model of the hue, saturation, value color system. This model is most closely related to how humans perceive color.

```
Color_Convert, hue, sat, light, r, g, b, /HSV_RGB
```

The first three parameters are input parameters, and the next three are output parameters containing the vectors after conversion. You must set the proper keyword to switch from one color system to another. See the on-line help for *Color_Convert* for more details.

Chapter 4



Creating Line Plots

A Basic Line Plot

Perhaps the most basic of the traditional graphics commands is a line plot. We collect or obtain some data and we want to see how that data varies with respect to another data set. Sometimes the second data set is nothing more than “time,” or the order in which we collected the data. In any case, we plot one data against the second, and what we end up with is a line plot that gives us some indication of the relationship between the two variables. A line plot is created in IDL with the *Plot* command.

Line Plot Colors

We are going to spend a lot of time talking about the details of line plots in this chapter. For now, though, let’s just draw the simplest line plot and talk about some of its properties. For example, you can type this code at the IDL command line to see a plot of a data vector.

```
IDL> vector = [3, 5, 2, 9, 10, 12]
IDL> Plot, vector
```

It is extremely likely that your graphics display window shows a line plot drawn in white on a black background. This is the default color scheme for IDL traditional graphics commands. You see an example in Figure 1.

This is an unfortunate default for graphics output and is just the opposite of most modern graphics systems, which normally display black lines on a white background. And, of course, this is trouble if we want to save our graphics in a form we can print on a piece of paper, since the black back-

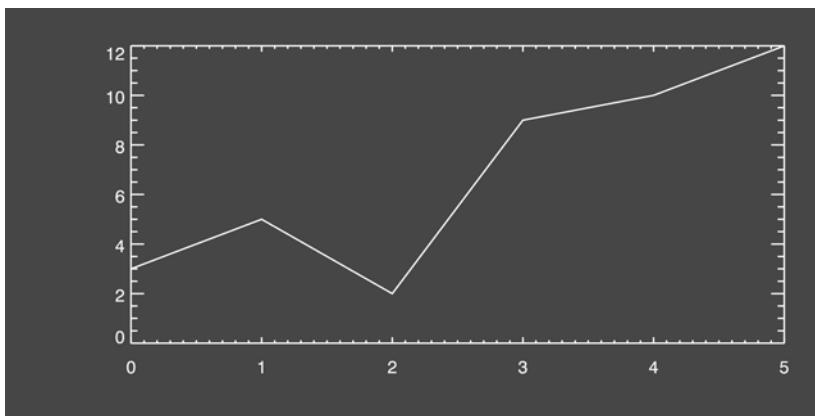


Figure 1: A traditional line plot in the white on black color scheme.

ground will quickly send printing expenses through the roof, with cases of black printer cartridges required to keep up with demand.

There are further complications. For example, PostScript output in IDL has the opposite convention. It uses black lines on a white background. So if we want to write graphical programs that both run on our display *and* work as PostScript output we are going to find it extremely difficult to select drawing colors that work well for both types of output. Black backgrounds, of course, require light drawing colors, and white backgrounds require dark drawing colors. As you will see later in the book, producing PostScript output is essential for creating high quality output for presentations, web and book illustrations, and journal submissions. (See “Creating PostScript Output” on page 333.) It is critical that we learn to write programs that work equally well on the display and in a PostScript file.

Since one of my goals is to show you how to write an IDL graphics program that works on all devices all the time, we need to eliminate this black background inconvenience as soon as we can. Unfortunately, many of the ways people have been taught to solve this problem in the past have made it harder, rather than easier, to write device-independent programs.

How IDL Chooses Drawing Colors

The choice of which color to use for line plots and other graphics is set by the IDL system variable `!P`. This is the *plotting* system variable and it contains fields that supply default values for many aspects of line plots and

other graphics commands. To see a list of the fields in the *!P* system variable, you can type the *Help* command, like this.

```
IDL> Help, !P, /Structure
```

In the list of plot properties, you will see two of interest to us currently: *Color* and *Background*. The *!P.Color* field determines the color that will be used for drawing the plot itself. In more recent versions of IDL, it will be set to the value 16777215. In older versions of IDL it was set to the value 255. Of course, the value 16777215 is a 24-bit value that can be decomposed into three 8-bit values. Those three values are 255, 255, and 255. In other words, the 24-bit value for the color white. (See “Understanding IDL Color Models” on page 36 for more information about decomposed and indexed color models.)

If you are using the indexed color model in IDL, rather than the default decomposed color model, the value 16777215 will be read by IDL as the color index 255 (it will read just the first 8 (or red) bits of the 24-bit value. This color index also points to a white color in most of the color tables supplied with IDL.

The *!P.Background* field is set to the value 0 by default. This is the 24-bit value for black if using the decomposed color model, and is also the index for the color black in most color tables supplied with IDL. Thus, when most of us enter IDL and draw a line plot, we see a white plot on a black background.

One common way to solve the black background problem, then, is to put yourself in indexed color mode, where *!P.Color=255* (essentially) and *!P.Background=0*, and then simply reverse the color table, so that the color loaded at color table index 255 is black and the color loaded at index 0 is white.

```
IDL> Device, Decomposed=0 ; Index color model.  
IDL> LoadCT, 0 ; Normal black to white color table.  
IDL> TVLCT, r, g, b, /Get  
IDL> TVLCT, Reverse(r), Reverse(g), Reverse(b)  
IDL> Plot, vector
```

This produces the black on white plot you were expecting, as shown in Figure 2, below. (This figure was actually produced as a PostScript file which uses the black on white color scheme by default, but you should be seeing the same thing on your display now.)

All is well *until* you try to produce high-quality PostScript output with this code. The PostScript device will switch the *!P.Color* and *!P.Background*

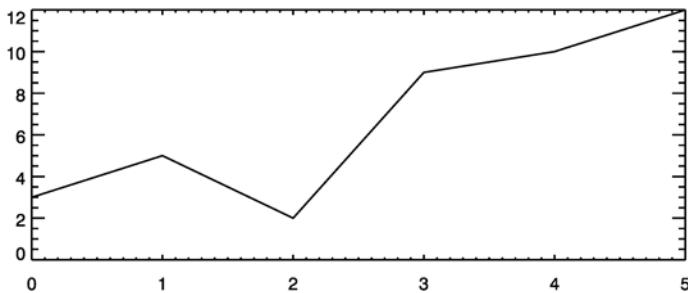


Figure 2: A simple line plot with the vector drawn in black on a white background.

indices (i.e., `!P.Color` will now be set to index 0 and `!P.Background` will now be set to index 255), and then will completely ignore the `!P.Background` value! In PostScript, the joke goes, you can have any background color you like, as long as you *really* like white.

What this means, of course, is that you will be drawing a white plot on a white background, and your PostScript output will be, uh, a little hard to see. In fact, what you *will* see coming out of your PostScript printer is what appears to be a blank sheet of paper!

We will talk about producing PostScript output in detail later. And you will see that you *can* have background colors other than white in PostScript, although you will have to go to some trouble to do so. My point here is that if you want to write IDL code that is device independent (and, believe me, you do!), this common method of producing graphics windows with white backgrounds is *not* the way to do it.

Where to Load Drawing Colors

The black and white colors we are talking about here are what I like to call *drawing colors*. Drawing colors are the handful of colors we use, as in this case, to draw graphics, to annotate plots, to draw attention to details, and to otherwise add some life to drab graphical output. Typically, you will need no more than five or ten drawing colors. Some-times we need more.

One of the secrets of producing device independent graphics programs in IDL is expressed in Coyote's Rule for Drawing Colors: *Load drawing colors wherever you like in the color table. But don't, under any circumstances, load them at color table index 0 or 255!*

Another way of saying this is you want to be very, very careful if you are going to use `!P.Color` and `!P.Background` to specify your drawing colors,

since these normally point to exactly the wrong indices to be helpful to you.

Truth be told, I am hoping to convince you to *never* load drawing colors in the color table, unless it is absolutely necessary. And sometimes it *is* necessary. For example, you might be dealing with IDL graphics commands that predate the 1980s (e.g., the *Shade_Surf* command) or you might work with colleagues who learned to write programs from their graduate student advisor, who learned to write programs in the 1970s. Whatever it is, there are occasions when you need to load drawing colors in the color table. Just don't ever, if you can help it, load them at indices 0 or 255. It will make your life considerably easier.

Producing Black on White Plots

Okay, so what is the proper way to produce black on white plots that *are* device independent?

As usual in IDL, there are several ways to solve this problem. For example, we could just point *!P.Color* and *!P.Background* to color indices (assuming we are still using the indexed color model) that are not 0 and 255, where we have loaded the black and white drawing colors. In this example, we load white at color 253 and black at color 254 and make these system variables point to the proper colors.

```
IDL> Device, Decomposed=0
IDL> TVLCT, [255,0], [255,0], [255,0], 253
IDL> !P.Background = 253
IDL> !P.Color = 254
IDL> Plot, vector
```

This code works for all IDL graphics devices to produce black plots on white backgrounds.

Or, if we don't want to load drawing colors in the color table at all, we can use the decomposed color model and let these system variables specify the colors black and white directly.

```
IDL> Device, Decomposed=1
IDL> !P.Background = 'FFFFFF'xL
IDL> !P.Color = '000000'xL
IDL> Plot, vector
```

This code works correctly on all 24-bit graphics devices, but not on 8-bit graphics devices. And, unfortunately, IDL still has a number of 8-bit graphics devices. For example, until IDL 7.1 the PostScript device was an 8-bit device and used only the indexed color model. Even today, that is the default setting for the PostScript device. The Z-graphics device is still an 8-bit device in its default mode.

It would be nice if we could set these system variable colors to the correct 8-bit or 24-bit value, depending upon the device we are using. Fortunately, that is exactly what the [Coyote Library](#) routine `cgColor` is designed to do.

If we set the system variables immediately before we draw the line plot, then `cgColor` will supply them with the proper value for the current graphics device. We no longer have to worry about which device, or even which color model we are currently using. `cgColor` takes care of all of those details for us.

```
IDL> !P.Background = cgColor('white')
IDL> !P.Color = cgColor('black')
IDL> Plot, vector
```

Of course, with this construction we have the additional benefit of knowing what colors we are trying to use, since we can read the names of the colors in our code.

Color Keyword Equivalents to System Variables

The problem with using system variables to set drawing colors is that system variables are global variables. And once set they remain in effect until they are specifically changed or you exit IDL. It is easy to forget what values they are set to. So, for example, if you set them when you are using the decomposed color model, and then you switch to the indexed color model and forget to reset them, your plots will certainly be displayed with incorrect colors, and vice versa.

Note: It is much easier to get colors right if you use the keyword equivalents to these system variables.

The fields of the plotting system variable, `!P`, (as well as the axes system variables, `!X`, `!Y`, and `!Z`) nearly always have keyword equivalents that can be used directly on the graphics command to set properties of the plot. If the system variable property is different from the keyword property, the keyword property prevails.

In this case, for example, we can produce the device independent, black on white plot we desire in a single command, by using the keywords *Color* and *Background*.

```
IDL> Plot, vector, Color=cgColor('black'), $
      Background=cgColor('white')
```

In my own IDL programming, this is how I have produced black on white plots in 99 percent of the programs I have ever written. These days, I create plots with the [Coyote Library](#) program `cgPlot`, which also uses this

method to create black on white plots as the default color scheme. (You will learn more about this program in the section “Using a Refurbished Line Plot Command” on page 116.)

This said, I am not going to use these two keywords for the most part in the examples that follow in this chapter. I am going to assume you have selected one or the other of the color models to work in (and I *highly* recommend you use the default decomposed color model) and that you have set your drawing color system variables appropriately. This way, I can avoid typing the *Color* and *Background* keywords on each and every command I illustrate in this book. I am going to assume you are seeing black on white illustrations, as shown in this book and in Figure 2.

```
IDL> Device, Decomposed=1
IDL> !P.Background = cgColor('white')
IDL> !P.Color = cgColor('black')
```

Alternatively, you can substitute `cgPlot` for the `Plot` command in any of the examples in this chapter, if you would prefer to do that. The `cgPlot` command draws in black on a white background by default.

Specifying Data Colors

You will notice in Figure 2 that the data is drawn in the same color as the plot axes. The `!P.Color` system variable, or the *Color* keyword sets the color for the axes, the data, the axis annotations, and the plot title. Everything but the background is set to the plot color.

Sometimes we would prefer that the data be plotted in a different color from the axes and annotations. To do this in traditional graphics, we have to draw the axes and annotations in one color, and then overplot the data in another color. This requires that we use the *NoData* keyword on the `Plot` command to suppress the drawing of the data, and then overplot the data with the `Oplot` command. The code looks like this.

```
IDL> Plot, vector, Color=cgColor('black'), /NoData
IDL> Oplot, vector, Color=cgColor('red')
```

Colors can be layered on. For example, we can add green symbols to indicate the actual data values in the plot.

```
IDL> Oplot, vector, Color=cgColor('Forest Green'), $
      PSym=2, SymSize=1.5
```

You see the result in Figure 3.

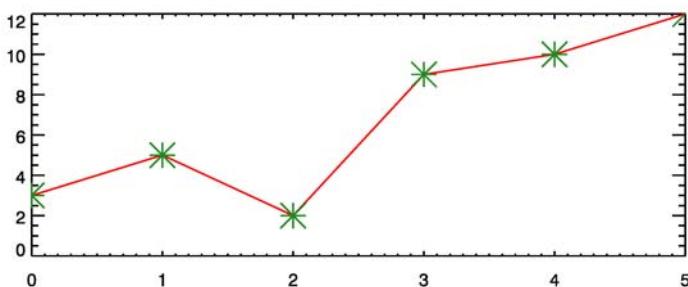


Figure 3: Axes and axis annotations are drawn in one color with the *Plot* command, while data is over-plotted onto the axes in other colors with the *OPlot* command.

Plotting Data with Line Plots

When we plot a vector as in Figure 3, the X or horizontal axis represents the independent data, and the Y or vertical axis represents the dependent data. If we only supply one argument or positional parameter to the *Plot* command, that argument is taken to be the dependent data set.

That is why the Y axis is labelled from 0 to 12. It represents the range of the dependent data, which in this case is actually 2 to 12. If the minimum value of the dependent data is greater than zero, then the Y axis will always start at zero unless you specifically indicate not to. For example, if your dependent data was in the range 1200 to 1400, you would probably not want your Y axis to start at zero, you would prefer that it start at, say, 1200. You could suppress this default behavior by setting the *YNoZero* keyword on the *Plot* command. Here are two plots, drawn one above the other to show you the difference. (We use the system variable *!P.Multi* to accomplish this multiple plot feat, but we don't discuss *!P.Multi* until later in “Creating Multiple Line Plots” on page 93.)

```
IDL> !P.Multi = [0,1,2]
IDL> data = [1230, 1367, 1286, 1300, 1375]
IDL> Plot, data
IDL> Plot, data, /YNoZero
IDL> !P.Multi = 0
```

You see the result in Figure 4. Notice that the upper Y range in the two plots is different, too, not just the lower Y range. We will discuss why this is the case in just a moment.

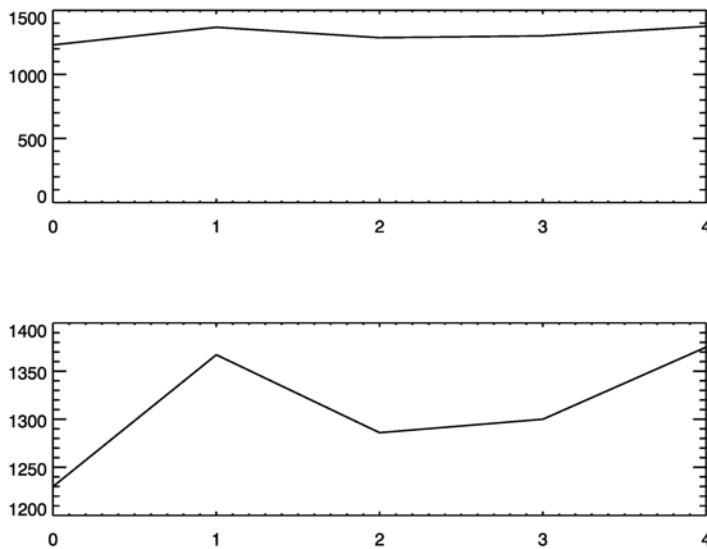


Figure 4: The plot on the top is a normal IDL plot. The plot on the bottom was created with the `YNoZero` keyword set. Notice that the plots have different dependent data ranges.

But, let's go back to our original vector again now, so we can learn just a bit more about how the simplest *Plot* command works.

Normally, in science, we are not interested in plotting just a single variable. We are interested in plotting one variable against another to see the relationship between them. When we plot two variables, one is designated the independent variable, and one is designated the dependent variable. In IDL the independent variable is plotted on the X axis and the dependent variable is plotted on the Y axis.

In fact, in IDL it is not possible to plot a single variable. The *Plot* command *always* plots a dependent variable against the independent variable. However, if you don't supply the independent variable, IDL has to create one to plot the dependent variable against. The one it creates always has the same number of elements as the dependent data, and it is a vector that starts at 0 in its first element and increases by one for each successive element.

In the case of our dependent vector (named *vector*), IDL creates the independent vector by creating, essentially, a command like this.

```
IDL> indepVector = IndGen(N_Elements(vector))
```

The *IndGen* command (*Index Generator*) creates a vector of five elements (the number of elements in *vector*), ranging in value from 0 to 4. Explicitly, the *Plot* command looks like this.

```
IDL> Plot, indepVector, vector
```

Note the positional order of the two vectors. If you supply both the independent and dependent vector, then the independent vector is the first positional parameter and the dependent vector is the second. If you supply only the dependent vector (as we did before), then IDL assumes the first positional parameter is the dependent data and it creates the independent data vector itself. In either case, the X axis is labeled according to the values of the independent data vector, which in the default case always extends from 0 to one less than the number of elements in the dependant data vector.

Normally, the dependent data is collected in some sort of “experiment.” I use the term loosely to describe some kind of data collection operation. Often, data is collected at some specific location, or over some specific time, or jointly with another variable (for example, wind speed and direction). Suppose, for example, this vector data was collected over a period of 15 minutes in regular intervals. We might want to create an independent variable that showed the elapsed time of the experiment. We can create the vector like this.

```
IDL> elapsedTime = (IndGen(N_Elements(vector)) * 15.0) $  
      / (N_Elements(vector)-1)  
IDL> Print, Min(elapsedTime), Max(elapsedTime)  
0.0000  15.0000
```

Now, we can use this vector in our plot.

```
IDL> Plot, elapsedTime, vector, XTitle='Elapsed Time', $  
      YTitle='Signal Strength', Title='Experiment 1A'
```

Notice in Figure 5 that we are now annotating the axes with text by using the *XTitle* and *YTitle* keywords to the *Plot* command. We have also put a title on the overall plot with the *Title* keyword.

It is now obvious that the X axis is labeled according to the values of the independent data vector. Adding axes annotations and a plot title is one way we can customize line plots, which is discussed in detail in the next section.

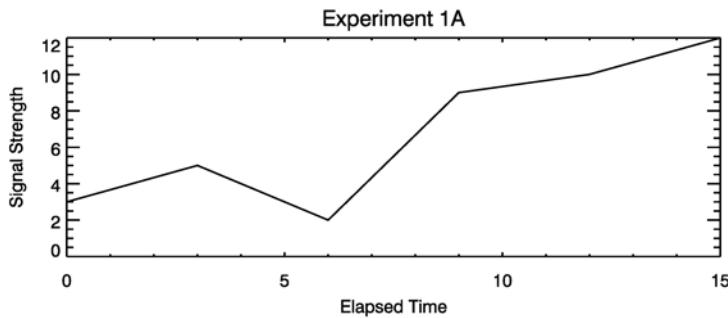


Figure 5: The independent data, `elapsedTime`, plotted against the dependent data. We have used keywords to add axis annotations to the plot.

Annotating Line Plots

If you reference the `Plot` command in the IDL on-line help, you will find there are 50-60 different keywords that will modify the appearance of a line plot. You will want to memorize a dozen or so of these (happily most of these will apply to other graphics commands, too!), and there are maybe a dozen more you will want to vaguely remember are available. The rest you will probably never use in a long IDL programing career. I'll introduce you to the most important ones in the next few pages.

You can't do everything with traditional graphics commands in IDL. But I find most people are surprised with what you *can* do with them. If you have something in mind, I encourage you to look through the keywords available. You may be surprised at what you find.

Line Plot Titles

Probably the most often used keywords are those that label axes, such as those you just added in Figure 5. In addition to labeling the X and Y axis of a line plot, you can also label the plot itself by means of the `Title` keyword.

```
IDL> Plot, elapsedTime, vector, XTitle='Elapsed Time', $  
YTitle='Signal Strength', Title='Experiment 1A'
```

Several auxiliary keywords are used in conjunction with titles. These keywords typically set the size of the titles. The first keyword you need to know is the `Charsize` keyword. The `Charsize` keyword has its counterpart in the `!P.Charsize` system variable and it sets the character size of the text

used for the plot annotations. Actually, it sets the character size for the axis annotations. The plot title will always be set to 1.25 times the character size. The default character size is 1.0. If you set *Charsize*, for example, to 0.8, your axes annotations are now four-fifths the normal size, and the plot title is now set to 1.0.

The keywords */XYZ/Charsize* apply to the character size of the axis annotations themselves. But, what you have to realize is that these sizes are not absolute sizes, like *Charsize*, but rather they modify the *Charsize* value. So, for example, if you want the plot title to be the same size as the axes annotations, you might try something like this.

```
IDL> Plot, elapsedTime, vector, XTitle='Elapsed Time', $  
      YTitle='Signal Strength', Title='Experiment 1A', $  
      Charsize=1.0, XCharsize=1.25, YCharsize=1.25
```

The axes labels in the following command will be the same size as the plot title, no matter what character size you choose.

```
IDL> Plot, elapsedTime, vector, XTitle='Elapsed Time', $  
      YTitle='Signal Strength', Title='Experiment 1A', $  
      Charsize=0.75, XCharsize=1.25, YCharsize=1.25
```

You see the result in Figure 6.

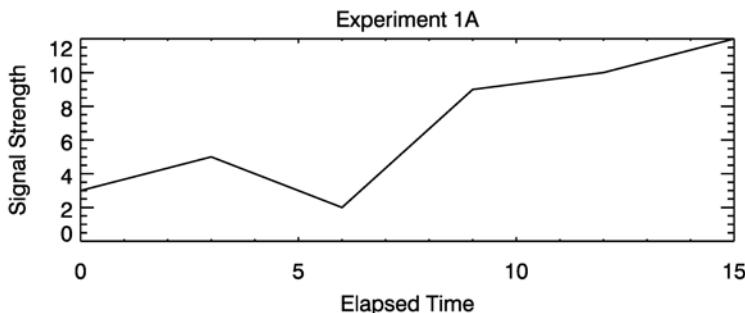


Figure 6: Keywords can modify the size of the axis and plot annotations.

As I have grown older, I prefer larger character sizes for plot annotations. I compensate for poorer eyesight in several ways. First, I change the “default” character size (the character size that is represented by a character size of 1.0) by using the *Set_Character_Size* keyword of the *Device* command. I issue this command in my IDL start-up file, but you can issue the command from the IDL command line any time you want larger “default” fonts.

```
IDL> Device, Set_Character_Size=[8,12]
```

The values passed to the *Set_Character_Size* keyword are a bit complicated to explain, but basically you are setting the pixel size of a rectangle enclosing an “average” character in the first element, and the amount of spacing (or leading) between character lines in the second element. The values 8 and 12 seem to work for me. You should experiment with your own values. These two values set the !D.X_CH_SIZE and !D.Y_CH_SIZE system variables, respectively. The normal, unadjusted, values are 7 and 10 for my Windows machine.

Another way I compensate is to use the program [cgDefCharSize](#) to set the character size. This program is also fairly complicated and sets the character size based on a number of factors, some of which we have not discussed yet. I’ll leave a detailed discussion of this program for later, but one of the purposes of this program is to duplicate the look and feel of IDL 8 function graphics programs, which use larger default character sizes than traditional graphics commands.

```
IDL> Plot, elapsedTime, vector, XTitle='Elapsed Time', $  
      YTitle='Signal Strength', Title='Experiment 1A', $  
      CharSize=cgDefCharSize()
```

The Coyote Graphics System programs ([cgPlot](#), [cgContour](#), etc.) all use [cgDefCharSize](#) to set their default character size.

Creating Your Own Axis Labels

It is also possible to add your own axis labels to a line plot, with the */XYZ/TickName* keywords. For example, if instead of labeling the independent axis with the elapsed time, you want to label it with the number of the measurement, you might do something like this.

```
IDL> tickLabels = ['1st', '2nd', '3rd', $  
                  '4th', '5th', '6th']
```

You will notice in Figure 6, however, that we have four tick labels and we have six measurements. To label our ticks by measurement, rather than by elapsed time, we have to make sure we match the number of tick marks with the number of tick labels. We do this with the */XYZ/Ticks* keyword, which sets the number of tick *intervals* to draw. (Note, it does *not* set the number of tick marks to draw.) We want five tick intervals, or six tick marks.

There is one other complication. When labeling axes with tick names we have to make sure we do not allow auto-scaling of the axis range. We will talk about what this means shortly, but for now we are going to turn X axis auto-scaling off by setting the *XStyle* keyword to 1.

Our final code looks like this.

```
IDL> Plot, elapsedTime, vector, XTitle='Measurements', $  
      YTitle='Signal Strength', Title='Experiment 1A', $  
      CharSize=0.75, XCharSize=1.25, YCharSize=1.25, $  
      XTickName=ticklabels, XTicks=5, XStyle=1
```

You see the result in Figure 7.

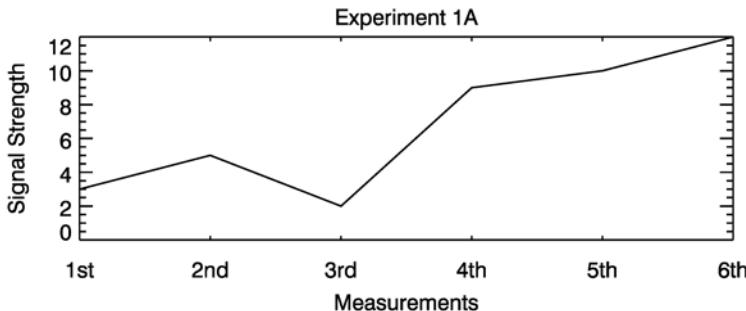


Figure 7: The independent data axis is labeled with the counts of the measurements.

Formatting Axis Labels

If you need to format tick label output, you can use the */XYZ/TickFormat* keywords. The format specification is a string, enclosed in parenthesis, and is identical to format statements you might use for outputting data into files or for printing. For example, if you want to have the elapsed time displayed to two decimal points, you can type this command. The format "(F0.2)" means a floating point format of any "natural" length (this is what the 0 means), with two digits to the right of the decimal point.

```
IDL> Plot, elapsedTime, vector, XTitle='Elapsed Time', $  
      YTitle='Signal Strength', Title='Experiment 1A', $  
      XTickFormat='(F0.2)'
```

We can also use the */XYZ/TickFormat* keywords to specify the name of a tick formatting function that we write in the IDL programming language. Suppose, for example, that we want to label our vector measurements with a letter and number, but we want a more general approach than is possible with the */XYZ/TickName* keyword.

Maybe we want each odd numbered measurement to have an A in front of the measurement number and each even numbered measurement to have an L in front of the measurement number. We can write such a labeling function. Let's call it *OddEvenLabels*. We write it like this. The three posi-

tional parameters, *axis*, *index*, and *value* are required to be defined, and we must write the routine as a function that returns a string that will be the tick label.

The *axis* variable is the axis number: 0 for an X axis label, 1 for a Y axis label, and 2 for a Z axis label. The *index* variable is the tick mark index. Indices start with the number 0 and increase by 1 up to the number of tick marks minus 1. The *value* variable is the data value at the tick mark. It is a double-precision number.

You don't call this routine yourself. Rather, it is called by internal code when the *Plot* command needs to format tick label output. We call such a routine a *callback function*. Our simple function looks like this.

```
FUNCTION OddEvenLabels, axis, index, value
    odd = (index MOD 2) GT 0 ? 1 : 0
    IF odd THEN retValue = 'A-' + StrTrim(index,2)
    ELSE retValue = 'L-' + StrTrim(index,2)
    RETURN, retValue
END
```

If we save the function in a location on our IDL path, we need do nothing more with the function to use it. Notice in this case that we don't do anything with either the *axis* or *value* variable in the function. But we do need to define them nonetheless. The code to use the tick formatting function looks like this.

```
IDL> Plot, elapsedTime, vector, XTitle='Measurements', $
      YTitle='Signal Strength', Title='Experiment 1A', $
      CharSize=0.75, XCharSize=1.25, YCharSize=1.25, $
      XTickFormat='OddEvenLabels', XTicks=5, XStyle=1
```

You see the result in Figure 8.

Suppressing Axis Labels

Tick labeling can also be completely suppressed with the *[XYZ]TickFormat* keywords. Unfortunately, I don't know why this works. The secret was entrusted to me over 20 years ago by an old, crusty IDL programmer who prefers to remain anonymous. I don't think he will mind if I pass the secret on to you. Some things should remain a mystery.

You can suppress axis labeling by setting the *[XYZ]TickFormat* keywords to the format '(A1)', like this.

```
IDL> Plot, elapsedTime, vector, XTitle='Measurements', $
      YTitle='Signal Strength', Title='Experiment 1A', $
      CharSize=0.75, XCharSize=1.25, YCharSize=1.25, $
      XTickFormat='(A1)'
```

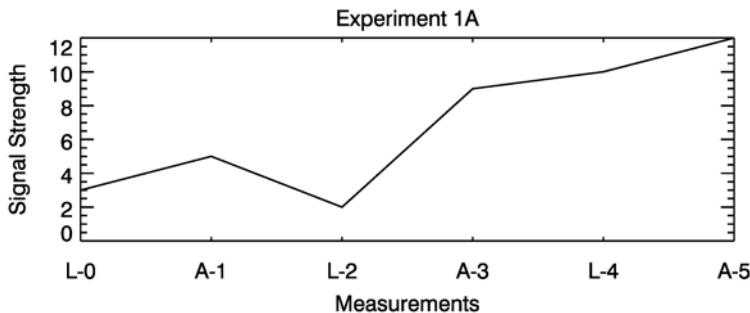


Figure 8: Tick labels can be formatted by means of a user-written IDL function.

You see the result in the top panel of Figure 9.

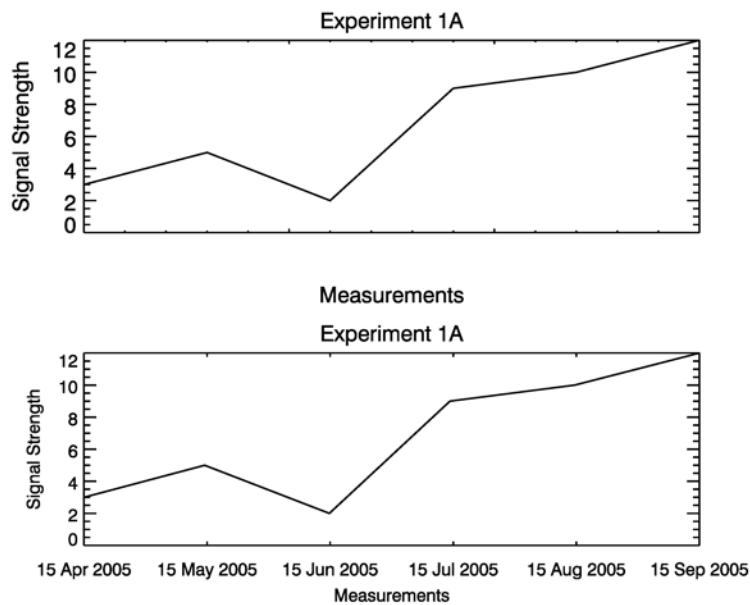


Figure 9: Tick labels can be suppressed entirely with the [XYZ]TickFormat keywords, as shown in the top figure. Or, you can use calendar dates to label the axis, as shown in the bottom figure.

Using Calendar Dates as Axis Labels

Another use of the */XYZ/TickFormat* keywords is to label axes with calendar dates. This is a two step operation. The first step involves calling the *Label_Date* function in IDL to set up a date format. The second step involves setting the *Label_Date* function as the tick formatting function. For example, suppose our data was collected on the 15th of the month, for six months, starting in April, 2005.

First, let's set up our time vector. We can use the IDL *TimeGen* command to build the time vector. The dates will be Julian dates.

```
IDL> time = TimeGen(N_Elements(vector), $  
Start=Julday(4, 15, 2005, 0, 0, 0), $  
Units='MONTHS')
```

Next, we call *Label Date* to set up a date format. The date format uses codes such as %D for the day, %M for the month, and %Y for the year. There are more formats than I discuss here. See the IDL on-line help for details. You can be quite elaborate in how you format your date string.

```
IDL> void = Label_Date(Date_Format='%D %M %Y')
```

Finally, we issue the *Plot* command using the *Label Date* function to format our X axis.

```
IDL> Plot, time, vector, XTitle='Measurements', $  
YTitle='Signal Strength', Title='Experiment 1A', $  
CharSize=0.75, XCharSize=1.00, YCharSize=1.00, $  
XTickFormat='Label_Date', XTicks=5, XStyle=1
```

You see the result in the bottom panel of Figure 9.

Using Irregular Tick Spacing and Values

You can use irregular tick spacing and values on any axis using a combination of the */XYZ/TickV* keyword for setting tick values and the */XYZ/Ticks* keyword for setting tick intervals. For example, consider this IDL code.

```
IDL> irtime = [0.8, 3.5, 5.8, 7.2, 12.1, 15]  
IDL> Plot, elapsedTime, vector, XTickV=irtime, XTicks=5
```

You see the result in Figure 10.

Using Symbols and Line Styles

Another way you can customize the appearance of line plots is to use different symbols and line styles for your plots. Lines can be drawn with six

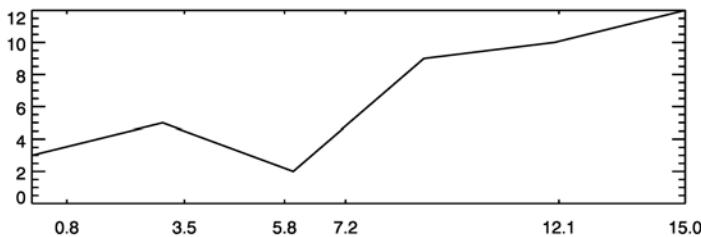


Figure 10: It is possible to label axes with irregular tick marks and spacing using a combination of [XYZ]TickV and [XYZ]Ticks keywords.

different line styles, as shown in Table 1. To choose a different line style, you set the *LineStyle* keyword to the index number of the line style you want to use.

For example, to draw this line plot with a dashed line style, you set the *LineStyle* keyword to 2. You see the result in the top panel of Figure 11.

```
IDL> Plot, elapsedTime, vector, LineStyle=2
```

Index Number	Line Style
0	Solid
1	Dotted
2	Dashed
3	Dash Dot
4	Dash Dot Dot
5	Long Dash

Table 1: The line style can be changed by assigning an index number to the *LineStyle* keyword.

Symbols are treated in a similar way. IDL is supplied with just 8 different symbols, as shown in Table 2 on page 87. You select a plot symbol with the *PSym* keyword, like this.

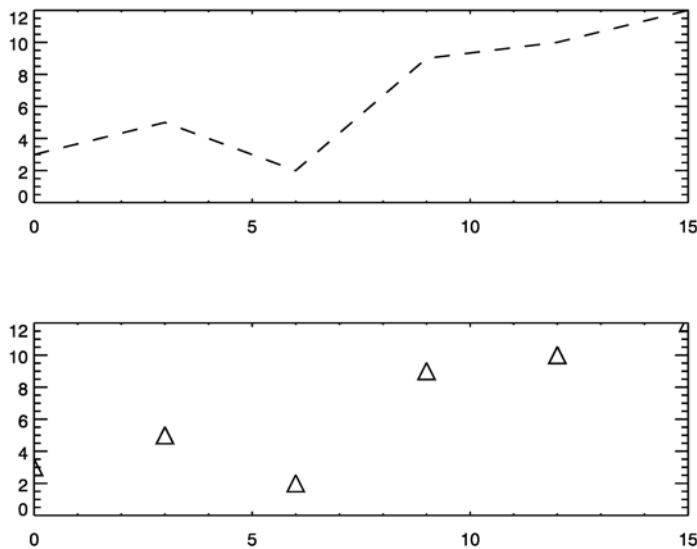


Figure 11: The figure at the top was created by setting the `LineStyle` keyword to 2. The figure at the bottom was created by setting the `PSym` keyword to 5.

```
IDL> Plot, elapsedTime, vector, PSym=5
```

You see the result in the bottom panel of Figure 11.

If you want to connect plot symbols with lines, you simply give a negative value to the `PSym` keyword. Then the selected symbol is drawn with lines connecting them.

```
IDL> Plot, elapsedTime, vector, PSym=-5
```

The lines that connect the symbols can have whatever line style you like. So, if you want to connect the symbols with dashed lines, you can type this.

```
IDL> Plot, elapsedTime, vector, PSym=-5, LineStyle=2
```

You see an example of the last two commands in Figure 12.

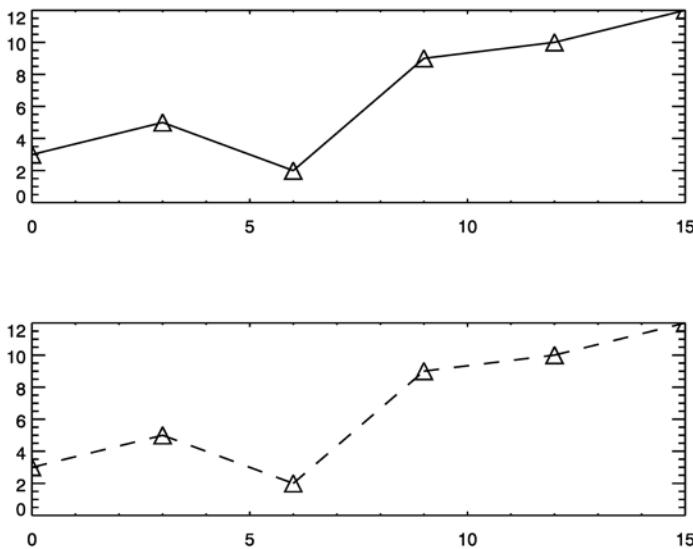


Figure 12: Symbols can be connected by lines by using a negative symbol number. Different line styles can be used together with symbols.

Creating Your Own Plot Symbols

Obviously, the available symbols supplied by default with IDL are quite limited. But IDL does provide an easy way to create your own plot symbols with the *UserSym* command. The way I build a symbol, typically, is to draw my symbol on a piece of graph paper with a two unit square distance and the origin of my XY coordinate system in the center of the symbol. I then locate each vertex of my symbol in this square space and trace the symbol, noting the X and Y coordinate of each vertex. Symbols created like this are “standard” size. These are stored in X and Y vectors that are passed to the *UserSym* command.

Symbols can be of any size, although I would encourage you to use a two unit square for defining them, unless you have a good reason for doing something else. The symbol can have up to 49 vertices. Here, for example, is the code I use to create a star symbol.

```
IDL> x = [0.0, 0.5, -0.8, 0.8, -0.5, 0.0]
IDL> y = [1.0, -0.8, 0.3, 0.3, -0.8, 1.0]
IDL> UserSym, x, y, /Fill
```

The symbol is selected by using the *PSym* keyword value 8 to select a user-defined symbol. Only one user-defined symbol can be defined at a time, so in practice this means you must create the symbol using *UserSym* just before you use the symbol in a line plot. You can use the *SymSize* keyword to increase or decrease the size of your symbol. All sizes are calculated with respect to the default symbol size of 1.0.

In the following commands, I make the symbol bigger than normal and color it red. Normally, all graphics output is clipped to the interior of the plot boundaries. By setting the *NoClip* keyword to 1, I allow the symbols at either end of the plot to be drawn outside the plot boundaries.

Index Number	Symbol Drawn on Plot
0	No symbol. Points are connected by lines.
1	Plus sign
2	Asterisk
3	Period
4	Diamond
5	Triangle
6	Square
7	X
8	User defined with <i>UserSym</i> .
9	Not used.
10	Data drawn in "histogram" mode.
-PSym	Negative values of <i>PSym</i> connect symbols with lines.

Table 2: These are the default index numbers you can use with the *PSym* keyword to produce different plotting symbols on your plots. Note that if you use a negative value for *PSym* the symbols are connected by lines.

```
IDL> Plot, elapsedTime, vector
IDL> OPlot, elapsedTime, vector, PSym=8, SymSize=2.5, $
    Color=cgColor('red'), NoClip=1
```

You see the result in the top panel of Figure 13.

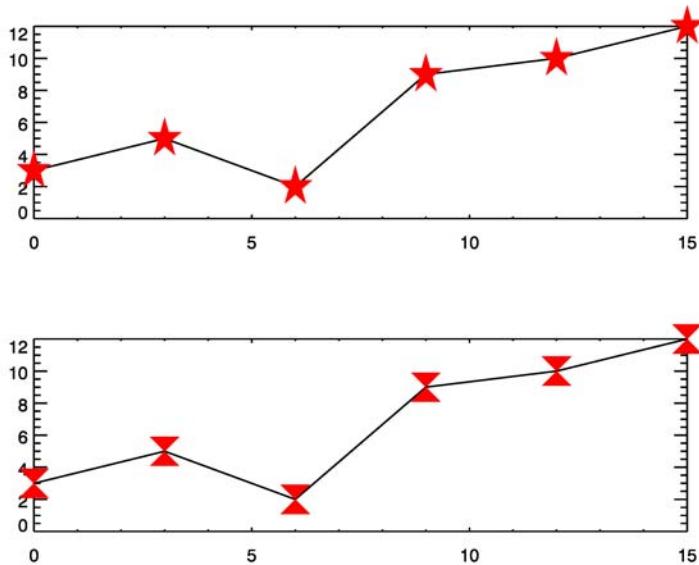


Figure 13: User-defined symbols are easy to create with `UserSym`.

The [Coyote Library](#) contains a symbol catalog file, named `SymCat`, which defines 46 different plotting symbols, including the default IDL plotting symbols. You can see a list of the symbols by reading the documentation header of the program. About half of these symbols are filled symbols. For example, we can produce the same figure as in the top panel of Figure 13, except with filled hourglass symbols, like this.

```
IDL> Plot, elapsedTime, vector  
IDL> OPlot, elapsedTime, vector, PSym=SymCat(22), $  
           SymSize=1.75, Color=cgColor('red'), NoClip=1
```

You see the result in the bottom panel of Figure 13.

You can use a negative value for `SymCat` to connect these symbols with lines, as you normally do with the `PSym` value. In other words, this command does exactly the same thing as the two commands above, except that the symbols are not colored with this command.

```
IDL> Plot, elapsedTime, vector, PSym=-SymCat(22), $  
           SymSize=2.5, NoClip=1
```

Overplotting Additional Data Vectors

Now that you know how to use different line styles, symbols, and colors to display your data, it's probably a good time to talk about overplotting additional data vectors on your line plots. There is no limit to the number of vectors you can plot on a line plot in IDL. Each additional data set is added to the original plot with the *OPlot* command.

There are a couple of points to understand, however, before you can do this successfully in all cases. The first point to consider is that the first graphics command that "erases the window" establishes a data coordinate system for the graphics window. I don't literally mean "erases the window," but often a graphics command (such as the *Plot*, *Contour*, and *Surface* commands, but not the *TV* command) has the effect of erasing what is currently in the graphics window before it draws its graphical output.

One of the side effects of issuing a command like this is that a data coordinate system is established for that graphics window by that command. For a line or contour plot, you can see the extent of the data coordinate system by looking at how the axes are labeled. The "knowledge" of the data coordinate system created in this way is stored in IDL system variables.

In particular, the system variables *!X.S* and *!Y.S* (in the case of a line or contour plot) contain *scaling* parameters that tell IDL how to take a value in the data coordinate system and scale and translate that value so that the proper pixel in the graphics window, which uses a *pixel* or *raster* or *device* (the names are synonymous) coordinate system, can be set to the proper color value.

You do not need to be concerned with how this is done. What you do need to be concerned about is the fact that the means to do it are stored in *system* or *global* variables. In effect, what this means is that if you want to put something in a graphics window, using the data coordinate system that is in effect in a global variable, then you better know how that data coordinate system was created, or you don't have any hope at all of getting things in their proper location.

You will learn how to save and restore data coordinate systems when we talk about more sophisticated traditional graphics operations in later chapters. For now, it is enough to know that the first line plot you draw establishes the data coordinate system for every subsequent data operation you perform in that graphics window, until you type a command that "erases" the graphics window.

Consider the following two vectors.

```
IDL> vec_1 = Smooth(RandomU(-60L, 31) * 20 - 10.0, 9)
IDL> vec_2 = Smooth(RandomU(seed, 31) * 50 - 25.0, 9)
```

These vectors each have the same number of elements, so we won't worry for the moment about the independent data vector. Let's just plot these two vectors like this.

```
IDL> Plot, vec_1
IDL> Oplot, vec_2, LineStyle=2
```

You see the result in Figure 14. At first glance the plot may appear to be correct. But on closer inspection, you can see that the second vector (the one drawn in a dashed line style) is being clipped. Indeed, the minimum and maximum value of the second vector is less than -10 and greater than 10, respectively.

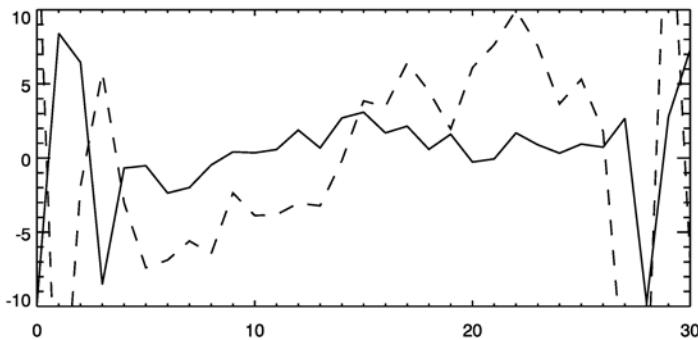


Figure 14: There is no limit to the number of data vectors that can be placed on a line plot, but the first plot establishes the data coordinate system for all subsequent plot actions. This plot has an incorrect Y data range, causing the over-plotted vector to be clipped.

```
IDL> Print, Min(vec_2), Max(vec_2)
-18.2861      18.0642
```

To draw this plot correctly, we want the dependent data range (i.e., the Y range) to extend from at least -20 to 20. We can set this range on the first *Plot* command with the *YRange* keyword.

```
IDL> Plot, vec_1, YRange=[-20,20]
IDL> Oplot, vec_2, LineStyle=2
```

You see the result in Figure 15.

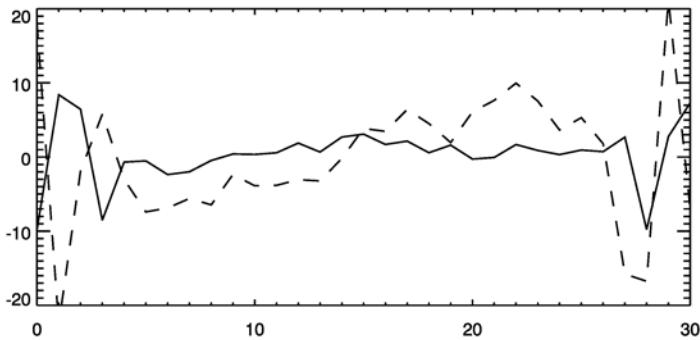


Figure 15: By setting the Y axis range, both plots fit completely within the plot boundaries.

Now both vectors are correctly located in the plot. Or, we could do this in a more general way by comparing the minimum and maximum values of all the data vectors we want to include in the plot. Our code might look like this.

```
IDL> minYRange = Min([Min(vec_1), Min(vec_2)])
IDL> maxYRange = Max([Max(vec_1), Max(vec_2)])
IDL> Plot, vec_1, YRange=[minYRange, maxYRange]
IDL> Oplot, vec_2, LineStyle=2
```

You can easily see how this can be extended to calculate the minimum and maximum data range for any number of data vectors.

We will have more to say about how IDL chooses the data range (the so-called “auto-scaling” of axes) later in this chapter.

Positioning Line Plots

If you think of the graphics display window, or the window you create on the PostScript page, as the canvas on which you are going to display your traditional graphics commands, then it is helpful to think of that canvas as

having a *normalized* coordinate system associated with it. A normalized coordinate system is one that extends from 0 to 1 in both the X and Y directions of the canvas, no matter what size the canvas is or its actual dimensions.

We often position a line plot on the canvas by specifying the plot's position with respect to this normalized coordinate system. In fact, we use a *Position* keyword to do this. The *Position* keyword is set equal to a four-element vector that describes the lower-left and upper-right corner, in X and Y, of the plot axes on the canvas. In pseudo-code, it might look like this.

```
position = [X_LL, Y_LL, X.UR, Y.UR]
```

Suppose, for example, you want to have a multiple line title on your line plot. IDL does not leave enough room for such a title in its default positioning of line plots. There is too little room at the top and sometimes bottom of the plot. You can leave more room for multiple line titles by positioning the plot with the *Position* keyword, like this.

```
IDL> Plot, elapsedTime, vector, $  
      YTitle='Signal Strength', $  
      XTitle='Measurements!C(Experimental)', $  
      Position=[0.1, 0.25, 0.9, 0.85], $  
      Title='Experiment 1A!C(October 5, 2010)'
```

You see the result in Figure 16. Note the use of !C in the title keywords, which inserts a carriage return into the text output.

The *Position* keyword can also be used to put multiple plots in the graphics display window at arbitrary locations. The only thing you have to keep in mind when you do this is that most IDL traditional graphics commands (e.g., *Plot*, *Contour*, *Surface*, etc.) will first erase the window before drawing their graphics output. If you are going to use multiple graphics commands in the same window, you will have to suppress this behavior by setting the *NoErase* keyword on all the graphics commands which follow the first one.

Here, for example, is how you might put a small plot inside the boundaries of a larger plot.

```
IDL> Plot, elapsedTime, vector, XTitle='Elapsed Time', $  
      YTitle='Signal Strength', Title='Trial 1A', $  
      Position=[0.125, 0.125, 0.9, 0.9]  
IDL> temperature = Randomu(-3L, 21)*5 + 25  
IDL> time = FIndGen(21)*15/20.  
IDL> Plot, time, temperature, YTitle='Temperature', $  
      XTitle='Elapsed Time', CharSize=0.65, /NoErase, $  
      Position=[0.20, 0.60, 0.45, 0.85], /YNoZero
```

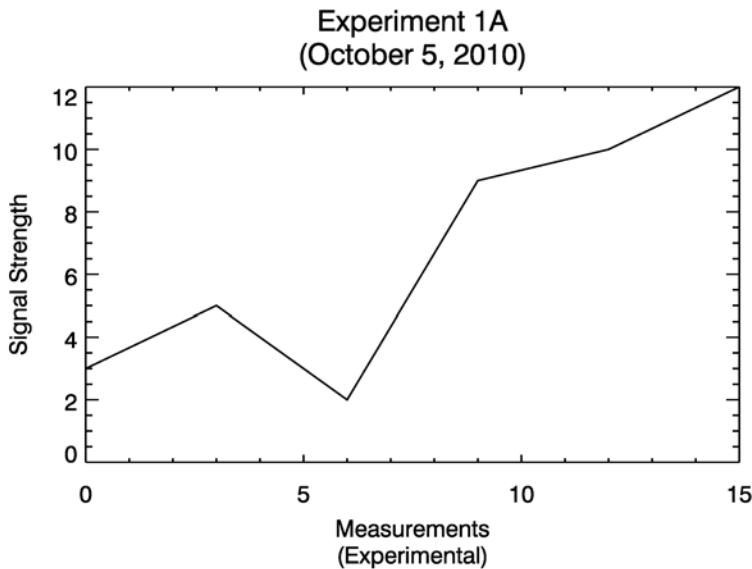


Figure 16: The `Position` keyword can be used to leave extra room in the plot window for multiple line titles.

You see the result in Figure 17.

Creating Multiple Line Plots

In the plot above, you not only have to keep track of exactly where you are positioning each graphic, but you have to worry about character sizes and other small details associated with producing nice looking graphics output. The fact that you can do this is a huge bonus, but sometimes having so much power in your hands is overkill.

In fact, much of the time we don't so much want inset graphics, as in Figure 17, but rather we are interested in seeing several plots on a single page of output. Such plots are sometimes called *small multiple plots*, and they give us the opportunity to compare several plots at once. We can, of course, create small multiple plots by the methods described above, but keeping track of all the plot positions and other details is tedious and we might prefer to avoid this kind of work. And, needless to say, if we later decide to change the plot layout, we can forget about going out for beers with the guys after work. We will be working late!

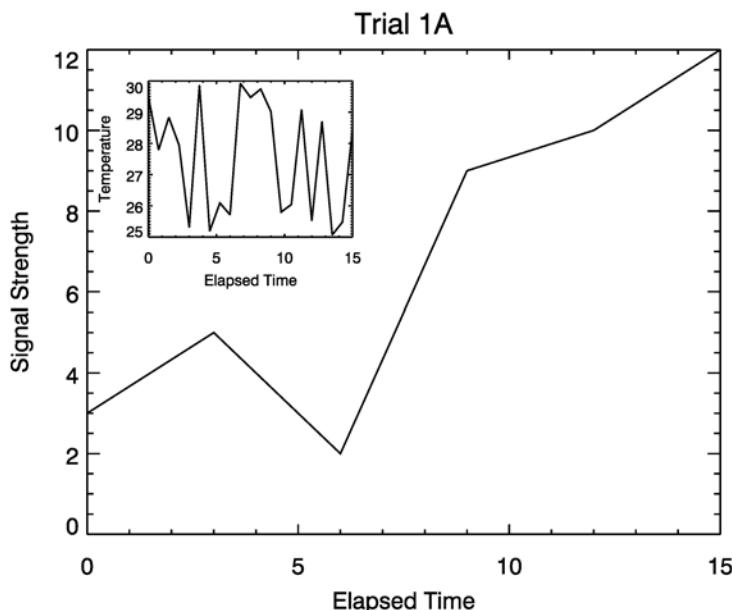


Figure 17: Using the Position and NoErase keywords, you can put plots and other IDL graphics wherever you like in a graphics window.

Fortunately, IDL provides an easy way to create multiple plots, provided that your plots can be laid out on a regular grid. This is implemented by the system variable `!P.Multi`. This system variable is a five-element array, with each element defined as follows.

- | | |
|--------------------------|--|
| <code>!P.Multi[0]</code> | The first element of <code>!P.Multi</code> contains the number of graphics plots <i>remaining to be plotted</i> on the graphics display or PostScript page. This is a little non-intuitive, but you will see how it is used in a moment. It is normally set to 0, meaning there are no more graphics plots remaining to be output on the display. Thus, the <i>next</i> graphics command will erase the display and output the first of the multiple graphics plots. |
| <code>!P.Multi[1]</code> | This element specifies the number of graphics columns on the page. |
| <code>!P.Multi[2]</code> | This element specifies the number of graphics rows on the page. |

!P.Multi[3]	This element specifies the number of graphics plots stacked in the Z direction. (This applies only if you have established a 3D coordinate system in IDL, and is usually just set to 0.)
!P.Multi[4]	This element specifies whether the graphics plots are going be displayed by filling up each row ($!P.Multi[4]=0$) before moving to the next row or by filling up each column ($!P.Multi[4]=1$) before moving to the next column. That is to say, whether the output will be displayed in row order or in column order. The default is row order.

Suppose, for example, you want to display four line plots in a two-column by two-row grid, with the plots being drawn in column order. You would set up the `!P.Multi` system variable like this.

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
```

Now IDL will take care of the details of arranging the plots in this space, setting character sizes appropriately for the plots, and handling all of the myriad other details for you. What is *essential*, however, is that you let IDL do everything for you. I'm going to be overly dramatic here to make a point, but don't, under any circumstances, take care of the details yourself! Don't use the `Position` keyword on your `Plot` commands, don't try to set the character size of your plots (it is okay to set the default character size), and so forth. If you do, `!P.Multi` will get very, very confused. In fact, *you* will get confused, because nothing will go where you expect it to go or look the way you expect it to look! Give up being stubborn and let IDL do it for you. If this is hard for you, re-read the section on the `Position` and `NoErase` keywords.

Okay, having given up some control, you are still free to use other keywords you have used before (and some you haven't encountered yet) to control how your plots look on the page. Colors, line styles, titles, overplotting, and other accoutrements are all acceptable for your graphics commands. Be aware, however, that if want to overplot onto a plot you just created with `!P.Multi`, you *must* do this before you move to the next plot. (This is not absolutely true, but true enough for now. You will learn other techniques later.) Here "moving to the next plot" means executing an IDL graphics command that would normally erase the display window prior to displaying the graphics output.

Note: The values of `!P.Multi` are “sticky”. That is to say, they persist in the IDL session until they are changed. This can cause great confusion, because once set, all plots, not just the ones you want to plot as multiple plots, will use the `!P.Multi` settings. Be absolutely sure you set `!P.Multi` back to its default settings when you are done with it. The easiest way to do this is to set the variable to 0, as shown below.

In the code below, I am going to use the [Coyote Library](#) routine `cgDemoData` to provide random data to plot. Here is how we can create our four plots.

```
IDL> Plot, cgDemoData(17), Title='Plot 1', $  
      XTitle='Time', YTitle='Signal'  
IDL> Plot, cgDemoData(17), Title='Plot 2', $  
      XTitle='Time', YTitle='Signal'  
IDL> Plot, cgDemoData(17), Title='Plot 3', $  
      XTitle='Time', YTitle='Signal'  
IDL> Plot, cgDemoData(17), Title='Plot 4', $  
      XTitle='Time', YTitle='Signal'  
IDL> !P.Multi = 0
```

You see the result in Figure 18. Note how I turned `!P.Multi` off. By setting the entire variable to 0, I have “zeroed out” each field in `!P.Multi`. In this case, this will set the `!P.Multi` system variable to its default values. Subsequent `Plot` commands will display themselves normally.

Notice, too, how the figures in the plot completely fill up the canvas. If you want to put an overall title on this plot page, you will not have room to do so.

If we want to leave room around the outside margins of a multiple plot layout, we can use the “outside margin” or `OMargin` field of the `!X` and `!Y` axis system variables to do so. The outside margin fields are only used when `!P.Multi` is set to something other than its default values.

Take care, however, because the outside margin system variables use a strange unit of measure. They are expressed in units of character size. In practice, this means you might have to experiment a little to get the margins correct for your particular layout. And, if you settle on margins of, say, 4 and 10 when your character size is 1.0, then these margins will be twice as big if you later change your character size to 2.0. This may not be what you had in mind.

I often use outside margins to save room on my plot layout for color bars (`!X.OMargin=[1,10]`) or for page titles (`!Y.OMargin=[0,4]`). The margin

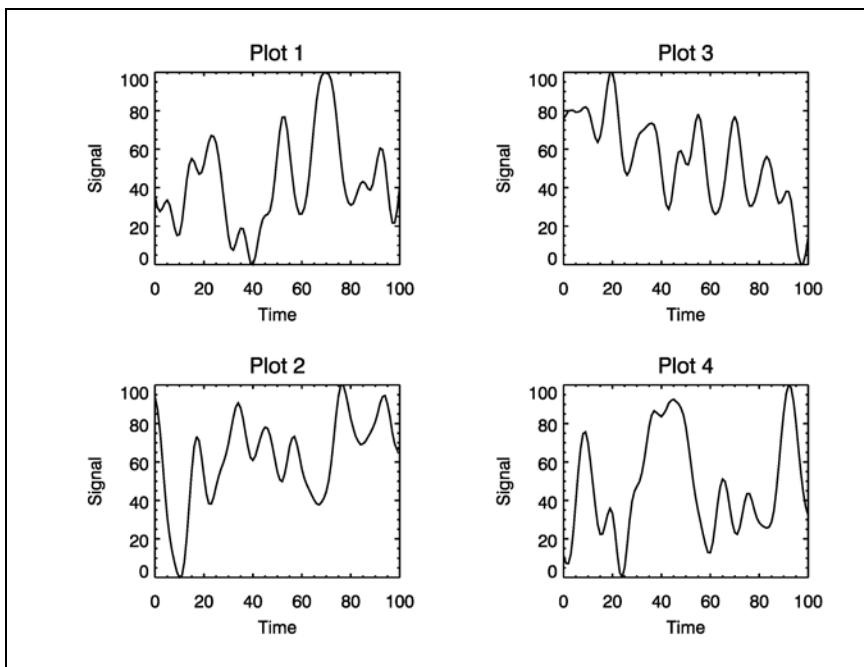


Figure 18: Small multiple plots can be created with the `!P.Multi` system variable. The system variable takes care of plot placement, character sizes, and other plot details.

values are specified as `[left, right]` for `!X.OMargin`, and as `[bottom, top]` for `!Y.OMargin`.

Here is how we can put a page title on our plot layout. Note that I zero out the `!Y.OMargin` system variable when I am finished with it, just as I zero out the `!P.Multi` system variable. If you don't do this, you may not be pleased with subsequent graphics output. Remember, these are *global* system variables. They affect everything!

```
IDL> !P.Multi = [0, 2, 2, 0, 1]
IDL> !Y.OMargin = [0, 3]
IDL> Plot, cgDemoData(17), Title='Plot 1', $
      XTitle='Time', YTitle='Signal'
IDL> Plot, cgDemoData(17), Title='Plot 2', $
      XTitle='Time', YTitle='Signal'
IDL> Plot, cgDemoData(17), Title='Plot 3', $
      XTitle='Time', YTitle='Signal'
IDL> Plot, cgDemoData(17), Title='Plot 4', $
      XTitle='Time', YTitle='Signal'
```

```
IDL> XYOuts, 0.525, 0.96, /Normal, 'Experiment 1', $  
      CharSize=1.5, Alignment=0.5, $  
      Color=cgColor('red')  
IDL> !Y.OMargin = 0  
IDL> !P.Multi = 0
```

You see the result in Figure 19.

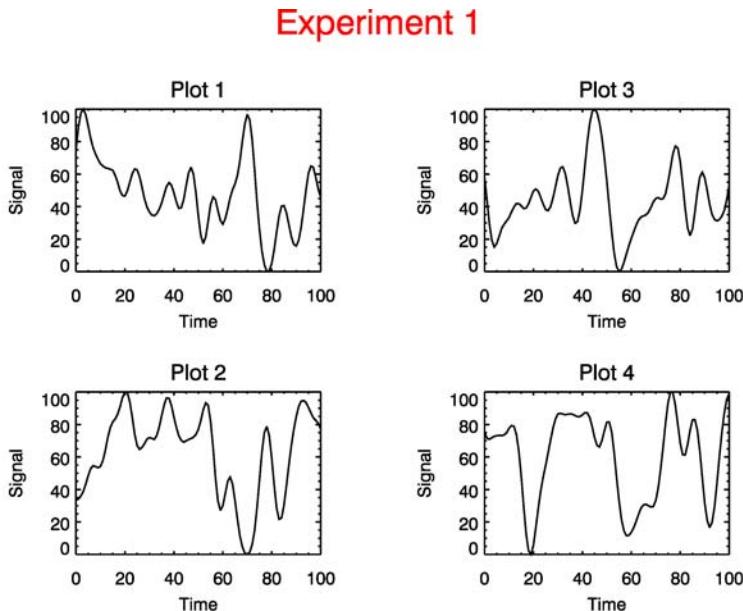


Figure 19: Outside margin system variables can be used to allow room around the multiple plots for color bars, titles, and other plot annotations.

Not all five values of `!P.Multi` need to be set every time. If you decide you are happy with no plots in the Z direction and displaying plots in row order, but you want three columns and two rows, you can configure `!P.Multi` like this.

```
IDL> !P.Multi = [0, 3, 2]
```

The fourth and fifth elements will maintain their current values, since they are “sticky.” You will see this version of the command a great deal in the examples in this book, since I don’t usually need to set the fourth and fifth

elements of the variable. Remember to “zero out” the system variable before you move on to other examples in the chapter.

```
IDL> !P.Multi = 0
```

Customizing Line Plot Axes

There are any number of ways the appearance of a line plot can be changed in IDL. We will discuss some of the more popular ways in this section, but we will by no means exhaust the possibilities. IDL programmers are only limited by their own imaginations in this regard.

Axis Range Scaling

Consider a plot of the following two vectors, representing a dependent and independent data vector.

```
IDL> dep = Smooth(Smooth(RandomU(-6L, 101) * 50, 7), 3)
IDL> indep = IndGen(101) * 10.0 / 100.
IDL> Plot, indep, dep
```

You see the result in Figure 20. The data coordinate system established by IDL has set a Y range from 0 to 50, and an X range from 0 to 10.

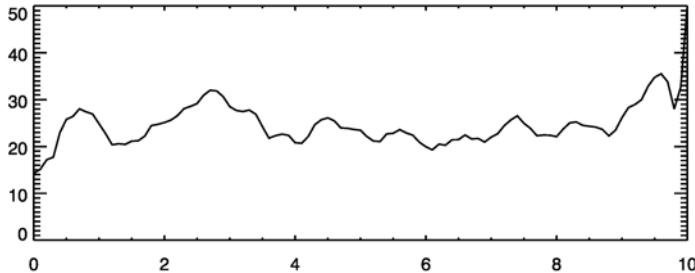


Figure 20: A plot command establishes a data coordinate system.

This is as expected, since the minimum and maximum values of the data vectors fall within this range.

```
IDL> Print, Min(dep), Max(dep)
      14.2394      49.9223
IDL> Print, Min(indep), Max(indep)
      0.000000      10.0000
```

Notice, however, that while the X data range is *exactly* equal to the minimum and maximum of the independent data vector (fortuitously, as it happens!), the Y data range is only approximately equal to the minimum and maximum of the dependent data vector. In fact, IDL picks data coordinate ranges that are guaranteed to contain the actual data in the line plot, but which might be larger than the actual data range, if it can produce a more aesthetically pleasing data axes by choosing a larger range. By “aesthetically pleasing” I mean a data axis that can be nicely divided by an integer number of tick intervals.

Another way of saying this is that IDL auto-scales the data axes to produce axes that have an integer number of tick intervals. Quite often, we are happy to have IDL do this for us. But there are occasions—maybe more than you think—when you do *not* want IDL to do this, and it is important to know how to turn auto-scaling of axes off.

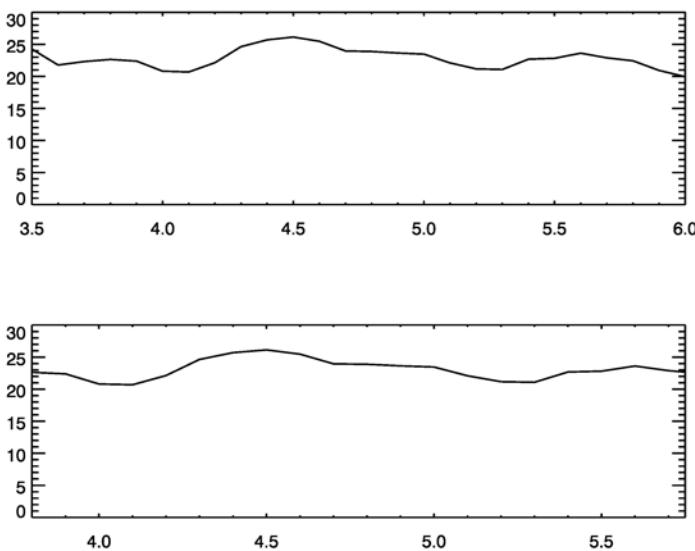


Figure 21: Zooming into a portion of the plot with auto-scaling of axes turned on (top) and turned off (bottom) by setting the `XStyle` keyword to 1.

Zooming Into a Line Plot

Consider this simple example. Suppose you want to “zoom into” a portion of your data in order to examine a particular feature more closely. Say you are interested in the portion of data in the X range between 3.8 and 5.75. You can zoom into this portion of your data vectors, by setting the *XRange* keyword to these values.

```
IDL> Plot, indep, dep, XRange=[3.8, 5.75]
```

You see the result in the top panel of Figure 21. The X axis range is now 3.5 to 6.0. This is close to what you asked for, but not exactly what you asked for. To make it exact, you must turn axis auto-scaling off. This is done by means of the */XYZStyle* keywords. In our case, to turn auto-scaling of the X axis off, we must set the *XStyle* keyword to 1.

```
IDL> Plot, indep, dep, XRange=[3.8, 5.75], XStyle=1
```

You see the result in the bottom panel of Figure 21. Note that one could argue that the aesthetics of the X axis is compromised, but I believe this is more than offset by the darn axis actually doing what you asked it to do! It’s not so critical here, but it will be critical later on, so it’s best to know how to do it.

Incidentally, if you are interested in interactively zooming into a line plot, you might find the [Coyote Library](#) routine, [FSC_ZPlot](#), of interest. You zoom into the plot by clicking, holding, and dragging the left mouse button inside the plot. You zoom all the way back out by clicking the right mouse button inside the plot. The [FSC_ZPlot](#) window is completely resizable.

```
IDL> FSC_ZPlot, indep, dep
```

You see the result of zooming a plot with [FSC_ZPlot](#) in Figure 22.

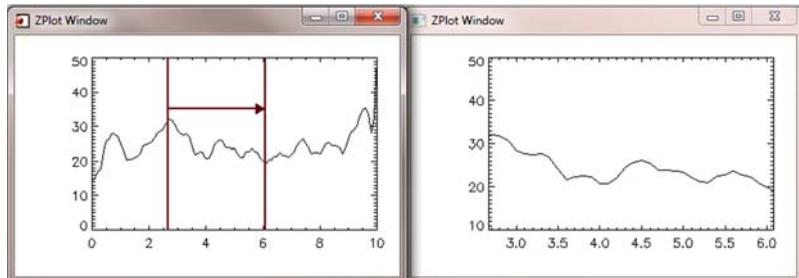


Figure 22: The [FSC_ZPlot](#) command allows interactive zooming of line plots.

Reversing Line Plot Axes

Sometimes you would like to reverse the range of an axis. For example, atmospheric pressure is usually plotted with high values at the bottom of the Y axis and low value at the top of the Y axis. This is easily accomplished in IDL by just reversing the order of the range values in the *[XYZ]Range* keywords. To display these two vectors with the Y axis reversed, we can type this.

```
IDL> Plot, indep, dep, YRange= [Max (dep), Min (dep) ]
```

Suppressing Axes

The *[XYZ]Style* keywords do more than just turn axis auto-scaling on and off. These keywords are actually bit values, and the look of a particular axis depends upon which bit (or bits) in these keywords are set. You see in Table 3 what each bit means, and what value you should assign to the keyword to set a particular bit.

Suppose you want to draw a plot in which you didn't draw the X axis. You would set the *XStyle* keyword to 4. But then the data would be drawn with auto-scaling of axes turned on. If you want to turn the axis off, and draw the plot with exact axis scaling, you will set the *XStyle* keyword to 4 + 1, or 5. This will set both bit 0 and bit 2. Here are a couple of different looking plots

```
IDL> Plot, indep, dep, XStyle=4 + 1, YStyle=8
IDL> Plot, indep, dep, XStyle=8 + 1, YStyle=8
```

You see the result in Figure 23. The plot on the top has had the Y box style axis turned off, and the X axis turned completely off. The plot on the bottom has had the box style turned off for both the X and Y axes

Bit	Value	Meaning
0	1	If this bit is set, do not auto-scale axis. Use exact axis scaling.
1	2	If this bit is set, extend the axis by 5% of the data range in each direction.
2	4	If this bit is set, do not draw the axis and/or axis annotation.
3	8	If this bit is set, do not use a box-style axis. Use a single axis.
4	16	Inhibit setting the Y axis minimum value to 0. Mostly obsolete. This effect is accomplished with the YNoZero keyword.

Table 3: Axis style bit values and their meaning.

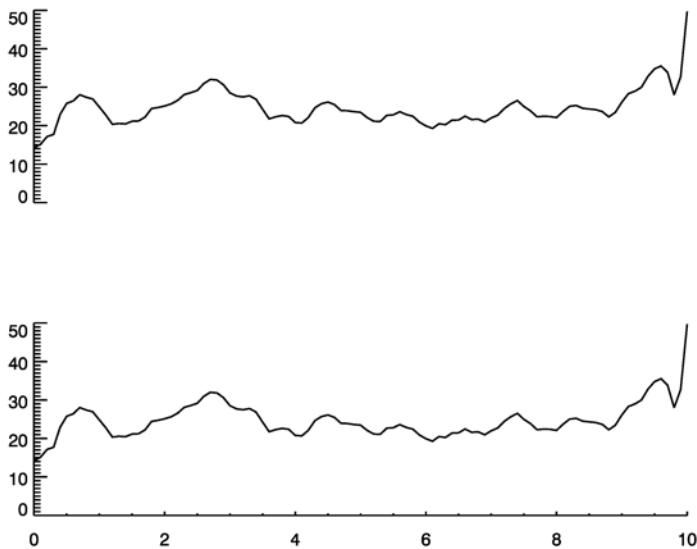


Figure 23: Axes can be turned on or off, or box style axes can be turned on or off, with the [XYZ]Style keywords.

Using Logarithmic Axes

So far, the axes we have considered have all been linear axes. But IDL can also use logarithmic axes on graphical output. Logarithmic axes are set by means of the /XYZ/Log keywords. Consider the dependent variable *dep_y*, which we define like this.

```
IDL> dep_y = (FIndGen(101) * (10.0/100.0) + 1)^2
IDL> Print, Min(dep_y), Max(dep_y)
      1.00000      121.000
```

We can plot this dependent data vector using either a linear or logarithmic axis.

```
IDL> Plot, indep, dep_y
IDL> Plot, indep, dep_y, /YLog
```

You see the result in Figure 24.

One weakness of IDL's logarithmic axes is that only major tick marks are labeled. Quite often you would prefer minor tick marks in logarithmic

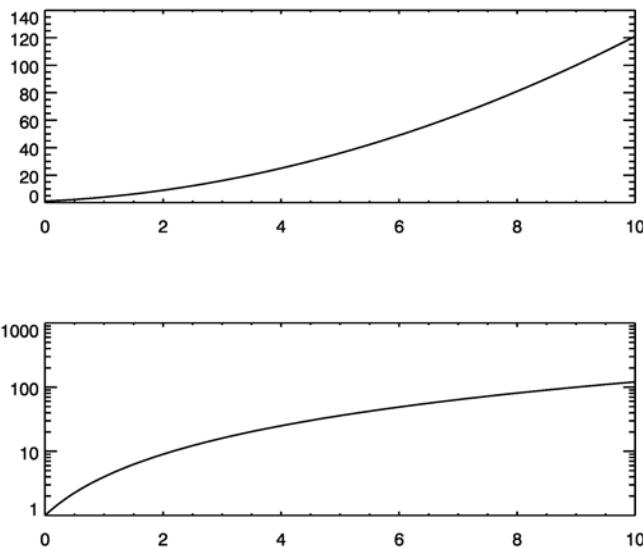


Figure 24: The dependent vector plotted with a linear axis (top) and a logarithmic axis (bottom).

plots to be labeled, too. The only way to accomplish this is to write your own tick formatting function.

Fortunately, the IDL user community has stepped in to address this deficiency. Martin Schultz has written a nice logarithmic labeling function, named *LogLevel*, which allows great control over how logarithmic axes are labeled. You can learn more about this topic, and find the *LogLevel* source code here. (You can learn more about tick formatting functions on page 80.)

http://www.idlcoyote.com/graphics_tips/minorlog.html

Dealing with a Transient Logarithmic Bug

There is one obscure detail you ought to be aware of when working with logarithmic axes. When an */XYZLog* keyword is used, it actually sets the *Type* field in the appropriate axis system variable (e.g. *!X*, *!Y*, or *!Z*) And this type bit occasionally gets “stuck.” In other words, unlike almost all other properties that get set from the IDL command line, the logarithmic

property in say, `!Y.Type`, is sometimes sticky. It seems to get set and stay set, unless you specifically turn it off.

But, oddly enough, you can't turn it off by, say, setting `YLog=0` on a `Plot` command. You have to turn it off by actually setting the `Type` field in the axis system variable. You must do something like this to free it.

```
IDL> !Y.Type = 0
```

I am not sure whether this always happens, or happens under some circumstances and not others, or in some versions of IDL but not others. I do know I run into strange problems with logarithmic axes enough to remember this little bit of trivia. If your logarithmic axes are giving you some trouble, see if you can fix the problem with this axis "type" trick.

Adding Additional Axes to a Line Plot

Sometimes you want to display two dependent vectors in the same line plot, but their axis range is such that there is little overlap. If you displayed them using the same Y scale they would be far apart, or the scale would be so large as to smooth any interesting features in the data out of existence.

In these circumstances, we sometimes use a second dependent data axis to display the second data vector. The procedure is simple. Draw the first vector in the normal way, but leave a little extra room at the right hand side of the plot for a second axis, and then draw the second axis (in fact, you can draw as many as you like) with the `Axis` command.

To demonstrate, we can draw the two plots in Figure 24 in a single plot, using a linear and a log axis. We start by drawing the first plot with a linear axis, using the `Position` keyword to leave room at the right side of the plot for a second logarithmic axis.

```
IDL> Plot, indep, dep_y, YTitle='Linear Axis', $  
      YStyle=8, Position=[0.15, 0.15, 0.85, 0.95], $  
      Color=cgColor('red8')
```

Next, we add the second axis with the `Axis` command. The only trick here is that we have to use the `Save` keyword in order to save the proper axis scaling values (i.e., in `!Y.S`) to allow overplotting onto the new Y axis. If we don't save the scaling factors, the overplot will use the data coordinates that were established by the initial `Plot` command, which is not what we want. We will draw the second axis and data vector in a different color so you can clearly see what is happening.

```
IDL> Axis, YAxis=1, /YLog, YRange=[1,500], YStyle=1, $  
      /Save, Color=cgColor('blu8'), $  
      YTitle='Log Axis'
```

And, finally, we draw the data in the logarithmic data range.

```
IDL> Oplot, indep, dep_y, Color=cgColor('blu8')
```

You see the result in Figure 25.

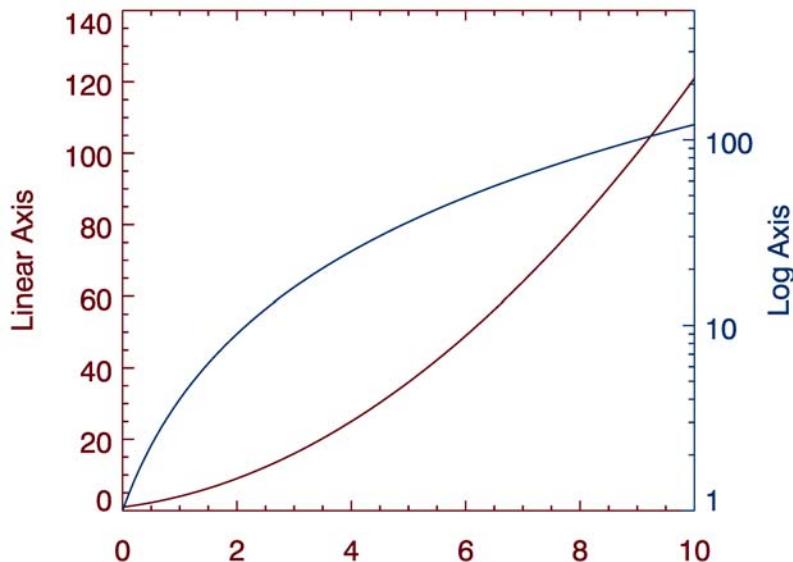


Figure 25: IDL will allow you to use one or more additional axes with the `Axis` command.

Adding Other Annotation to Line Plots

It is often important to know how to draw lines and text on line plots. For example, we may want to draw attention to a portion of our data by drawing a box around it, or we may want to add a legend to our plot. Three fairly low-level graphics commands are useful in traditional graphics.

These are *PlotS*, *XYSOutS*, and *PolyFill*. (These routines have their counterparts in the device and color model independent [Coyote Library](#) routines `cgPlotS`, `cgText`, and `cgColorFill`.)

The *PlotS* command is sometimes used to overplot symbols on a line plot, but is more often used as a general purpose tool for drawing lines and

curves in IDL graphics windows. *XYOutS* is used to output a string at some (x,y) location. And *PolyFill* is used to fill polygons with textures or colors.

Consider that in displaying our data, we might want to indicate where the median value of our vector lies on the plot of the vector. We can use *PlotS* to draw a line across the plot at the median value location.

PlotS takes two positional parameters in the 2D case and three positional parameters in the 3D case. In the 2D case, these will be vectors of X and Y locations that should be connected to one another. In other words, the point (X_0, Y_0) will be connected to point (X_1, Y_1) and so on.

The first step is to calculate the median value of the vector.

```
IDL> medianValue = Median(vector)
```

We want to draw the line at this median value across the plot, the length of the X axis. We may not know exactly what the end points of the line we want to draw are, since IDL may calculate different axis end points from those we specify if we have axis auto-scaling turned on. But, fortunately, once we draw a plot, IDL stores the location of the axis endpoints in the axes system variables. In our case, the endpoints will be stored in *!X.CRange* (the *Calculated Range*, or the range after all of IDL's calculating was done). We must get this value *after* we draw the plot.

We can draw the line in any style supported by IDL. Let's draw the line here in a dashed line style and red color.

```
IDL> Plot, elapsedTime, vector, /NoData
IDL> Oplot, elapsedTime, vector, Color=cgColor('blu8')
IDL> xvalues = !X.CRange
IDL> yvalues = [medianValue, medianValue]
IDL> PlotS, xvalues, yvalues, LineStyle=2, $
      Color=cgColor('red8')
```

We can finish this off by labeling the median line with *XYOutS*. We sometimes have to locate labels by inspection. (I'll teach you a simple method I often use in just a moment.) But, in this case, it looks like the label could be centered about the point (2.5, 10) in the data coordinate space. We use the *Alignment* keyword to *XYOutS* to pick a centered alignment (*Alignment*=0.5). We could choose to align the text to the left of the point (*Alignment*=0.0) or to the right of the point (*Alignment*=1.0).

I am going to convert the median value to a string to include it in the label. I could have used *StrTrim* to do the conversion to a string, but I choose to use the *String* function here, because I want to format the string number

to two decimal places, rather than the five decimal places I would have gotten with *StrTrim*.

```
IDL> text = 'Median Value ( ' + String(medianValue, $  
Format='(F0.2)' ) + ')'  
IDL> XYOutS, 2.5, 10, text, Alignment=0.5, $  
Color=cgColor('red8')
```

You see the result in Figure 26.

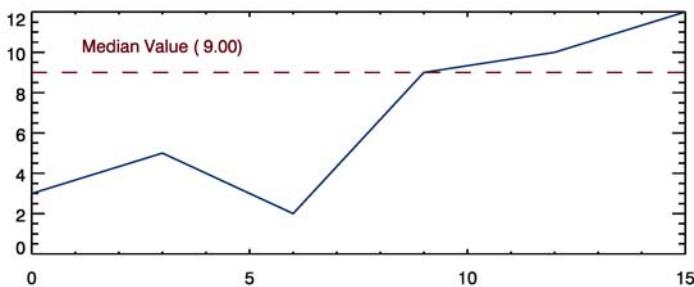


Figure 26: Lines and text can be drawn on line plots with *PlotS* and *XYOutS*.

Understanding Coordinate Systems

Up to now, we have been using a data coordinate system to locate where we place the line and the label on the plot. IDL, however, has three coordinate systems we can use: data, normalized, and device (sometimes also called “pixel”). The normalized coordinate system has coordinates that range from 0 to 1 in the graphics window space. The device coordinate system is based on the number of pixels in the graphics window.

The default coordinate system for the *PlotS* and *XYOutS* commands, as for almost all IDL graphics commands (except for those dealing with images), is the data coordinate system. To specify locations in one of the other coordinate systems, we have to set the appropriate *Normal* or *Device* keyword to indicate which coordinate system we want.

As a general rule for producing device independent IDL programs, you should always locate plot annotations with data or normalized coordinates. Do not ever use device coordinates if you can avoid it. I almost always prefer to use normalized coordinates if I can. But data coordinates often work, too. IDL makes it easy to convert values from one coordinate system to another with the *Convert_Coord* command.

For example, if you want to know what the data value (2.5,10) is in normalized coordinates, so you can use normalized coordinates to locate the median value label, you can write the code below. You use keywords to *Convert_Coord* to indicate which coordinate system your data is in currently, and which coordinate system to convert it to. Note that the return value is always a 3 x N array, where N is the number of values you are converting (just one, in this case). Each value is converted to an (x,y,z) triple. In this case, the Z value will be zero, since we are not passing a Z value to be converted.

```
IDL> normalCoords = Convert_Coord(2.5, 10, /Data, $  
           /To_Normal)  
IDL> Print, normalCoords  
0.277260      0.777783      0.000000  
IDL> XYOutS, normalCoords[0], normalCoords[1], text, $  
           Alignment=0.5, Color=cgColor('red8'), /Normal
```

To see why normalized coordinates are often useful, suppose you want to label the plot we just made with the date in the lower-right corner of the plot, just inside the plot axes. You want the date to be close to the axes, but not too close, nor too far away. It is much easier to estimate a “reasonable” distance using normalized coordinates than it is to do so with data coordinates, which might well vary from plot to plot.

But, how do you know where the lower-right corner of the plot is in normalized coordinates? Well, if you used the *Position* keyword to position the plot you would know. It would be at the location indicated by (*position[2],position[1]*). But, we didn’t use the *Position* keyword in this case.

Fortunately, IDL knows exactly where it put the axes in the plot. And it not only keeps track of the end points of the axes in data coordinate space (*!X.CRange* and *!Y.CRange*), it keeps track of the axes in normalized coordinate space, too, in the form of the system variables *!X.Window* and *!Y.Window*.

The location we are looking for is (*!X.Window[1]*, *!Y.Window[0]*). With this information, we can estimate that “reasonably close” is within, say, 0.025 of this location. So we put the date on the line plot with these commands. Note that we are right-justifying the text.

```
IDL> xloc = !X.Window[1] - 0.025  
IDL> yloc = !Y.Window[0] + 0.025  
IDL> XYOutS, xloc, yloc, /Normal, SysTime(), $  
           Alignment=1.0
```

Selecting Label Points Interactively

I mentioned previously that when we are labeling a plot for the first time, we sometimes need to select the location for a label by inspection. We

need to find a location in the plot, for example, which doesn't obscure the underlying data or detract from the overall design of the plot.

One of my favorite ways to select the location for a label is to do it interactively. I display the plot, without a label, in the graphics window, then I "audition" several locations by clicking in the window and drawing the label there. I almost always use normalized coordinates for this operation. I use the *Cursor* and *XYOutS* commands like this.

```
IDL> Plot, elapsedTime, vector
IDL> Cursor, x, y, /Down, /Normal & XYOutS, x, y, $
    /Normal, 'Test Label', Alignment=0.5
```

When I click the mouse button down in the graphics window, my "Test Label" is drawn, centered, at that location. If I don't much care for that location, I "erase" it, and click my cursor somewhere else. I "erase" the label by drawing over it in the background color. This is not really erasing anything, but it often has the same effect if the label is placed in a location where only the background color is present.

```
IDL> XYOutS, x, y, /Normal, 'Test Label', $
    Alignment=0.5, Color=!P.Background
IDL> Cursor, x, y, /Down, /Normal & XYOutS, x, y, $
    /Normal, 'Test Label', Alignment=0.5
```

When I am satisfied, I incorporate this location into the program that displays the line plot.

Creating a Histogram Line Plot

A simple application of the *Plot*, *PlotS*, *PolyFill* and other simple graphics commands can produce a dramatically better histogram plot than you can produce in IDL by using the normal off-the-shelf commands.

The standard IDL method to create a histogram plot is to use the symbol index 10 as the plotting symbol in a *Plot* command. As an example, we plot the histogram of an image we load with [cgDemoData](#). The histogram is calculated with the IDL *Histogram* command. The *Histogram* command has a justifiably famous reputation among IDL programmers for solving a number of difficult programming challenges quickly and easily. It forms the heart of programming in what is called *The IDL Way*. But, it's not for the faint of heart. (See JD Smith's infamous *Histogram Tutorial* on the [Coyote's Guide to IDL Programming](#) web page for additional information. This is one of those articles it pays to read a couple of times every year,

whether you think you need to or not. I learn something new every time I read it!)

In the code below, the `Scale_Vector` command is used to construct an independent data vector from the minimum and maximum values used to construct the histogram. These values are returned by the `OMin` and `OMax` output keywords to `Histogram`. The `NaN` keyword is set so that any non-finite data in the input is ignored in building the histogram.

```
IDL> image = cgDemoData(7)
IDL> h = Histogram(image, Binsize=5B, /NaN, $
                   OMin=omin, OMax=omax)
IDL> x = Scale_Vector(Findgen(N_Elements(h)), $
                      omin, omax)
IDL> Plot, x, h, PSym=10, Max_Value=15000, XStyle=1, $
      YTitle='Number of Pixels', XTitle='Pixel Value'
```

You see the result in Figure 27.

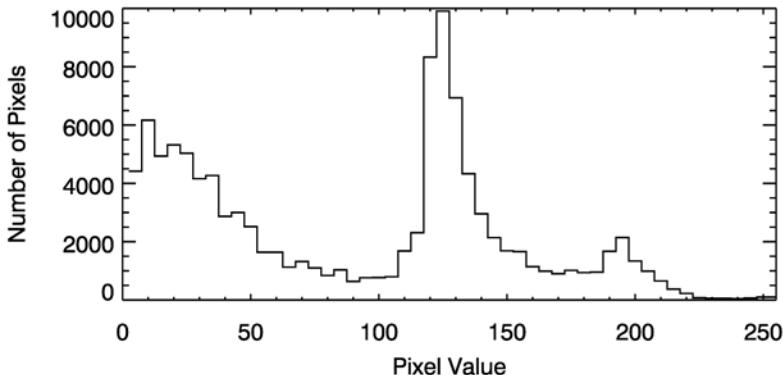


Figure 27: A histogram plot in IDL is created by using the `Plot` command and setting the `PSym` keyword to 10.

A more sophisticated histogram plot can be constructed using IDL traditional graphics commands. One example is the `cgHistoplot` command from the [Coyote Library](#). Here is the same data plotted with `cgHistoplot`.

```
IDL> cgHistoplot, image, /Fill, Max_Value=10000, $
      YTitle='Number of Pixels', XTitle='Pixel Value'
```

You see the result in Figure 28. The `cgHistoplot` command has many features that are not available in the normal histogram plotting function. You

can use different colors, different fill patterns, overlay histograms on top of other histograms and so on. You can even display the plot in a resizeable graphics window by setting the *Window* keyword.

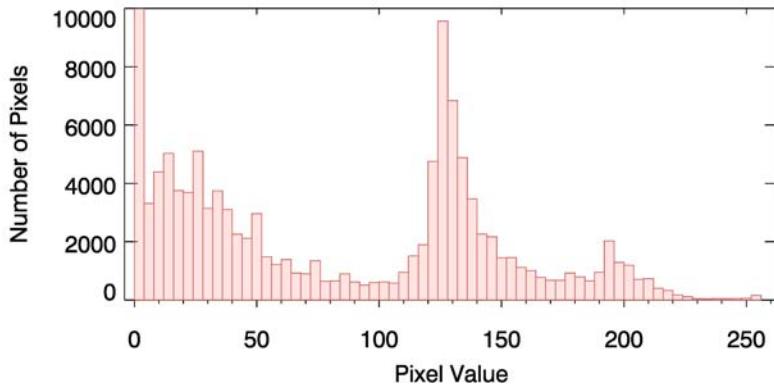


Figure 28: The `cgHistoplot` command creates a more refined histogram plot than simply setting the `PSym` keyword to 10.

Here are some commands you can try.

```
IDL> data = RandomU(5L, 200)*20
IDL> !P.Multi = [0, 2, 2]
IDL> cgHistoplot, data, BinSize=1.0
IDL> cgHistoplot, data, BinSize=1.0, /Fill, $
      PolyColor=['charcoal', 'dodger blue']
IDL> cgHistoplot, data, BinSize=1.0, /LINE_FILL, $
      PolyColor=['charcoal', 'dodger blue'], $
      Orientation=[45, -45]
IDL> cgHistoplot, data, BinSize=1.0, /Fill, $
      PolyColor='sky blue', MinInput=0
IDL> moredata = RandomU(3L, 80)*20
IDL> cgHistoplot, moredata, BinSize=1.0, /Fill, $
      PolyColor='royal blue', /Oplot, MinInput=0
IDL> !P.MULTI = 0
```

You see the result in Figure 29.

It is also possible to obtain and/or plot the cumulative probability distribution of the histogram. The cumulative probability vector, scaled into the data range 0 to 1, is returned via the *Probability* keyword. If you want to plot the cumulative probability vector, set the *OProbability* keyword. You

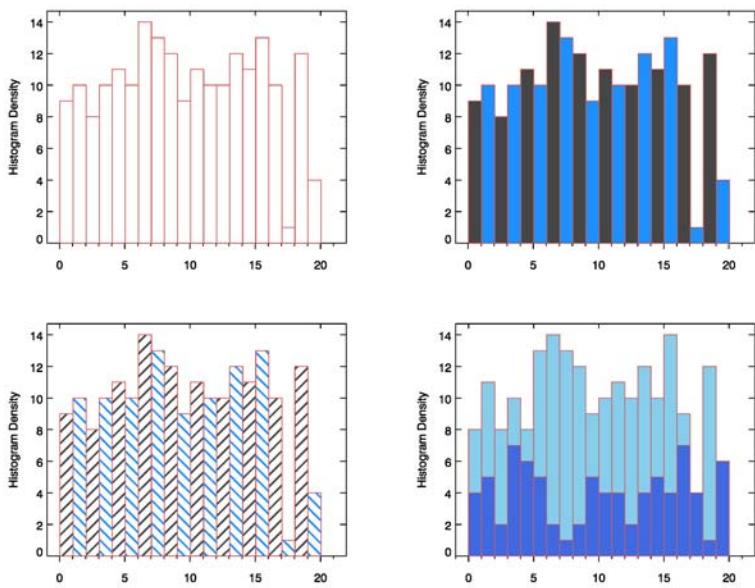


Figure 29: The `cgHistoplot` command adds new features to plotting data histograms, including colors, textures, and the ability to overlap histogram plots on the same set of axes.

can chose the color for the probability vector with the `ProbColorName` keyword.

```
IDL> cgHistoplot, cgDemoData(7), /Fill, $  
      ProbColorName='blu6', /OProbability
```

You see the result in Figure 30.

Handling Missing Data in a Line Plot

There are several possible ways to handle missing data in a line plot. To see how this works, let's first simulate integer type data, where the missing data values are represented by a large positive integer, say 999. Type these commands.

```
IDL> data = Long(cgDemoData(1))  
IDL> data[40:49] = 999  
IDL> Plot, data
```

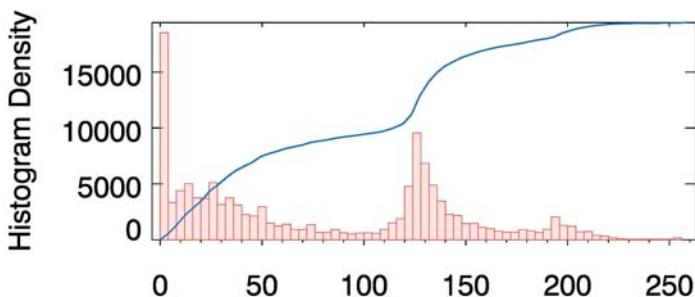


Figure 30: The `cgHistoplot` command with the cumulative probability function over-plotted in blue.

You see the result in the top panel of Figure 31. Note how the actual data is “washed out” by the very large values of the missing data and is barely visible.

One way to handle this missing data is to use the `Max_Value` keyword to the `Plot` command to screen out any data greater than the maximum value designated by this keyword. (There is a corresponding `Min_Value` keyword, too, if your missing data is designated by a very small or negative number.)

```
IDL> Plot, data, Max_Value=100
```

You can see in the bottom panel of Figure 31 that there is a gap in the line plot where the data is greater than the maximum value.

Another, and in many ways more useful, method of handing missing data is to set the missing values to the bit representation of Not-A-Number, frequently written as NaN, and pronounced as “naan”. Unfortunately, these NaN bit patterns apply *only* to floating point and double precision floating point numbers. In IDL the bit patterns are represented by the system variables `!Values.F_NaN` and `!Values.D_NaN`.

To emphasize the point that they apply *only* to single or double precision floats, let’s try to apply them to the integer data we have now. No keywords are required if the input to the `Plot` command contains NaN values. The `Plot` command will simply ignore the NaN values when drawing the plot.

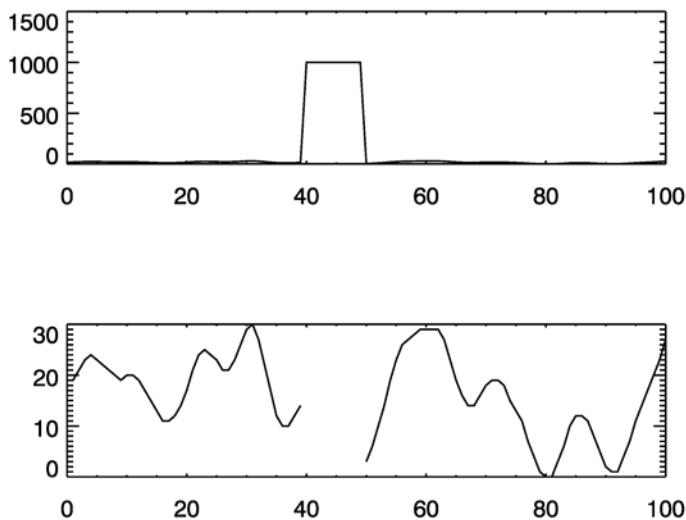


Figure 31: Missing data can be “screened out” of line plots by using the `Max_Value` or `Min_Value` keywords to the `Plot` command.

```
IDL> data = Long(cgDemoData(1))
IDL> data[40:49] = !Values.F_NaN
IDL> Plot, data
```

You see the result in Figure 32. Whoa! What has happened here!?

What has happened is that this NaN bit pattern, which is supposed to be for floating point numbers, has been interpreted as a long integer. That long integer happens to be a very large negative number.

```
IDL> Print, data[40]
-2147483648
```

To use NaNs correctly, apply them only to single and double precision floating point variables. In practice, this means you might have to cast your variable to floating point type first.

```
IDL> data = Float(data)
IDL> data[40:49] = !Values.F_Nan
IDL> Plot, data
```

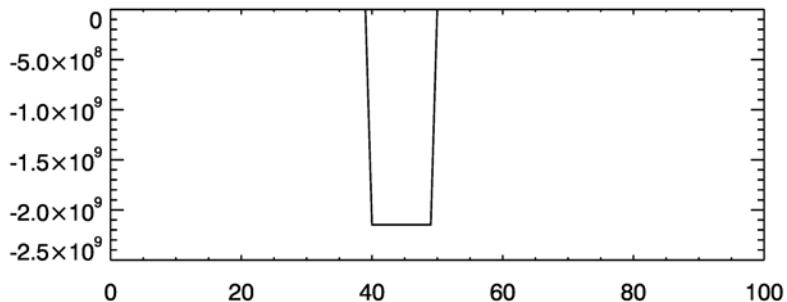


Figure 32: If you apply the !Values.F_NaN bit pattern to integer variables, the results can be unexpected.

You see an example of the correct way to use NaNs for missing data in Figure 33.

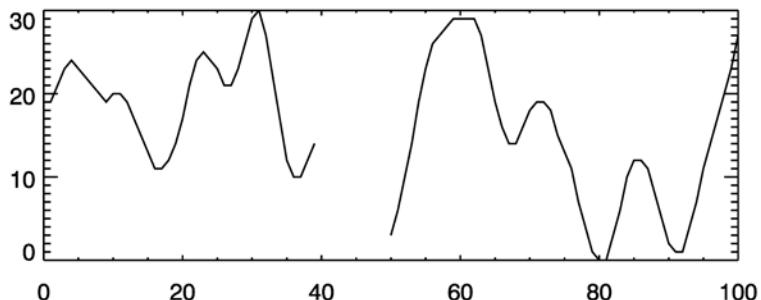


Figure 33: The data must be cast to single or double precision floats before using NaNs for missing data.

Using a Refurbished Line Plot Command

The `cgPlot` command from the [Coyote Library](#) is a device independent, color model independent wrapper for the *Plot* and *Oplot* commands. It is part of a suite of programs, collectively called the Coyote Graphics System (which includes `cgContour`, `cgSurf`, `cgColorFill`, `cgText`, and other programs), for creating traditional graphics output that work and look the same on all devices and in any color decomposition state. They all also work in the resizable graphics window, `cgWindow`. `cgPlot` takes care of many of the details necessary to write device independent graphics pro-

grams, works with colors in a more natural way by specifying colors directly, and has features that are not available in the *Plot* command.

The Coyote Graphics System commands produce, by default, black output on white backgrounds. This is the opposite of IDL traditional commands, but makes it possible to use these commands to produce (as much as possible) identical looking PostScript output. (You can return to the traditional color scheme of white on black by setting the *Traditional* keyword on the command.)

You will notice that textual output from these commands is slightly larger than normal. There are two reasons for this. First, larger output more closely matches the look and feel of the corresponding PostScript output when PostScript or TrueType fonts are used. And, second, larger output more closely matches the look and feel of similar object graphics programs in IDL 8.

One of the unique features of *cgPlot* is that you can independently color the plot symbols (with the *SymColor* keyword), the plot axes and annotation (with the *AxisColor* keyword), and the data itself (with the *Color* keyword). Here, for example, is how to create a plot with green axes, with the data drawn in red, with blue symbols. You can use any of the 46 symbols supported by the symbol catalog *SymCat*. In this example, we use filled circles.

```
IDL> data = cgDemoData(1)
IDL> cgPlot, data, PSym=-16, Color='red', $
      SymColor='blue', AxisColor='dark green'
```

You see the result in Figure 34.

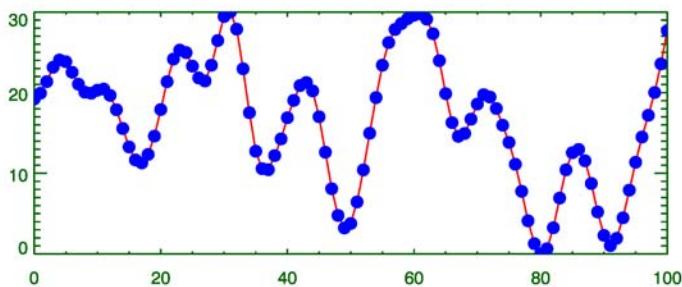


Figure 34: With *cgPlot* you can independently color the axes, data, and symbols.

Another interesting feature is that you can set the aspect ratio of the plot (ratio of height to width). This is sometimes important when creating PostScript output and you want to be able to measure a distance using a ruler on a piece of paper, for example. If you want to make a plot with a height two-thirds the width, you can type a command like this.

```
IDL> cgPlot, data, Aspect=2/3.0, Title='Aspect Plot'
```

You see the result in Figure 35. Note that the plot is centered in the display window.

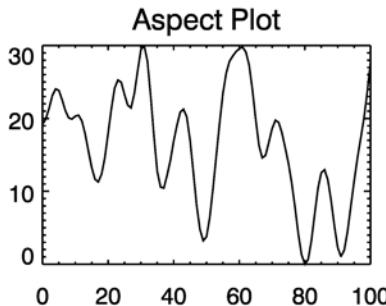


Figure 35: Plots containing a specified aspect ratio can be created by setting the `Aspect` keyword of `cgPlot`.

Finally, it is easy to create multi-colored lines and symbols when `cgPlot` is used in conjunction with `cgPlotS`, another [Coyote Library](#) program for creating device independent graphics. In `cgPlotS` the `Color`, `SymColor`, and `SymSize` keywords can all be vectors (this is not possible with `PlotS`), giving the user the opportunity to display other “dimensions” of their data with size or color. Here is a contrived example to make the point.

```
IDL> data = cgDemoData(1)
IDL> time = Scale_Vector(Findgen(101), 0, 6)
IDL> LoadCT, 33, /Silent
IDL> cgPlot, time, data, /NoData
IDL> cgPlotS, time, data, PSym=-16, $
    Color=BytScl(data), SymColor=BytScl(data), $
    SymSize=Scale_Vector(data, 1.0, 2.5)
```

You see the result in Figure 36.

One huge advantage of the `cgPlot` command is that all drawing of graphics is done using decomposed color whenever possible, so that no drawing colors are loaded into the one physical color table. This is possible for any

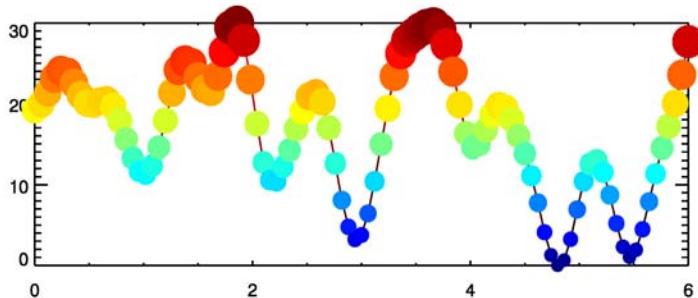


Figure 36: `cgPlotS` accepts vectors for the `Color`, `SymColor` and `SymSize` keyword, allowing you to add extra “dimensions” to your line plots.

24-bit graphics display device, which includes nearly every computer, the Z-graphics buffer since IDL 6.4 and the PostScript device since IDL 7.1. This greatly eliminates the possibility of “corrupting” the color table, which is normally used to display images.

The `cgPlot` command produces identical results, including the correct background color, on your display, in the Z-graphics device, and—unlike the `Plot` command, which can only produce a white background—in the PostScript device. For example, this command produces exactly the same result in all common graphics devices.

```
IDL> cgPlot, data, Background='rose', Color='navy'
```

You see the PostScript result in Figure 37.

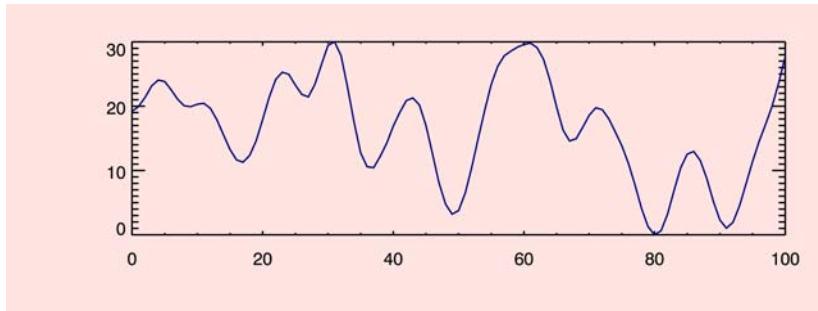


Figure 37: The `cgPlot` command produces the asked for background color even in the PostScript device, unlike the `Plot` command.

Line Plots in Resizeable Graphics Windows

If you want to see a line plot by itself in a resizeable graphics window, with controls for making hardcopy output files, set the *Window* keyword. The plot is displayed in `cgWindow`.

```
IDL> cgPlot, time, data, /NoData, /Window
IDL> cgPlotS, time, data, PSym=-16, $
      Color=BytScl(data), SymColor=BytScl(data), $
      SymSize=Scale_Vector(data, 1.0, 2.5), /AddCmd
```

Note that the second, `cgPlotS`, command is “added” to the `cgWindow` program by setting the *AddCmd* keyword. There is no limit to the number of commands you can add to `cgWindow` to be “executed” when the graphics window is resized. It is possible to send the output (i.e., the commands added to the program) of `cgWindow` directly to a PostScript file, or to save the contents of the graphics window as a raster image file to share with colleagues, post on a web page, or include in a presentation. Use the *Save As...* button in the upper-left corner of the graphics window. It is also possible to save the contents of the graphics window (the *visualization*) to a file, which you could e-mail to a colleague or simply open at some later time to view again.

You see the result in Figure 38.

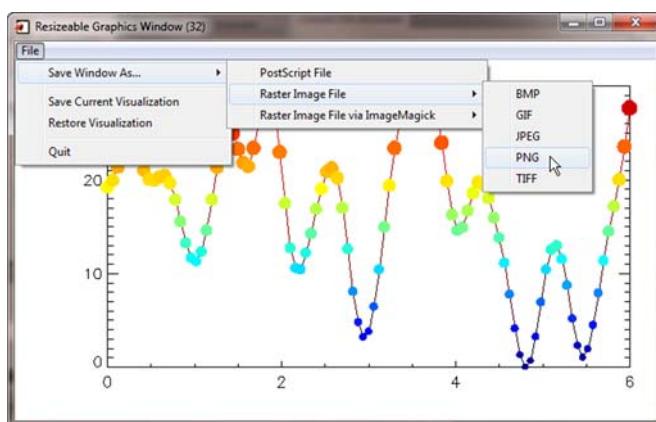


Figure 38: `cgWindow` can be used to create a line plot in a resizeable graphics window. Contents can be saved in various raster image formats and in a PostScript file. The contents of the window (the visualization) can be saved to a file and restored at a later time.

Chapter 5



Creating Contour Plots

Basic Contour Plots

Nearly everyone is familiar with what a contour plot is from looking at topographical maps. A contour plot is a way of representing a three-dimensional surface on a flat, two-dimensional surface. The third dimension is represented by contour lines, usually drawn, at least on topographical maps, at some equally-spaced contour interval.

Almost any two-dimensional data set can be contoured in IDL. The values of the two-dimensional data will represent the “height” or third dimension to be contoured. To learn how contour plots are created, we will load a simple 2D array with the `cgDemoData` command, a [Coyote Library](#) routine you downloaded to use with this book. You can use the *Help* command to determine that this is a 41 x 41 floating point array. The *Min* and *Max* commands allow you to determine the array’s data range, which in this case is 0 to 1550.

```
IDL> data2D = cgDemoData(2)
IDL> Help, data2D
      DATA2D          FLOAT      = Array[41, 41]
IDL> Print, Min(data2D), Max(data2D)
      0.000000      1550.00
```

A contour plot in its most basic form can be created just by passing this array as the argument of the IDL traditional graphics command *Contour*, like this.

```
IDL> Contour, data2D
```

You see the result in Figure 1. This is a PostScript rendition of the contour plot. On the display, this is rendered as white lines on a black background.

(We will have more to say about colors in just a moment.) Not terribly impressive, I admit. But, wait, it gets better.

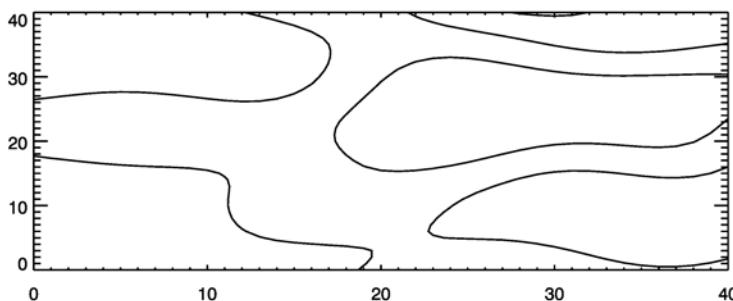


Figure 1: The most basic contour plot in IDL. There are 60+ keywords that you can use to enhance the information content of the *Contour* command to make it significantly more valuable to you.

There are more than 60 different keywords you can use with the *Contour* command to make the contour plot significantly more valuable to you. Before we start to talk about the 15-20 of those keywords that you will absolutely want to know how to use to add more features to a contour plot, notice what is already there. The axes are labeled!

You will notice that both the horizontal or X axis, and the vertical or Y axis have ranges that extend from 0 to 40. This should remind you of what happens with the *Plot* command when you only pass the dependent data set to the command. IDL creates the independent data to plot the dependent data against.

The same thing is going on here, except that IDL needs to create both an X and a Y vector that specify the locations of each value in the 2D data array. By default, these two vectors are just index vectors that are the same size as the dimensions of the data array. In other words, IDL essentially executes these commands, when the *Contour* command is called as it was in the statement above.

```
IDL> s = Size(data2D, /Dimensions)
IDL> xvec = IndGen(s[0])
IDL> yvec = IndGen(s[1])
IDL> Contour, data2D, xvec, yvec
```

The X and Y data vectors must be monotonically increasing (or decreasing) in value, but they do *not* have to be regularly spaced. The X and Y data

parameters also don't have to be vectors. They can be 2D arrays with the same dimensions as the data array. Each element of the X and Y arrays will then locate the corresponding element in the data array.

Normally, the X and Y data vectors represent some physical property of the data or the way it was collected. For example, X and Y might represent the longitude and latitude locations of the data, or they might represent physical properties like temperature and pressure.

Let's assume we are contouring satellite data and the X and Y vectors represent longitude and latitude locations. We want to express them, of course, in map units. Normally, for these kinds of map contours, the X and Y vectors would be expressed in terms of projected meters. If we assume this data represents, say, atmospheric pressure collected by satellite and gridded to 25 km grid cells, centered over Boulder, Colorado, USA, in an orthographic map projection, then the projected meter values we are talking about extend from -512500 to +512500 in both X and Y. (Don't get too hung up on these details. I have some points I want to make about contour plots, and sometimes it is easier to make those points using imagination rather than reality. You will have plenty of time to use the techniques I am going to show you with messy real data.)

If I want my data range to go from -512500 to +512500, then the center of the first grid cell will be at -500000 and the center of the last grid cell will be at +500000. I can make my vectors like this.

```
IDL> s = Size(data2D, /Dimensions)
IDL> lon = IndGen(s[0]) * 25000L - 500000L
IDL> lat = IndGen(s[1]) * 25000L - 500000L
IDL> Print, Min(lon), Max(lon)
      -500000      500000
IDL> Print, Min(lat), Max(lat)
      -500000      500000
```

These are large values to display on an axis, so it would be better to divide these by, say, 1,000 and indicate this in the plot annotation. We could write code like this.

```
IDL> lon = lon / 10^3
IDL> lat = lat / 10^3
IDL> xtitle = 'Longitude (projected meters x 1000)'
IDL> ytitle = 'Latitude (projected meters x 1000)'
IDL> Contour, data2d, lon, lat, XTitle=xtitle,
      YTitle=ytitle, Title='Atmospheric Pressure'
```

You see the result in Figure 2.

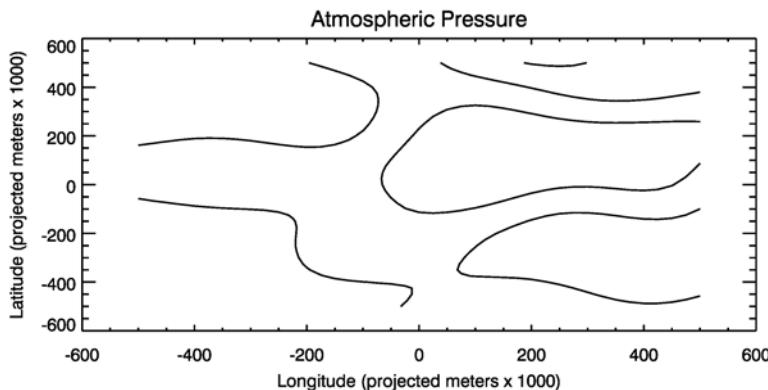


Figure 2: This plot is not a whole better than the first, except that the axes are labeled properly. Or are they? Note our old friend axis auto-scaling rearing its ugly head!

Visually, this contour plot is not much better than the previous plot, but at least the axes are labeled properly. Or, are they? Here we see a *very* common problem with contour plots in IDL that we first encountered with line plots. Namely, the contour axes are auto-scaled. They are setting end points that reflect IDL's aesthetic sensibilities more than they do our desire for a decent looking contour plot. I would say this is a problem in contour plotting about 80 percent of the time.

The solution, of course, is the same as the solution for line plots: turn axis auto-scaling off by setting the *[XYZ]Style* keywords to 1. The command you want to use is this.

```
IDL> Contour, data2d, lon, lat, XTitle=xtitle, $  
      YTitle=ytitle, Title='Atmospheric Pressure', $  
      XStyle=1, YStyle=1
```

Before we look at the new contour plot results, let's fix one other problem with the plot. Contour plots are not very useful to us unless the contour lines are labeled. And, usually, we need more contour lines than the default contour plot gives us. We can choose the number of contour lines we want to draw (sort of, I'll explain this in detail in just a moment) by setting the *NLevels* keyword. And we can choose which contour lines to label by setting the *C_Labels* keyword. The *C_Labels* keyword is a vector, whose value is set to 1 if you want that contour level labeled and to 0 if you do not want it labeled.

To draw, say, 12 contour levels and label them all, we can type this command. Note the use of the *Replicate* command, which replicates the value 1 twelve times and returns a 12-element vector.

```
IDL> Contour, data2d, lon, lat, XTitle=xtitle, $  
YTitle=ytitle, Title='Atmospheric Pressure', $  
XStyle=1, YStyle=1, NLevels=12, $  
C_Labels=Replicate(1, 12)
```

You see the result in Figure 3.

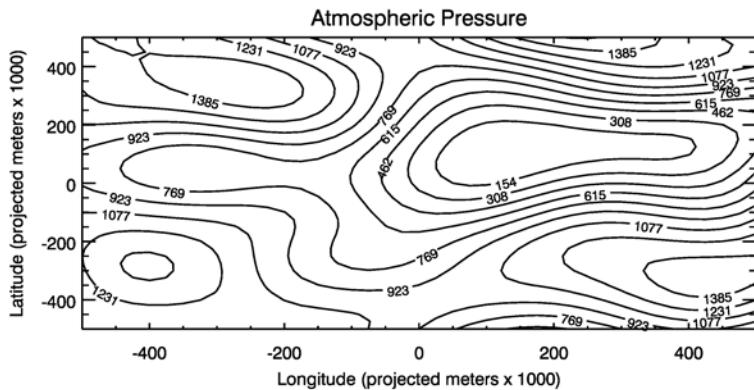


Figure 3: Finally, the contour plot is getting into the realm of the useful for us. Maybe the contour labels are too close together in certain areas of the contour plot. But this is a problem that can be addressed.

This contour plot is starting to contain the kind of information that will make it useful to us. But there are still modifications we can make to improve it even more. One problem we can fix, for example, is that the contour labels appear to be too close together in some areas of the plot, particularly in the upper right-hand corner. But we will address and solve this problem shortly.

Before we do, let's address another problem. The *Contour* command is rapidly becoming too long to type!

Advantages of Keywords versus System Variables

In addition to the 50 or so general purpose graphics keywords that apply to most IDL traditional graphics commands (you were introduced to perhaps 20 of these in the line plot chapter), there are perhaps 10-12 more

keywords that apply specifically to contour plots. Most of these additional keywords start with a “*C*_” prefix to identify them specifically as contour plot keywords. The *C_Labels* keyword is an example . These keywords can be discovered by consulting the IDL on-line help for the “CONTOUR procedure.” IDL on-line help can be summoned like this.

```
IDL> ? contour
```

In IDL 8 and above, this will lead you straight to the new graphics IDL contour *function*, which is not what you need here. Locate the *Index* of the IDL on-line help application and find the *IDL Contour procedure*.

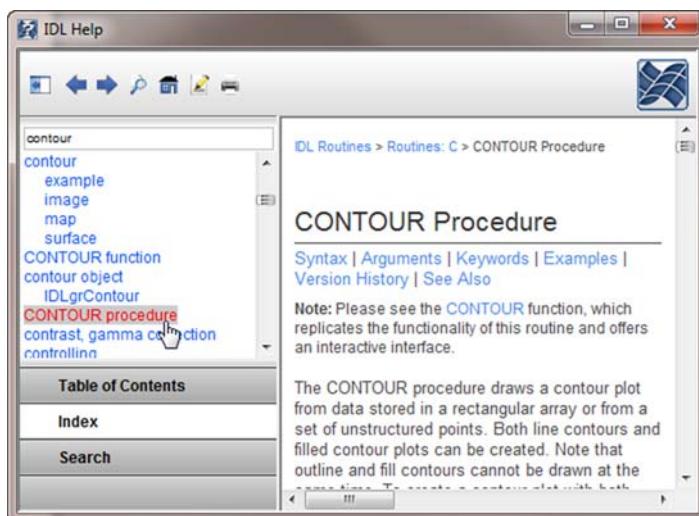


Figure 4: To learn which keywords are available for the IDL traditional graphics Contour command, be sure you locate the right entry in the IDL on-line help application. This shows the IDL help application for IDL 8.0.

Many of the standard graphics commands have counterparts in the IDL system variables, *!P*, *!X*, *!Y*, and *!Z*. I don’t, in general, recommend setting these system variable counterparts because it is too easy to become confused about what is happening to your IDL graphics output when these system variables are set. Frankly, you will forget you have set them. Or, even if you remember you have set them, you will forget what you set them to.

In my own programming, I stay away from setting plotting system variables whenever possible. (And one of the reasons it takes me *hours* to

debug a problem with a colleague's IDL graphics program is that they *do* use graphics system variables in a program far removed from the one they have sent me to debug. Aaaauugghhh!)

But, that said, I am going to break my rule. We are going to be using the same data, displayed much the same way, throughout this chapter. I don't want to type long contour commands any more than you do. And, yet, I want the examples in this book to be simple enough to type at the IDL command line.

So, here are the system variables I am going to use.

```
IDL> !X.Title = 'Longitude (projected meters x 1000)'
IDL> !Y.Title = 'Latitude (projected meters x 1000)'
IDL> !P.Title = 'Atmospheric Pressure'
IDL> !P.Color = cgColor('black')
IDL> !P.Background = cgColor('white')
IDL> !X.Range = [-500, 500]
IDL> !Y.Range = [-500, 500]
IDL> !X.Style = 1
IDL> !Y.Style = 1
```

Now, to produce the same plot as before, I only have to use the contour plot specific keywords. The command looks like this.

```
IDL> Contour, data2d, lon, lat, NLevels=12, $
      C_Labels=Replicate(1, 12)
```

Note: Remember, when you are typing an IDL keyword you only have to type enough letters to make it a unique keyword for the command. You don't have to type the full name of the keyword, although I always do for the sake of clarity. I wouldn't think of using a shortened keyword name if I was writing an IDL program. Short keyword names will make your programs especially hard for you and others to read and debug later. But, it's certainly okay to shorten keywords if you are noodling around at the IDL command line.

Customizing Contour Plots

Before we do anything else, let's fix the problem we identified earlier. The contour labels are too close together in some areas of the contour plot, particularly in the upper right-hand corner.

One way we can attempt to solve this problem is to make the contour labels themselves smaller. The contour labels are drawn by default with a

character size of 0.75. But we can set the size with the contour plot specific keyword *C_CharSize*. We could try, for example, a label size of 0.5.

```
IDL> Contour, data2d, lon, lat, NLevels=12, $  
      C_Labels=Replicate(1, 12), C_CharSize=0.5
```

You see the result in Figure 5. This improves the situation somewhat, but almost makes the contour labels too small to read now.

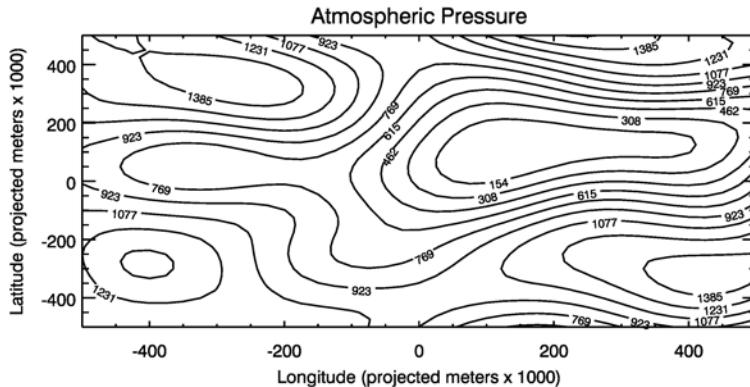


Figure 5: The contour labels can be sized independently of the contour plot titles and other annotations.

Unfortunately, it is not possible to determine exactly where the labels appear in traditional graphics contour plots. We do not have access to the algorithm that determines this property. But another solution we can try is to label every other contour interval. To do this, we need a vector of alternating 1s and 0s to pass to the *C_Labels* keyword. A quick way to create such a vector is like this.

```
IDL> everyOther = Reform(Rebin([1,0], 2, 6), 12)  
IDL> Print, everyOther, Format='(12I2)'  
1 0 1 0 1 0 1 0 1 0 1 0
```

And we use it like this.

```
IDL> Contour, data2d, lon, lat, NLevels=12, $  
      C_Labels=everyOther
```

Another way to label every other contour level takes us all the way back to IDL 4, but you still see it in use today. This method sets the *Follow* keyword. Originally, the *Follow* keyword would select the “contour-following,” rather than the “cell-filling” algorithm for creating the contour

plot. One side-effect of the contour-following drawing algorithm was to label every other contour line. Today, all contour plots use the contour-following algorithm to draw the contours, but the keyword persists because of this useful side-effect. This command, then, has the same effect as the previous command.

```
IDL> Contour, data2d, lon, lat, NLevels=12, /Follow
```

You see the result in Figure 6.

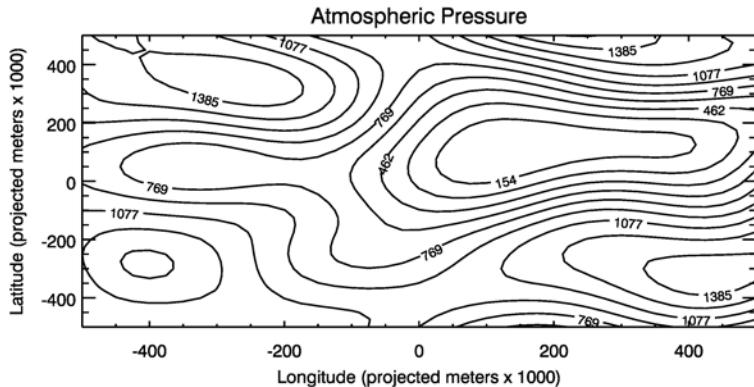


Figure 6: Labeling every other contour line can be accomplished with the modern *C_Labels* keyword or the much older *Follow* keyword.

Notice that neither of the contours in the lower-left corner of the plot are labeled, even though they are adjacent to one another. The smaller contour inside the larger should be labeled. But contour lines must have a non-specified minimum length to be labeled. This contour line is too short to be labeled. You don't have control over this, and the only solution is to make your contour plot larger, so that particular contour line will become long enough to be marked for labeling.

Selecting Contour Levels

The IDL documentation describes the *NLevels* keyword to the *Contour* command as representing the “number of equally spaced contour levels”. If this keyword is not used, then “approximately six levels are drawn.” This turns out to be a bit of fiction, as I will demonstrate in just a moment. In fact, the number of contour levels specified with the *NLevels* keyword is

highly dependent on the data being contoured and is rarely, if ever, equal to the number of levels you set with this keyword. It's true that as this number gets larger, more contour lines are drawn, but don't rely on the number of lines being accurate.

Consider the default case of "approximately six levels".

```
IDL> Contour, data2d, lon, lat, C_Labels=Replicate(1,6)
```

As you can see in Figure 7, the contour plot drew (maybe if we are being generous) three levels, at most. But nothing like six.

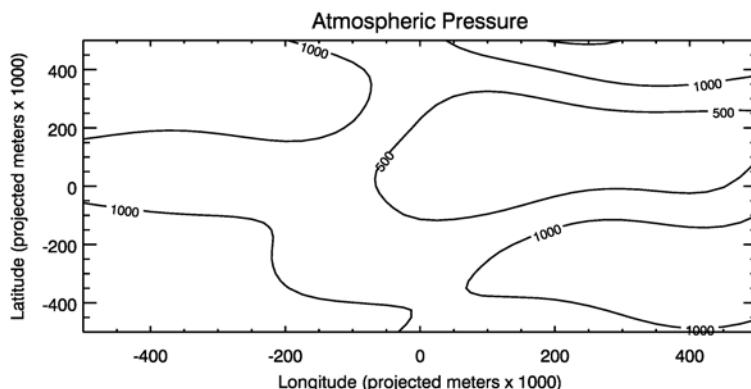


Figure 7: The NLevels keyword does not, generally, give you “N contouring levels” as claimed by the IDL documentation. Treat this as an approximation to the number of contour levels.

As it turns out with this particular data set, that when we ask for 12 contour levels, we actually get nine. But I will postpone this demonstration until the section dealing with filled contour plots, when this will become evident.

The bottom line is this. If you want N contour levels, then you need to define those levels yourself and specify them with the *Levels* keyword to the *Contour* command, rather than with the *NLevels* keyword. The *Levels* keyword is how you specifically select levels in your data to contour.

For example, if you want to draw contours at 250, 500, 750, and 1000 pressure units in this contour plot, you would type these commands.

```
IDL> levels = [250,500,750,1000]
IDL> Contour, data2d, lon, lat, Levels=levels, $
      C_Labels=Replicate(1,4)
```

You see the result in Figure 8.

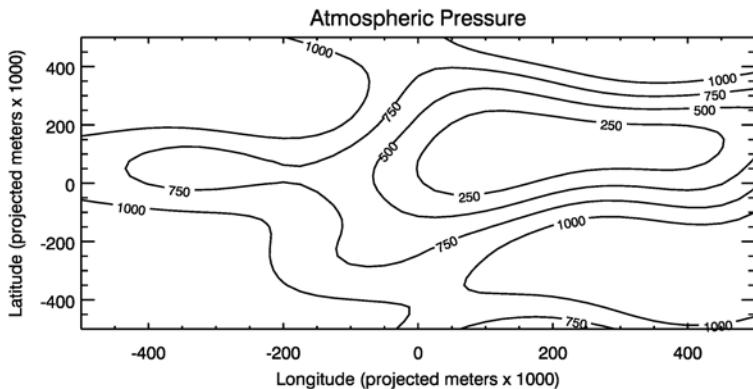


Figure 8: Contour levels can be selected and specified directly with the `Levels` keyword. For consistent results, the `Levels` keyword should be used instead of the `NLevels` keyword to specify contour levels.

If you want exactly 12 contour levels, you would have to calculate those levels yourself. Do not use the `NLevels` keyword. You would do it like this.

```
IDL> nlevels = 12
IDL> step = (Max(data2d) - Min(data2d)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(data2d)
IDL> Contour, data2d, lon, lat, Levels=levels, /Follow
```

You see the result in Figure 9. You can compare this with Figure 6 to see that this result is different from creating the contour plot as we did before with the keyword `NLevels=12`.

Note that the contour levels do not always have to be labeled with the value of the contour level. You can use the `C_annotation` keyword to choose alphanumeric annotations for the contour levels. For example, you could label the contour levels 250, 750, and 1200 as "low," "medium," and "high," like this.

```
IDL> threeLevels = [250, 750, 1200]
IDL> annotations = ['Low', 'Medium', 'High']
IDL> Contour, data2d, lon, lat, Levels=threeLevels, $
      C_annotation=annotations
```

You see the result in Figure 10.

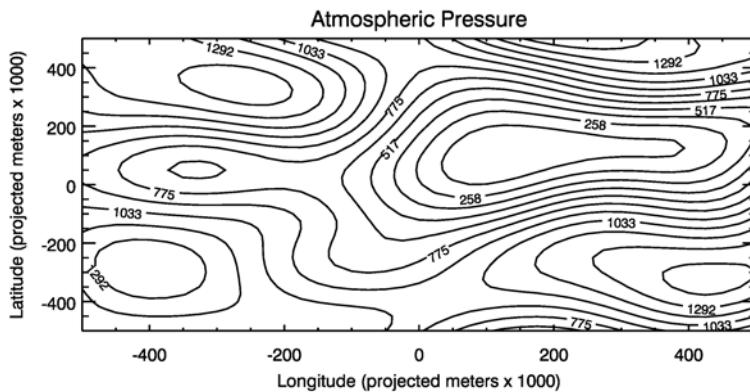


Figure 9: To get exactly 12 contour levels, you must calculate the levels yourself, rather than specifying 12 levels with the `NLevels` keyword.

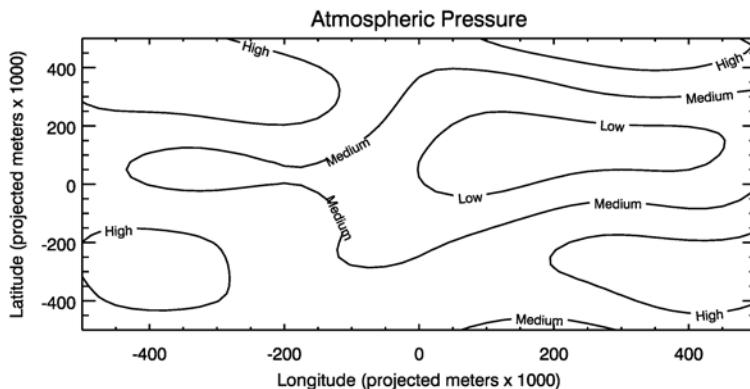


Figure 10: Contour levels can be labeled with alphanumeric labels.

Modifying Contour Lines

Contour lines can be drawn in the usual battery of line styles (see page 84). For example, to draw every other contour line in a dashed line style, we can use the `CLineStyle` keyword like this.

```
IDL> Contour, data2d, lon, lat, Levels=levels, $
      CLineStyle=[0,2], C_Labels=everyOther
```

You see the result in Figure 11.

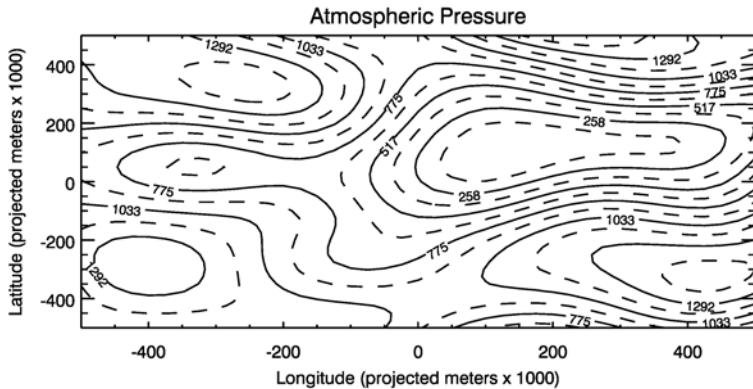


Figure 11: Contour lines can be displayed in different line styles. Use the *C_Linestyle* keyword to select a line style for each contour level.

Note that the *CLineStyle* keyword is a “wrapping” keyword. In other words, its values “wrap around” and can be used over again if there are more than just two contour lines to draw. Most of the contour keywords are wrapping keywords. The one notable exception to this rule is the *CLabels* keyword. If you set this keyword with just two values, as you just did for the *CLineStyle* keyword, then only the first two contour lines will be labeled according to the values present in the vector. I don’t know a good reason why the *CLabels* keyword defies the rule. I’ve often had cause to wish it didn’t.

Sometimes you want to make a thicker contour line at regular intervals. For example, we can make every third line a thicker line by using the *CThick* keyword like this.

```
IDL> everyThird = Reform(Rebin([0,0,1], 3, 4), 12)
IDL> Contour, data2d, lon, lat, Levels=levels, $
      C_Thick=[1,1,2], C_Labels=everyThird
```

You see the result in Figure 12.

It still might be difficult to determine which direction in a contour plot is the downhill direction. We can use the *Downhill* keyword to put small tick marks in the downhill direction of the contour lines, like this.

```
IDL> Contour, data2d, lon, lat, Levels=levels, $
      C_Thick=[1,1,2], C_Labels=everyThird, /Downhill
```

You see the result in Figure 13.

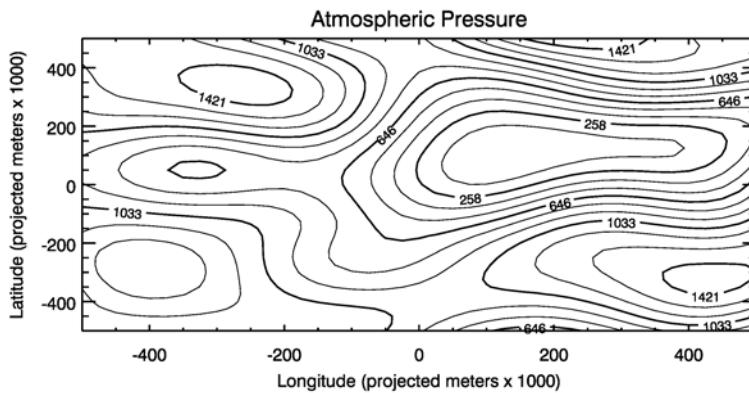


Figure 12: Every third line in the contour plot is made thicker with the `C_Thick` keyword.

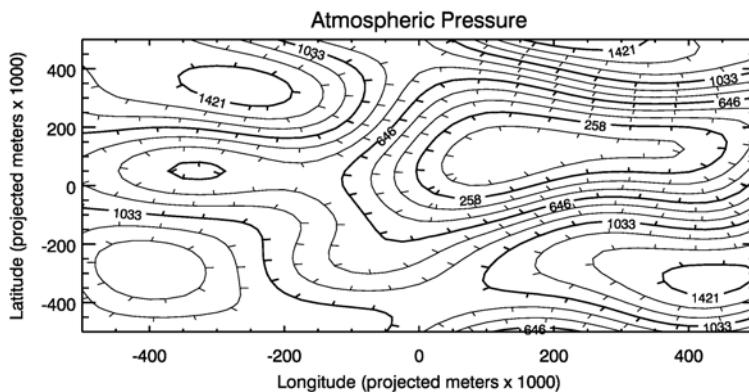


Figure 13: Ticks are placed in the downhill direction using the `Downhill` keyword.

Adding Color to Contour Plots

Contour plots can be drawn in a different color by setting the `Color` keyword. But, as with line plots, this will affect both the contour lines and the contour plot axes and annotations. Normally, when we are adding color to a contour plot, we want the axes to be one color and the contour lines to

be drawn in some other color or colors. The contour line colors can be selected with the *C_Colors* keyword to the *Contour* command.

The *C_Colors* keyword is a vector that describes, almost always, the color table indices with which each contour line is to be drawn. The only exception I am aware of is when these color indices are translated into 24-bit integers by `cgColor` so that the contour plot can be drawn using the color decomposition model. This, in fact, is how `cgContour`, a program you will learn more about later in this chapter, manages to draw color contour plots without ever loading colors into the physical color table.

This implies two very important points about adding color to contour plots. First, we need to load colors into the color table at the indices we specify in the *C_Colors* vector. And, then, we typically need to set ourselves up to be in indexed color mode to be able to see the proper colors. (See “Understanding IDL Color Models” on page 36 for additional information about color modes and models.) If we are using the default decomposed color mode, and we load color indices as the contour colors, then no matter what colors we have loaded into the color table, the contour colors will be displayed in shades of red!

I typically load contour colors at the bottom of the color table, so they don’t interfere with the default locations of other drawing colors loaded with `cgColor`. I also often use Brewer color tables, rather than the ones supplied with IDL. If you do use IDL color tables, be careful which ones you select. Most of the IDL-supplied color tables use white and black as colors at either end of the color table. These are generally not the colors you want to use in a contour plot. If you use `cgLoadCT` to load IDL color tables, rather than `LoadCT`, you can use the *Clip* keyword to clip these troublesome colors from either end of the color table. (Black and white colors are sometimes used, however, to indicate either missing or totally saturated areas in the data.)

In the example here, there are 12 contour levels, so I need 12 contour colors. I choose to load these colors at the bottom of the color table, starting in color index 1. I do not ever load contour colors into color index 0 or 255, or it becomes impossible to produce PostScript output correctly. Since I depend on PostScript output to produce nice looking IDL plots for presentations, web output, and books, this is critical to me. Here is how I load colors from the IDL Blue-Red color table.

```
IDL> LoadCT, 33, NColors=12, Bottom=1
```

This color table has blue at one end of the color table and red at the other, so it is appropriate for contour colors. If I had wanted, say, to use the Standard Gamma II color table, I might have done something like this.

```
IDL> LoadCT, 5, NColors=12, Bottom=1
```

This color table has a black color at one end of the color table and a white color at the other end, as shown in Figure 14.

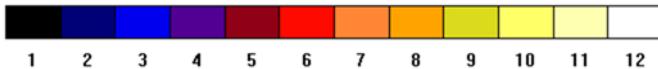


Figure 14: The twelve colors loaded by `LoadCT` for contour colors. Notice the black and white color at either end of the color range.

I can clip the black and white colors from this color table with `cgLoadCT` and the `Clip` keyword. In this example, colors are selected uniformly from the color table using the range of color indices 30 to 240, rather than from the usual 0 to 255.

```
IDL> cgLoadCT, 5, NColors=12, Bottom=1, Clip=[30,240]
```

You see the result in Figure 15.



Figure 15: The same color table as before, but with the black and white colors “clipped” from each end by loading the colors with `cgLoadCT` and using the `Clip` keyword.

Using Colors in Contour Plots

Next, I set IDL to indexed color mode, saving the current color mode so I can set it back after I draw the contour plot.

```
IDL> Device, Get_Decomposed=currentMode
IDL> Device, Decomposed=0
```

Note: It is not possible to use both keywords at the same time with the `Device` command. If I do use both keywords, it will work correctly on Windows machines (the current mode will be fetched before it is changed to another mode), but it will fail on UNIX machines (the mode will be changed before the current mode is fetched). This operation must always be done in two steps to work in a device independent way. (Learn about the `SetDecomposedState` command below, which solves this problem for you.)

Finally, I draw the contour plot, using the *NoData* keyword to suppress drawing the contour lines. The contour lines are placed on the contour plot using the *Overplot* keyword and are drawn in color by setting the *C_Color* keyword to the proper color index values (in this case, 1 to 12). When I am done drawing the contour plot, I return to the color mode in effect before I changed the mode.

```
IDL> LoadCT, 33, NColors=12, Bottom=1
IDL> Contour, data2d, lon, lat, Levels=levels, /NoData, $
      Background=cgColor('white'), $
      Color=cgColor('black')
IDL> Contour, data2d, lon, lat, Levels=levels, $
      /Overplot, C_Colors=IndGen(12)+1, $
      C_Labels=everyOther
IDL> Device, Decomposed=currentMode
```

You see the result in Figure 16.

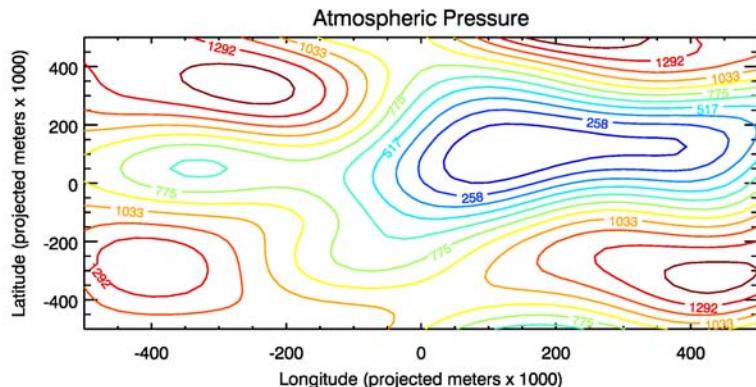


Figure 16: Contour lines can be drawn with colors, using the *C_Colors* keyword to indicate which color index in the color table should be used to select the color. This implies the indexed color mode is in effect when the plot is drawn.

Setting the Color Decomposition State

When we start to work with color table indices, such as those specified with the *C_Colors* keyword, in contour plots, we put ourselves in a position where we *must* use the indexed color model to display the colors. This is a problem with the code we have just written because it sets the

color decomposition state with the *Decomposed* keyword to the *Device* command.

Of course, this works perfectly on the display, because both normal display devices (i.e., X and WIN) accept a *Decomposed* keyword for the *Device* command. But some devices, such as older PostScript devices (older than IDL 7.1), do not. On those devices, the code above will throw an error on the first or second line of code!

The way we get around this problem is to use two programs from the [Coyote Library](#) that encapsulate the device and version dependencies so that we can obtain the current decomposition state, and set the decomposition state of the current graphics device in a device and version independent way. The two programs are [GetDecomposedState](#) to get the current color decomposition state or mode, and [SetDecomposedState](#) to set the current decomposition state or mode. The [SetDecomposedState](#) program uses [GetDecomposedState](#) to obtain the current color decomposition state before it is changed. This also solves the problem of having to get the decomposition state in one command for UNIX machines and set it with another (see “Setting the Color Model” on page 51 for more information). You will see these programs used in the code throughout the rest of this chapter.

Naturally, we can do other things with color and contour lines. For example, we could draw most lines blue, and every third line red, using code like this.

```
IDL> TVLCT, cgColor('blu4', /Triple), 1
IDL> TVLCT, cgColor('red4', /Triple), 2
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, Levels=levels, /NoData, $
      Background=cgColor('white'), $
      Color=cgColor('black')
IDL> Contour, data2d, lon, lat, Levels=levels, $
      /Overplot, C_Colors=[1,1,2], C_Labels=everyThird
IDL> SetDecomposedState, currentState
```

You see the result in Figure 17.

Creating Color Filled Contour Plots

It is often when creating color filled contour plots that people run into the, well, peculiarities of the traditional graphics *Contour* command. It might be helpful to see how most people approach the problem of creating a

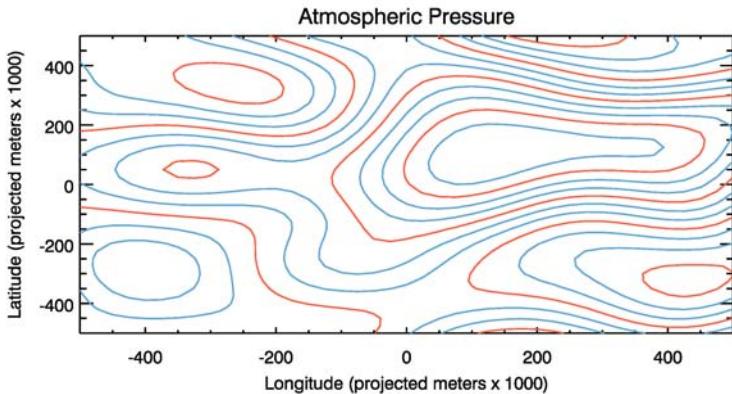


Figure 17: The C_Labels keyword is also a wrapping keyword. Here we draw contour lines in blue, except every third contour line is drawn in red.

color filled contour plot to see what some of the difficulties are and how to work around them.

Most people who want to create a color filled contour plot in IDL simply read the on-line documentation and see the directions call for loading a color table and setting the *Fill* keyword, and that's about it. Job done. Maybe they set *NLevels* to the number of contour levels they want to have in their filled contour plot.

Their first command might look something like this. Note I am adding a color bar to the contour command plot. This is common with filled color contour plots to give the user a sense of what the colors mean in the contour plot. IDL doesn't come with a traditional graphics color bar command, so I am going to use the [cgColorbar](#) program from the [Coyote Library](#) to produce the color bar above the contour plot. (Another alternative would be to use the discrete color bar command, [cgDCBar](#), if I didn't have too many contour levels.) Notice how I save room for the color bar by using the *Position* keyword on the *Contour* command.

```
IDL> LoadCT, 33
IDL> Contour, data2d, lon, lat, /Fill, NLevels=12, $
       Position=[0.125, 0.20, 0.95, 0.75], $
       Background=cgColor('white'), $
       Color=cgColor('black'), XStyle=1, YStyle=1
IDL> Contour, data2d, lon, lat, /Overplot, $
       Color=cgColor('black'), NLevels=12, $
       C_Labels=everyOther
```

```
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
    Divisions=12, XTicklen=1, XMinor=0, $
    AnnotateColor='black', CharSize=0.75, $
    Position=[0.125, 0.92, 0.95, 0.96]
```

What many people see on their display when they create this kind of color filled contour plot in IDL is shown in Figure 18.

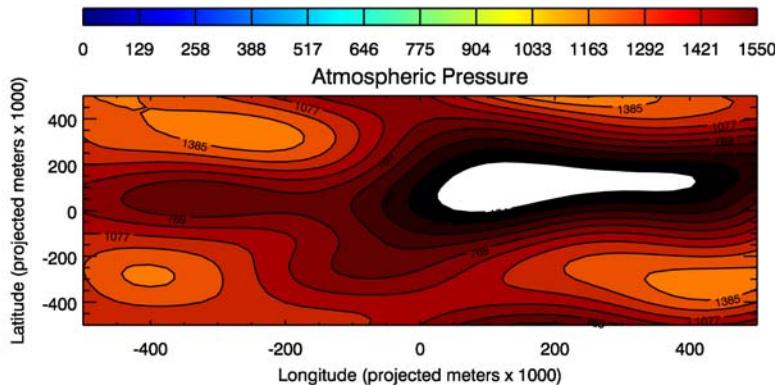


Figure 18: This is a typical result for many IDL users when they create a filled contour plot for the first time.

A number of things appear to be wrong with this contour plot. It certainly didn't use the colors we loaded into the color table. (You can see the contour plot is using different colors from those used in the color bar.) And there appears to be some kind of hole in the plot where we see the white background color. What in the world is going on here!?

Let's see if I can explain. The traditional graphics *Contour* command is very old. It was one of the first graphics commands added to IDL when the language was first written in the early 1980s. It was purchased from another software company to be added to IDL. Because it was purchased, the *Contour* command has always been something of a black box to IDL developers. Frankly, the code is not documented very well. It works, but no one seems to know exactly how it works!

Because of this, the *Contour* command has not changed much over the years. One way it hasn't changed is that in nearly 100 percent of the IDL programs I've ever seen, contour colors are expressed as indices into a color table. This means to use the *Contour* command successfully with

contour colors, you *must* be in indexed color mode. If you are not in indexed color mode, then you are by definition in decomposed color mode (the IDL default), and indexed colors in this mode will *always* appear in shades of red, no matter what colors you have loaded in the color table. The `cgColorbar` command is smart enough to put itself into the correct mode before it displays colors, but the `Contour` command is not.

So, you can solve the color problem and get yourself closer to what you want by just putting yourself in indexed color mode to use the `Contour` command.

```
IDL> LoadCT, 33
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, /Fill, NLevels=12, $
      Position=[0.125, 0.125, 0.95, 0.80], $
      Background=cgColor('white'), $
      Color=cgColor('black'), XStyle=1, YStyle=1
IDL> Contour, data2d, lon, lat, /Overplot, $
      Color=cgColor('black'), NLevels=12, $
      C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
      Divisions=12, XTicklen=1, XMinor=0, $
      AnnotateColor='black', Charsize=0.75, $
      Position=[0.125, 0.915, 0.955, 0.95]
```

You see the result in Figure 19.

We still have problems with this plot. For example, what about the colors themselves? We obviously are expecting 12 colors, since we asked for 12 contour levels with the `NLevels` keyword. In fact, (ignoring the white hole for a moment), we only find nine colors being used in the contour plot. On the other hand, the color bar looks like it is using 256 colors, but there appear to be some “extra” colors in the color bar itself. For example, there is a white line at about 388 and an extra black line about 517. Where are these colors coming from?

Two things are going on here. First, we have allowed IDL to choose colors from the color table for us. This is almost always a bad idea. By just setting the `Fill` keyword and not specifying which color indices we want to use with the `C_Colors` keyword we learned about in the previous section, IDL has “selected” some colors out of the color table. Which ones? Who knows! It would be better to load 12 colors at some defined location in the color table and use those specific 12 colors for both the contour plot and the color bar. We can modify the commands above to do this easily. We use the `NColors` and `Bottom` keywords when we load the colors, and we use

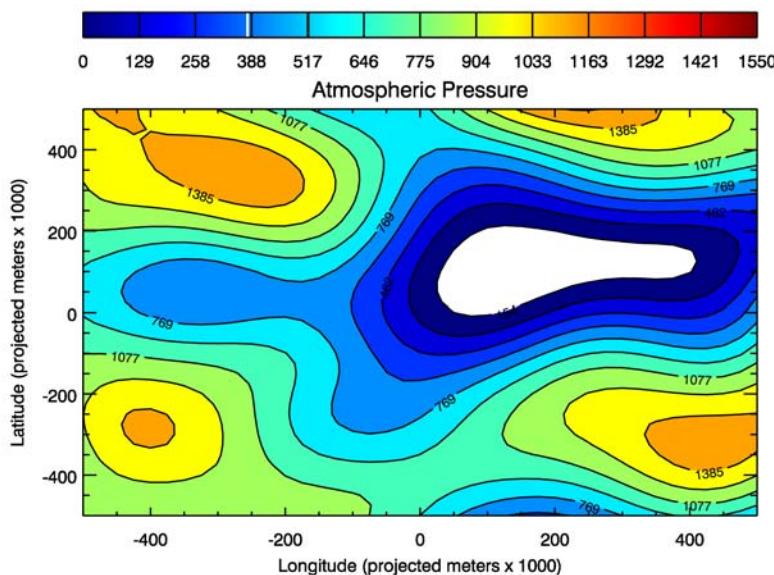


Figure 19: By running the *Contour* command in indexed color mode, the contour colors are (mostly!) correct.

C_Colors on the *Contour* command to specify those twelve color indices, starting at index 1 to be the colors we use in the contour plot.

```
IDL> LoadCT, 33, NColors=12, Bottom=1
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, /Fill, NLevels=12, $
    Position=[0.125, 0.125, 0.95, 0.80], $
    Background=cgColor('white'), $
    Color=cgColor('black'), XStyle=1, YStyle=1, $
    C_Colors=IndGen(12)+1
IDL> Contour, data2d, lon, lat, /Overplot, $
    Color=cgColor('black'), NLevels=12, $
    C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $
    Divisions=12, XTicklen=1, XMinor=0, $
    AnnotateColor='black', NColors=12, Bottom=1, $
    Position=[0.125, 0.915, 0.955, 0.95], $
    Charsize=0.75
```

You see the result in Figure 20.

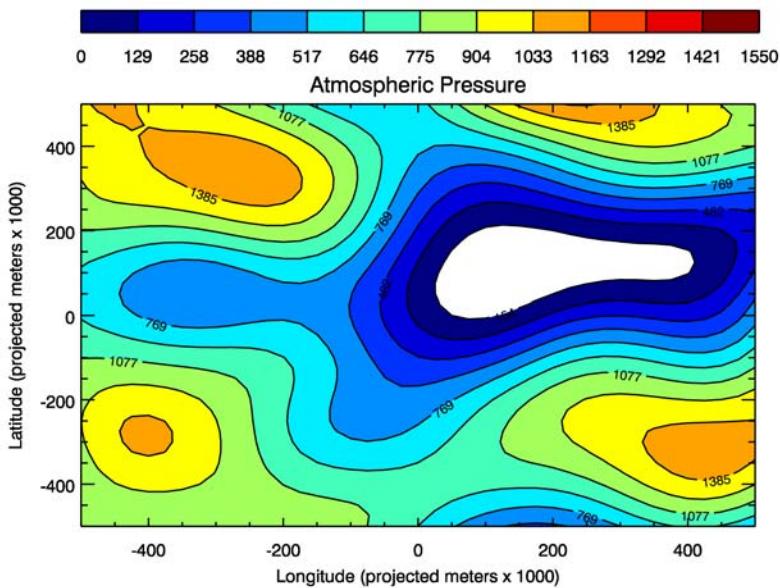


Figure 20: By choosing and loading only 12 colors, we can match the colors in the contour plot with the colors in the color bar. Well, almost. We still have a small problem that we are not using 12 colors in the contour plot!

If you look carefully, you will notice that the problem of extra lines in the color bar has gone away in this version of the contour plot. How come?

What was happening is that when you use indexed color mode your color table can become “dirty” or “contaminated” from other programs using it. There is only one color table and all programs running in IDL in indexed color mode get their colors from that same color table. If a program is forced to load colors in the color table, and you use that table without first loading it with the colors you expect to be there, those colors could well be wrong.

In our case, since we set ourselves in indexed color mode, and then used `cgColor` to load drawing colors for the contour plot, `cgColor` was forced to use the color table and those corrupted colors later showed up when we used all 256 colors for the color bar.

There is an absolute rule about using indexed color mode that you want to memorize. When using indexed color mode, always, *always* load the col-

ors you want to use from the color table *immediately* prior to using them. Don't assume they will be correct!

By restricting the colors to the bottom of the color table as we are doing in the last command, we don't run into, or use, colors that may have been loaded by `cgColor`. We are corrupting the color table for sure, but it is not doing us any harm, since we have separated the drawing colors from the contour fill colors. (If you want to see the colors you have loaded in the current color table, type `CIndex`.)

Note: I should mention that it is not absolutely required that indexed colors be used in contour plots. It is possible to use decomposed colors. In fact, `cgContour`, which you will learn about later in this chapter (on page 162), uses decomposed colors whenever possible. It uses `cgColor` to convert color table index values to decomposed color values. (See "Using `cgColor` With Color Table Indices" on page 45.) But I have seen thousands of IDL programs with contour plots in them, and `cgContour` is the only one I have ever seen that works with decomposed colors. That's why I think it is important to know how to set up indexed contour colors correctly.

With this plot it is now obvious we are only using nine contour intervals, not 12. For example, there are no red colors in the plot, even though we have red colors in the 12 colors of the color bar. It is also obvious that the labels on the contour lines don't match the values on the color bar. Why not?

The reason is that we have let IDL choose the contour intervals for us with the `NLevels` keyword, and whatever algorithm IDL is using to do this is not giving us the kind of results we expect. To create 12 contour intervals *exactly*, and have the color bar values match the contour labeling, we must create the contour levels ourselves. Then, we must use the `Levels` keyword to specify the levels and forget we ever heard about the `NLevels` keyword.

The code becomes something like this.

```
IDL> nlevels = 12
IDL> LoadCT, 33, NColors=nlevels, Bottom=1
IDL> step = (Max(data2d) - Min(data2d)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(data2d)
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data2d, lon, lat, /Fill, Levels=levels, $
    Position=[0.125, 0.125, 0.95, 0.80], $
    Background=cgColor('white'), $
    Color=cgColor('black'), XStyle=1, YStyle=1, $
    C_Colors=IndGen(nlevels)+1
```

```

IDL> Contour, data2d, lon, lat, /Overplot, $  

      Color=cgColor('black'), Levels=levels, $  

      C_Labels=everyOther  

IDL> SetDecomposedState, currentState  

IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $  

      Divisions=12, XTicklen=1, XMinor=0, $  

      AnnotateColor='black', NColors=12, Bottom=1, $  

      Position=[0.125, 0.915, 0.955, 0.95], $  

      CharSize=0.75

```

You see the result in Figure 21.

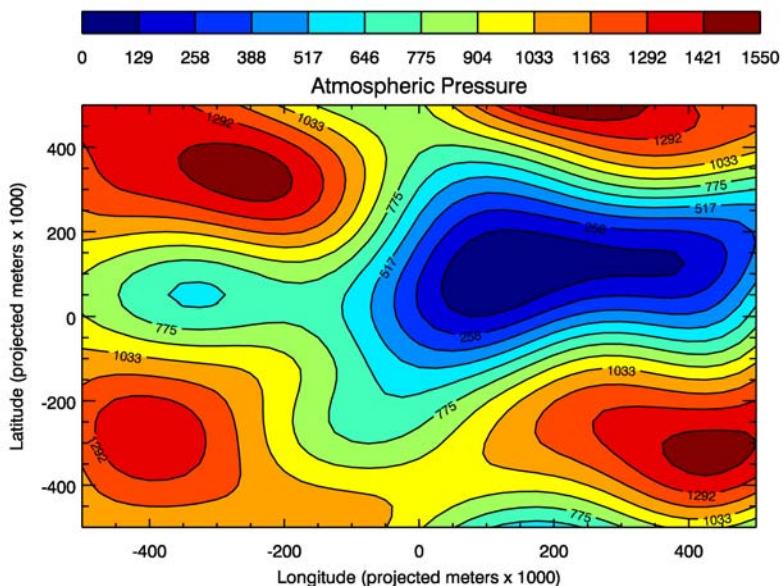


Figure 21: By specifying your own contour levels, the contour labels match the values in the color bar, and the hole in the contour plot disappears.

Notice that one consequence of defining your own contour levels is that the hole in the contour plot disappears! The other major benefit, of course, is that now your contour labels agree with the values in the color bar, and all 12 contour levels are colored with the appropriate colors.

Choosing a Different Fill Algorithm

There are occasions when the *Fill* keyword is not appropriate for creating filled contour plots. The algorithm that the *Contour* command uses to fill

contours can become confused if there are “open” contours to fill. Open contours are sometimes created by missing data and are often created by drawing filled contours on map projections set up with the *Map_Set* command. If you are having trouble with your filled contour plots, always try replacing the *Fill* keyword with the *Cell_Fill* keyword.

The *Cell_Fill* keyword uses a “cell filling” algorithm that is slightly less efficient and slower than the algorithm used by setting the *Fill* keyword. The upside, however, is that it is sometimes more accurate because it draws more contour polygons to fill. In the case of the data we have been using, the results would be identical for both keywords.

Filled Contours on Map Projections

If you are placing filled contour plots on map projections (especially those created with the *Map_Set* command), an excellent rule of thumb is to *always* use the *Cell_Fill* keyword rather than the *Fill* keyword. It is notoriously easy to inadvertently create open contours with map projections, and the danger is that if you don’t use *Cell_Fill*, the colors of your contours may be incorrect. And, what is worse, it will be extremely difficult to tell this is the case.

The only trick in putting filled contour plots on map projections is that all three positional parameters are required parameters in this case, and you must overplot the filled contours onto the map by setting the *Overplot* keyword.

Suppose we use this same data set, but adjust the X and Y positional parameters so they represent locations in the United States, represented by a longitude range of -124° to -66° and a latitude range of 25° to 50°. Recall that the data variable, *data2d*, is a 41 x 41 array. Normally, when I have to create variables of a particular size, with explicit endpoints, I use the [Coyote Library](#) routine [Scale_Vector](#) to do so. We can create the proper X and Y vectors like this.

```
IDL> lats = Scale\_Vector(Findgen(41), 25, 50)
IDL> lons = Scale\_Vector(Findgen(41), -124, -66)
```

Now we are ready to set up the map projection data coordinate space using the *Map_Set* command, like this. Note that we have to erase the window with a background color with a map projection to get the background color we want. The *NoErase* keyword on the *Map_Set* command is essential here to prevent the background color from being erased.

```
IDL> Erase, Color=cgColor('white')
IDL> Map_Set, /Mercator, 37.5, -95, /NoBorder, $
    Limit=[25, -124, 50, -66], $
```

```
Color=cgColor('black'), $  
Position=[0.05, 0.05, 0.95, 0.80], /NoErase
```

We put the filled contour command on the map like this. Here I am using a Brewer color table, whose colors are designed to work well on maps. I load the Brewer color table with the [Coyote Library](#) routine `cgLoadCT`.

```
IDL> nlevels = 12  
IDL> step = (Max(data2d) - Min(data2d)) / nlevels  
IDL> levels = IndGen(nlevels) * step + Min(data2d)  
IDL> cgLoadCT, 4, NColors=nlevels, Bottom=1, /Brewer, $  
/Reverse  
IDL> SetDecomposedState, 0, CurrentState=currentState  
IDL> Contour, data2d, lons, lats, /Cell_Fill, $  
Levels=levels, C_Colors=IndGen(nlevels)+1, $  
/Overplot  
IDL> Contour, data2d, lons, lats, /Overplot, $  
Color=cgColor('grey'), Levels=levels, $  
C_Labels=everyOther  
IDL> SetDecomposedState, currentState
```

Finally, we can add the map outlines and the color bar to complete the plot.

```
IDL> Map_Continents, Color=cgColor('charcoal')  
IDL> Map_Continents, Color=cgColor('charcoal'), /USA  
IDL> cgColorbar, Range=[Min(data2d),Max(data2d)], $  
Divisions=nlevels, XTicklen=1, XMinor=0, $  
AnnotateColor='black', NColors=nlevels, $  
Bottom=1, Position=[0.1, 0.87, 0.9, 0.90], $  
Title='Atmospheric Pressure', Charsize=0.75
```

You see the result in Figure 22.

Contouring Irregularly Sampled Data

The data we have passed to the `Contour` command so far has always been gridded two-dimensional data. Sometimes our data are not like that. For example, if we are collecting atmospheric pressure data, as we have been supposing in the examples so far, we may be sending weather balloons up to collect the data. Such data will be distributed in random locations over the sampling area.

Such a data array *must* be gridded before it can be contoured, but we have a couple of ways to proceed. We can either grid the data array ourselves and then pass it to the `Contour` command. Or, we can simply ask the `Con-`

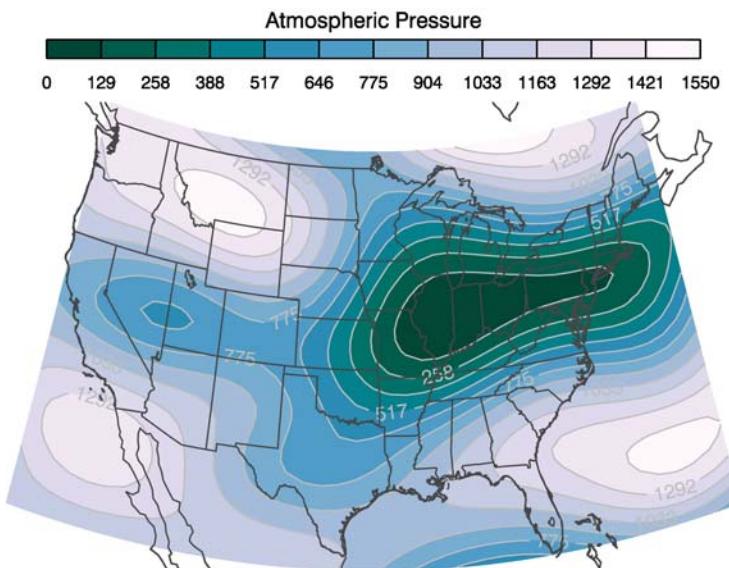


Figure 22: Be sure to use the `Cell Fill` keyword to add filled contour plots to map projections created with the `Map_Set` command. Map projections often create open contours that the `Fill` keyword doesn't handle well.

tour command to grid the data array for us, before contouring it, by setting the `Irregular` keyword. We have more control, and more options, if we grid the data array ourselves, so I will show you a simple gridding method you can use with your data.

It might be interesting to sample the data set we are using now, just to see how well we can reproduce the results with irregularly sampled data. To do so, we need to make two-dimensional arrays of our X and Y vectors. We can do that like this.

```
IDL> lat2d = Rebin(Reform(lat, 1, 41), 41, 41)  
IDL> lon2d = Rebin(lon, 41, 41)
```

Let's create 200 random points from this data set, add some random noise to the data, so they don't fall directly on a grid, and display them in a plot to see where they are located.

```
IDL> pts = Round(RandomU(-3L, 200) * 41 * 41)  
IDL> dataIrr = data2d[pts]
```

```
IDL> lonIrr = lon2d[pts] + RandomU(5L, 200) * 50 - 25
IDL> latIrr = lat2d[pts] + RandomU(8L, 200) * 50 - 25
IDL> Plot, lonIrr, latIrr, PSYM=4, XRange=[-500, 500], $
      YRange=[-500,500], XStyle=1, YStyle=1, $
      Title='Random Sampling Locations'
```

Note: In the code above I used particular numbers for the random number seed (i.e., $-3L$, $5L$, and $8L$). This is so your results will be identical to the results shown here. If you want truly random values, use the variable “seed” as the seed in the *RandomU* commands above.

You see the result in Figure 23.

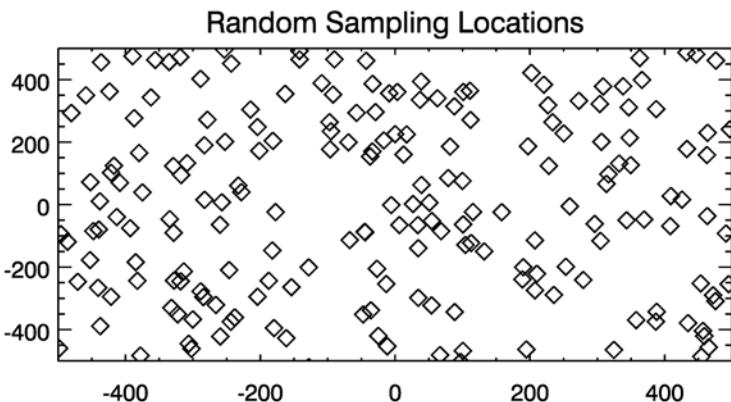


Figure 23: Here are the simulated random sampling locations of the data values we want to contour.

The simplest way to grid irregular data yourself is to use the two IDL routines *Triangulate* and *Trigrid*. This is the method used by the *Contour* command itself, although you don't have access to the parameters of the routines when doing the gridding directly with the *Contour* command.

The *Triangulate* command is used to produce a set of Delaunay triangles that the *Trigrid* command can use to produce the gridded data. Delaunay triangulation is an algorithm for producing a set of triangles from a set of points, such that a circle formed by connecting the vertices of any triangle does not contain any other point. Triangulations are sometimes plagued by collinear points. (One of the reasons contouring irregular data directly with the *Contour* command will sometimes fail.) You can use the *Repeat*

and *Tolerance* keywords to the *Triangulate* command to deal with circumstances like this.

To create the set of Delaunay triangles and return them in an output *triangles* variable, we use the *Triangulate* command.

```
IDL> Triangulate, lonIrr, latIrr, triangles
```

Now we are ready to pass these triangles to the *Trigrid* command. We can set the size of the output grid we want with the *NX* and *NY* keywords. The default grid size is 51 by 51. Since we are trying to reproduce the original data, we will set these keywords to produce a grid of 41 by 41.

The important thing in gridding irregular data to contour is to return from *Trigrid* the X and Y vectors that specify the locations of the gridded data. These can be obtained from the *XGrid* and *YGrid* output keywords. The command will look like this.

```
IDL> gridData = Trigrid(lonIrr, latIrr, dataIrr, $  
triangles, NX=41, NY=41, $  
XGrid=xgrid, YGrid=ygrid)
```

We can display this gridded data as a filled contour plot.

```
IDL> nlevels = 12  
IDL> step = (Max(gridData) - Min(gridData)) / nlevels  
IDL> levels = IndGen(nlevels) * step + Min(gridData)  
IDL> LoadCT, 33, NColors=nlevels, Bottom=1  
IDL> SetDecomposedState, 0, CurrentState=currentState  
IDL> Contour, gridData, xgrid, ygrid, /Cell_Fill, $  
Levels=levels, Background=cgColor('white'), $  
Position=[0.125, 0.125, 0.95, 0.80], $  
Color=cgColor('black'), XStyle=1, YStyle=1, $  
C_Colors=IndGen(nlevels)+1  
IDL> Contour, gridData, xgrid, ygrid, /Overplot, $  
Color=cgColor('black'), Levels=levels, $  
C_Labels=everyOther  
IDL> SetDecomposedState, currentState  
IDL> cgColorbar, Range=[Min(gridData),Max(gridData)], $  
Divisions=12, XTicklen=1, XMinor=0, $  
AnnotateColor='black', NColors=12, Bottom=1, $  
Position=[0.125, 0.915, 0.955, 0.95], $  
Charsize=0.75
```

You see the result in Figure 24. Compare this result with Figure 21. Not too bad, considering we used less than 15 percent of the original data points to construct the gridded data set.

This result should not look too different from doing the gridding directly with the *Contour* command by setting the *Irregular* keyword.

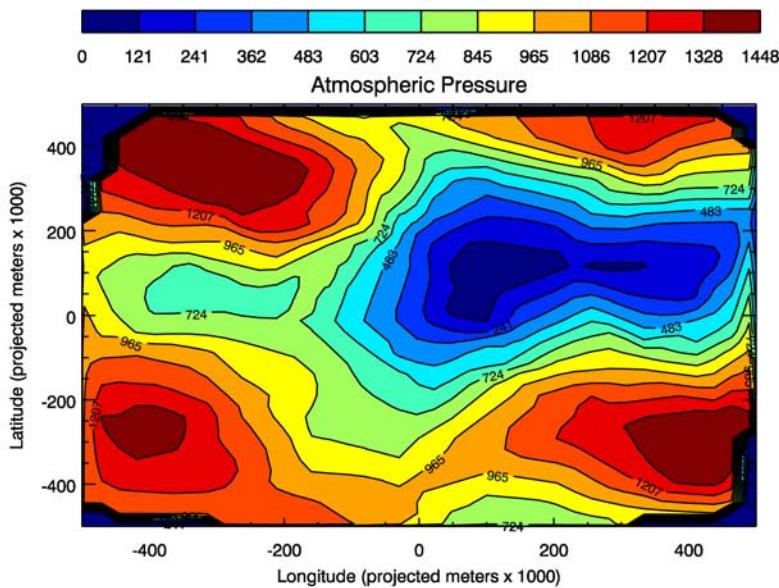


Figure 24: Irregular data can be gridded with Triangulate and Trigrid before being contoured.

```

IDL> nlevels = 12
IDL> step = (Max(dataIrr) - Min(dataIrr)) / nlevels
IDL> levels = IndGen(nlevels) * step + Min(dataIrr)
IDL> LoadCT, 33, NColors=nlevels, Bottom=1
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, dataIrr, lonIrr, latIrr, /Fill, $
      Levels=levels, Background=cgColor('white'), $
      Position=[0.125, 0.125, 0.95, 0.80], $
      Color=cgColor('black'), XStyle=1, YStyle=1, $
      C_Colors=IndGen(nlevels)+1, /Irregular
IDL> Contour, dataIrr, lonIrr, latIrr, /Overplot, $
      Color=cgColor('black'), Levels=levels, $
      C_Labels=everyOther, /Irregular
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(gridData),Max(gridData)], $
      Divisions=12, XTicklen=1, XMinor=0, $
      AnnotateColor='black', NColors=12, Bottom=1, $
      Position=[0.125, 0.915, 0.955, 0.95], $
      Charsize=0.75

```

You see the result in Figure 25 .

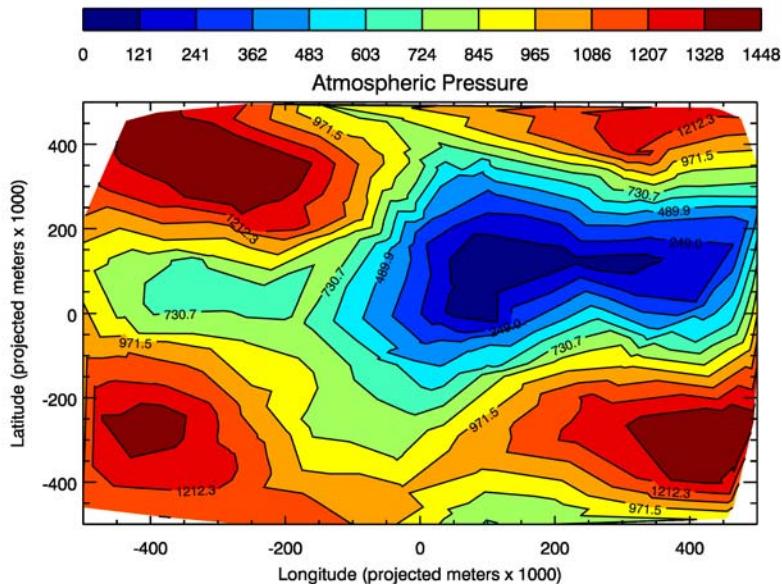


Figure 25: The same irregular data gridded by the `contour` command directly by setting the `Irregular` keyword.

Comparing Figure 25 to Figure 24, you see one obvious difference. The blue border around the outside of Figure 24 is missing in Figure 25. What is the `Contour` command doing that we didn't do when we gridded the data ourselves?

It turns out that the `Contour` command sets all gridded data that falls outside the Delaunay triangle boundary (or, another way of say this is outside the “convex hull” of the data points) to the missing value `Nan` (not a number). We can do this ourselves with the `Missing` keyword to `TriGrid`. Because we didn't use the `Missing` keyword, all our missing data values got set to 0. We probably want to set our missing values to something other than 0, but this introduces several other complications I think you should know about.

Suppose we decide to set all data outside the convex hull to the missing value `Nan`. Floating point `Nans` are represented in IDL with the system variable `!Values.F_NAN`. We would set the missing values like this.

```
IDL> gridData = Trigrid(lonIrr, latIrr, dataIrr, $  
triangles, NX=41, NY=41, $  
XGrid=xgrid, YGrid=ygrid, Missing=!Values.F_NAN)
```

But, in the rest of our code, we have to be careful to handle missing data correctly. For example, we have these two lines in our display code.

```
IDL> step = (Max(gridData) - Min(gridData)) / nlevels  
IDL> levels = IndGen(nlevels) * step + Min(gridData)
```

Look what happens when we check the values of these variables.

```
IDL> Help, step  
STEP      FLOAT   =  NaN  
IDL> Print, levels  
NaN  NaN
```

Yikes! Those kinds of values are not going to create a very good looking contour plot. To create the correct values, we are going to have to be careful to set the *NaN* keyword on these functions to properly exclude *NaN* values.

```
IDL> step = (Max(gridData, /NaN) - Min(gridData, /NaN)) $  
/ nlevels  
IDL> Help, step  
STEP      FLOAT   =  118.689  
IDL> levels = IndGen(nlevels) * step + $  
Min(gridData, /NaN)  
IDL> Print, levels  
23.9834      142.672      261.360      380.049  
498.737      617.426      736.115      854.803  
973.492      1092.18      1210.87      1329.56
```

We have several of these functions in our display code. Here is the modified code.

```
IDL> nlevels = 12  
IDL> step = (Max(gridData, /NaN) - Min(gridData, /NaN)) $  
/ nlevels  
IDL> levels = IndGen(nlevels) * step + $  
Min(gridData, /NaN)  
IDL> LoadCT, 33, NColors=nlevels, Bottom=1  
IDL> SetDecomposedState, 0, CurrentState=currentState  
IDL> Contour, gridData, xgrid, ygrid, /Fill, $  
Levels=levels, Background=cgColor('white'), $  
Position=[0.125,0.125,0.95,0.80], $  
Color=cgColor('black'), XStyle=1, YStyle=1, $  
C_Colors=IndGen(nlevels)+1  
IDL> Contour, gridData, xgrid, ygrid, /Overplot, $
```

```

Color=cgColor('black'), Levels=levels, $
C_Labels=everyOther
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Range=[Min(gridData, /Nan), $ 
      Max(gridData, /NaN)], Charsize=0.75, $ 
      Divisions=12, XTicklen=1, XMinor=0, $ 
      AnnotateColor='black', NColors=12, Bottom=1, $ 
      Position=[0.125, 0.915, 0.955, 0.95]

```

You see the result in Figure 26.

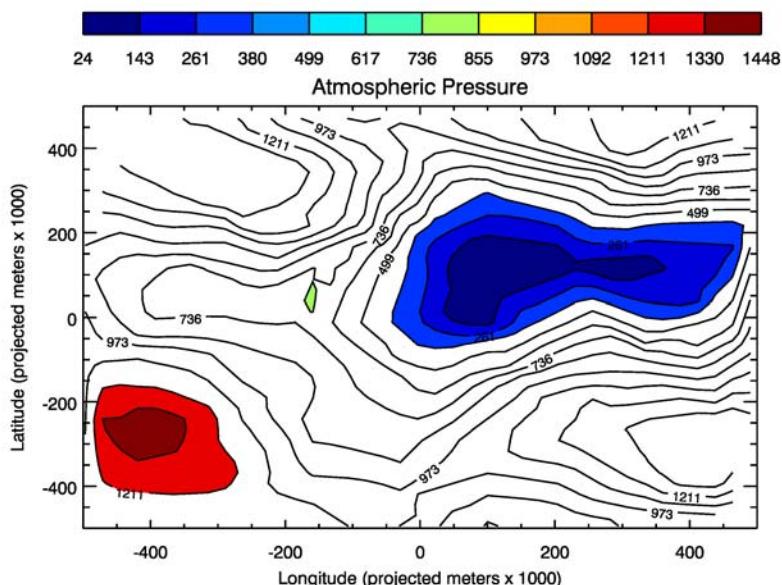


Figure 26: Something bad has happened to our filled contour plot!

Something bad is going on here! What has happened is that by introducing NaNs into the data, we have caused “open contours” to appear in the contouring algorithm. The *Fill* keyword to the *Contour* command selects an algorithm that doesn’t know how to cope with open contours very well. To correct the problem, we have to choose the *Cell_Fill* keyword instead.

Making just this one change in the code above, produces the result you see in Figure 27.

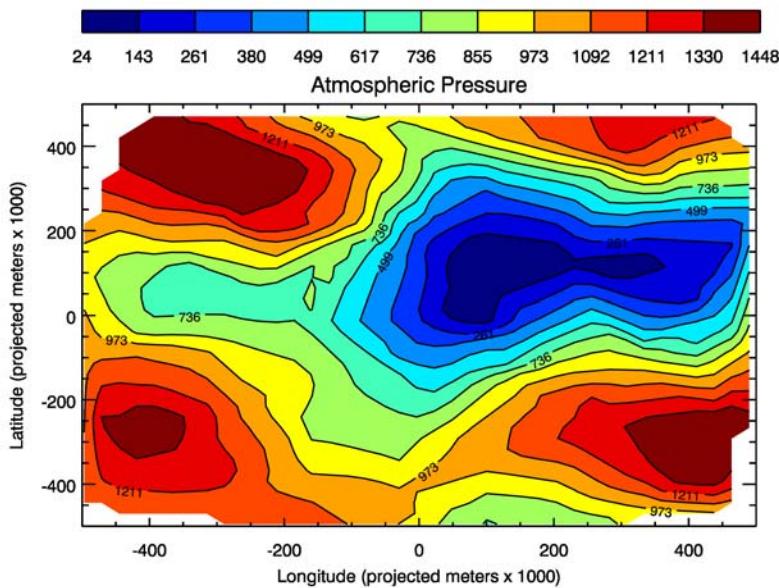


Figure 27: By changing the *Fill* keyword to *Cell Fill*, we produce the correctly filled contour plot.

This result can now be compared favorably to the *Contour* result itself in Figure 25.

Gridding Irregular Data

An even more powerful method of gridding data, because it has many more options than the *Triangulate/Trigrid* method we just discussed, is to use the IDL routine *GridData* in combination with the *QHull* command which can build Delaunay triangles. Let's try to use it to build a similar grid for contouring.

First, we build the set of Delaunay triangles, using *QHull*.

```
IDL> QHull, lonIrr, latIrr, triangles, /Delaunay
```

Next, we grid the data, using the triangles we just created. We have our choice of several different gridding methods, among which are inverse distance, kriging, linear interpolation, nearest neighbor, polynomial fitting and so on. We are going to use the inverse distance method, in which data points closer to the grid are weighted more heavily than grid points further away.

```
IDL> gridData = GridData(lonIrr, latIrr, dataIrr, $  
    Method='InverseDistance', Dimension=[41,41], $  
    Start=[-500, -500], Delta=[25, 25], $  
    Triangles=triangles, Missing=!Values.F_NaN)
```

So that we can use our current *lon* and *lat* vectors, we set the *Start*, *Delta*, and *Dimension* keywords to produce a grid that is identical to our original gridded data set.

We display the data like this.

```
IDL> nlevels = 12  
IDL> step = (Max(gridData, /NaN) - Min(gridData, /NaN)) $  
    / nlevels  
IDL> levels = IndGen(nlevels) * step + $  
    Min(gridData, /NaN)  
IDL> LoadCT, 33, NColors=nlevels, Bottom=1  
IDL> SetDecomposedState, 0, CurrentState=currentState  
IDL> Contour, gridData, lon, lat, /Fill, $  
    Levels=levels, Background=cgColor('white'), $  
    Position=[0.125,0.125,0.95,0.80], $  
    Color=cgColor('black'), XStyle=1, YStyle=1, $  
    C_Colors=IndGen(nlevels)+1  
IDL> Contour, gridData, lon, lat, /Overplot, $  
    Color=cgColor('black'), Levels=levels, $  
    C_Labels=everyOther  
IDL> SetDecomposedState, currentState  
IDL> cgColorbar, Range=[Min(gridData, /Nan), $  
    Max(gridData, /NaN)], Charsize=0.75, $  
    Divisions=12, XTicklen=1, XMinor=0, $  
    AnnotateColor='black', NColors=12, Bottom=1, $  
    Position=[0.125, 0.915, 0.955, 0.95]
```

You see the result in Figure 28. You have to be somewhat careful in how you interpret contouring results. Look, for example, in the lower right corner of this figure. Are there really two peaks in the data? Possibly, but we can't be sure. It is not that the contouring algorithm is wrong or that the result is incorrect; it is that the resolution of the data does not always lend itself to unambiguous results. It is always a good idea to filter the result of any data analysis through your own head before you tell the world about your amazing discovery.

Contouring a Real World Example

Naturally, contour plots always work great when you are learning about them in a book. But real world examples are more difficult. They tend to

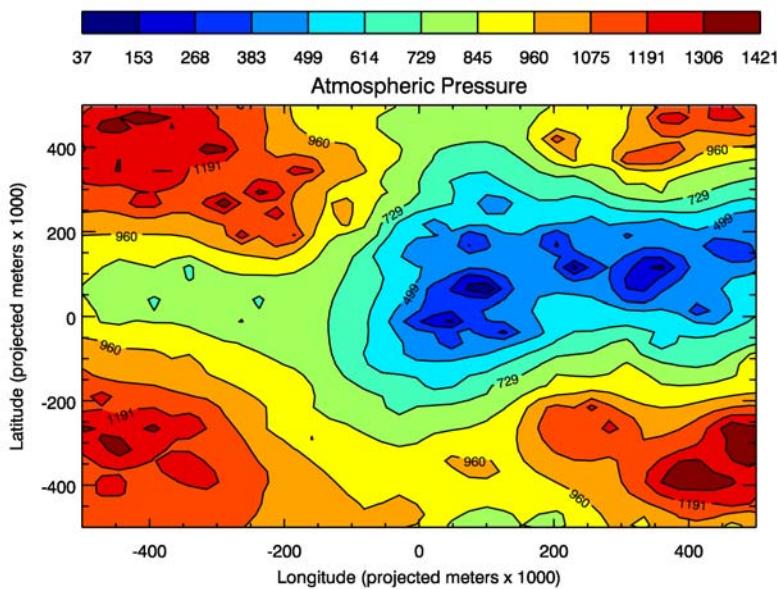


Figure 28: The same irregular data gridded with GridData using an inverse weighting gridding method.

throw curve balls, or—as my cricket playing Australian colleagues say—“wrong’uns.” They are like projects around the house. You can plan on three trips to the hardware store, no matter how simple you think the job is going to be.

So, here is a real world example that gives you some practice applying the techniques you have learned so far in this chapter. The data set we will use is a publicly available NASA Goddard Satellite-based Surface Turbulent Fluxes (GSSTF) data set (<http://disc.sci.gsfc.nasa.gov/measures/documentation/Readme.GSSTF2b.pdf>). I am going to show you how to contour the latent heat flux from a data set from 1 January 2008, which is stored in an HDF-EOS5 scientific data format (an HDF5 format). The file is named *GSSTF.2b.2008.01.01.he5*. If you like, you can download the file from my web page, although this is not necessary. The data file is quite large and I have saved the required variables from the file in an IDL save file, which I will describe shortly, and which is much faster to download. Here is the URL to obtain the data file itself, if you care to use it.

<http://www.idlcoyote.com/books/tg/data/GSSTF.2b.2008.01.01.he5>

The latent heat flux variable in the file is named “E” and the fill value is named “_FillValue.” Case is important in scientific data format files. Here is the IDL code I used to read the data, the fill value, and the dimensions of the data from the file. It is necessary to reset the IDL session before we start to clear away any system variable values we were using earlier in the chapter. (This is not necessary if you are starting your IDL session with this example. Nor is it necessary to type these commands. You can skip this command block and find an easier way to obtain the relevant data in the paragraph below the command block.)

```
IDL> .RESET
IDL> file = 'GSSTF.2b.2008.01.01.he5'
IDL> fileID = H5F_Open(file)
IDL> dataName = '/HDFEOS/GRIDS/SET1/Data Fields/E'
IDL> dataID = H5D_OPEN(fileID, dataName)
IDL> dataspaceID = H5D_GET_SPACE(dataID)
IDL> dims = H5S_GET_SIMPLE_EXTENT_DIMS(dataspaceID)
IDL> lon_dim = dims[0]
IDL> lat_dim = dims[1]
IDL> latentHeat = H5D_READ(dataID)
IDL> fill_valueID = H5A_OPEN_NAME(dataID, '_FillValue')
IDL> fill_value = H5A_READ(fill_valueID)
IDL> H5A_Close, fill_valueID
IDL> H5D_Close, dataID
```

If you chose not to download the data file and type the commands listed above, you can follow along with this example by restoring the file *heat_flux.sav* from among the data sets you downloaded to use in this book. The file can be found here if you haven’t yet downloaded it.

http://www.idlcoyote.com/books/tg/data/heat_flux.sav

The variables *latentHeat*, *fill_value*, *lon_dim*, and *lat_dim* will be restored.

```
IDL> Restore, File='heat_flux.sav'
IDL> Help, latentHeat, fill_value, lon_dim, lat_dim
      LATENTHEAT      FLOAT      = Array[360, 180]
      FILL_VALUE      FLOAT      = Array[1]
      LON_DIM        ULONG64   = 360
      LAT_DIM        ULONG64   = 180
```

The first curve ball you see here is that the *fill_value* variable is a one-element array, instead of the scalar value you were probably expecting. This is going to set up a classic “gotcha” situation with the *Where* function in

IDL, which is what we must use to find these “missing” or fill values in the data if we plan to exclude them from further processing.

Look what happens if you are not aware that the *fill_value* variable is a one-element array.

```
IDL> indices = Where(latentHeat EQ fill_value, count)
IDL> Print, count
      1
```

Compare this result to using a scalar value for the *fill_value* variable.

```
IDL> indices = Where(latentHeat EQ fill_value[0], count)
IDL> Print, count
      38774
```

Quite a difference! While you can get away with using a one-element array for a scalar most of the time in IDL, here is one example where you definitely cannot. When IDL evaluates the expression with the EQ operator and two vectors, it quietly truncates the result to match the smaller of the two vectors. The *Where* function is the unfortunate victim in this operation. Yikes!

So, the first thing to do is to take care of this scalar versus array problem by making the fill value a scalar value.

```
IDL> fill_value = fill_value[0]
```

The fill value essentially identifies the “missing” data values in this file. In order to calculate the proper levels for contouring, we need to remove these missing values from consideration. The normal way to do this is to set these missing values to NaNs. The code looks like this.

```
IDL> indices = Where(latentHeat EQ fill_value, cnt)
IDL> IF cnt GT 0 THEN latentHeat[indices] = !Values.F_NAN
```

Note that we didn’t have to convert the *latentHeat* variable to a floating variable first, because it already is a floating point array.

Now we can find the minimum and maximum of the remaining “good” data. Remember to set the *NaN* keyword.

```
IDL> minData = Min(latentHeat, MAX=maxData, /NaN)
IDL> Print, minData, maxData
      -13.5637      583.412
```

Normally, latent heat flux is shown in about 8 regular divisions of about 50 watts per square meter. We could easily create 8 contour levels for this data set, but if we start from the minimum value of the data, as we have done previously in this chapter, the levels will not be “natural” divisions.

They will have arbitrary floating point values such as -13.5637, 63.5637, 133.5637, etc. It would be better to have 8 levels that started at 0 and went up in units of 50 (0, 50, 100, etc.), but then we have the problem that some of the data in the file will have values less than the minimum level or greater than the maximum level.

We could solve this problem by adding two additional “levels” or contour colors, one for values less than 0 and one for values greater than the highest contour level we are interested in. In other words, we divide the data into 10 colors or divisions, with the lowest and highest division representing the values that are outside the range of values we are particularly interested in. We can create 10 divisions by specifying the nine levels for the contour plot like this. (The top division, the 10th, represents all the values greater than the value of our last contour level.)

```
IDL> levels = [minData, IndGen(8)*50]
```

Next, this data is world-wide data on a one-degree grid, so we can set up the longitude and latitude vectors we need to display the data with like this.

```
IDL> lon = Scale_Vector(Findgen(lon_dim), -180, 179) + 0.5
IDL> lat = Scale_Vector(Findgen(lat_dim), -90, 89) + 0.5
```

The extra 0.5 we add to each element is to align the left (longitude) or bottom (latitude) edge of the grid cell with a whole degree. The centers of the grid cells then fall on half-degree increments (-179.5°, -178.5°, -177.5°, etc.). The advantage of this kind of gridding is that it avoids the non-physical extra half degree “below” the South Pole (at -90.5°) and the small gap just below the North Pole (at 89.5°).

We are going to display this in a window that is wider than it is tall, so we need a *Window* command. But, we want this program to work in the PostScript device, too, so we need to protect this *Window* command and only issue the command if we are on a device that supports windows.

```
IDL> IF (!D.Flags AND 256) NE 0 THEN $
      Window, XSize=1000, YSize=700
```

Another way to do this is to use the `cgDisplay` command, which automatically protects the command in a PostScript device. In fact, in a PostScript device, it creates a “window” having an aspect ratio of 700/1000, just like this window on the display device.

```
IDL> cgDisplay, 1000, 700
```

I would like to display the filled contour plot in a window with a white background. Since I want to put this on top of a map projection, and map projections don’t allow me to use a *Background* keyword to set the back-

ground color, I have to erase the window with the background color, and then use a `NoErase` keyword on my map projection command. I set the map projection (a cylindrical projection, in this case) up like this.

```
IDL> Erase, Color=cgColor('white')
IDL> Map_Set, Position=[0.05,0.05,0.95,0.75], /NoErase
```

Next, we load the 16 colors for the contour plot, and display the filled contour plot. We will use a Brewer color table, since Brewer colors were designed to work well on maps. The code looks like this.

```
IDL> cgLoadCT, 25, /Brewer, NColors=10, Bottom=1, /
      Reverse
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, latentHeat, lon, lat, /Overplot, $
      /Cell_Fill, C_Colors=IndGen(10)+1, $
      Levels=levels, Color=cgColor('black')
IDL> Contour, latentHeat, lon, lat, /Overplot, $
      Levels=levels, Color=cgColor('black')
IDL> SetDecomposedState, currentState
```

Note: We are using the `Cell_Fill` keyword rather than the `Fill` keyword in this example. This is essential on maps, generally, which tend to create open contours, and even more so when there are open contours as a result of missing values (`NaN`) in the data being contoured.

The next step is to add map annotations. I would like to display this with box style axes, and I want the character size in a PostScript file to be slightly smaller than the character size I use on the display. I set the character size like this.

```
IDL> charsize = (!D.Name EQ 'PS') ? 0.65 : 1.0
IDL> Map_Continents, Color=cgColor('tan'), /Fill
IDL> Map_Continents, Color=cgColor('black')
IDL> Map_Grid, /Box_Axes, Color=cgColor('black'), $
      CharSize=charsize
```

Finally, we add a color bar to the plot to indicate the data values and their associated colors. In this case, we use the discrete color bar routine `cgDCBar` from the [Coyote Library](#).

```
IDL> labels = ['< 0', '0-50', '50-100', '100-150', '$
      '150-200', '200-250', '250-300', '300-350', '$
      '350-400', '>400']
IDL> cgDCBar, NColors=10, Bottom=1, Color='black', $
      Position=[0.05, 0.9, 0.95, 0.94], Rotate=-45, $
      Labels=labels, CharSize=charsize, $
      Title='Latent Heat (watts/meter^2)'
```

You see the result in Figure 29.

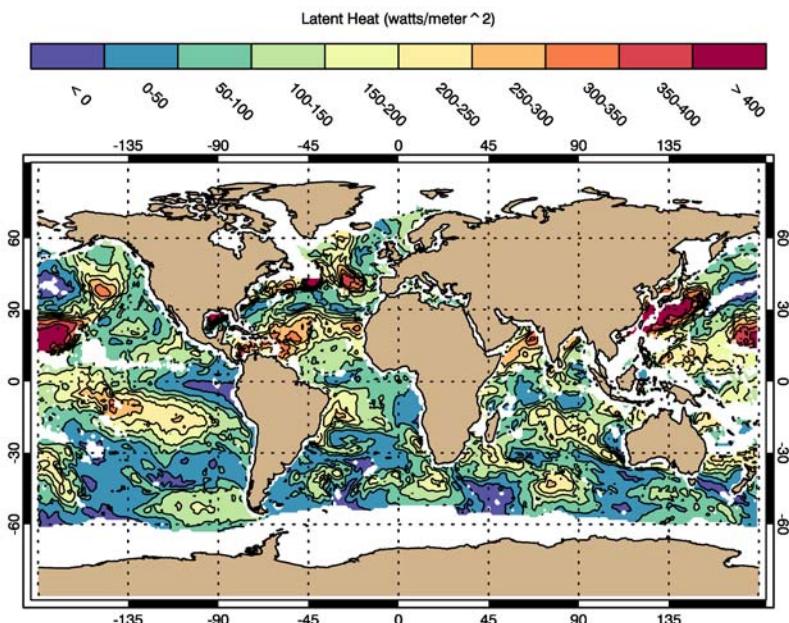


Figure 29: A real world example of contouring latent heat flux from a NASA satellite data set.

Using a Refurbished Contour Command

You will find that many of the problems identified with the traditional *Contour* command in IDL have been corrected in the [cgContour](#) command from the [Coyote Library](#). The [cgContour](#) command is part of a suite of programs, collectively called the Coyote Graphics System (which includes [cgPlot](#), [cgPlotS](#), [cgSurf](#), [cgText](#), and [cgWindow](#)), for creating traditional graphics output that work and look the same on all devices and in any color decomposition state. The [cgContour](#) command takes care of many of the details necessary to write device independent graphics programs, works with colors in a more natural way by specifying colors directly, and has features that are not available in the *Contour* command. It can easily be displayed in the resizeable graphics window, [cgWindow](#).

The Coyote Graphics commands produce, by default, black output on white backgrounds. This is the opposite of IDL traditional commands, but makes it possible to use these commands to produce (as much as possible) identical looking PostScript output. (You can return to the traditional

color scheme of white on black by setting the *Traditional* keyword on the command.)

You will notice that textual output from these commands is slightly larger than normal. There are two reasons for this. First, larger output more closely matches the look and feel of the corresponding PostScript output when PostScript or TrueType fonts are used. And, second, larger output more closely matches the look and feel of similar object graphics programs in IDL 8.

The `cgContour` program has been specifically written to address the following problems with the *Contour* command:

- The *NLevels* keyword should specify exactly N contour levels.
- There should be no “hole” in a filled contour plot.
- There should be an easy selection method for which contour levels to label.
- It should draw graphics output in decomposed color mode, when possible, so color tables do not become “polluted” with drawing colors.

Here are some side-by-side comparisons to give you a sense of how `cgContour` works. Notice that, by default, all contour levels are labeled. Here is an example of the default output of *Contour* and `cgContour`. (If you have been working the examples in this chapter, and you haven’t already done so, you might want to start a fresh IDL session by entering the `.Reset` executive command at the IDL command prompt before you begin with these commands.)

```
IDL> data = cgDemoData(2)
IDL> LoadCT, 0, /Silent
IDL> Window, 0, XSize=400, YSize=400
IDL> Contour, data, Title='Normal Contour Plot', $
      XTitle='X Title', YTitle='Y Title'
IDL> Window, 1, XSize=400, YSize=400
IDL> cgContour, data, Title='Coyote Contour Plot', $
      XTitle='X Title', YTitle='Y Title'
```

You see the result in Figure 30.

Here is a similar comparison with basic filled contour plots. First, set up the common elements.

```
IDL> cgLoadCT, 17, /Brewer
IDL> cgLoadCT, 4, NColors=10, Bottom=1, /Brewer, /Reverse
```

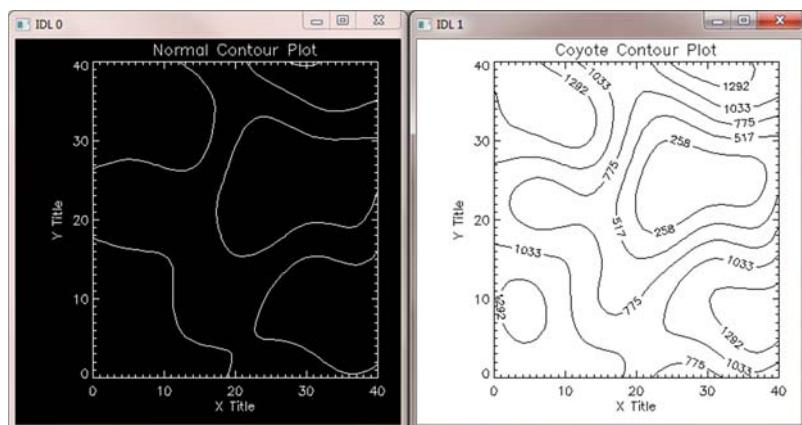


Figure 30: A side-by-side comparison of the basic *Contour* command versus the basic *cgContour* command.

```
IDL> c_colors = IndGen(10) + 1
IDL> position = [0.1, 0.1, 0.9, 0.8]
```

Now, draw the contour plot with the *Contour* command. Make sure you preserve the color model and that you draw the filled contours using the indexed color model.

```
IDL> Window, 0, XSize=400, YSize=400
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Contour, data, NLevels=10, /Fill, $
      Position=position, C_Colors=c_colors
IDL> Contour, data, NLevels=10, /Overplot
IDL> SetDecomposedState, currentState
IDL> cgColorbar, Divisions=10, NColors=10, Bottom=1, $
      Range=[Min(data),Max(data)], TickLen=1.0, $
      Position=[0.1,0.90,0.90,0.94], CharSize=0.75
```

Now, draw the same filled contour plot with *cgContour*.

```
IDL> Window, 1, XSize=400, YSize=400
IDL> cgContour, data, NLevels=10, /Fill, $
      Position=position, C_Colors=c_colors
IDL> cgContour, data, NLevels=10, /Overplot
IDL> cgColorbar, Divisions=10, NColors=10, Bottom=1, $
      Range=[Min(data),Max(data)], TickLen=1.0, $
      Position=[0.1,0.90,0.90,0.94], CharSize=0.75
```

You see the result in Figure 31. Notice you don't have to worry about setting the color decomposition state, since the `cgContour` program takes care of this automatically.

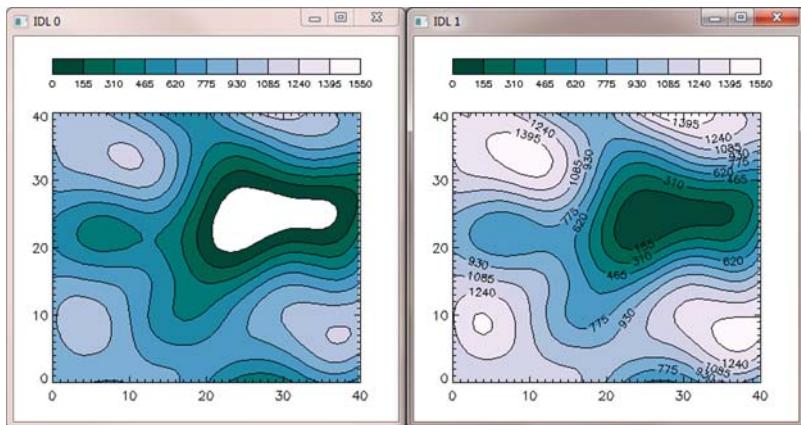


Figure 31: A side-by-side comparison of a filled contour plot with the `Contour` command and a filled contour plot with the `cgContour` command.

The `cgContour` command also has the ability to accept a color palette so that contour colors are completely independent of the colors loaded in the current color table. The easiest way to obtain a color palette is with the `cgLoadCT` command and its `RGB_Table` output keyword. If this keyword is used, colors are not loaded into the current color table, but are simply loaded into a color palette and returned to the user. The plot on the right in Figure 31 can be reproduced with a Standard Gamma II color table like this.

```
IDL> cgLoadCT, 5, RGB_Table=pal
IDL> Window, 2, XSize=400, YSize=400
IDL> cgContour, data, NLevels=10, /Fill, $
    Position=position, Palette=pal
IDL> cgContour, data, NLevels=10, /Overplot
IDL> cgColorbar, Divisions=10, NColors=10, Bottom=1, $
    Range=[Min(data),Max(data)], TickLen=1.0, $
    Position=[0.1,0.90,0.90,0.94], Charsize=0.75, $
    Palette=pal
```

You see the result in Figure 32.

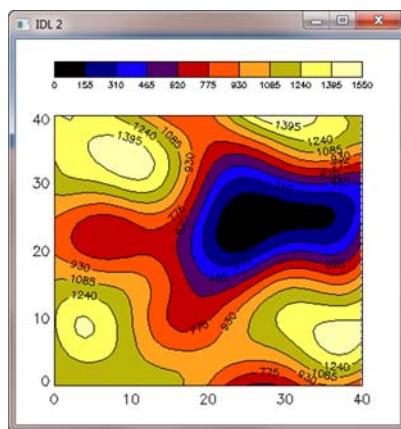


Figure 32: The `cgContour` command allows you to specify a color table palette that keeps your contour plot colors completely independent of the colors in the current color table.

Labeling Contour Intervals

The `cgContour` command also makes it easy to choose which contours to label. If you want every other contour, or every third contour, labeled, you simply set the new *Label* keyword to 2 or 3. Set it to 0 to turn contour labeling completely off, like the default for the *Contour* command. The default for `cgContour` is 1, which labels every contour level.

Using Colors in Contour Plots

The `cgContour` command makes it very easy to use colors in contour plots. The axis and annotation color can be different from the contour colors. And, these colors can be expressed as color names that are among the 200 color names recognized by `cgColor`. Consider these commands that show how colors can be used in a different way than with the *Contour* command.

```
IDL> Window, 0, XSize=400, YSize=400
IDL> cgContour, data, NLevels=10, AxisColor='blue', $
      Color='red'
IDL> Window, 1, XSize=400, YSize=400
IDL> cgContour, data, NLevels=5, AxisColor='brown', $
      C_Colors=['aquamarine', 'dark green', 'orange', '$
                  'crimson', 'purple']
```

You see the result in Figure 33.

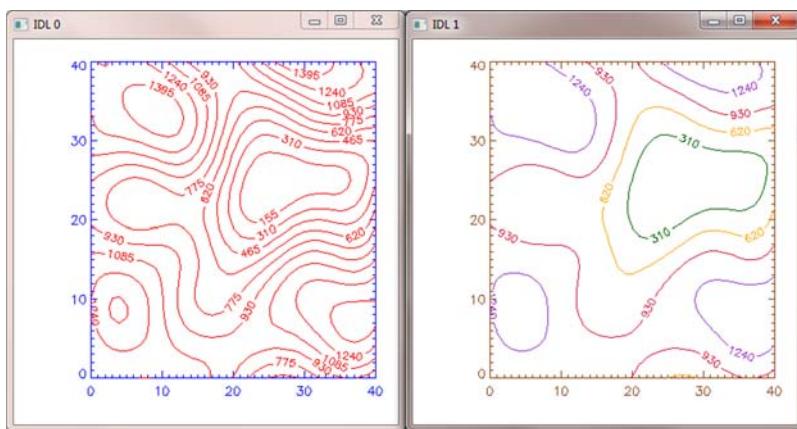


Figure 33: The `cgContour` command allows color names to be used to specify contour colors.

Contour Plots in Resizeable Graphics Windows

If you want to see the contour plot by itself in a resizeable graphics window, with controls for making hardcopy output files, set the `Window` keyword. The contour plot is displayed in `cgWindow`.

```
cgContour, data, NLevels=10, AxisColor='navy', $  
Color='red', /Window
```

You see the result in Figure 34.

The `cgWindow` application allows you to write the graphics commands to a PostScript file, or save the display in any of five different raster file formats. If you have ImageMagick installed on your machine, you have the option of creating raster files by converting PostScript files to raster output with ImageMagick's *convert* command. This results in raster files of significantly higher quality, especially in the quality of the fonts used for annotation. You will learn more about this topic in “Presentation Quality by Leveraging PostScript” on page 405.

You can add as many graphics commands as you like to a `cgWindow` program window. Here, for example, is how to display a filled contour plot with contour lines overlaid, and with a color bar at the top of the plot. Simply set the `Window` and/or `AddCmd` keywords on the commands to display the contour plot in a resizeable graphics window.

```
IDL> data = cgDemoData(2)  
IDL> cgLoadCT, 4, /Brewer, /Reverse, NColors=12, $  
Bottom=1
```

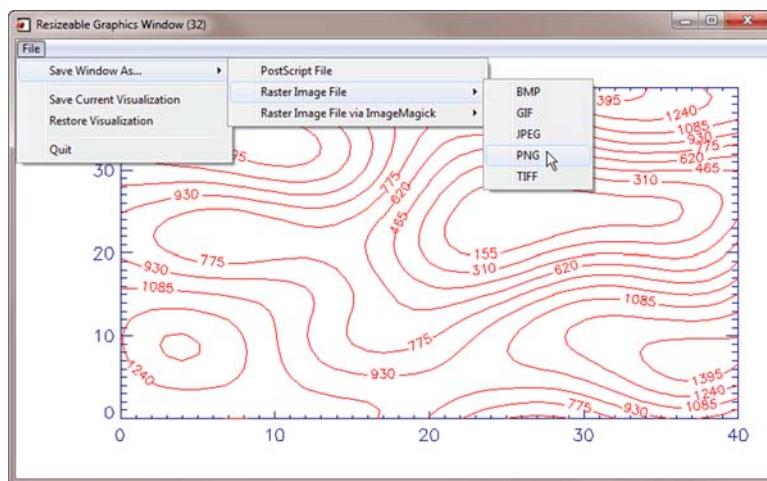


Figure 34: The `cgContour` command can be displayed in a resizable graphics window, where you can save the graphics window in five different raster file formats and as a PostScript file. This resizable graphics window is created with the Coyote Library program `cgWindow`.

```
IDL> cgContour, data, NLevels=12, $
      /Fill, C_Colors=IndGen(12)+1, $
      Position=[0.1,0.1,0.9,0.75], /Window
IDL> cgContour, data, NLevels=12, $
      Color='Charcoal', /Overplot, /AddCmd
IDL> cgColorbar, Divisions=12, $
      Range=[Min(data), Max(data)], NColors=12, $
      Bottom=1, XMinor=0, XTicklen=1.0, /AddCmd
```

You see the result in Figure 35.

If you would like to list the commands you have loaded into the `cgWindow` command list, just set the `ListCmd` keyword. The commands are printed in the console window.

```
IDL> cgWindow, /ListCmd
0. cgContour, p1, C_COLORS=value, FILL=value, $
      NLEVELS=value, POSITION=value
1. cgContour, p1, COLOR=value, NLEVELS=value, $
      OVERPLOT=value
2. cgColorbar, BOTTOM=value, DIVISIONS=value, $
      NCOLORS=value, RANGE=value, XMINOR=value, $
      XTICKLEN=value
```

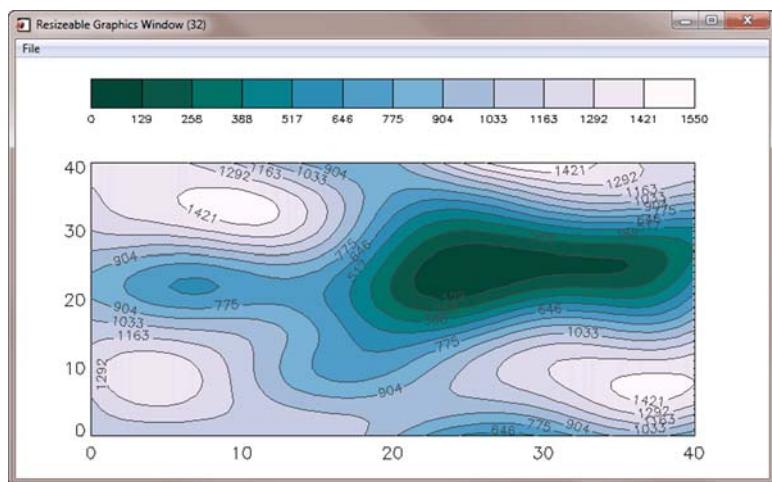


Figure 35: The `cgWindow` program allows you to add an unlimited number of graphics commands to the graphics window. The graphics window is resizeable, and the window content can be sent to a PostScript file or saved in any of five different raster file formats.

You can use the resizeable graphics window `cgWindow` just like you use other graphics windows in IDL. Use `cgSet` to select a `cgWindow` to add commands to, and `cgDelete` to delete `cgWindow` programs you are finished with. You can have as many `cgWindow` programs on the display as you like, each with a different display, as in normal IDL graphics windows.

For example, here is how you can save the window index number of the first `cgWindow`, open a second `cgWindow`, and display another contour plot.

```
IDL> windowIndex = cgQuery(/Current)
IDL> cgWindow
IDL> cgContour, cgDemoData(18), NLevels=20, Label=2, $
      Color='charcoal', /AddCmd
```

The `cgQuery` function is used to obtain information about the `cgWindow` programs currently on the display. You can determine their window index numbers, the widget identifiers of their top-level base widgets, the window titles, or obtain their object references by setting appropriate keywords. If the keyword *Current* is set, as it was here, this information is returned for just the current `cgWindow` (i.e., the last one created).

To display another contour plot in the first `cgWindow`, type this.

```
IDL> cgSet, windowIndex
IDL> cgContour, cgDemoData(18), NLevels=20, Label=2, $
      Color='dodger blue', /Window
```

You see the result in Figure 36.

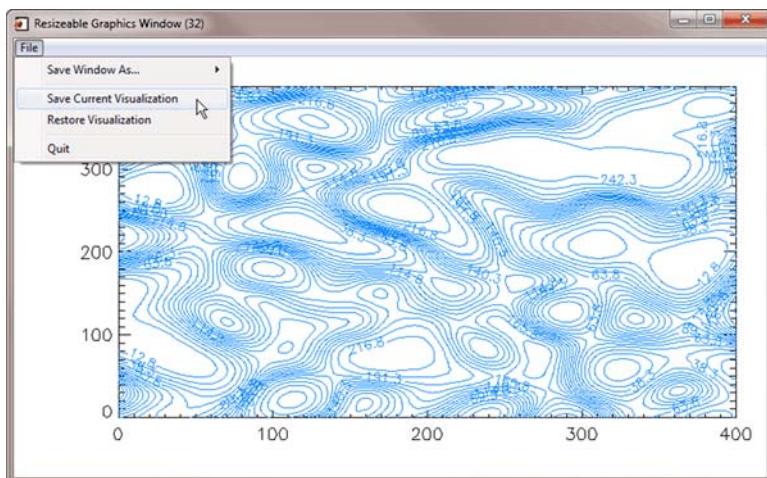


Figure 36: You can use resizable graphics windows in exactly the same way you currently use normal graphics windows. Here a new command was added to the window.

The graphics display (the *visualization*) in a `cgWindow` can be saved to a file, where it can be e-mailed to a colleague so he or she can view the same visualization you see, or so you can restore it later to view it again. Simply choose the Save Current Visualization button from the pull-down File menu, as illustrated in Figure 36.

To delete the first `cgWindow`, type this.

```
IDL> cgDelete, windowIndex
```

The second `cgWindow` will be sent forward on your display so you can view it easily.

You can delete all the `cgWindow` applications currently on the display by setting the *All* keyword.

```
IDL> cgDelete, /All
```

Chapter 6



Creating Surface Plots

Basic Surface Plots

In theory, any two-dimensional array can be displayed as a three-dimensional surface plot in IDL using the traditional graphics command *Surface*. In practice, because of rendering time and the sheer pixel limitations of most displays, 2D arrays of less than about 300 elements on a side are generally used to produce surface plots. Larger arrays are often reduced to a size of this magnitude using the *Rebin* or *Congrid* command before being passed to the *Surface* command.

Because surface plots and contour plots are often related to one another, we will use the simple 2D array we used to produce contour plots in the previous chapter to produce surface plots in this chapter. Recall that we used the [Coyote Library](#) routine `cgDemoData` to load the 2D data set. The data is a 41 x 41 floating point array with values ranging from 0 to 1550.

```
IDL> data2D = cgDemoData(2)
IDL> Help, data2D
      DATA2D           FLOAT      = Array[41, 41]
IDL> Print, Min(data2D), Max(data2D)
      0.000000      1550.00
```

A surface plot in its most basic form can be created just by passing this array as the argument to the traditional *Surface* command. Note that hidden line removal is done automatically.

```
IDL> Surface, data2D, Color=cgColor('black'), $
      Background=cgColor('white')
```

You see the result in Figure 1.

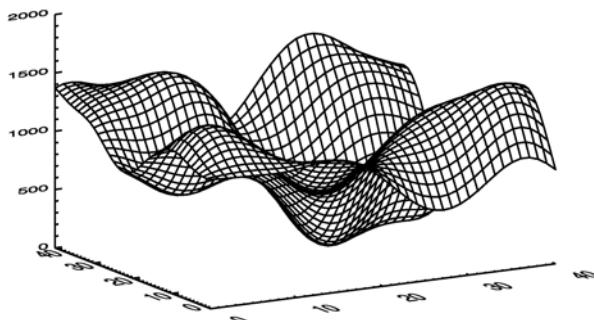


Figure 1: The basic Surface plot in IDL. The X and Y axes are labeled according to the size of the 2D array. The Z axis is always vertical in the output of the traditional graphics Surface command.

As with the contour plot in the previous chapter, the axes are labeled with the number of elements in the 2D array. The Z axis, which is the vertical axis in Figure 1, is labeled with the values of the 2D data set. Annotation keywords that you used for line and contour plots can also be used to annotate surface plots. /XYZ/Title keywords all work in the normal way for surface plots. The Title keyword, however, works a bit differently. It is rotated to appear in the XZ plane of the three-dimensional plot. Consider this command.

```
IDL> Surface, data2D, Color=cgColor('black'), $  
      Background=cgColor('white'), $  
      XTitle='Longitude', YTitle='Latitude', $  
      ZTitle='Atmospheric Pressure', $  
      Title='Pressure Over USA'
```

You see the result in Figure 2.

Most of the time, we prefer a plot title to be facing toward the person viewing the plot. Sometimes this orientation is called an “on the glass” orientation. The traditional graphics *Surface* command cannot produce an “on the glass” title like this, so if this is required, the title is usually left off the *Surface* command, and is added to the graphics output with the *XYOutS* command, like this.

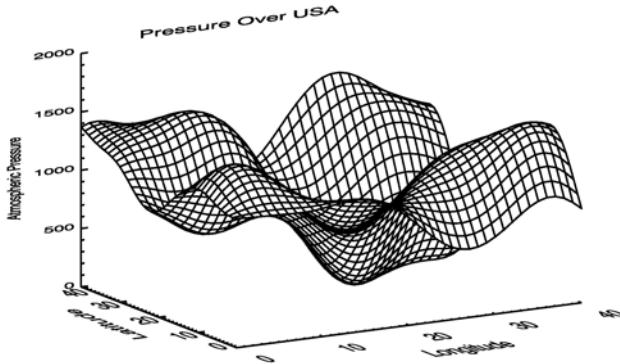


Figure 2: Title keywords can be used to annotate a basic surface plot. Note how the graphics title is rotated in the XZ plane of the 3D surface plot.

```
IDL> Surface, data2D, Color=cgColor('black'), $  
      Background=cgColor('white'), $  
      XTitle='Longitude', YTitle='Latitude', $  
      ZTitle='Atmospheric Pressure'  
IDL> XYOutS, 0.5, 0.95, /Normal, 'Pressure Over USA', $  
      Alignment=0.5, Color=cgColor('black')
```

You see the result in Figure 3.

As with the *Contour* command, we can create X and Y vectors of values that correspond to the physical quantities or properties of the surface plot. In the case of the surface plot, the values in these vectors will determine the intersections of the lines that constitute the surface mesh. We can make vectors of longitude and latitude like this.

```
IDL> s = Size(data2D, /Dimensions)  
IDL> lon = IndGen(s[0]) * 25000L - 500000L  
IDL> lat = IndGen(s[1]) * 25000L - 500000L
```

These are large values to display on an axis, so it would be better to divide these by, say, 1,000 and indicate this in the plot annotation. We could write code like this.

```
IDL> lon = lon / 10^3  
IDL> lat = lat / 10^3  
IDL> xtitle = 'Longitude (meters x 1000)'  
IDL> ytitle = 'Latitude (meters x 1000)'
```

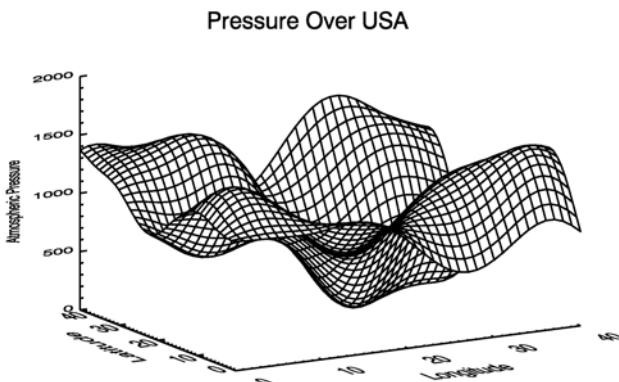


Figure 3: An “on the glass” title is added with the `XYSOutS` command.

We can use these values in the surface plot like this.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure'
```

You see the result in Figure 4. Note the auto-scaling of the axes. In surface plots we don’t mind this so much, but be aware it is occurring. Of course, it can be turned off by setting the `[XYZ]Style` graphics keywords. The best looking surface plots are sometimes created by turning auto-scaling off for the X and Y axes, but leaving it on for the Z axis.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0
```

The Surface Command Is Not True 3D

The IDL traditional graphics `Surface` command is a contemporary of the `Contour` command and is quite old. (Well, in dog years, anyway.) It has been around since the early 1980s. In those days the state of the art in computer graphics was not true 3D representations, but a transformation that

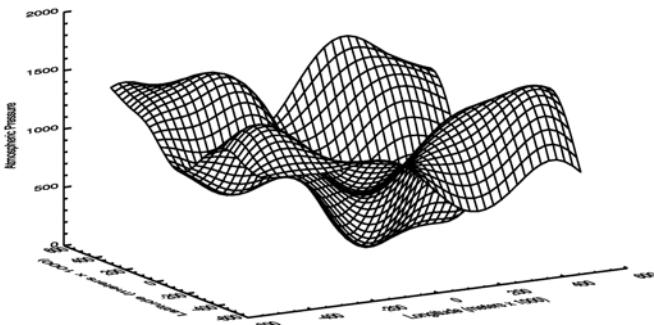


Figure 4: The surface plot with all three positional parameters passed. Note the axes auto-scaling. Auto-scaling is controlled by the [XYZ]Style graphics keywords.

has come to be known as a “two-and-a-half D” representation (2.5D). This imposes limitations on the *Surface* command.

In particular, it is not possible to rotate the three orthogonal axes of the *Surface* command in true 3D space. Rather, rotations are only allowed that end up with the Z axis in a vertical position when rendered on a flat 2D plane. You will find, for example, that you can rotate a surface about the X axis, with the *AX* keyword, and about the Z axis, with the *AZ* keyword, but you can’t rotate the surface about the Y axis. There is no *AY* keyword.

If true 3D rotations of orthogonal axes are important to you, then you will want to use a surface command that takes advantage of the object graphics system in IDL (the *cgSurface* command in the [Coyote Library](#), the *iSurface* iTool command, or the new graphics *Surface* function in IDL 8.0, for example). The object graphics system is a true 3D graphics system.

Nevertheless, the traditional graphics *Surface* command is still useful in a number of circumstances. It is particularly useful when a surface plot is to be used in combination with one or more of the other traditional graphics commands. It is not possible to combine traditional graphics line and contour plots, for example, with true 3D surface commands in the same IDL graphics window. The two graphics systems are completely separate and use different kinds of graphics windows.

Rotating the Surface

The default rotation of a surface plot of the sort we have created so far is to rotate the surface 30 degrees about the X axis and 30 degrees about the Z axis. If, however, you want to rotate the surface by 45 degrees about X and 60 degrees about Z, you can use the *AX* and *AZ* keywords, like this.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0, $
      AX=45, AZ=60
```

You see the result in Figure 5. Notice how rotations about the Z axis have the effect of “flattening” the surface by making the vertical Z axis shorter.

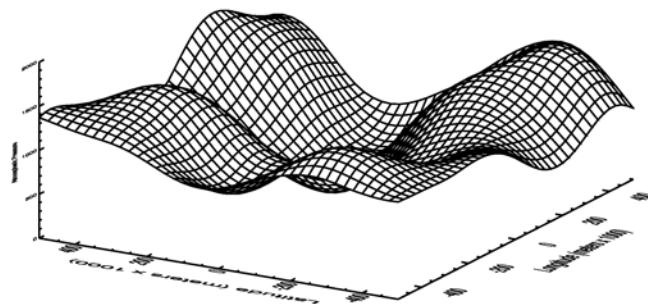


Figure 5: Rotations can only be applied to the X and Z axes. Rotations about the Y axis are not allowed in a 2.5D system.

Sometimes, especially if other graphics commands are going to be used with the *Surface* command, the 3D data coordinate space needs to be saved so other commands can use it. To save the 3D transformation from this command, set the *Save* keyword. The 3D transformation matrix produced by the command is saved in the system variable *!P.T*. Other graphics commands can use this 3D transformation matrix to position themselves in 3D space. Usually you tell a command to use the *!P.T* transformation matrix by setting a *T3D* keyword on the command itself.

For example, here is how we display both a surface and a contour plot of the same data. Note the use of the *T3D* keywords on the *Contour* commands.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      ZTitle='Atmospheric Pressure', $
      XStyle=5, YStyle=5, ZStyle=0, $
      AX=45, AZ=60, /Save
IDL> Contour, data2D, lon, lat, /T3D, XStyle=1, $
      YStyle=1, XTitle=xtitle, YTitle=ytitle, $
      /NoErase, Color=cgColor('black'), /NoData
IDL> Contour, data2D, lon, lat, /T3D, /Overplot, $
      NLevels=12, C_Label=Replicate(1,12), /NoClip, $
      ZValue=0.0, /Downhill, Color=cgColor('red8')
```

You see the result in Figure 6. You will see later how to make this plot even better by using the Z-graphics buffer to perform hidden line removal and clean up the overlap between the surface and contour plot.

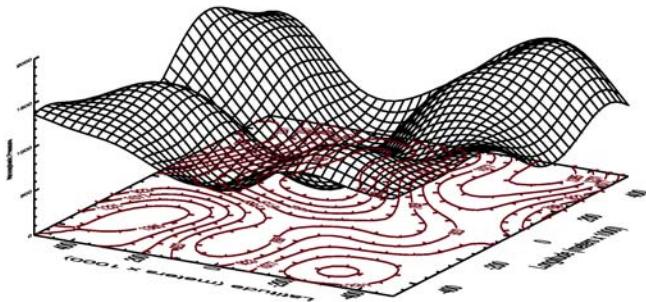


Figure 6: The *Surface* command can be used to set up a 3D transformation matrix for other graphics commands to use.

Customizing Surface Plots

There are some 70 keywords that can modify the appearance of a surface plot. You have already learned many of these keywords in the chapters on creating line and contour plots. But there are a few others that apply only to surface plots.

One of these keywords is the *Skirt* keyword that is used to place a skirt around the outside of the surface plot. *Skirt* should be set to a Z value where you want the skirt to be drawn.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0, $
      AX=30, AZ=60, Skirt=0
```

You see a surface with a skirt in Figure 7.

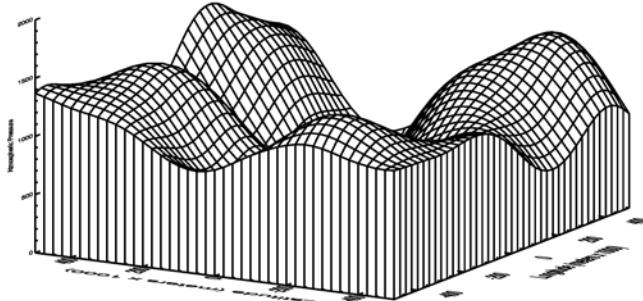


Figure 7: A skirt is added to the surface plot with the *Skirt* keyword.

Note that the skirt is always drawn from the edge of the surface to the value set by the *Skirt* keyword. For example, it is customary to draw a pressure axis reversed from the normal Z axis. This is easily done by simply reversing the *ZRange* keyword, but notice how the skirt is drawn now.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=1, $
      AX=30, AZ=60, Skirt=0, ZRange=[Max(data2D), 0]
```

You see the result in Figure 8.

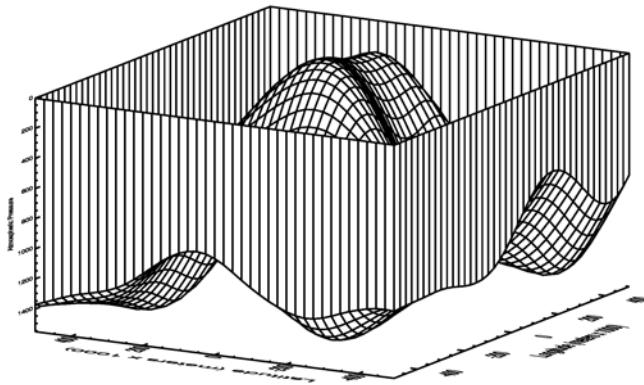


Figure 8: A skirt doesn't make as much sense if the Z axis is reversed!

In this case, you want to set the skirt to the bottom of the Z axis, or `!Z.CRange[0]`, like this.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=1, $
      AX=30, AZ=60, Skirt=!Z.CRange[0], $
      ZRange=[Max(data2D), 0]
```

You see the result in Figure 9.

The vectors that describe the intersection of the mesh lines in a surface plot have to be monotonically increasing, but they do not have to be evenly spaced. For example, we can randomly sample the longitude vector like this.

```
IDL> randomSamples = RandomU(-3L, 41) * 1000 - 500
IDL> newLon = randomSamples[Sort(randomSamples)]
```

Now we draw the surface plot like this with the X vector specifying random locations in the X direction.

```
IDL> Surface, data2D, newlon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0, AX=60
```

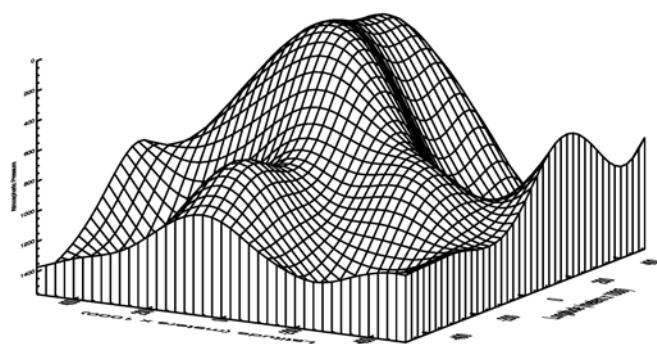


Figure 9: A skirt applied correctly when the Z axis is reversed.

You see the result in Figure 10.

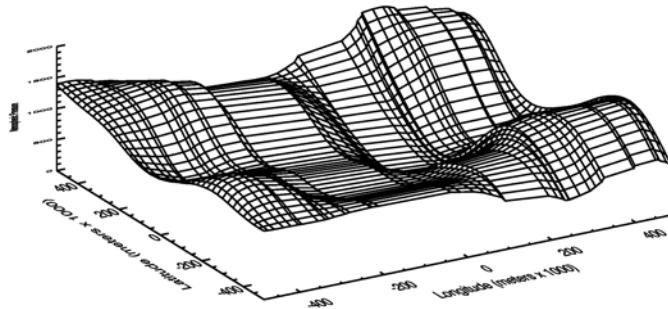


Figure 10: The X and Y vectors must be monotonically increasing (or decreasing), but they do not have to be evenly spaced.

Adding Color to Surfaces

Color can be added to surface plots in much the same way it was added to contour plots. For example, if you would like the surface axes to be drawn in black, and the surface itself to be drawn in green, you can type com-

mands like the following. First, display the axes in black by setting the *NoData* keyword so the surface is not drawn.

```
IDL> Surface, data2D, lon, lat, $  
      Color=cgColor('black'), $  
      Background=cgColor('white'), $  
      XTitle=xtitle, YTitle=ytitle, $  
      ZTitle='Atmospheric Pressure', $  
      XStyle=1, YStyle=1, ZStyle=0, /NoData
```

There is no *Overplot* keyword for the *Surface* command like there is for the *Contour* command. Rather, we have to perform a trick. We have to draw the surface plot over again, but this time with the axes turned off and the surface turned on! Recall that we turn axes off by setting the “4-bit” in the *[XYZ]Style* keywords. The second *Surface* command looks like this. Be sure to set the *NoErase* keyword, or your second *Surface* command will erase the first.

```
IDL> Surface, data2D, lon, lat, $  
      Color=cgColor('grn5'), /NoErase, $  
      XTitle=xtitle, YTitle=ytitle, $  
      ZTitle='Atmospheric Pressure', $  
      XStyle=5, YStyle=5, ZStyle=4
```

You see the result in Figure 11.

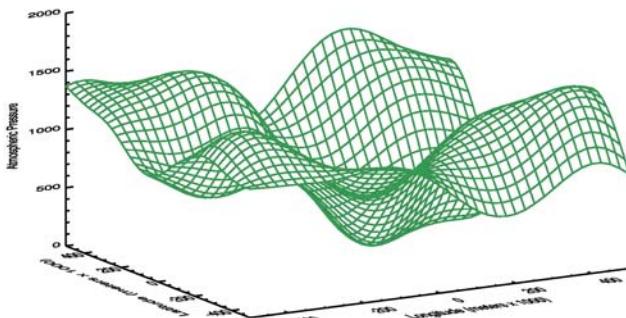


Figure 11: Surfaces can be drawn in color by using two Surface commands to draw different parts of the surface plot.

If you want the bottom surface drawn in a different color from the top surface, you can use the *Bottom* keyword.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0

IDL> Surface, data2D, lon, lat, $
      Color=cgColor('grn5'), /NoErase, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=5, YStyle=5, ZStyle=4, $
      Bottom=cgColor('red5')
```

You see the result in Figure 12.

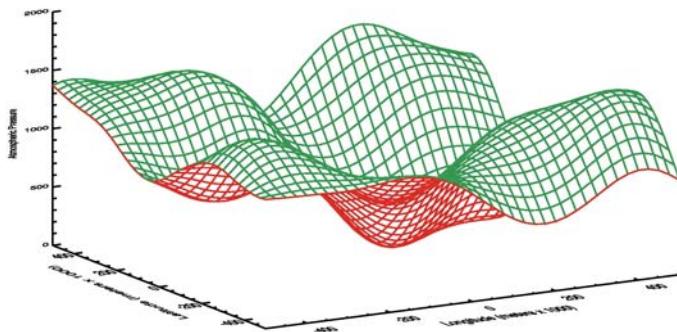


Figure 12: A second color can be added to the surface by using the **Bottom** keyword to specify a color for the bottom surface.

If you want to show you were a hot-shot traditional graphics programmer, you can add a red skirt to this surface plot by typing three surface commands like this.

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0

IDL> Surface, data2D, lon, lat, $
      Color=cgColor('red5'), /NoErase, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=5, YStyle=5, ZStyle=4, /Skirt
```

```
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('grn5'), /NoErase, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=5, YStyle=5, ZStyle=4
```

You see the result in Figure 13.

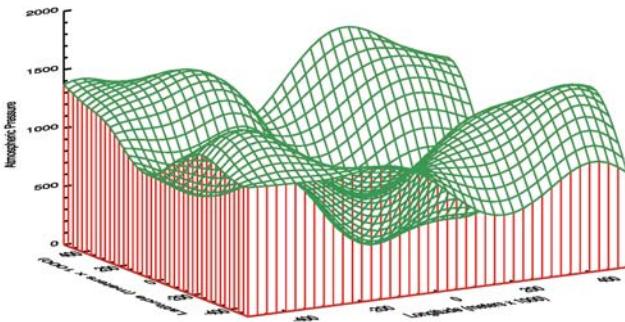


Figure 13: A three color surface plot is constructed from three separated Surface commands, each drawing part of the plot.

Rather than display the surface in a single color, we often want to display the surface in colors that might represent another property of the surface. A fourth dimension, in effect.

Suspend belief for a moment and imagine this surface we are dealing with represents the elevation values of a mountain valley near Boulder, Colorado. If you also had snow pack data, you might want to display the surface elevations with colors that represented the depth of the snow pack. Essentially, you would have added or visualized another dimension on top of the three-dimensional elevation data. The only requirement for doing this is that the dimensions of the data sets match.

Consider, for example a second data set available though the [cgDemoData](#) program. Let's think of this data set as snow pack depth for a moment.

```
IDL> snowDepth = cgDemoData(3)
IDL> Help, data2D, snowDepth
      DATA2D          FLOAT      = Array[41, 41]
      SNOWDEPTH       FLOAT      = Array[41, 41]
```

To color the original data set with this second data set, we use the *Shades* keyword to add the colors. The variable we pass to the *Surface* command via the *Shades* keyword will be a 2D array, scaled into color indices. That is to say, the values will represent indices into a color table. If the value of the *Shades* array is 128, for example, then that same point in the *data2D* array will be set to the color loaded into the color table at color index 128.

This implies, of course, that we are in indexed color mode when we perform this operation. Otherwise, no matter what colors we have loaded into the color table, the surface will be drawn in red colors only.

But, we have another problem, too. We are using `cgColor` to choose the background and drawing color for the surface plots we have created so far. If we are in indexed color mode, these drawing colors have to be loaded into the color table. But, where? In general, we have no idea. (I don't even know where they are loaded! I've never cared about it.)

When we chose contour colors, we loaded the contour colors at the lower end of the color table, specifically so they didn't conflict with the drawing color indices used by `cgColor`. But with contour colors it was no problem restricting ourselves to, say, 30 colors at most. Here we might want to use many more colors. How can we make sure the drawing colors we chose don't interfere with the colors we want to use for coloring the surface plots?

Here is how it is done. Suppose I need a handful of drawing colors. It is rare that I need more than five. I will restrict the drawing color to color indices 250 to 254. I *never* load drawing colors at indices 0 or 255. Then I will use the *Set_Shading* command in IDL to restrict the number of surface colors I want to use for shading or coloring surfaces to the indices 0 to 249. I'll use the *Values* keyword for this. In other words, I'll have 250 colors set aside for coloring the surface.

Here is how I add my *snowDepth* colors to the surface plot. Note how I load the color table for the colors and how I specify my background and axis colors with `cgColor`. The second positional parameter tells `cgColor` in which color index to load the color. Note also how I byte scale the *snowDepth* data into the color range of 0 to 249.

```
IDL> Device, Get_Decomposed=currentMode
IDL> Device, Decomposed=0
IDL> Set_Shading, Values=[0,249]
IDL> LoadCT, 33
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black', 250), $
      Background=cgColor('white', 251), $
```

```

XTitle=xtitle, YTitle=ytitle, $
ZTitle='Atmospheric Pressure', $
XStyle=1, YStyle=1, ZStyle=0, $
Shades=BytScl(snowDepth, Top=249)

IDL> Device, Decomposed=currentMode
IDL> cgColorbar, NColors=250, Divisions=8, $
      Range=[Min(snowDepth), Max(snowDepth)], $
      XMinor=0, Format='(F0.2)', $
      AnnotateColor='black', XTitle='Centimeters', $
      Position=[0.2, 0.92, 0.8, 0.95], $
      Title='Snow Depth'

```

You see the result in Figure 14.

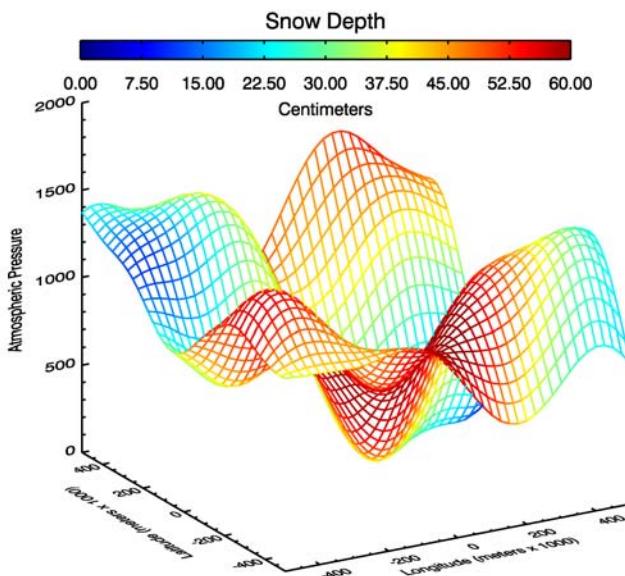


Figure 14: A fourth dimension can be added to the surface plot by coloring or shading the 3D data with a second data set.

Setting the Color Decomposition State

When we start to work with color indices, such as those specified with the *Shades* keyword, in surface plots, we put ourselves in a position where we *must* use the indexed color model to display the colors. This becomes a problem with the code we have just written because it sets the color decomposition state with the *Decomposed* keyword to the *Device* command.

Of course, this works perfectly on the display, because both normal display devices (i.e., X and WIN) accept a *Decomposed* keyword for the *Device* command. But some devices, such as older PostScript devices (IDL version 7.0 and below), do not. On those devices, the code above would throw an error on the first or second line of code!

The way we get around this problem is to use two programs from the [Coyote Library](#) that encapsulate the device and version dependencies so that we can obtain the current decomposition state, and set the decomposition state of the current graphics device in a device and version independent way. The two programs are [GetDecomposedState](#) to get the current color decomposition state or mode, and [SetDecomposedState](#) to set the current decomposition state or mode. The [SetDecomposedState](#) program uses [GetDecomposedState](#) to obtain the current color decomposition state before it is changed. You will see these programs used in the code throughout the rest of this chapter.

Another common use of the *Shades* keyword is to perform “elevation shading”. That is, to color the surface according to its elevation or height in the surface plot. In this case, we simple pass the *Shades* keyword a byte scaled version of the data itself. The code looks like this.

```
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Set_Shading, Values=[0,249]
IDL> LoadCT, 33
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black', 250), $
      Background=cgColor('white', 251), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0, $
      Shades=BytScl(data2d, Top=249)
IDL> SetDecomposedState, currentState
IDL> cgColorbar, NColors=250, Divisions=8, $
      Range=[Min(data2D), Max(data2D)], $
      XMinor=0, Format='(I0)', $
      AnnotateColor='black', XTitle='', $
      Position=[0.2, 0.92, 0.8, 0.95], $
      Title='Pressure Units'
```

You see the result in Figure 15.

Another type of surface plot is a “lego” plot. This kind of surface plot can be created by simply setting the *Lego* keyword on the *Surface* command.

```
IDL> Surface, data2D, lon, lat, /Lego, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
```

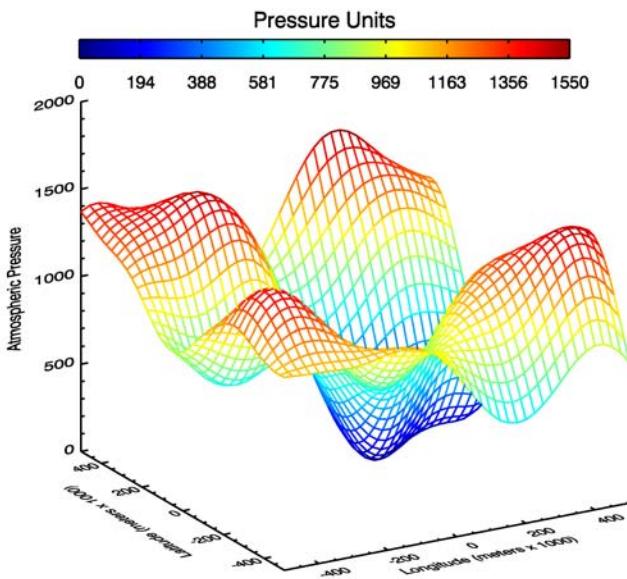


Figure 15: Here the surface is shaded by its own elevation using a technique called “elevation shading.”

```
XTitle=xtitle, YTitle=ytitle, $  
ZTitle='Atmospheric Pressure', $  
XStyle=1, YStyle=1, ZStyle=0
```

You see the result in Figure 16.

Creating Shaded Surface Plots

The surface plots we have drawn so far have been what we call “wire mesh” surface plots. But sometimes we prefer a solid surface. In IDL traditional graphics, these solid surfaces are called “shaded surfaces” and are created with the *Shade_Surf* command. The keywords available for the *Shade_Surf* command are identical to the keywords you use with the *Surface* command.

For example, the basic shaded surface is created with commands like this. (Note I have added commands to create the data from the same commands used in previous sections. You don’t need to re-type these commands if you already have this data in IDL’s memory.)

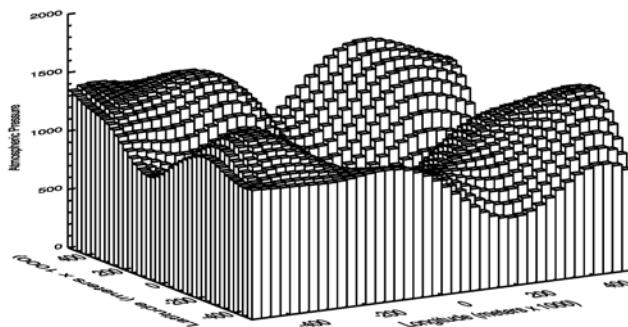


Figure 16: A “lego” surface plot is created simply by setting the `Lego` keyword to the `Surface` command.

```
IDL> data2d = cgDemoData(2)
IDL> s = Size(data2D, /Dimensions)
IDL> lon = IndGen(s[0]) * 25000L - 500000L
IDL> lat = IndGen(s[1]) * 25000L - 500000L
IDL> lon = lon / 10^3
IDL> lat = lat / 10^3
IDL> xtitle = 'Longitude (meters x 1000)'
IDL> ytitle = 'Latitude (meters x 1000)'
IDL> Shade_Surf, data2D, lon, lat, $
      Color=cgColor('black'), $
      Background=cgColor('white'), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0
```

What you see when you type this last command depends on several factors. The most important factor is the color model you are using in the IDL session when you type the command. If you are using a decomposed color model (the default model for IDL), you will see the display shown in the top panel of Figure 17. If you are using an indexed color mode, you might well see something that looks like the display in the bottom panel of Figure 17.

The `Shade_Surf` command is a contemporary of the `Contour` and `Surface` commands, and as such, prefers an indexed color environment. But, the gray scale colors are odd to see in a decomposed color environment. We

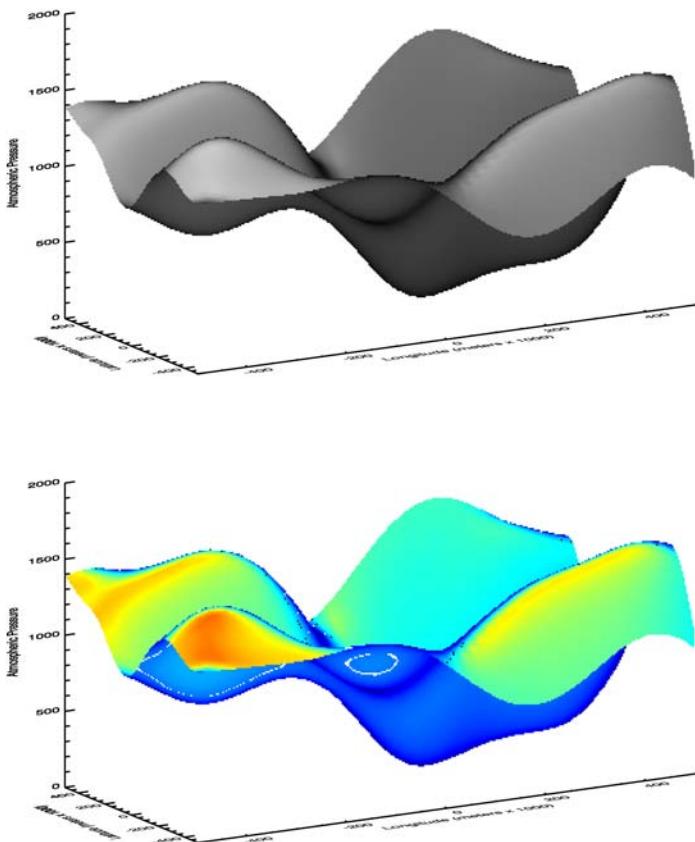


Figure 17: What you see when you execute a `Shade_Surf` command depends on several factors, the most important of which is the color model you are using in your IDL session.

naturally expect colors in the color table to be ignored in a decomposed color environment. But, normally, when we ignore the color table, the colors we display are expressed in shades of red, not shades of gray. What's going on here!?

It turns out that what you are seeing on your display is not really a surface plot, as if you used the `Surface` command in IDL. Instead, you are viewing an *image* of the surface plot! And 2D images, when they are displayed with

the decomposed color model on, are always displayed in shades of gray. (We discuss this in detail in the next chapter.)

Normally, it wouldn't matter whether you were looking at an actual surface plot or an image of the surface plot, because they both look the same on the display. But, the fact that this is an image does occasionally cause subtle (and sometimes not so subtle!) problems when we are working with shaded surfaces. I'll say more about this later, but just keep this fact in mind as we move through the examples.

Perhaps the clearest way to illustrate that this is an image we are looking at, with axes tacked onto it, is to type this same command in PostScript output, using the color decomposition model. (This only became available in IDL 7.1.) You see what happens when I do this in Figure 18.

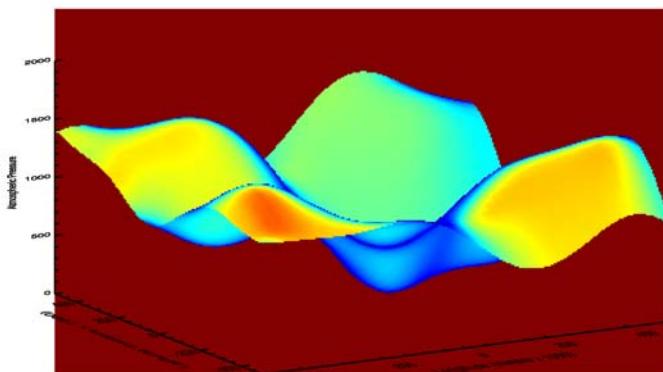


Figure 18: The same `Shade_Surf` command rendered in PostScript with the decomposed color model on shows clearly that the shaded surface is rendered as an image with axes tacked onto it.

Okay, a mess. How do we solve this problem?

We solve it the same way we just solved it for the `Surface` command. We make sure we always use the indexed color model when we issue the command, and we restrict the surface colors to some portion of the color table that doesn't interfere with other drawing colors. We restrict colors with the `Set_Shading` command and the `Values` keyword, and we use index parameters with `cgColor` to keep drawing colors away from the shading colors.

Here is the same shaded surface displayed properly. (Although with a less garish color table!) This code produces identical results both on your display and in a PostScript file.

```
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Set_Shading, Values=[0,249]
IDL> cgLoadCT, 5, /Brewer, /Reverse, NColors=250
IDL> Shade_Surf, data2D, lon, lat, $
      Color=cgColor('black', 250), $
      Background=cgColor('white', 251), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0
IDL> SetDecomposedState, currentState
```

You see the result in Figure 19.

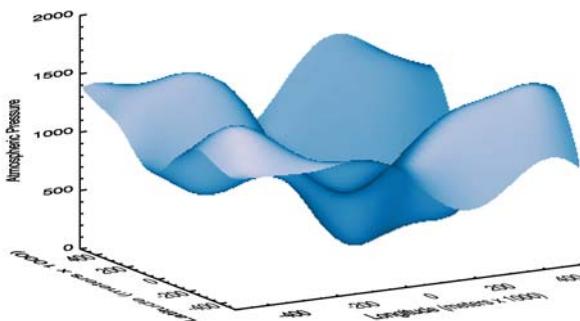


Figure 19: To properly display a shaded surface, use indexed colors and keep the surface colors separate from the drawing colors in the color table. Set_Shading can be used to restrict the color space.

The shading used by the *Shade_Surf* command is called “Gouraud shading.” It uses one light to illuminate the surface, and it is the only option available in this traditional graphics command. It is possible, though, to move the light source that illuminates the surface to achieve slightly different colors on the surface. Normally, the light is positioned at [0,0,1] in normalized axis coordinates. This makes the light rays parallel to the Z axis. If you want the surface to be lighted from the side, you can position

the light at [0.0,1.0,0.5] with the *Light* keyword to the *Set_Shading* command.

```
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> Set_Shading, Values=[0,249], Light=[0.0,1.0,0.5]
IDL> cgLoadCT, 5, /Brewer, /Reverse, NColors=250
IDL> Shade_Surf, data2D, lon, lat, $
      Color=cgColor('black', 250), $
      Background=cgColor('white', 251), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0
IDL> SetDecomposedState, currentState
```

You see the result in Figure 20. Compare this result to the result in Figure 19. There are clear differences from changing the light source.

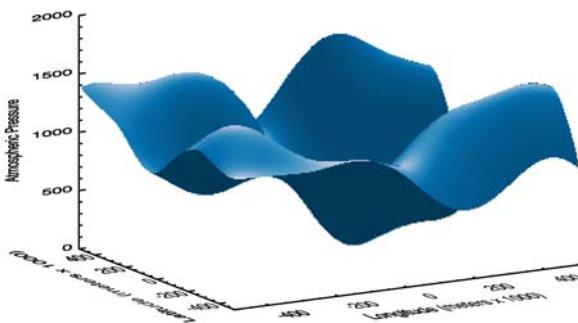


Figure 20: By moving the light source, you can create different effects with Gouraud shading.

Note: The *Set_Shading* parameters are “sticky.” That is to say, once set, they remain in effect in that IDL session until changed. You will have to set them back to default values if that is what you want. Otherwise all shaded surfaces created in this IDL session will use these new parameters.

Shading With Another Data Set

As with the *Surface* command, another data set can be used for the “colors” or shading values. Simply use the *Shades* keyword, as before, to

accept a byte scaled 2D array of the same size as the data you are shading. Unfortunately, this is not “texture mapping” in which another image is warped directly onto the surface. It just sets different values to use for the shading parameters.

For example, consider this code.

```
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> snowDepth = cgDemoData(3)
IDL> cgLoadCT, 5, /Brewer, /Reverse, NColors=250
IDL> Shade_Surf, data2D, lon, lat, $
      Color=cgColor('black', 250), $
      Background=cgColor('white', 251), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0, $
      Shades=BytScl(snowDepth, Top=249)
IDL> SetDecomposedState, currentState
```

You see the result in Figure 21. It is difficult to tell at a glance that this surface is being shaded with a second data set.

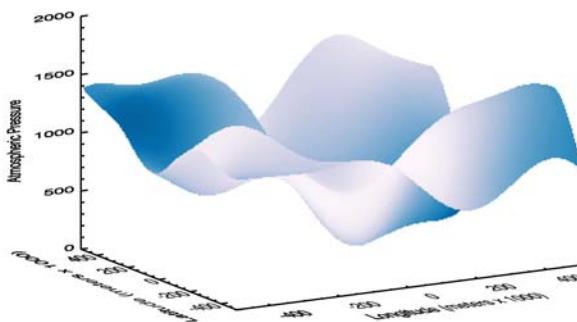


Figure 21: It is difficult to tell another data set is shading this surface.

It would be easier to tell, perhaps, if we performed elevation shading on the original data, then added the second data set on top of the first in another color, and also used a wire mesh surface instead of a shaded surface.

First, we create the elevation shaded surface of the original data.

```
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> cgLoadCT, 5, /Brewer, /Reverse, NColors=250
IDL> Shade_Surf, data2D, lon, lat, $
      Color=cgColor('black', 250), $
      Background=cgColor('white', 251), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=1, YStyle=1, ZStyle=0, $
      Shades=BytScl(data2D, Top=249)
IDL> SetDecomposedState, currentMode
```

Then, we load a different color table and display a wire mesh version of the second data set on top of the first. Be sure to set the *NoErase* keyword on this second *Surface* command.

```
IDL> SetDecomposedState, 0, CurrentState=currentState
IDL> cgLoadCT, 33, NColors=250
IDL> Surface, data2D, lon, lat, $
      Color=cgColor('black', 250), $
      Background=cgColor('white', 251), $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=5, YStyle=5, ZStyle=4, /NoErase, $
      Shades=BytScl(snowDepth, Top=249)
IDL> SetDecomposedState, currentState
```

You see the result in Figure 22. Now it is easier to see the second data set on top of the first.

Setting Up a 3D Scatter Plot

A *Surface* command can sometimes be used to set up a three-dimensional coordinate space for other types of graphical displays. For example, it can be used to set up a coordinate system for displaying a 3D scatter plot. A scatter plot resembles a surface in some respects, but is usually used when the data is randomly distributed and not easily gridded.

To see how this is done, let's randomly sample the data we have been using in this chapter. We use the same technique for doing this that we used at the end of the contouring chapter. Here is the code for creating 200 random data points from the *data2D* array.

```
IDL> lat2d = Rebin(Reform(lat, 1, 41), 41, 41)
IDL> lon2d = Rebin(lon, 41, 41)
IDL> pts = Round(RandomU(-3L, 200) * 41 * 41)
```

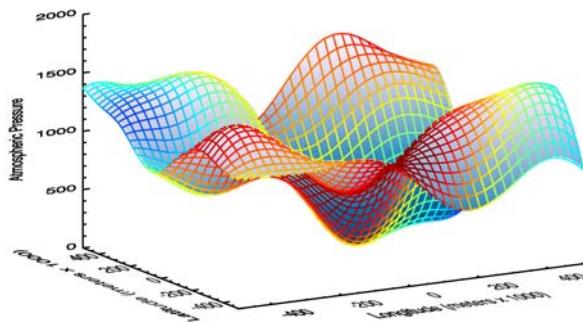


Figure 22: Using a surface command to add a wire mesh surface to a shaded surface makes it easier to see the second data set values.

```
IDL> dataIrr = data2d[pts]
IDL> lonIrr = lon2d[pts] + RandomU(5L, 200) * 50 - 25
IDL> latIrr = lat2d[pts] + RandomU(8L, 200) * 50 - 25
```

The data and the latitude and longitude vectors are now 200-element vectors, randomly distributed. If we examine the minimum and maximum values of the vectors, we see that the latitude and longitude vectors range from about -550 to 550 in value, and the data ranges from about 0 to 1500 in value. We can use the *Surface* command to set up a 3D coordinate space for displaying the scatter plot of this data.

Here I use the IDL *Dist* function to create a throw-away 2D data set for the *Surface* command. It is “throw-away” because I don’t care what the actual values are, I am using the *NoData* keyword to suppress the drawing of the surface anyway, and I am using the *[XYZ]Range* keywords to set up the actual data range of the axes. Note the *Save* keyword. This is required to save the 3D transformation matrix in the *!PT* system variable. It will be important in subsequent commands.

```
IDL> Surface, Dist(100), XStyle=1, YStyle=1, /NoData, $
      XRange=[-550,550], YRange=[-550,550], $
      ZRange=[0,1500], Color=cgColor('black'), $
      Background=cgColor('white'), $
      ZTitle='Atmospheric Pressure', $
      XTitle=xtitle, YTitle=ytitle, $
      Position=[0.075, 0.075, 0.925, 0.825], /Save
```

The scatter plot will be enhanced by adding a few more axes. Let's add a box-type set of axes in the XY plane, and let's add two more Z axes, just to provide a little more context to the plot for the observer. We use the *Axis* command to add axes to the plot.

```
IDL> Axis, XAxis=1, /T3D, XStyle=1, $  
      Color=cgColor('black')  
IDL> Axis, YAxis=1, /T3D, YStyle=1, $  
      Color=cgColor('black')  
IDL> Axis, ZAxis=0, /T3D, 550, 550, $  
      Color=cgColor('black')  
IDL> Axis, ZAxis=0, /T3D, 550, -550, $  
      Color=cgColor('black'), $  
      ZTitle='Atmospheric Pressure'
```

Now, we are ready to draw the random data points of the scatter plot.

First, we set up the colors for the plot.

```
IDL> symColors = Fix(BytScl(dataIrr))  
IDL> LoadCT, 33, /Silent
```

Notice a small trick on the first line of code above. We are going to pass these symbol color values to `cgColor` later on, so we can draw the colors in a device independent way. To do so, we have to convert them to a string, since that is what `cgColor` expects. If we convert byte values to strings in IDL, we get very strange results. In fact, we get the ASCII character representation of the value, not the “number” value we want. We have to convert the byte values to short integers before we convert them to strings to get the right “number” value we need to pass to `cgColor`.

We will draw the points with a filled circle (using `SymCat` to create this symbol) colored according to its Z-value. Before we draw the symbols, however, we will draw lines that connect the symbols to the “ground” of the XY plane of the plot, and we will draw small dots at the intersection of the line and the XY plane as an aid in visualizing where the symbols are located on the plot.

We draw the lines before the symbols, because we want the symbols placed “on top of” the plot. That is, these are the most important parts of the plot and should not be overwritten by other graphics. First we draw the light gray lines.

```
IDL> num = N_Elements(dataIrr)  
IDL> FOR j=0,num-1 DO Plots, [lonIrr[j], lonIrr[j]], $  
      [latIrr[j], latIrr[j]], [dataIrr[j],0], $  
      Color=cgColor('light gray'), /T3D
```

Then we draw a dot (*PSym=3*) at the bottom of the light gray lines.

```
IDL> For j=0,num-1 DO Plots, lonIrr[j], latIrr[j], 0, $  
    /T3D, PSym=3, SymSize=2.0, $  
    Color=cgColor('black')
```

We draw a filled circle symbol (*PSym=SymCat(16)*) in color at the top of the light gray lines.

```
IDL> FOR j=0,num-1 DO PlotS, lonIrr[j], latIrr[j], $  
    dataIrr[j], /T3D, PSym=SymCat(16), SymSize=2.0, $  
    Color=cgColor(StrTrim(symColors[j],2))
```

And we finish the plot by drawing a color bar to show the values the colors represent.

```
IDL> LoadCT, 33  
IDL> cgColorbar, Divisions=6, Format='(I0)', $  
    AnnotateColor='black', Charsize=0.8, $  
    Title='Atmospheric Pressure', $  
    Position=[0.2, 0.88, 0.8, 0.92]
```

You see the results in Figure 23.

Surfaces in PostScript Output

I want to say just a word about creating surfaces in PostScript output. PostScript output is a topic that has its own chapter in this book, but I want to point out that to obtain the proper rotation of surface axis labels you have to use either Hershey or TrueType fonts. Normally, this means setting the *Font* keyword to -1 or 1 on the *Surface* command, or setting the *!P.Font* system variable to -1 or 1 before the *Surface* command is executed.

Because I want the code in this book to run both on the display and in the PostScript device (all of the figures in the book are made by pasting the code to the IDL command line), I set the PostScript device up with the [Coyote Library](#) routine [PS_Start](#). In other words, to check the code, I paste the code to the IDL command line. If it works as I expect it to, I call [PS_Start](#), paste the code to the PostScript device, then call [PS_End](#) to close the PostScript file and produce the high resolution PNG file that I import into the book as a figure.

One of several things [PS_Start](#) does is to set the system variable *!P.Font* to 0 so that PostScript hardware fonts are used in the PostScript device. It also sets *!P.Charsize* to either 1.25 on Windows computers or to 1.5 on Unix computers. The reason for changing the character size is that typi-

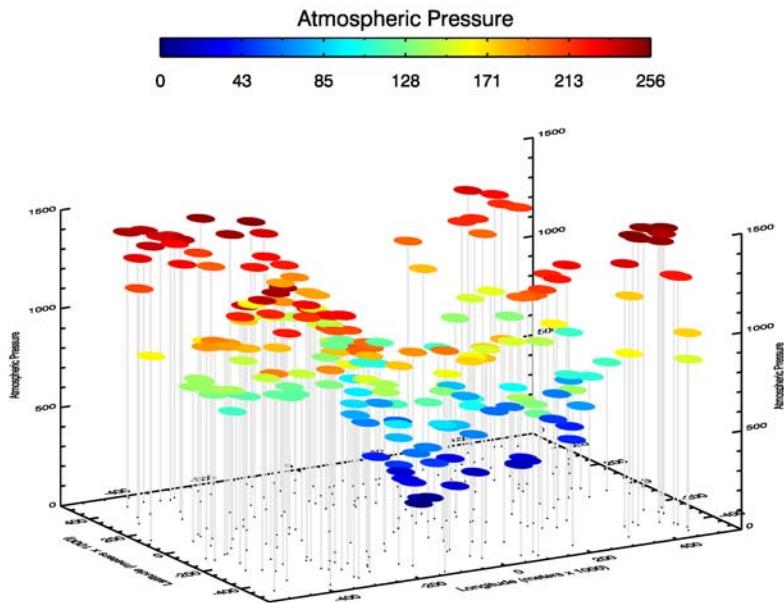


Figure 23: The `Surface` command can be used to set up 3D coordinate spaces for other types of 3D graphics, in this case a scatter plot.

calligraphic PostScript fonts are a bit smaller than “normal” fonts. They look better if they are made a bit larger than I would prefer them on the display. TrueType fonts (`Font=1`) are even smaller yet, and require an even larger character size.

You will learn more about this later, but I want to alert you to this fact in case you tried to produce a surface plot in PostScript and were confused about why your plot labels are not rotated properly. If you use `PS_Start` to display a surface, set the `Font` keyword to 1 or -1.

```
IDL> PS_Start, Font=1, Charsize=cgDefCharsize()
IDL> Surface, data2D, lon, lat
IDL> PS_End
```

Using a Refurbished Surface Command

The `cgSurf` command from the [Coyote Library](#) is a device independent, color model independent wrapper for both the `Surface` and `Shade_Surf`

commands. It is part of a suite of programs, collectively called the Coyote Graphics System (which includes `cgPlot`, `cgPlotS`, `cgContour`, `cgColorFill`, and `cgText`, etc.), for creating traditional graphics output that work and look the same on all devices and in any color decomposition state. `cgSurf` takes care of many of the details necessary to write device independent graphics programs, works with colors in a more natural way by specifying colors directly, and has features that are not available in the *Surface* or *Shade_Surf* commands.

The Coyote Graphics System commands produce, by default, black output on white backgrounds. This is the opposite of IDL traditional commands, but makes it possible to use these commands to produce (as much as possible) identical looking PostScript output. (You can return to the traditional color scheme of white on black by setting the *Traditional* keyword on the command.)

You will notice that textual output from these commands is slightly larger than normal. There are two reasons for this. First, larger output more closely matches the look and feel of the corresponding PostScript output when PostScript or TrueType fonts are used. And, second, larger output more closely matches the look and feel of similar object graphics programs in IDL 8.

Most of the surface displays shown in this chapter can be accomplished more easily with `cgSurf`, and with fewer commands. For example, here is how to produce a surface display with different colors for the axes, the top of the surface, and the bottom of the surface. Note that the title of the surface plot is presented “on-the-glass”, rather than rotated as it is with the *Surface* command. The keywords *TSize* and *TSpace* are available to tweak the size and location of the title to accommodate a variety of page sizes and layouts.

```
IDL> data = cgDemoData(3)
IDL> cgSurf, data, Color='red', Bottom='blue', $
      Title='Colorful Surface', TSize=1.0
```

You see the result in Figure 24. Note that you didn’t have to worry about using `SetDecomposedState` to get and set the color decomposition state. This is handled by `cgSurf` itself.

Here are the commands needed to create the shaded image with the wire overlay in Figure 22.

```
IDL> cgLoadCT, 5, /Brewer, /Reverse
IDL> cgSurf, data2D, lon, lat, $
      XTitle=xtitle, YTitle=ytitle, $
```

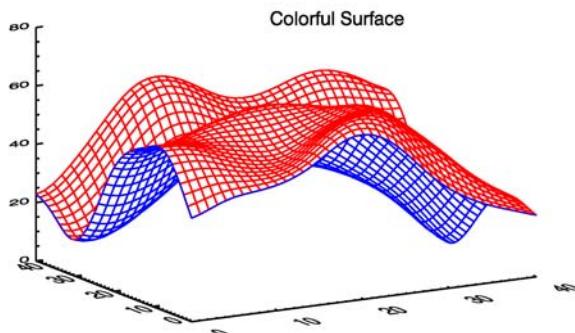


Figure 24: Colored surfaces are created in a more natural way and with fewer commands with the alternative surface command `cgSurf`.

```

ZTitle='Atmospheric Pressure', $
XStyle=1, YStyle=1, ZStyle=0, $
Shades=BytScl(data2D), /Shaded
IDL> cgLoadCT, 33
IDL> cgSurf, data2D, lon, lat, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      XStyle=5, YStyle=5, ZStyle=4, /NoErase, $
      Shades=BytScl(snowDepth)

```

Notice how many fewer commands you have to type and how much less careful you have to be with colors. No need to worry about your drawing colors interfering with the surface colors. The `cgSurf` program keeps track of all of that for you.

You see the result in Figure 25.

All drawing of graphics in `cgSurf` is done using the decomposed color model whenever possible. This avoids the problem of contaminating color tables by loading drawing colors in them. As part of this “do no harm” philosophy, you can also load your own color vectors into the program, so that the surface is always displayed in the correct colors, no matter what colors are currently loaded in the hardware color table. (This is especially important if the surface is to be displayed in a resizeable graphics window, as I will describe in just a moment.)

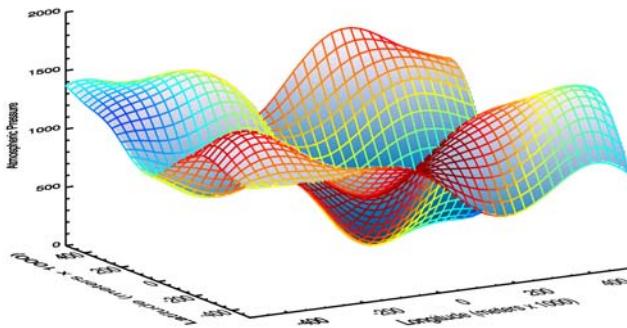


Figure 25: The `cgSurf` command allows you to create both shaded and wire mesh surfaces easily.

The way to load your own color vectors is to use the *Palette* keyword to pass an N by 3 (or 3 by N) byte array containing the red, green, and blue color vectors. One way to obtain such vectors is with `cgLoadCT` and its *RGB_Table* keyword. If used with this keyword, `cgLoadCT` doesn't actually load the colors into the color table. Rather, it just packages the color vectors that would be loaded into an N by 3 array that can be loaded with `TVLCT` or used with the *Palette* keyword that is available on other Coyote Graphics System routines.

```
IDL> cgLoadCT, 5, RGB_Table=palette  
IDL> cgSurf, data, Palette=palette, /Elevation
```

You see the result in Figure 26.

Surface Plots in Resizeable Graphics Windows

If you want to see the surface plot by itself in a resizeable graphics window, with controls for making hardcopy output files, set the *Window* keyword. The surface plot is displayed in `cgWindow`.

```
IDL> cgSurf, data, Color='red', Bottom='blue', $  
Title='Colorful Surface', TSize=1.0, /Window
```

The `cgWindow` application allows multiple commands to be executed in the window. For example, here is how you might display the shaded surface in Figure 25 in a resizeable graphics window.

```
IDL> cgLoadCT, 5, /Brewer, /Reverse, RGB_Table=pal_1
```

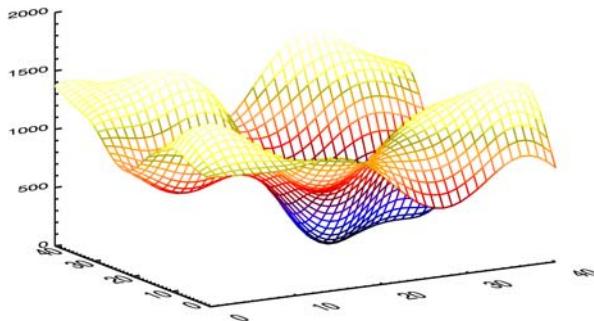


Figure 26: The `cgSurf` command can display itself with its own color or table vectors if the RGB vectors are passed into the program with the `Palette` keyword.

```
IDL> cgSurf, data2D, lon, lat, Palette=pal_1, $  
      XTitle=xtitle, YTitle=ytitle, $  
      ZTitle='Atmospheric Pressure', $  
      XStyle=1, YStyle=1, ZStyle=0, $  
      Shaded=BytScl(data2D), /Shaded, /Window  
IDL> cgLoadCT, 33, RGB_Table=pal_2  
IDL> cgSurf, data2D, lon, lat, Palette=pal_2, $  
      XTitle=xtitle, YTitle=ytitle, $  
      ZTitle='Atmospheric Pressure', $  
      XStyle=5, YStyle=5, ZStyle=4, /NoErase, $  
      Shaded=BytScl(data), /AddCmd
```

You see the result in Figure 27.

The `cgSurf` command allows you to position multiple plots with the *Layout* keyword, like other Coyote Graphics System commands. Here, for example, is how you can display a contour plot and a surface plot of the same data side-by-side in a resizeable graphics window.

```
IDL> cgWindow, WXSsize=800, YWSsize=400  
IDL> cgLoadCT, 33, /Window  
IDL> cgSurf, data2D, /Elevation, Layout=[2,1,1], $  
      /Shaded, /AddCmd  
IDL> cgContour, data2D, NLevels=12, /Fill, $  
      Layout=[2,1,2], /AddCmd  
IDL> cgContour, data2D, NLevels=12, Color='charcoal', $  
      Layout=[2,1,2], /AddCmd
```

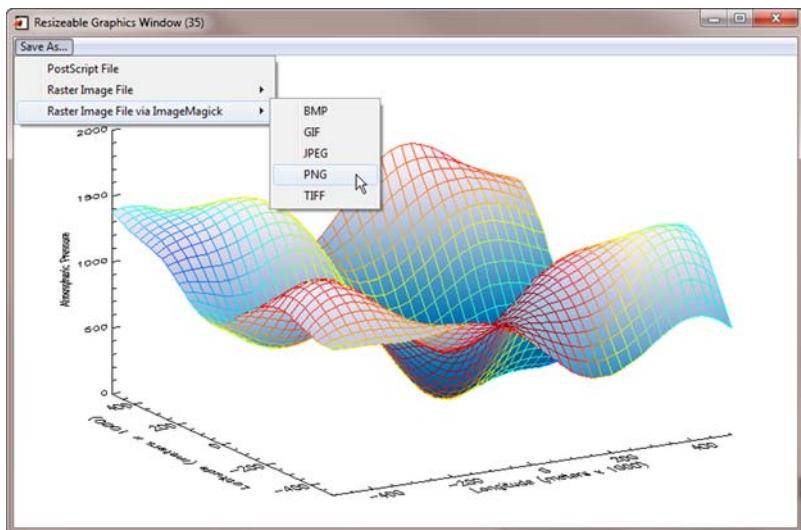


Figure 27: The `cgWindow` program allows you to add an unlimited number of graphics commands to the graphics window. The graphics window is resizable, and the window content can be saved in a PostScript file or in any of five different raster file formats.

You see the result in Figure 28.

A True 3D Surface Command

While `cgSurf` is a much better solution when writing device and color independent programs than the *Surface* command, it suffers from the same fundamental problem that plagues the *Surface* command. Namely, the *Surface* command is not a 3D command. (Technically, it is called a 2.5D command.) The surface is prohibited from rotating about all three surface axes simultaneously.

To use a true 3D coordinate space in IDL, you have to either work exclusively in the Z-graphics buffer or you have to use the object graphics system in IDL. This book is not about the object graphics system, of course. It is about IDL traditional graphics commands which were written well before the object graphics system was introduced in IDL 5.

But, that said, there is one object graphics program that I use so often I think of it as a “traditional” graphics command. It is `cgSurface`, and this

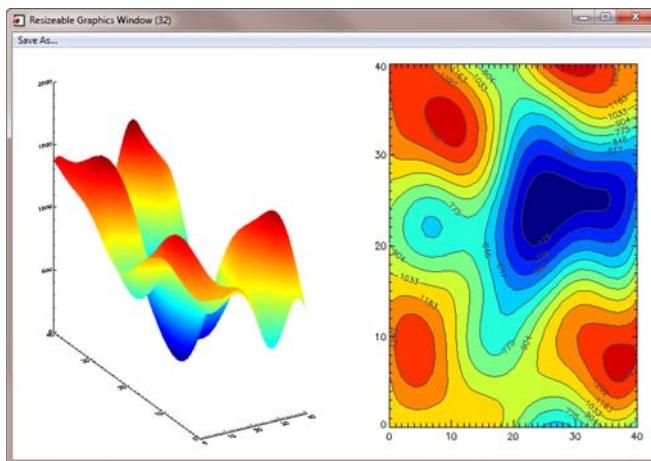


Figure 28: The `Layout` keyword can be used with `cgSurf` and other Coyote Graphics System routines to create multiple plots in resizable graphics windows.

command is written specifically to duplicate the functionality and way of working of the `Surface` and `cgSurf` commands. It uses many of the same keywords, has many of the same properties, and yet provides some of the essential functionality that the `Surface` command lacks.

In particular, the `cgSurface` command allows you to interactively rotate and move the surface in the display window, zoom in and out of the surface view, and change surface features by means of pull-down menus from the menu bar of the display window. You can display different types of surfaces (e.g., wire mesh, shaded, lego, dot surface, etc.) and you can shade the surface with elevation shading or by draping any 2D or true-color image on top of the surface.

You see in Figure 29 what the default surfaces look like as a wire-mesh and as a shaded surface.

```
IDL> cgSurface, data2d, lon, lat, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure'
IDL> cgSurface, data2d, lon, lat, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', /Shaded
```

If you click your mouse anywhere on the surface, you can rotate the surface in 3D space. If you click your mouse on any axis, you can move or

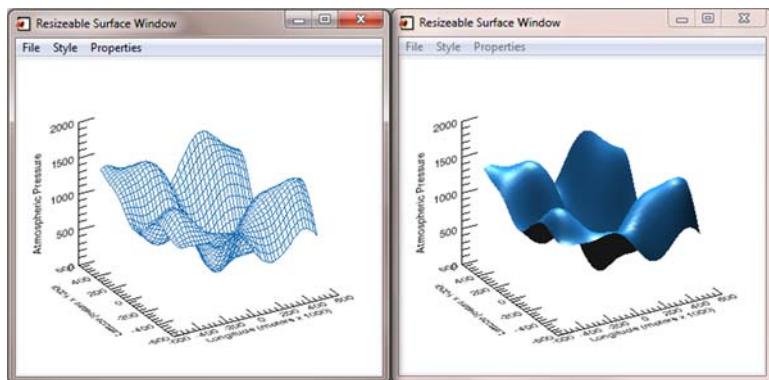


Figure 29: The default wire mesh and shaded surface plots, using the non-traditional `cgSurface` command.

translate the surface in the display window. You can do the same with the plot title (specified by means of a *Title* keyword) if it is displayed in the display window.

If you right-click the mouse on the surface, you will zoom into the surface. If you middle-click the mouse on the surface you will zoom out of the surface. By this means, you can rotate and position the surface in the display window for the best effect. And, of course, the display window is completely resizeable.

You can access and change surface properties by means of pull-down menus activated by selecting the buttons in the menu bar above the surface plot. You will find that you can change surface colors, turn elevation shading on or off, and save the surface in a variety of image file formats, including JPEG, PNG, and PostScript. The PostScript file is a true, vectorized PostScript file that can be opened in PostScript editing software for further manipulation if that is desired.

A shaded surface is illuminated with four different light sources and you can access these lights via a light controller to change the color and intensity of each light source, or to turn the light source off altogether.

Changing the light properties can cause the surface to be displayed with interesting effects. You see an example of the light controller in Figure 30.

Adding Surface Texture Maps

Another nice feature of `cgSurface` is the ability to drape any 2D or true-color image on top of a surface by means of the *Texture_Image* keyword.

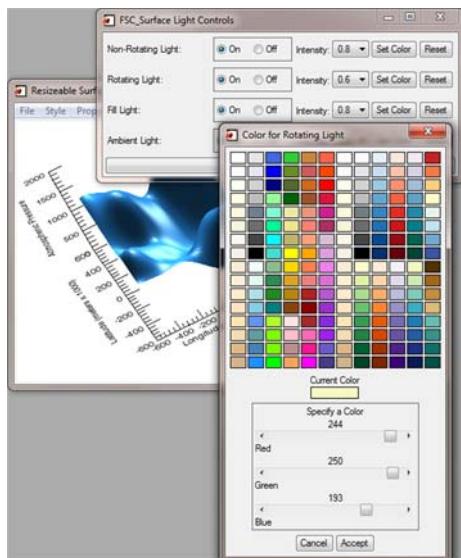


Figure 30: Lights can be turned on or off, and set to different colors and intensities with the Light Controller.

If a 2D image is used, the color table for displaying the texture image can be specified. The `ZScale` keyword in the command below “flattens” the surface, making it easier to see the texture.

```
IDL> earth = cgDemoData(7)
IDL> cgSurface, data2d, lon, lat, ZScale=0.5, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      Texture_Image=earth, CTable=4, /Brewer, $
      Title='2D Image as Texture'
```

You see the result in the left panel of Figure 31.

A three dimensional or true-color image can also be used.

```
IDL> rose = cgDemoData(16)
IDL> cgSurface, data2d, lon, lat, ZScale=0.5, $
      XTitle=xtitle, YTitle=ytitle, $
      ZTitle='Atmospheric Pressure', $
      Texture_Image=rose, $
      Title='True-Color Image as Texture'
```

You see the result in the right panel of Figure 31.

The ability to add a surface texture has practical applications. For example, it would be possible to add a MODIS true-color satellite image as a

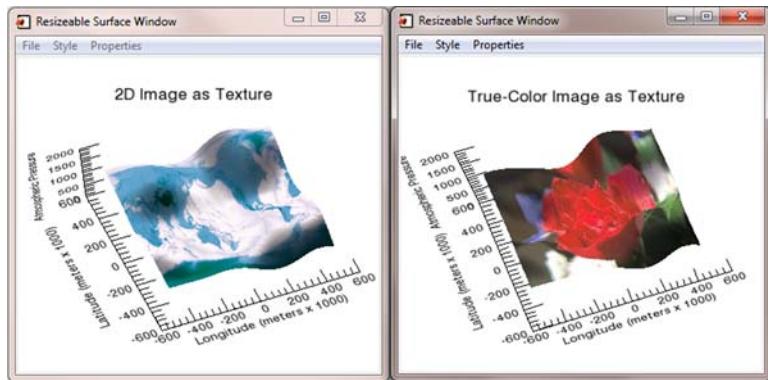


Figure 31: The `cgSurface` command can accept any 2D or true-color image as a texture map to be draped on top of the surface plot.

texture map to a digital elevation model (DEM) of a region of the Earth, so you can see the terrain in real-world colors.

There is such a digital elevation model and image in the IDL *examples* directory. First, let's read the image.

```
IDL> file = Filepath(Subdir=['examples', 'data'], $  
    'elev_t.jpg')  
IDL> Read_JPEG, file, image  
IDL> Help, image  
IMAGE      BYTE = Array[3, 512, 512]
```

Then, let's read the 64x64 DEM byte data for the same region.

```
IDL> file = Filepath(Subdir=['examples', 'data'], $  
    'elevbin.dat')  
IDL> dem = BytArr(64,64)  
IDL> OpenR, lun, file, /Get_Lun  
IDL> ReadU, lun, dem  
IDL> Free_Lun, lun
```

Now, we simply place the image onto the DEM as a texture map.

```
IDL> cgSurface, dem, Texture_Image=image, ZScale=0.5
```

You can now rotate the surface around for better viewing, turn the axes off from the program controls, zoom into the surface, and so on. You see the result in Figure 32.

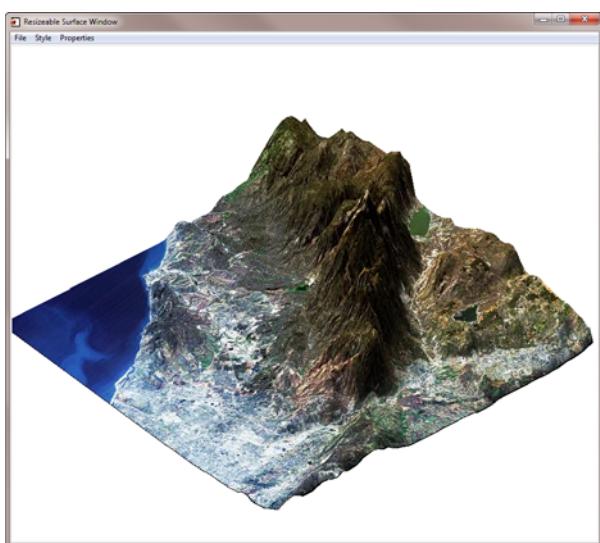


Figure 32: A visible color satellite image is draped on a digital elevation model of the same region, giving a life-like appearance to the data.

Chapter 7



Creating Image Plots

Reading Image Data

IDL got its start as a language for handling and processing image data, so it is no surprise that scientists and engineers the world over use it for this reason. The images we will be working with in this chapter as examples can be found, for the most part, in the *examples/data* sub-directory of the IDL distribution. There are several different ways you can read the data in these image files.

(It is possible to argue that data is only an “image” when it is displayed as an image. Otherwise, it is just data that can be read, manipulated, and displayed in other ways, such as in a contour or surface plot. This is a valid point, but since this chapter is about displaying data as images, I use the terms “image” and “data” to mean the same thing throughout the chapter. We should be clear, though, that “image” files store “data,” which do not always have to be displayed as images.)

If the files are in a standard image format (e.g., JPEG, PNG, etc.) then you can read them with the IDL *Read_Image* command. You *must* pass the name of the file, spelled correctly and in a format your computer operating system recognizes (including case, if your operating system, UNIX, for example, is case sensitive), to the *Read_Image* command. To help with formatting file names in a machine independent way, we often format the file name with the *Filepath* command in IDL.

The *Filepath* command produces a machine appropriate (the path is formatted with the proper forward or backward slashes, etc.), fully-qualified file name without having to remember how file names are formatted on

machines you don't normally work on. *Filepath* always starts looking for files in the main IDL directory, unless told otherwise with the *Root_Dir* keyword. The sub-directories are specified as a string array. You will see this command again, because it is an extremely useful command in IDL. *Filepath* does not check the case of your file names, so be careful with this if you are using an operating system (e.g., UNIX) that is case sensitive.

```
IDL> file = Filepath('muscle.jpg', Root_Dir=!Dir, $  
SubDirectory=['examples', 'data'])  
IDL> image = Read_Image(file)  
IDL> Help, image  
IMAGE      BYTE = Array[652, 444]
```

If users of your programs have difficulty typing file names (and many of them do!), then you may prefer to have them specify a file name with a file selection tool. The *Dialog_Pickfile* command is tailor-made for this purpose. It uses the file selection tool that is native for the window system you are running IDL on. This selection method should be familiar to you and your users. It is possible to configure the dialog to start with a particular file in a particular directory, as in this command, so the user only has to click the Accept button if the selection is correct. Otherwise, the user can make whatever changes are desired to select a file.

```
IDL> file = Dialog_Pickfile(File='muscle.jpg', Path=!Dir)
```

You see an example of *Dialog_Pickfile* on a Windows machine in Figure 1.

Note: There is a bug in *Dialog_Pickfile* (through at least IDL 8.0) that causes only the last 14 or 15 letters in a file name to be highlighted and displayed in the file name selector. If you use a file name longer than this, the entire file name is there, but it is displayed and highlighted incorrectly. This can cause confusion for users.

Notice the *Cancel* button in the bottom right corner of the dialog. Learn to fear that button! Normally, if buttons are provided in a graphical user interface, users will click them. *Dialog_Pickfile* is perfect for allowing user to make a file selection. But, if they *don't* make a selection, if they click the *Cancel* button instead, you have to anticipate that, too, or your programs will throw an error when you try to read the file.

This is of almost no importance when you are noodling around on the IDL command line, but it makes a huge difference in the programs you write. If the user clicks the *Cancel* button, a null string is returned.

```
IDL> Help, file  
FILE      STRING = ''
```

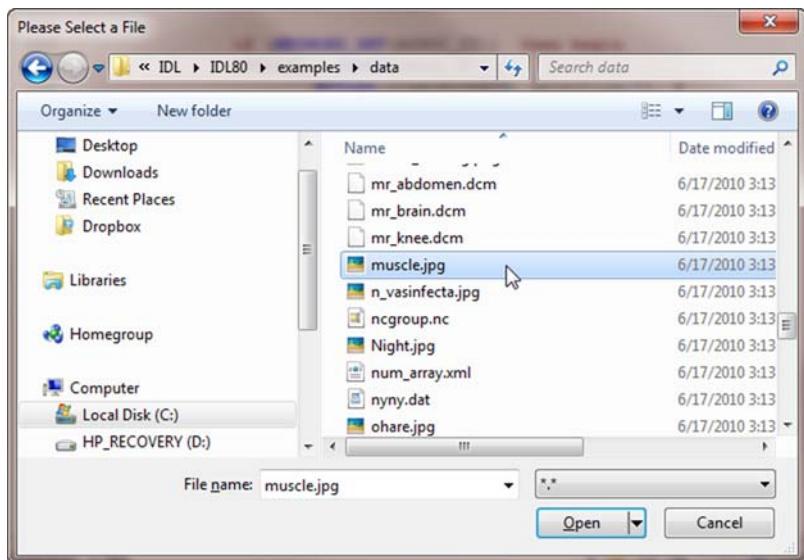


Figure 1: The file selection tool *Dialog_Pickfile*, shown here for a Windows machine.

If the user clicks the *Open* button, nothing, oddly, is actually opened. But what *does* happen is a fully qualified, machine-appropriate file name is returned to you by the dialog.

```
IDL> Print, file
```

```
C:\Programs\ITT\IDL\IDL80\examples\data\muscle.jpg
```

Most of the time, if we include a *Dialog_Pickfile* in a program, we just check for this null string and return out of the program if the null string is returned from the dialog. Usually, not much can be done in a program if the file we need to open isn't available.

```
file = Dialog_Pickfile(File='muscle.jpg', Path=!Dir)  
IF file EQ '' THEN RETURN
```

If you are only interested in opening JPEG files, for example, it doesn't make much sense to see all the TIFF or PNG files that also reside in a directory. You can use the *Filter* keyword to show only those files you are particularly interested in.

```
IDL> file = Dialog_Pickfile(File='muscle.jpg', $  
Filter='*.jpg', Path=!Dir)
```

The filter can use any number of regular expression wildcard characters to make the filter as specific or as general as you need it to be.

The command above does not really get you to the directory where you want to start looking for the *muscle.jpg* file. (The file actually resides in the */examples/data* sub-directory of the starting directory *!Dir*.) We want to be able to specify the path (with the *Path* keyword) in a machine appropriate way. We can use the *Filepath* command, along with another useful IDL command, *File_Dirname*, that separates the path or directory from a file name.

```
IDL> startPath = File_DirName(Filepath(Root_Dir=!Dir, $  
Subdirectory=['examples', 'data'], '*'))  
IDL> file = Dialog_Pickfile(File='muscle.jpg', $  
Filter='*.jpg', Path=startPath)
```

Note: You may want to use the *Coyote Library* routine *FSC_Pickfile* in place of *Dialog_Pickfile*. *FSC_Pickfile* is a *Dialog_Pickfile* wrapper with several useful properties. One is that it “remembers” the last directory and (optionally) the last file the user selected in that IDL session and when it is called a subsequent time, will position itself automatically in the last directory where the previous file was selected. This avoids quite a lot of typing, since often images we want to work with reside in the same directory.

Once you have a file name, of course, and it points to a file in one of the common image formats, you can read it with *Read_Image*. The *Read_Image* program uses the file extension as an indicator of what kind of file to read.

```
IDL> image = Read_Image(file)
```

Read_Image is a wrapper for lower-level file reading routines in IDL. These have names such as *Read_TIFF*, *Read_GIF*, *Read_JPEG*, *Read_PNG*, and so on. Search the IDL on-line help for “*Read_*” to see a complete list of the image file types IDL can read. All of the common image formats will be listed there. Some of the image reading routines will be procedures and some will be functions. There is no rhyme or reason for this, it is just a fact. (Computer programming styles change about as often as hemlines, and you will see vestiges of these styles in the way IDL programs are written, how keywords are formulated, and how procedures and functions are named. If you decide to give up programming, but want to remain in science, there is an anthropology Ph.D. thesis waiting for you here.)

```
IDL> Read_JPEG, file, image
```

```
IDL> image = Read_PNG(png_file)
IDL> image = Read_TIFF(file, r, g, b, GeoTIFF=geo)
```

If you are familiar with keywords used for a particular reading function, then you can use those same keywords with *Read_Image* if you are reading a file of that type. Positional and keyword parameters are also passed along by *Read_Image* to the lower-level image reading routine.

```
IDL> image = Read_Image(file, r, g, b, GeoTIFF=geo)
```

Information About Image Files

It is often useful to know more about the image file before you read the file. All of the common image formats in IDL have “query” routines associated with them that will query the image file for specific image information. These have names that correspond to the *Read_** routines discussed in the previous section. You will find *Query_TIFF*, *Query_GIF*, *Query_JPEG*, *Query_PNG*, and so on. These low-level query routines can be accessed as a group by calling the *Query_Image* function (the counterpart of the *Read_Image* function). *Query_Image* uses the file extension to determine which low-level file query routine to call.

These routines are functions that return a 1 if the image file is a valid file of the specified type and a 0 otherwise. Information about the file is returned in the form of a structure. In this example, the variable *info* is an output parameter that upon returning has information about the *muscle.jpg* file.

```
IDL> validFile = Query_Image(file, info)
IDL> Help, info, /Structure
      ** Structure <6fe4d450>, 7 tags:
          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'
```

The output shows there is one image in this JPEG file, the pixel type is 1, which corresponds to IDL variable type “byte,” there is no color palette or color table in this file, and it is a single channel image (in other words, an 8-bit byte image). The *dimensions* field can be printed to obtain the size of the image.

```
IDL> Print, info.dimensions
652    444
```

There is a program in the [Coyote Library](#) named [ImageSelect](#) that takes advantage of *Dialog_Pickfile*, and the *Read_** and *Query_** routines to

allow you to select common image files to read. One advantage of [ImageSelect](#) is that as you select the files, you are able to see small thumbnail images of the original images, along with information about the image that might be helpful to you. The *Demo* keyword automatically locates you in the IDL *examples/data* sub-directory.

```
IDL> image = ImageSelect(Filter='*.jpg', /Demo)
```

You see an example of [ImageSelect](#) in Figure 2.

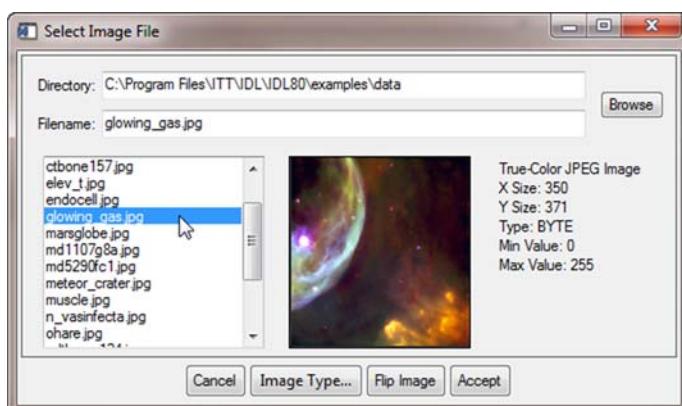


Figure 2: The [ImageSelect](#) program allows the user to see a small thumbnail size image of the original image, with image information on the right-hand side of the display.

Notice the [Flip Image](#) button on the [ImageSelect](#) dialog. This allows you to reverse the image in the Y direction. We will have more to say about this in just a moment, but IDL uses the convention that the [0,0] pixel in an image is displayed in the lower-left corner of the graphics window. Most other image display software displays this pixel in the upper-left corner of the graphics window. To work with images created in other software, the image must almost always be ‘flipped’ in IDL. This is especially true of TIFF images.

Reading Images In Other File Formats

Naturally, all images are not in common image formats. Some images are in scientific data formats (netCDF, HDF, CDF) and some images are in what we sometimes call flat binary formats of various types. Fortunately, IDL has general purpose tools that allow the user to open and read almost any type of image data file. If IDL itself doesn’t provide a specific program

for reading a file, you can often find programs written in IDL on the Internet for reading files of a particular format.

For example, many astronomers use FITS image files (Flexible Image Transport System). IDL does not provide a built-in program for reading FITS files, but several such programs, written in IDL, can be found in the NASA IDL Astronomy Library, one of the best maintained, free, and respected IDL program libraries in the world. The library can be found here: <http://idlastro.gsfc.nasa.gov/>.

Users of passive microwave data collected by satellite and distributed by the National Snow and Ice Data Center (NSIDC) can obtain an IDL program named the DataViewer to read image files in a variety of flat file and scientific data formats (<http://nsidc.org/data/ease/tools.html>). Source code for the DataViewer is also available as part of the Catalyst Library (<http://www.idlcoyote.com/catalyst/catlib.html>), another source, along with the [Coyote Library](#), of IDL programs on the Internet.

It is certainly possible to read flat, binary image files in IDL, but to do so you need to know more about the data in the file. You need to know, for example, if there is a header in the file (or, more exactly, you need to know if there is a byte offset to the data in the file), you need to know the type of data in the file (e.g., byte, integer, float, etc.), and you need to know how the data is organized in the file (e.g., the dimensions of the image). This information is required to read *any* image file, of course, but common image file formats and scientific data formats (e.g., netCDF or HDF) store this information in the image file itself, whereas flat, binary files usually do not.

There are several flat, binary image files among the image files in the IDL *examples/data* directory. How would you read them? With great difficulty, probably, because you don't know anything at all about them. These image files are stored as an extremely long sequence of 0s and 1s. If you don't know what those values mean, there is almost no hope of reading the image files properly.

Fortunately, ITTVIS has anticipated this problem and includes a file named *data.txt* in the same *examples/data* directory. If you opened this file in a text editor, for example, you could read a description of the data file and how the data is organized. All of the image data in these flat, binary data files in this directory are byte type data.

Here is the entry for the *head.dat* data file.

```
head.dat
80 100 57
57 MRI slices of the human head
```

To read this data into IDL, we can use the *Read_Binary* command.

```
IDL> filename = Filepath('head.dat', $  
Subdirectory=['examples','data'])  
IDL> image = Read_Binary(filename, Data_Type=1, $  
Data_Dims=[80,100,57], Endian='native')  
IDL> Help, image  
IMAGE      BYTE = Array[80, 100, 57]
```

The *Data_Type* keyword should be set to the type code for the particular data type stored in the file. In this case, type 1 is byte data. The type codes correspond to the normal IDL data type codes. See the on-line help for the *Size* function, called with the *Type* keyword, for an explanation of data type codes.

Byte Swapping: Big Endian Versus Little Endian

The *Endian* keyword refers to the byte ordering in the file. The values in flat, binary files are stored in a machine specific way that is referred to as “little endian” or “big endian”. For example, there are two ways to store a two-byte integer value. With the first byte first (little endian) or with the second byte first (big endian). If you are on a little endian machine and you are reading data that was created on a big endian machine, or vice versa, then you will have to swap the bytes in the values you read from the file. If you don’t, the values will be wildly wrong. In fact, this is how you usually recognize there is a problem. The values don’t make the least bit of sense to you!

IDL has the ability to swap bytes as it reads the data from the file, and that is what the *Endian* keyword allows you to do. The value *native* indicates that no byte swapping is required, and is the default setting. Other possible values are *little* and *big*, which describe the data in the file, not the architecture of your machine. If the data is little endian and your machine is little endian, then the data will not get byte swapped. If the data is little endian and your machine is big endian, then if the *Endian* keyword is set to *little*, the data will be byte swapped. If you forget to swap the bytes as you read the data from the file, you can do it later with the *ByteOrder* command. Naturally, if your data is byte type there is no need to worry about byte swapping. A byte is a byte the world over on all machine architectures.

Most of the machines running IDL these days are little endian, but not all. There is the increasingly rare Solaris machine, which is big endian. And, some file formats, such as the FITS (Flexible Image Transport System) files, used widely in astronomy applications, store their data in a big endian format.

If you want to know if your machine is big endian or little endian, you can type these commands to find out.

```
IDL> little_endian = (Byte(1, 0, 1))[0]
IDL> IF (little_endian) $
      THEN Print, "I'm little endian." $
      ELSE Print, "I'm big endian."
```

The [Coyote Library](#) program [GetImage](#) provides a user-friendly interface for reading flat, binary files of this type.

```
IDL> filename = Filepath('head.dat', $
                           Subdirectory=['examples','data'])
IDL> image = GetImage(filename)
```

You see an example of [GetImage](#) in Figure 3.

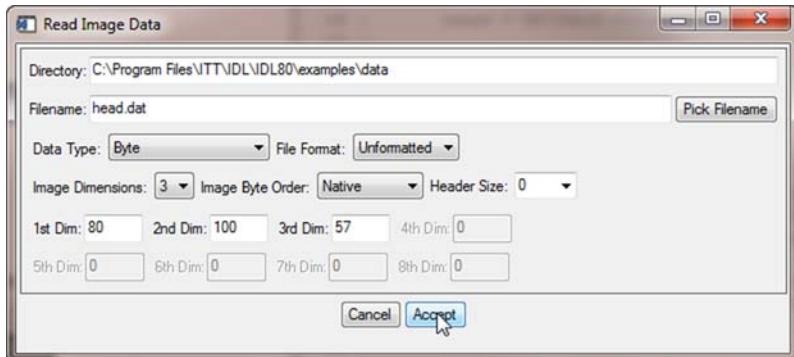


Figure 3: The [GetImage](#) program has a user-friendly interface for selecting parameters required for reading flat, binary files.

For the purposes of the examples in this chapter, most of the time we will load images using the [Coyote Library](#) program [cgDemoData](#), the same program we have used in previous chapters. [cgDemoData](#) is just a program that “knows about” the image files found in the *examples/data* directory in the IDL distribution.

```
IDL> head = cgDemoData(8)
IDL> Help, head
HEAD      BYTE = Array[80, 100, 57]
```

Using Traditional Image Display Commands

The traditional IDL graphics commands for displaying images are *TV* and *TVScl*. The commands are identical in nearly every way, including the keywords that can be used with them. They differ in only one respect: *TVScl* byte scales the data passed it into the number of colors in the hardware color table (set by *!D.Table_Size*, but normally 256 for machines with 24-bit graphics cards, as almost all machines have today) before the data is displayed. *TV* simply takes its data and truncates the data to bytes before it is displayed. In both cases, what ends up on the display are byte values in the range 0 to 255.

You might think, then, that *TVScl* would be the logical command to use in preference to *TV*, since it is more likely to result in a “correct” display of non-byte image data. In fact, *TVScl* is almost useless for writing graphical programs that use images. Its only real utility is at the IDL command line where it is often used to have a “quick look” at image data. It would be a mistake, in my opinion, to use a *TVScl* command in a graphics program.

“But,” you protest, “if I display my floating point image data with a *TV* command, it looks screwy.” Yes, of course. Because the *TV* command simply displays the first byte of any image pixel value. In other words, it truncates image pixel values to byte values before the image is displayed. If the data values are not byte values, the *TV* command will almost certainly display the data “incorrectly.”

“Ah, ha!” you say, “this proves your argument is lame.” On the contrary, it simply proves that I’ve spent more time than you have around scientists and engineers who struggle mightily to understand how colors and images work together to produce information that they can understand and explain. *TVScl*, believe me, causes more misunderstanding, anguish, and wasted work hours than almost any other command in the IDL language.

Yes, the *TV* command requires byte data if the image is going to be “correctly” displayed. This simply means that you have to provide byte data as an argument. But, and this is a *huge* advantage, by forcing yourself to do this, you will begin to write image display programs that make sense to you. And, you will avoid the misunderstandings and errors in perception that *TVScl* often causes.

In just a moment, I’ll show you how to use the *BytScl* command to byte scale your image data for display with the *TV* command, a necessary skill if you take my advice and abandon the *TVScl* command. But, first, more bad news.

I don't know any other way to say this, so I'll just say it straight out. I also think it is a mistake to use the *TV* command in an IDL graphics program.

"What!? Are you crazy!," I hear you saying. "You just told us the two commands for displaying images are *TV* and *TVScl* and now you say you don't use either of them? You are *muy loco, amigo!*!"

Probably, but I haven't used a *TV* or *TVScl* command in one of my IDL programs since about 1996, shortly after I bought my first 24-bit color monitor. And I am going to recommend you don't either.

But, you can decide for yourself. I'm going to show you how these commands work, and point out their limitations with respect to writing graphics display programs that work in a device and display independent manner. And I will also show you alternatives that will make your IDL programming life infinitely easier and more understandable.

Preparing Image Data for Display

Let's start at the beginning. What is the purpose of displaying an image? A lot of people answer this question by saying, "So I can see my data." But it is exactly this kind of thinking that gets these people in a lot of trouble. It is extremely unlikely that you will ever "see" your data on a computer display. What you see, nearly always, is a *representation* of your data. A representation that is greatly limited by the particular hardware capabilities of your computer.

For example, two-dimensional data sets collected in the real world seldom, if ever, contain data with values ranging from 0 to 255. And, yet, when we display a two-dimensional data set on a computer as an image, we *always* display pixel values that are in the range 0 to 255. Computers, with a very few high-end exceptions, cannot do anything else.

So, we take data that may have an extremely large range (e.g., 50000 to 150000) or data that may have an extremely small range (0.435 to 8.648), and we display it using a computer that scales *all* data, necessarily, into bytes that can only hold values 0 to 255.

What you are always looking at when you display an image is a representation of your data that you hope will show features in your data that will help you interpret or understand what your data means in the real world. Anything you can do to enhance your ability to interpret or understand your data, by manipulating the representation of your data, is generally considered to be a *Good Thing*. This forms the basis for many image processing techniques, as we will discuss later.

But for now, the single thing you can do to that will enhance the interpretation and understanding of your image data the most is to understand how to scale your data for display.

Using the *BytScl* Command

The command used most often for byte scaling data is, naturally, named *BytScl*. (Older users who remember file names restricted to no more than eight characters can explain the name to younger users.)

Consider floating point data with a range outside the range available to bytes.

```
IDL> data = cgDemoData(2)
IDL> Print, Min(data), Max(data)
      0.000000      1550.00
```

BytScl simply scales this data into the range 0 to 255, then converts any floating point value to a byte by truncation.

```
IDL> byteData = BytScl(data)
IDL> Print, Min(byteData), Max(byteData)
      0      255
```

On a modern computer with a 24-bit graphics card, these two image display commands are equivalent.

```
IDL> TV, BytScl(data)
IDL> TVScl, data
```

But it is rare that you would ever use a *BytScl* command without using a few of its keywords. Consider, for example, that you want to compare an image you collected in August with an image you collected in November. You could load a color table and display the two images like this:

```
LoadCT, 5
TVScl, augImage
TVScl, novImage
```

But, what would you see? You would see both images scaled into the 256 colors of your color table. But suppose you loaded a red color at color index 200. What value in your image would the red color represent?

In truth, you don't know. It depends entirely on what sort of data range the two images had to start with, since whatever that data range was, it was scaled into 256 values. If the two images had *different* data ranges, which is most likely, since that is why you are comparing them, then the red color represents *different* values in the two images.

So, now, you are looking at two images and they are scaled differently so that the same color represents different values, or another way of saying this, the same values (assuming there is some overlap) are displayed in different colors in the two images. You can see how you might accidentally come to the wrong conclusions about what the data are telling you about the world!

To do this image comparison correctly, you cannot use *TVScl* to scale and display your data. You must use *BytScl* to scale the data, and you must make sure both image data sets are scaled to the same measuring stick. That is to say, the red color in the two images must represent the *same* value in both images.

The way to make sure you use the same measuring stick is to use the *Min* and *Max* keywords to the *BytScl* command. Any data value less than or equal to the value specified with the *Min* keyword is set equal to 0 in the result. Any data value greater than or equal to the value specified with the *Max* keyword is set equal to 255 in the result. (It is actually set equal to the value specified by the *Top* keyword, which we will discuss in a moment, but the *Top* keyword value is 255 if the *Top* keyword is not used in the command.)

So, to compare these two images, we want to make sure we scale the values into the same data range, which could be the largest data range that includes the ranges in the two images. The code might look like this.

```
minRange = Min(augImage) < Min(novImage)
maxRange = Max(augImage) > Max(novImage)
TV, BytScl(augImage, Min=minRange, Max=maxRange)
TV, BytScl(novImage, Min=minRange, Max=maxRange)
```

Now, you are comparing apples with apples and not apples with oranges.

Note: If you are not familiar with the “less than” and “greater than” operators (i.e., `<` and `>`), they work by comparing the two values on either side of the operator and returning the smaller (`< operator`) or greater (`> operator`) of the two values.

The *Min* and *Max* keywords do not have to be set to include the entire data range of the two images. They can be set to arbitrary values. The point is, they should be set to the *same* values for all images that are going to be compared.

```
TV, BytScl(augImage, Min=100, Max=1400)
TV, BytScl(novImage, Min=100, Max=1400)
```

Note that if the *Min* and *Max* keywords are not used, then it is as if you set the *Min* keyword to the minimum value of the data, and *Max* keyword to the maximum value of the data.

Restricting the Number of Values

Now, it is an unfortunate fact of life that even in the second decade of the 21st century, people are still writing IDL programs that only work using the indexed color model. (See “Understanding IDL Color Models” on page 36.) In my opinion, this is because the *TV* command is so impossibly difficult to use effectively, but more on that later. The point is, they are restricting themselves to 256 of the 16.7 million colors available to them.

This is not many colors, and it is restricted even more by needing some colors in most graphics programs to do things other than display beautiful images. Line plots need to be drawn, images need to be annotated, and so on. In other words, we usually need *drawing colors* in addition to the usual image colors.

If you find yourself in this situation, you will need to restrict the number of image colors to something less than 256. You will have to save a few indices for drawing colors. I typically save the indices at the top of the color table for drawing colors, although some people prefer drawing color indices at the bottom of the color table. In either case, you create the space for drawing colors by using the *Top* keyword in the *BytScl* command.

For example, if I have five drawing colors, and I want to install them in the color table at color indices 250 through 254 (I *never* install drawing colors in color indices 0 or 255), then I want to load image colors in indices 0 through 249. In other words, I want 250 image colors.

To scale the image correctly into 250 colors, I have to set the *Top* keyword to 249, or one less than the number of colors I want to use to display the image.

```
IDL> byteData = BytScl(data, Top=249)
IDL> Print, Min(byteData), Max(byteData)
0    249
```

If I want to load my drawing colors at the bottom of the color table index, then I want to load them in color indices 1 through 5, and I want my image to be scaled into the range 6 through 255. I simply add a byte value of 6 to the scaled data.

```
IDL> byteData = BytScl(data, Top=249) + 6B
IDL> Print, Min(byteData), Max(byteData)
6    255
```

Another reason for using the *Top* keyword is to partition your image values into a particular number of “colors”. Suppose you want to display your image data in only 16 colors, as you might if you were doing a filled con-

tour plot. Then, you would set the *Top* keyword to one less than the number of colors you want to use.

```
IDL> byteData = ByteScl(data, Top=15)
IDL> Print, Min(byteData), Max(byteData)
      0      15
```

Scaling Images with Missing Data

Many, perhaps most, images contain “missing data.” This is certainly true of the satellite image data, and is probably true of most data collected with instrumentation. The missing data is usually identified by a number that is meant to represent such data. For example, it is extremely common for satellite images to be stored as two-byte signed integers. The missing data is often identified in such image files as the value -32767.

This kind of satellite image data usually comes with “calibration values” that can convert the stored 16-bit signed integer data to the actual floating point science data you are interested in. These calibration values are often called a “scale” and an “offset”, or in some cases they might be called the “slope” and the “intercept.” The image data is multiplied by the *scale* and then the *offset* is added to the result. The *scale* is usually a floating point number, so the resulting science data is a floating point array.

```
IDL> scienceData = (image * scale) + offset
```

It is the science data that you want to byte scale according to your display needs.

But, of course, this data cannot be scaled properly with all those missing data values in there! So, you have to deal with those values before you scale the data. You could try looking for the missing data now, after the data has been converted to science data, with the *Where* function.

```
IDL> indices = Where(scienceData EQ -32767, count)
IDL> Print, count
      0
```

But, you won’t find any! The missing data values in the image were changed to floats along with the real science data values in the calibration process

Finding a floating point value with the *Where* function is *extremely* problematic, since most floating point numbers cannot be represented exactly on a computer. The image data file I have in front of me as I write has a *scale* of 2e-6 and an *offset* of 0.05, so the floating point number you would have to look for is this.

```
IDL> missingFloat = -32767 * 2e-6 + 0.05
```

```
IDL> Print, missingFloat  
-0.0155340
```

Although this number is one of the floating point numbers that *can* be represented exactly on a computer, and you could search for this value in the science data and find it with the *Where* function, most floating point values could not be found this way. Normally, you have to search for a number that falls into a small range of values (a *delta* value) centered on the number you are looking for. This is usually more trouble than it is worth.

So, the rule of thumb is this, you want to search for missing data *before* you do anything else to the data. In our hypothetical case, we would search for the missing data values in the original 16-bit signed integer data.

```
IDL> indices = Where(image EQ -32767, count)
```

What we want to do next is turn these missing data values in the image into missing data values we can handle in computations. In computations, we want these image values to have the bit pattern that represents Not-a-Number (NaN). This bit pattern is available in the IDL system variable *!Values* as the field *F_NAN* and *D_NAN*. The *F_NAN* is a floating point bit pattern and the *D_NAN* is a double precision floating point bit pattern.

We could try setting these values like this.

```
IDL> IF count GT 0 THEN image[indices] = !Values.F_NaN
```

But this will *not* work! The problem is that the *image* variable is not a floating point array. It is an integer array. The floating point NaN value will get truncated to an integer and we won't be able to recognize it later in image computations. It is *extremely* important, then, that you cast your image data to floating point values *before* you set the missing data to NaNs. For example, you could use commands like this.

```
IDL> image = Float(image)  
IDL> IF count GT 0 THEN image[indices] = !Values.F_NaN
```

If your image comes with calibration data, it is likely that the calibration data turns the image into floating point values, so you apply the indices to the scaled data instead of to the original data.

```
IDL> indices = Where(image EQ -32767, count)  
IDL> scienceData = (image * scale) + offset  
IDL> IF count GT 0 THEN $  
      scienceData[indices] = !Values.F_NaN
```

The point is, once you have converted your missing data to the proper floating point bit pattern (and your image is now a floating point array),

you are ready to scale the image. But, if you try something like this, it will not work.

```
IDL> scaledImage = BytScl(scienceData, TOP=15)
IDL> MaxMin, scaledImage
      NaN  NaN
```

All of your *scaledImage* values will appear to be NaNs! NaNs can be used in computations in IDL, but in general, you have to alert the computational commands you use to their presence. In this case, you do this by setting the *NaN* keyword to the *BytScl* command.

```
IDL> scaledImage = BytScl(scienceData, TOP=15, /NaN)
```

Note: Not all versions of IDL and not all machine architectures display the same results when you use a NaN in the computation. For example, on a Windows machine, running IDL 8, computations with variables containing NaNs usually work as expected, although they throw warning messages about the program causing an arithmetic error (a floating illegal operand).

There is a difference of opinion in IDL programming circles as to whether or not the *BytScl* command (as well as other commands, such as *Min*, *Max*, etc.) should always check for NaNs, or should only check at the request of the programmer. Many programmers like the fact that the program will throw errors if unexpected NaNs show up in their data. I generally like it, too. But if I am writing programs for others to use and I am unsure about the kind of data the programs will be used with, I typically always set the *NaN* keyword on computational routines. That way they will do the right thing when NaN values show up in the data.

Missing Data and Image Colors

Often if you have missing data in an image, you would like to display the missing data in a color that is completely different from the colors you are using to display the actual image data with. I think of this “missing data color” as an annotation color or drawing color, and I load this color at the top of my color table and scale my image colors appropriately.

For example, suppose I want to display an image with 250 colors, using the Blue-Red color table (color table index 33), and I want to use a dark gray color for the missing data. I might load the colors and display the image like this. In this image, the missing values are represented by the value 0.

```
IDL> image = cgDemoData(5)
IDL> indices = Where(image EQ 0, count)
IDL> image = Float(image)
IDL> IF count GT 0 THEN image[indices] = !Values.F_NaN
```

```
IDL> scaledImage = BytScl(image, TOP=249, /NaN)
IDL> LoadCT, 33, /Silent, NColors=250
IDL> TVLCT, 110, 110, 110, 254
IDL> IF count GT 0 THEN scaledImage[indices] = 254
IDL> TV, scaledImage
```

You see an example of what the image looks like in Figure 4.

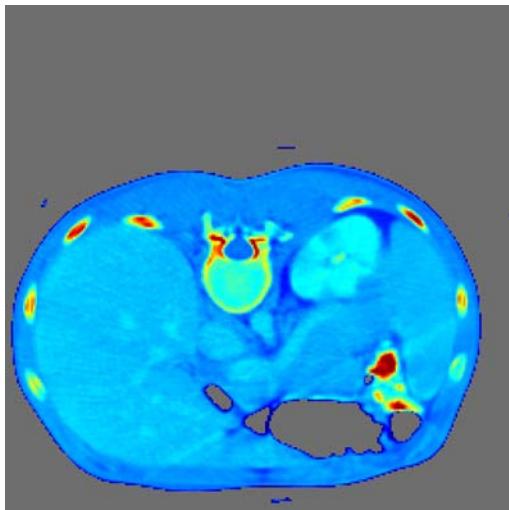


Figure 4: Missing data is set to a dark gray color in the display of this image.

Is the Image Upside-Down?

The image shown in Figure 4 of a slice though a human chest cavity is actually presented upside-down from the point of view of a physician. In fact, you will find that most images you read into IDL that were created with other software will appear upside-down to you when they are displayed in IDL. This is because IDL is one of the few image display programs that use the convention of locating the (0,0) pixel of the image in the lower-left corner of the display. The vast majority of image display software uses the convention of locating the (0,0) pixel of the image in the upper-left corner of the display.

This convention can be changed in IDL by setting the *!Order* system variable to 1, to change the convention for all images subsequently displayed

in that IDL session. Or, by setting the *Order* keyword on a *TV* or *TVSc* command, to change the convention for just that one command.

This is fine for *displaying* images, but it is a disaster for *interacting* with images! That is to say, if you ever want to use your cursor to locate a position in an image, or to find the value of an image at a particular location in an image, or even to draw something on top of an image, changing the display order with *!Order*, or with the *Order* keyword, can cause great confusion. This is because in the pixel space of an IDL graphics window, the (0,0) pixel is always in the lower-left corner, no matter what the value of *!Order*.

In my own programs, I *never* change *!Order* or use an *Order* keyword. If an upside-down image needs to be displayed right-side-up, I always reverse the actual data in the image with the *Reverse* command. This must be done immediately after reading the data, to avoid other problems. The code above should be modified like this. The “2” in the second parameter of the *Reverse* command indicates that it is the “second” or Y dimension that should be reversed.

```
IDL> image = cgDemoData(5)
IDL> image = Reverse(image, 2)
IDL> indices = Where(image EQ 0, count)
IDL> image = Float(image)
IDL> IF count GT 0 THEN image[indices] = !Values.F_NaN
IDL> scaledImage = BytScl(image, TOP=249, /NaN)
IDL> LoadCT, 33, /Silent, NColors=250
IDL> TVLCT, 110, 110, 110, 254
IDL> IF count GT 0 THEN scaledImage[indices] = 254
IDL> TV, scaledImage
```

You see the result in Figure 5.

This avoids confusion by aligning the positive Y direction of the coordinate system of the graphics window with the positive Y direction of the image. It also avoids the use of the *!Order* system variable, which is always a danger, in my opinion, since any code you write to interact with images will have to take this variable’s value into account. Life is much simpler if you assume the *!Order* value is 0 and simply reverse the image Y direction when required.

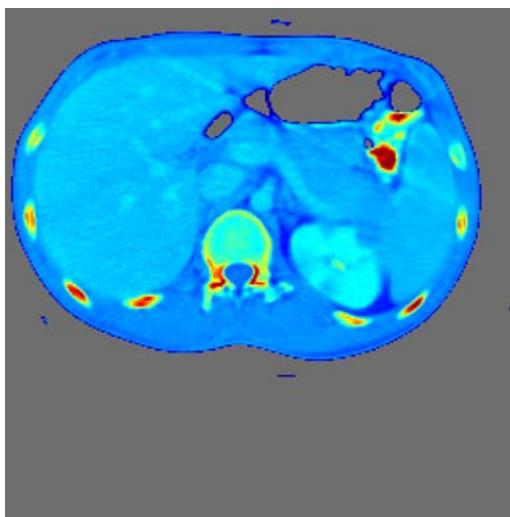


Figure 5: Rather than using `!Order` to change the $(0,0)$ pixel convention, you will have fewer problems with image interactions if you reverse the Y dimension of the image data with the `Reverse` command.

Displaying Images in Color

In truth, if you typed the commands above, it is possible that you are *not* looking at a color image, as shown in Figure 5, above. It is possible, indeed likely, that you are looking at a grayscale image on your display.

The `TV` command is perhaps the oldest of IDL's traditional graphics commands and, sadly, it hasn't undergone much updating in the past 25 years. One way it is deficient for displaying images on modern computers is that it only displays 2D images in color if the user has chosen to use the indexed color model.

As a result, you see a lot of `Device, Decomposed=0` commands in IDL user's start-up files. This, of course, forces the IDL user to choose only 256 colors from a palette of 16.7 million possible colors and greatly limits the possibilities of what can be accomplished with traditional graphics commands in IDL. This alone is enough to choose an alternative image display command, but, as you will see in a moment, the situation is even worse than this.

The bottom line is this. If you want to use the *TV* command to display a 2D image, then you need to use the indexed color model when you display the image. To do this correctly, all the time, requires a minimum of four IDL commands.

```
IDL> Device, Get_Decomposed=currentMode
IDL> Device, Decomposed=0
IDL> TV, scaledImage
IDL> Device, Decomposed=currentMode
```

This, of course, is why no one ever does it this way! Faced with this gruesome reality, most IDL users who want to use the *TV* command put themselves into indexed color mode (usually in their start-up files) and live with the unfortunate consequence of living in a time warp where they are trapped in the 1970s.

One of the ways being in indexed color mode causes problems is demonstrated by wanting to display a 24-bit image while in this mode. A 24-bit image is also called a “true-color image,” because in theory all 16.7 million colors are available all at once. Consider the rose image in the IDL distribution.

```
IDL> rose = cgDemoData(16)
IDL> Help, rose
ROSE   BYTE = Array[3, 227, 149]
```

A true-color image always has a 3 as the size of one of its dimensions. (Or a 4, if there is an alpha channel, but let's set this complication aside for now.) This indicates there are three image planes, each, if you like, a byte scaled 2D image in its own right. The three planes represent the red, green, and blue values that make up a color triple for specifying the color of a pixel. Because true-color images can express the red, green, and blue values independently, all 16.7 million colors are possible for each pixel in the image.

When the 3 is in the first position, as here, the image is said to be *pixel interleaved*. If the 3 is in the second position, the image is said to be *row interleaved*. And if the 3 is in the last position, the image is said to be *band interleaved*. When you display a 24-bit image with the *TV* command, you must know how the image is interleaved, so you can set the *True* keyword to 1, 2, or 3, respectively. In this case, the *True* keyword is set to 1.

```
IDL> TV, rose, True=1
```

Now, true-color images express their colors directly, there is no need to pass the image values through a color table to “look up” a display color. And yet, in many versions of IDL, this is exactly what happens when the

user is in the indexed color mode and wants to display a true-color image. The true-color image values are incorrectly routed through the color table to produce an image that is, well, weird, to say the least. You see an example of this in Figure 6, where the correct image is displayed in the top of the figure and a weirdly colored image is displayed below it.



Figure 6: The rose image displayed according to the colors in the image in the top figure, and in the wrong colors in some versions of IDL, and on some machine architectures in the bottom figure.

Routing true-color images through the color tables when in indexed color mode was a problem for almost all users in IDL version 5. It was corrected for Unix computers in IDL version 6. But the problem persisted in Windows computers until IDL version 7.1.

There are two ways to fix the problem if you insist on using the `TV` command. First, you can put yourself into decomposed color mode before you display a 24-bit color image.

```
IDL> Device, Get_Decomposed=currentMode  
IDL> Device, Decomposed=1  
IDL> TV, rose, True=1  
IDL> Device, Decomposed=currentMode
```

The problem with this solution is that it doesn't work with 8-bit devices like the PostScript device in versions of IDL prior to IDL 7.1. In other words, the PostScript device (and other 8-bit devices like older versions of the Z-graphics buffer) don't have a *Decomposed* keyword that can be set to 1 or any other number!

If you try to display a true-color image in PostScript, when you have a color table loaded, on those machines and in those versions of IDL that incorrectly route the image through the color tables, then you always end up with a weird image, as in the bottom of Figure 6. As you can imagine, it is hard to write device independent graphics programs under these conditions.

The other solution is to stay in indexed color mode to display the image, but to load a gray scale color table first, so that when the image is incorrectly routed through the color table vectors, the return values are the same as they were in the original image. This works for all machines in all versions of IDL, and in 8-bit devices like the PostScript device, but it changes the color table vectors, of course. So, you have to remember to set the color table back to its original values if you don't want to deal with color table problems. The code looks something like this.

```
IDL> Device, Get_Decomposed=currentMode
IDL> Device, Decomposed=0
IDL> TVLCT, r, g, b, /Get
IDL> LoadCT, 0, /Silent
IDL> TV, rose, True=1
IDL> TVLCT, r, g, b
IDL> Device, Decomposed=currentMode
```

Of course, even this solution leaves out the problem of having to figure out from the image how it is interleaved. That adds another dozen or so lines of code to your program if you want to do this correctly for all images on all machines and in all devices running IDL.

Here, for example, is the kind of code you might write (without error handling) if you are writing a program to display either a 2D or a true-color image with the *TV* command on your display.

```
ndims = Size(image, /N_Dimensions)
CASE ndims OF
  2: BEGIN
    true = 0
    Device, Get_Decomposed=currentMode
    Device, Decomposed=0
```

```
TV, image, True=true  
Device, Decomposed=currentMode  
END  
3: BEGIN  
  s = Size(image, /Dimensions)  
  index = Where(s EQ 3)  
  true = index[0] + 1  
  Device, Get_Decomposed=currentMode  
  Device, Decomposed=1  
  TVKCT, r, g, b, /GET  
  LoadCT, 0, /Silent  
  TV, image, True=true  
  TVLCT, r, g, b  
  Device, Decomposed=currentMode  
  END  
ENDCASE
```

Some true-color images, unfortunately, have alpha channels, so the “3” index is actually a “4”, but that’s a complication it probably doesn’t pay to go into here. The *TV* command wouldn’t display an alpha channel image properly in any case.

Basically, it takes on the order of about 25 lines of code to display an image properly with the *TV* command. And that is *only* if you want to get the colors right. If you want the image to show up in the right place in a PostScript file, or if you want to do other smart things, and believe me you do, you can multiply this number by a factor of 10 or so to get the number of commands you really need.

Of course, no one writes code like this. Most people just take their chances and hope they can at least get their programs to run on *their* computer. They live in fear they will be asked to run their programs on someone else’s computer. For example, the one used by their supervisor always seems to cause problems.

You can understand, then, why many IDL programmers are looking for better alternatives to the *TV* command.

Resizing and Positioning Images

In order to see a representation of image data on a display, most images typically have to be resized. They are either too big for the display, or too small to be seen easily. But, the *TV* command and *TVScl* commands do not resize anything. They simply take an image argument and blast its image pixels into the display window in a one-to-one fashion. Each pixel in the

image fills one pixel on the display. If the display window isn't at least as large as the image, then portions of the image will fall outside the display window and will not be seen. You will receive no warning or error message indicating this is what has happened.

In practical terms, this means that images larger than the resolution of your monitor must be resized to be displayed. It is frightening, in this era of high resolution CCD cameras and satellite sensing instruments, how large an image can be. It is often larger than your monitor. Adding to the problem is the fact that we often prefer to work with graphics windows smaller than our monitor, just so we can find our way around our desktop and get some work done.

Sometimes, we have the opposite problem. The image is so small, say 32x32, we have difficulty seeing the information in it.

In either case, the solution is to resize the image to a comfortable size before we display it. Here "comfortable size" typically means something about a quarter of the size of our display monitor, or perhaps a bit smaller.

Note: If you currently use an alternative TV command, or you plan to, you probably don't want to slog through this section of the book. All alternative TV commands I know about resize images on the display and in PostScript files automatically for you using the more natural Position keyword, which is used with all other traditional graphics commands. You only really need to read this section of the book if you want to continue to use TV and TVScl or you feel compelled to become an IDL expert. Otherwise, just skip ahead to the next section on page 241.

There are two commands you can use to resize image arrays in IDL: *Rebin* (a built-in command) and *Congrid* (a user library command, written in the IDL language).

Rebin is limited in that the resized image that returns as a result of the function must have size that is an integer factor of the size of the input image. In other words, you can make the output image twice the size of the input image, but you can't make it 1.5 times the size of the input image. The image can be reduced in size in one direction and expanded in size in another direction.

Consider this picture of David Stern, the person who wrote the first version of IDL in his attic office. We will use *Rebin* to make the image twice its original 192x192 size. By default, *Rebin* uses bilinear interpolation when it makes an image dimension larger, and nearest neighbor sampling when it makes an image dimension smaller.

Note: I am going to assume that everyone is using a decomposed color model in this section of the book, and that the !P.Background system variable has been set to a white color. This way, everyone can erase whatever is in the graphics window by simply typing the Erase command. Recall that image display commands do not, in general, erase the window before the image is displayed. They just transfer pixels to the display. To clear a window for the display of an image, you can type these commands. Afterward, you can just type “Erase”.

```
IDL> Device, Decomposed=1  
IDL> !P.Background = cgColor('white')  
IDL> Window, XSize=640, YSize=512  
IDL> Erase
```

Bilinear interpolation can cause the output image to have values that do not exist in the input image. Nearest neighbor sampling never creates image data values that don’t exist in the original image. To turn bilinear interpolation off for *Rebin* and force it to always use nearest neighbor sampling, you must set the *Sample* keyword.

```
IDL> stern = (cgDemoData(10)) [*,* ,1]  
IDL> s = Size(stern, /Dimensions)  
IDL> big_stern = Rebin(stern, 2*s[0] , 2*s[1])  
IDL> Help, big_stern  
BIG_STERN BYTE = Array [384, 384]  
IDL> TV, big_stern
```

You see the result in Figure 7. Note that the 384x384 image takes up just a portion of the 640x512 graphics window. The *TV* command simply transfers image pixels to the display in a one-to-one fashion, starting at the lower-left corner of the display.

If you want the image to fill the graphics window, you will have to use the *Congrid* command, which can make images of any size. Here we use the *!D* system variable to get the size of the current graphics window. (These window size values are not valid on UNIX machines until *after* a window is opened in the IDL session, an important point to remember if you are writing image display routines!)

```
IDL> big_stern = Congrid(stern, !D.X_Size, !D.Y_Size)  
IDL> TV, big_stern
```

You see the result in Figure 8. If you look closely, you can see the result looks a little “pixelated.” This is because *Congrid*, by default, always does nearest neighbor resampling, essentially replicating image pixels as needed. If you want to do bilinear interpolation with *Congrid*, you must set the *Interp* keyword.



Figure 7: The 384x384 image of David Stern, the programmer who created IDL, in a 640x512 graphics window. The TV command simply transfers image pixels to display pixels in a one-to-one fashion.

To more vividly show the difference between bilinear and nearest neighbor sampling, a blow-up of just a portion of this image is shown in Figure 9. The image on the left is displayed with bilinear interpolation and the image on the right is displayed with nearest neighbor sampling.

Notice, that the aspect ratio of the image in Figure 8 has changed as a result of the resizing. If you want to preserve the aspect ratio of the image and put it in the window in a way that would allow you to, say, put a set of axes on the image, then you have to go to a great deal more trouble.

Suppose we want to make a 400x400 image and center it in the window. We can write commands like this, assuming that the dimensions of the graphics window are larger than the proposed size of the image.

```
IDL> big_stern = Congrid(stern, 400, 400, /Interp)
IDL> xstart = (!D.X_Size - 400) / 2
IDL> ystart = (!D.Y_Size - 400) / 2
IDL> TV, big_stern, xstart, ystart
```

You see the result in Figure 10.



Figure 8: Congrid allows the image to be resized in arbitrary ways. Note that the aspect ratio of the image has changed in this 640x512 window.

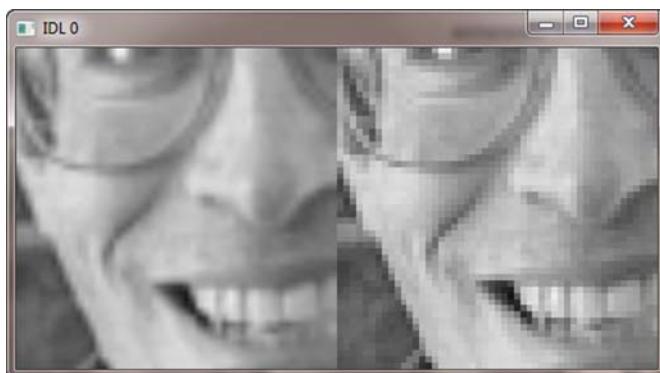


Figure 9: A comparison of a small portion of the image displayed with bilinear interpolation on the left and nearest neighbor sampling on the right. Note the more pixelated appearance of the image on the right.

But, where is this image located in the window? To put axes on the image, we have to use something like the *Plot* or *Contour* command, and that



Figure 10: Centering a large-as-possible image in a window requires computations based on the window size and aspect ratio of the image.

requires a position in the window in normalized coordinates. We will have to calculate what they are.

```
IDL> xloc = [xstart, xstart+400] / Float(!D.X_Size)
IDL> yloc = [ystart, ystart+400] / Float(!D.Y_Size)
IDL> position = [xloc[0], yloc[0], xloc[1], yloc[1]]
IDL> Print, position
      0.187500      0.109375      0.812500      0.890625
```

Note how the window size variables were cast to floats. If this isn't done, the arithmetic will be done with integers and the calculated locations will all be zero!

Now we are ready to draw axes in the image.

```
IDL> Plot, [0,1], /NoData, /NoErase, $
      Color=cgColor('black'), XRange=[0,192], $
      YRange=[0,192], XStyle=1, YStyle=1, $
      XTickLen=-0.035, YTickLen=-0.035, $
      Position=position
```

You see the result in Figure 11.

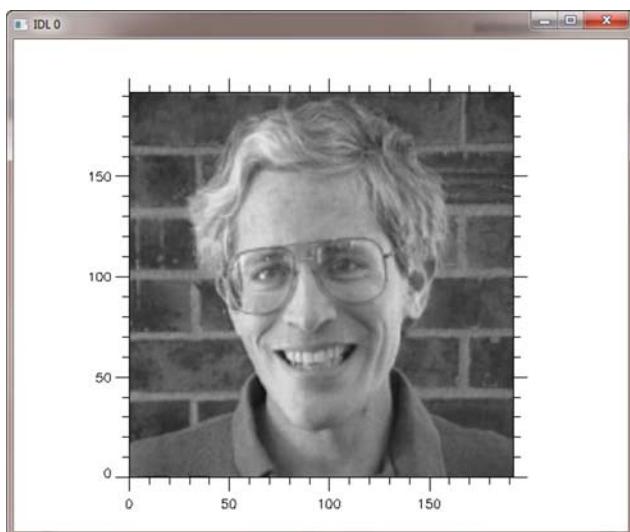


Figure 11: Axes are added to the image after calculating where the image is located in the graphics window.

Precise Resizing of Images

The *Congrid* command does a reasonably good job of resizing images, especially if the images have plenty of pixels. But sometimes images don't have plenty of pixels and you have to be a lot more careful about how you resize an image. For example, astronomers are often faced with having to work with images that don't have many pixels of interest.

The *Congrid* command assumes that a pixel's location is associated with the lower left corner of the pixel. Astronomers and others who care about precise positioning of image pixels prefer that the pixel's location be associated with the center of the pixel. *Congrid* will accommodate this convention by adding an offset of half a pixel width to each pixel's location if the *Center* keyword is set. This ensures the center of the image is still the center of the image after resizing. For two-dimensional images, the *Center* keyword is used only when interpolating. It is ignored when doing nearest neighbor sampling.

In addition, *Congrid*, under some conditions, will extrapolate an extra column or row to the output image. This, again, is dangerous when you are working with small images with few columns and rows. To prevent this from happening, you can set the *Minus_One* keyword. Be careful with this, though. The *Center* and *Minus_One* keywords were created to solve

the same problem, although they use slightly different algorithms to do so. The consensus among astronomers I talk to is that the *Center* keyword is the preferred method to solve this problem. Do not use both keywords together.

This means that to obtain precise resizing of an image with *Congrid*, you must set the *Center* keyword.

```
IDL> big_stern = Congrid(stern, 800, 800, /Center, $  
/Interp)
```

Precise Sizing of True-Color Images

Unfortunately, the discussion above applies only to two-dimensional images. *Congrid* has additional problems when precisely resizing true-color images. Recall that true-color images are 3D images in which the size of one of the dimensions is a three. Let's call this the "true-color dimension."

Congrid does not have the ability to tell the difference between a 3D array and a true-color image with a true-color dimension. As a result, all three dimensions of a 3D array are interpolated to the output size in the same way. This is incorrect for true-color images, as the true-color dimension should not be interpolated at all. (You can think of it as a placeholder, if you like.)

Again, this doesn't cause many problems in most images, because the colors are only a half pixel off and over an entire image it is not noticeable. But, it can be a factor in images with few pixels.

Before I show you how these problems can be corrected, I want to point out just one additional problem with *Congrid*. Three dimensional arrays are *always* interpolated with bilinear interpolation, no matter how the *Interp* keyword is set. This kind of interpolation, of course, can put values in the output image that don't exist in the input image. Astronomers and other scientists who work with satellite data almost always prefer that their image pixels be "pure" in the sense that their values always exist in the input image.

These precision resizing problems can be solved by using the [Coyote Library](#) program [FSC_Resize_Image](#). This program sends 2D images directly to *Congrid*, but with the *Center* keyword set by default. True-color images are treated as if the *Center* keyword is set, but they do not interpolate the true-color dimension of the true-color image. Finally, the program honors the *Interp* keyword, and performs nearest neighbor resampling of true-color images if this keyword is not set.

The [Coyote Library](#) alternative *TV* commands `cgImage` and `TVScale` resize all their images with `FSC_Resize_Image` before display.

Resizing Images in PostScript

None of this is too terribly difficult. The only real problem is that programs written this way only work on the display. They don't work at all when you are trying to create a PostScript file, because images are sized in a *completely* different way in a PostScript file.

This is a *huge* problem, because creating PostScript files, as you will see later in this book, is your only path to producing professional looking graphical output for presentations, web pages, and journal articles using traditional IDL graphics commands. *Everything* depends upon it! If your graphics programs don't work exactly the same on the display and in a PostScript file, or if you have to write different code for each device you want to display graphics on, then life is considerably more cumbersome than it has to be.

The basic problem is this. The PostScript device is a high-resolution device. On a Windows machine, for example, the monitor has a resolution of 72 pixels per inch (28.3 pixels per centimeter), whereas the PostScript device has a resolution of 1000 pixels per inch (394 pixels per centimeter). What this means is that if you try to resize an image in the PostScript device using pixel or device coordinates, the image will be something like 100 times larger than it will be on your display! Believe me, you will not be able to allocate enough contiguous memory in your IDL process to create an image that large.

Fortunately, there is no need to. PostScript is a device that has *scalable pixels*. This means, essentially, that a pixel can assume any rectangular shape it likes and is not confined to a fixed shape and size, as it is on the monitor. You can simply tell the PostScript device what size you want the image to be, and it will stretch the image pixels to be that size. There is no need to involve either the *Rebin* or *Congrid* commands.

The "stretching" is done with the *XSize* and *YSize* keywords to the *TV* command. The only difficulty here is that the "units" of image size are not pixels, as they are when you are talking about displaying an image on the monitor, they are "inches" or "centimeters," (the default unit).

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> TV, stern, XSize=4, YSize=4, /Inches
IDL> Set_Plot, thisDevice
```

Oh, dear! Suddenly, the problem of displaying an image in a PostScript file with a set of axes around it has gotten *considerably* more difficult. Not to mention the problem of writing a program that can be displayed on the monitor *and* in a PostScript file. Now you are going to have to calculate the position of the image in units of inches (or centimeters), which means you are going to have to know the “size” of the PostScript “window” in these units, which means...

Most IDL programmers give up right here. Too much work! Why go to all this trouble every time you have to write a graphics program that displays an image with the *TV* command? In truth, no one does. (Have I mentioned I haven’t used a *TV* command in an IDL program in over 15 years?)

IDL programmers generally fall into two camps. In the first camp are those who insist on using the *TV* command in their graphics programs. They struggle, struggle, struggle to write graphics programs with images and have probably never written a device independent graphics program with images in their entire career. In the other camp are those who have found an alternative image display command, who never think about these issues any more, and who work with images as naturally as they do line, contour, and surface plots. They have forgotten there was a difference between programs written for the display and programs that are written for a PostScript file. They write one program, and it works everywhere.

Which camp would you prefer to be in?

Yes, okay, me, too. So I am going to abandon any pretense that you want to use the *TV* command, and I am going to show you how easy it is to work with images in IDL if you have the right tools.

Alternative Image Display Commands

There are a number of alternative image display commands available on the Internet. They have various features that make them excellent alternatives to the *TV* command, and you will have to decide which you like best. But any one of them will display either a 2D or true-color image with the proper colors in a machine independent and device independent fashion. And, they will allow you to resize and position images in graphics windows, including PostScript windows, in a natural way.

I highly recommend two alternatives in particular. Liam Gumley has written an excellent *TV* alternative command named *ImDisp*. And the alternative command you will learn quite a lot about in this book is named

`cgImage`. It is from the [Coyote Library](#). `ImDisp` and `cgImage` are much alike in terms of features. You can find `ImDisp` on Liam's web page, <http://www.gumley.com/PIP/Programs/imdisp.pro>. Liam has also written an excellent book on IDL programming, called *Practical IDL Programming*, that you will find well worth reading.

Among the features `cgImage` and `ImDisp` have in common, and which are not found in the `TV` command, are these.

- The commands support and work identically on Windows, UNIX, and Macintosh computers, and in PostScript and the Z-Graphics buffer devices.
- 8-bit (e.g., PostScript) and 24-bit graphics devices are handled automatically. There is no need to know the interleaving property of true-color images.
- 2D and true-color images are always displayed in their proper colors on all devices and in all versions of IDL.
- The image is automatically sized to fit the display.
- The commands have provision for preserving the aspect ratio of the image.
- Images can be positioned on the display with a *Position* keyword, as other IDL graphics commands are. This facilitates using images with other graphics commands.
- The commands honor the *!P.Multi* system variable, as all other IDL graphics commands do, except the `TV` and `TVScl` commands.
- Resizing of images by nearest neighbor or bilinear interpolation methods is supported.
- Images can be byte scaled on the fly, using “normal” image scaling keywords: *MinValue*, *MaxValue*, *Top*, *Bottom*, and *NColors*.
- The commands can erase the display like other IDL graphics commands (usually via an *Erase* keyword).
- The commands honor a background color when erasing the display.
- Axes can automatically be drawn on the image, setting up a data coordinate space for the image.

There are a few minor differences between the two commands. For example, `ImDisp` can display “negative” images without reversing the color table. The `cgImage` command has more micro-controls for controlling how images are positioned with *!P.Multi* (room can be saved for image

titles or color bars, for example), and it can display images with alpha channels, automatically doing the required blending with the background image.

You will learn more about these features, and their importance in writing graphics programs, later in this chapter.

Positioning and Resizing Images

Nearly all IDL traditional graphics commands use a *Position* keyword to locate the graphics output in a display window. The *TV* and *TVScl* commands are the only exceptions. With other commands, the position is given as a four-element array in normalized coordinates (values that go from 0 to 1 in the display window). The x and y values of the lower-left hand corner of the position rectangle are given, followed by the x and y values of the upper-right hand corner of the position rectangle.

For example, a line plot is positioned with code like this.

```
IDL> Plot, Findgen(11), Position=[0.15, 0.15, 0.9, 0.9]
```

Alternative image display commands allow you to do the same thing with images. All the image resizing and positioning is done in a device and machine independent manner. This command works identically both on the display and in a PostScript file.

```
IDL> cgImage, stern, Position=[0.15, 0.15, 0.9, 0.9]
```

You see the result in Figure 12.

Naturally, the position in a graphics window, if specified with normalized coordinates, depends entirely on the size and shape of the graphics window. If you have a long skinny window, you can get a long, skinny image, as shown in Figure 13.

For this reason, among others, most alternative image display commands have provisions for maintaining the true aspect ratio of the image. In the case of *cgImage*, the *Keep_Aspect* keyword needs to be set. The image position, then, becomes the *starting position* in the window. The program calculates the largest image that can be fit into this starting position, while still maintaining the image's aspect ratio, and then centers the image in the starting position space.

This means, of course, that the *actual* position in the window might be different from the specified position in the window. (The actual position in the window is where the image finally ends up.) When you are working with images it is often essential to know where they are located in the win-



Figure 12: Alternative TV commands allow you to resize and position images in a display window using the **Position** keyword, like other IDL graphics commands.



Figure 13: Specifying the image position with a **Position** keyword can result in images stretched in strange ways in some graphics windows.

dow after they are displayed. For example, you might want to fit a color bar directly above an image and extending over the entire width of an image.

How you obtain the actual position in the window varies a bit among alternative TV commands, but with [cgImage](#) the value passed to the program via the **Position** keyword can be both an input and an output parameter. (If you prefer, the output position can also be obtained with the output

keyword *OPosition*.) This assumes, of course, that the *position* parameter is passed into the program as a variable and not as an expression, as in the previous command. Here is what the code looks like to both specify the starting position in the window, keep the image aspect ratio, and obtain the actual position where the image ended up.

```
IDL> position = [0.15, 0.15, 0.9, 0.9]
IDL> cgImage, stern, Position=position, /Keep_Aspect
IDL> Print, position
0.425000      0.150000      0.625000      0.900000
```

You see the result in Figure 14.



Figure 14: Here the position is the starting position, and a new position, the actual position, is calculated to indicate where the image ends up in the graphics window.

Note how much different the input position is from the output position. But now, the *position* parameter identifies exactly where the image is in the window, so this position can be used, for example, to draw axes on the image. Simply use the *position* variable that is returned from `cgImage` as the input position variable for the *Plot* command.

```
IDL> Plot, [0,1], /NoData, /NoErase, $
    Color=cgColor('black'), XRange=[0,192], $
    YRange=[0,192], XStyle=1, YStyle=1, $
    XTickLen=-0.035, YTickLen=-0.035, $
    Position=position
```

In fact, there is no need to use the *Plot* command for this purpose, since one of the features of almost all *TV* alternative commands is to automatically draw axes on images. With the `cgiImage` command, you simply set the *Axes* keyword. The data range for the axes can be set with the *XRange* and *YRange* keywords. *Plot* keywords that will apply to the axis, such as *[XY]TickLen* in the command above can be passed in with an *AxKeywords* structure parameter. The *Margin* keyword in this command puts a margin of 1/10th of the window size around the image.

```
IDL> cgImage, stern, Margin=0.1, /Keep_Axes, /Axes, $  
      XRange=[-10,10], YRange=[0,50], Color='navy', $  
      AxKeywords={XTickLen:-0.035, YTickLen:-0.035}
```

Once you have a data coordinate space set up, you can simply draw graphics in that space. For example, if you want to draw a box around the mouth in the image, you can see immediately from the image display that the X data coordinate will range from about -3 to 2.25 and the Y data coordinate will range from 13 to 18.

```
IDL> Plots, [-3.00, -3.00, 2.25, 2.25, -3.00], $  
      [13.00, 18.00, 18.00, 13.00, 13.00], $  
      Color=cgColor('red'), Thick=2
```

You see the result in Figure 15. Note that the image “data” coordinates will apply no matter where the image is displayed in the graphics window.

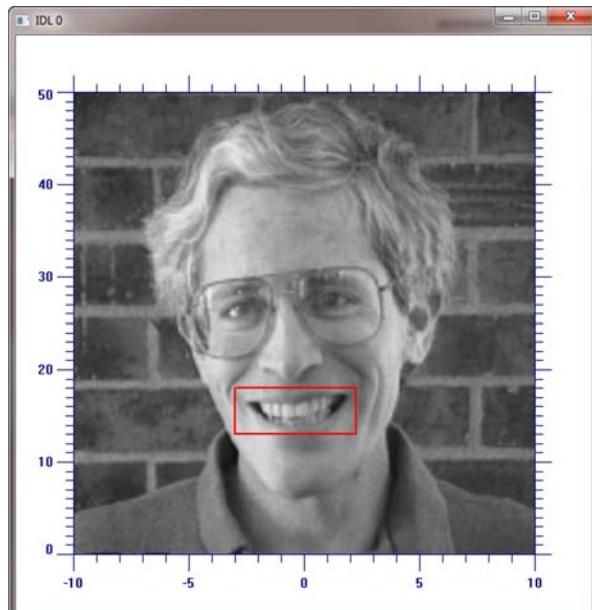


Figure 15: The `cgImage` command can display axes and set up a data coordinate system for displaying graphics on top of images.

Erasing the Graphics Window Before Display

The `TV` and `TVScl` commands do not erase the current graphics display before displaying the image. They simply put the pixels on top of whatever

is currently in the display. This is different from a *Plot* or *Contour* command, for example, and requires the user to either create a new graphics window before displaying an image, or to issue the *Erase* command before displaying an image. Often, especially if you are trying to write programs that work the same on the display and in the PostScript file, you prefer a white background for your images. This requires that you use the *Color* keyword to the *Erase* command.

```
IDL> Erase, Color=cgColor('white')
IDL> TV, stern
```

The *Erase* command does not work in a color model independent way. In fact, if used in indexed color mode, it will “dirty” the current color table. You may prefer to use the `cgErase` command, which does work in a color independent way, and also allows colors to be specified by name.

```
IDL> cgErase, 'white'
```

Alternative *TV* commands usually have the ability to erase the current graphics window before they display the image. The `cgImage` command, for example, sets the *Erase* keyword to 1 and the *Background* keyword to “white” automatically.

```
IDL> cgImage, stern, Margin=0.1
```

Image Backgrounds

Another nice feature of `cgImage`, not found in other alternative *TV* commands, is that the background color is honored on your display *and* in your PostScript file. (Most alternative *TV* commands allow any color you like for your PostScript background, as long as it is white.)

This command produces a wheat-colored image background both on your display and in your PostScript file. Note that by setting the axes color to “opposite,” `cgColor` will calculate a color that contrasts with (or is “opposite”) the background color

```
IDL> cgImage, stern, Background='wheat', Margin=0.15, $
      /Axes, Color='opposite'
```

You see the result in Figure 16.

Displaying Images with Colors

Normally, if you want to display a 2D image or array with colors, you load a color table into the hardware color table and display the image. With a 2D image, `cgImage` will put itself into the indexed color mode and use the colors in the hardware color table to create the colors for the image. This works well as long as the colors in the color table have not been contaminated by other programs or by loading drawing colors, for example. It is

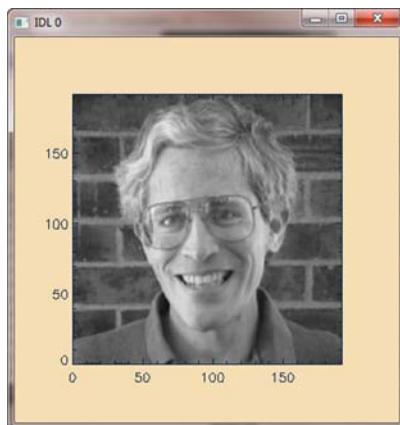


Figure 16: The `cgImage` command will produce the same background color both on the display and in a PostScript file.

always a good idea to load the image display color table immediately before displaying the image in the graphics window to avoid problems.

Another way to solve this problem is to load the color table vectors directly into the `cgImage` program. This way, you never have to worry about your color table being contaminated, since `cgImage` won't use it. It will load the color table vectors supplied to it immediately before it displays the image, and then re-load the old color table vectors after it has displayed the image.

The way to pass `cgImage` color table vectors is with the *Palette* keyword. The palette variable should be an N by 3 or a 3 by N byte array containing the red, green, and blue color vectors you want to use for the image display. One way to easily obtain the needed color palette variable is to use the *RGB_Table* keyword to `cgLoadCT`. When this keyword is used, `cgLoadCT` doesn't load any color vectors, it simply packages the vectors up into a 256 by 3 array that can be loaded into the hardware color table with *TVLCT* or used as a color palette for `cgImage` and other Coyote Graphics System programs.

```
IDL> cgLoadCT, 13, /Brewer, /Reverse, RGB_Table=palette
IDL> cgImage, stern, Background='wheat', Margin=0.15, $
   /Axes, Color='opposite', Palette=palette
```

You see the result in Figure 17.

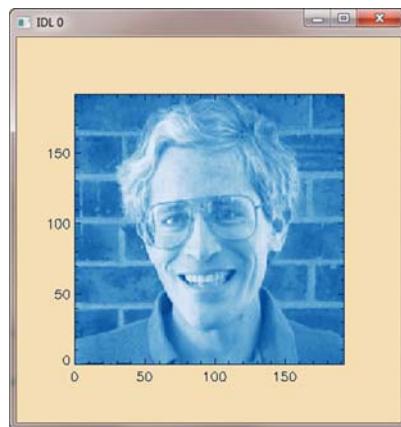


Figure 17: `cgImage` can manage its own color table vectors if the vectors are passed into the program with the `Palette` keyword. This avoids color table contamination problems caused by all programs having to use the same hardware color table.

Combining Images with Other Graphical Displays

The `cgImage` command makes it easy to combine images with other graphical displays. For example, an image can be displayed with contour lines over-plotted on top of the image, like this.

```
IDL> image = cgDemoData(18)
IDL> cgLoadCT, 4, /Brewer, /Reverse
IDL> cgImage, image, /Save, /Axes, /Keep_Aspect
IDL> cgContour, image, NLevels=10, Label=2, /Overplot
```

You see the result in Figure 18.

Interacting with Images

Normally, when we interact with images we put the image into a widget program (a program with a graphical user interface, or GUI) so we can track and handle cursor and button events. But `cgImage` has the functionality to offer a bit of interactivity to the user, even in a normal IDL graphics window. It does this by way of another [Coyote Library](#) program named `cgImageInfo`.

For example, if you have just typed the commands above, you see an image in the display window. Now, if you type the `cgImageInfo` command, IDL will be waiting for you to click the left mouse button inside the current graphics window. If you click inside the image, you will see printed out in

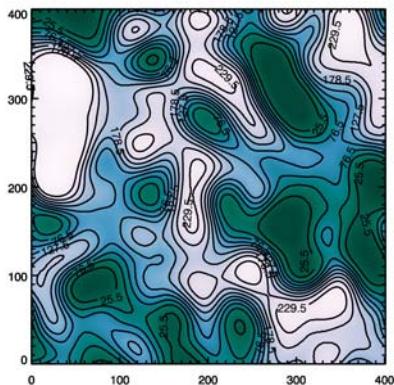


Figure 18: `cgImage` makes it easy to work with other graphics programs in the same display.

the IDL command log the pixel location in the image where you clicked, and the value of the image at that location. You can click in the image multiple times. When you are finished with the program, right click in the graphics window to exit `cgImageInfo`.

```
IDL> cgImageInfo, image
      Click cursor in image window to get image values.
      Use LEFT mouse button to inquire.
      Use RIGHT mouse button to exit program.

      Expecting cursor clicks in image window...
      Value at (90,159) is 111
      Value at (223,127) is 7
      Value at (302,282) is 178
      Value at (245,336) is 30

      Good-bye. TVINFO has returned control to IDL.
```

Be absolutely sure you right click to exit the program. Otherwise, the IDL command line will be blocked and you will not be able to type any more IDL commands.

How can `cgImageInfo` do this!? It does it primarily by “knowing” where the image is in the display window. `cgImage` actually stores the last known position of the image in the window in a common block that `cgImageInfo` accesses. But then it gets interesting.

What is on the display, of course, is a *representation* of the image, not the image itself. The image might not have the correct aspect ratio, it has

almost certainly been resized from its “normal” size, etc. How it is possible to pick out which pixel to display the “value” of when the user clicks inside the image.

Let’s step though the process, because this technique is useful in many situations involving image interaction. First, we have to find the cursor location in the window. In this case, we are simply going to use the IDL *Cursor* command, but most of the time when we interact with images, we are using some kind of widget program and the cursor location is coming to us as a button event from a draw widget. The important point is that somehow we get the location in the window.

Suppose we get the X and Y locations in device coordinates, like this.

```
IDL> Cursor, xloc, yloc, /Down, /Device
```

I need these locations in normalized coordinates, and I could get them directly in normalized coordinates by setting the *Normal* keyword to the *Cursor* command. But, since they always come to you in device coordinates in a button event from a draw widget, it is worth knowing how to deal with them as device coordinates.

Recall that the image position is given in normalized coordinates. I want to know, first of all, whether the user clicked inside the image “position.” To do that, I have to convert these device coordinates to normalized coordinates. I do this with the *Convert_Coord* command, which converts values from any IDL coordinate system to any other.

```
IDL> normalcoords = Convert_Coord(xloc, yloc, /Device, $  
           /To_Normal)  
IDL> xn = normalcoords[0]  
IDL> yn = normalcoords[1]
```

Now I can check to see if I am inside or outside the image position, which was returned from the image or stored in the `cgImage` common block. I am copying the actual position into the variable *p* here so the code is easier to type.

```
IDL> p = position  
IDL> inside = 0  
IDL> IF (xn GT p[0] AND (xn LE p[2]) AND (yn GT p[1]) $  
           AND (yn LE p[3])) THEN inside = 1  
IDL> IF inside THEN Print, 'Inside image!'
```

If I am inside the image, then I have to figure out where, exactly, I am located inside the image. That is to say, I need to know my image location in terms of the actual pixel locations of the image. I sometimes call this the *image pixel space*.

Identifying Locations in the Image

Now that we have identified the location of the cursor in the display, we must determine the corresponding location in the original image. In order to do this we must figure out a conversion between the normalized coordinates of the display and the image pixel space. Fortunately, a couple of tricks can help us find that transformation easily.

From the image dimensions, we know the number of pixels represented on the image. And from the position parameters, we know the normalized location of those pixels. Using `Scale_Vector` and `Image_Dimensions`, we can easily create a vector containing the normalized space location of each image pixel.

```
IDL> dims = Image_Dimensions(image, XSize=xsize, $  
YSize=ysize)  
IDL> xvec = Scale_Vector(Findgen(xsize+1), p[0], p[2])  
IDL> yvec = Scale_Vector(Findgen(ysize+1), p[1], p[3])
```

Now we have two vectors with normalized values, and we want to locate the X and Y locations returned from the cursor in these vectors. This is easily done with the `Value_Locate` command.

```
IDL> xpixel = Value_Locate(xvec, xn)  
IDL> ypixel = Value_Locate(yvec, yn)
```

From this pixel location, we find the value the image!

```
IDL> Print, image[xpixel,ypixel]  
111
```

This `Value_Locate` trick comes up again and again as a useful way to identify locations and values in images.

Zooming Images

To zoom into a portion of image, it is just a matter of using array subscripts to select a subregion of the image to zoom, and then resizing the subregion before displaying. The only trick, if there is a trick, is to be sure you scale the subregion in exactly the same way you scaled the original image. I mean by this, use the same `Min`, `Max`, and `Top` values in the `BytScl` command. What you don't want to do under any circumstances is use `TVScl` or some alternative `TVScl` command to display the subregion. If you do, there is no guarantee the "colors" of the subregion will match the "colors" of the original image.

For example, a zoomed image can be displayed like this.

```
IDL> subregion = stern[50:99, 50:99]
IDL> Window, XSize=400, YSize=400
IDL> cgImage, subregion
```

You see the image zoomed up by a factor of four on the left hand side of Figure 19.

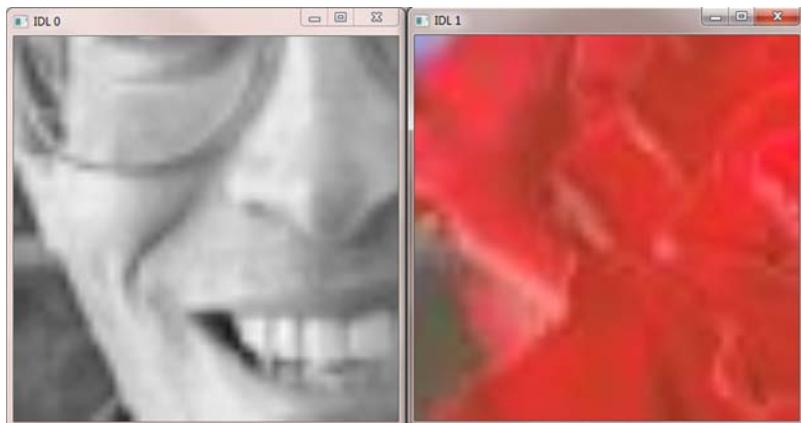


Figure 19: Zooming into an image is just a matter of selecting an image subregion and resizing it.

In this case, we didn't have to worry about scaling, since this image is a byte array. Most images in science are not byte arrays, of course. So, the more general sequence of commands to use would look something like this.

```
IDL> minimg = Min(stern, Max=maximg)
IDL> subregion = stern[50:99, 50:99]
IDL> Window, XSize=400, YSize=400
IDL> cgImage, BytScl(subregion, Min=minimg, Max=maximg)
```

If you had a true-color image, you would have to subset each of the three image planes.

```
IDL> rose = cgDemoData(16)
IDL> subregion = rose[*,50:100,50:100]
IDL> cgImage, subregion
```

Since true-color images are byte data by definition, you don't have to worry about scaling the subregion as a general rule. You see the zoomed rose image on the right hand side of Figure 19.

The [Coyote Library](#) contains a program named [FSC_ZImage](#) that will allow you to interactively zoom any 2D or true-color image, including images that are too large to display on your monitor. Right-clicking in the full-size image window will reveal controls that allow you to select a zoom factor between 2x and 16x. Right-click again to dismiss the controls and see the entire image. The zoomed image appears in a separate graphics window next to the full-size image. You see an example of [FSC_ZImage](#) in Figure 20.

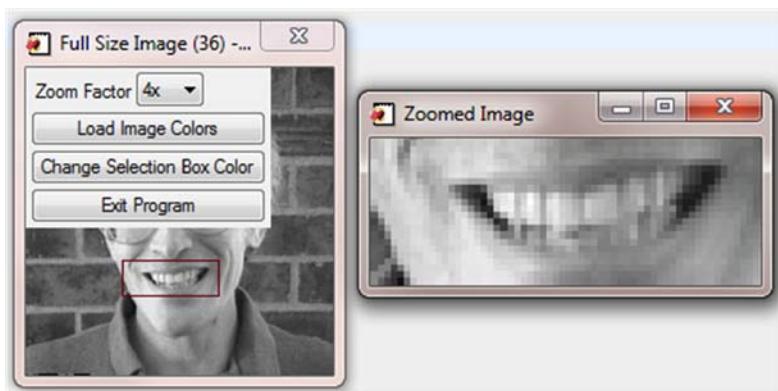


Figure 20: The interactive zoom image program [FSC_ZImage](#). Controls appear and disappear by right-clicking in the full size image window.

Displaying Multiple Images

The standard way of producing multiple IDL traditional graphics displays in a graphics window is with the system variable `!P.Multi`. The idea is to lay out a grid in the window, which can be filled with graphics plots. Neither the `TV` nor the `TVScl` command will honor the `!P.Multi` system variable settings, although most `TV` alternative commands will.

Consider the following commands, which will display an MRI series of the head. There are 57 images in the series, but the last three images do not contain meaningful data and will not be shown in this example. The images in this particular data set are “upside down” with respect to how a physician would view them. So, they must be reversed in the Y direction before being displayed. Outside margins for the title and the color bar are

created with the `!/XY].OMargin` keywords. These keywords must be reset at the end to their default values.

```
IDL> head = cgDemoData(8)
IDL> head = Reverse(head, 2)
IDL> Help, head
      HEAD      BYTE = Array[80, 100, 57]
IDL> s = Size(head, /Dimensions)
IDL> cgLoadCT, 0
IDL> cgDisplay, 9*s[0], 6*s[1]
IDL> !P.Multi = [0, 9, 6]
IDL> !X.OMargin = [0, 25]
IDL> !Y.OMargin = [0, 8]
IDL> FOR j=0,53 DO cgImage, head[*,*,j], $
      Background='white', MultiMargin = 0.25
IDL> !P.Multi = 0
IDL> cgColorbar, /Vertical, Range=[0,10], $
      Position=[0.94, 0.15, 0.97, 0.80], $
      Format='(F0.2)'
IDL> cgText, 0.45, 0.95, 'MRI Study of the Head', $
      /Normal, Alignment=0.5
IDL> !X.OMargin=0 & !Y.OMargin=0
```

You see the result in Figure 21.

While `cgImage` honors the `!P.Multi` system variable, it puts the image itself into the subplot position defined by the `!P.Multi` system variable. It does not account for the image's axis annotation, for example. This is often exactly what you want, but occasionally it is not. For example, if you want to display four images in a 2 x 2 grid, with axes, you might type commands like this.

```
IDL> LoadCT, 33, /Silent
IDL> !P.Multi = [0, 2, 2]
IDL> cgImage, cgDemoData(18), /Keep_Aspect, /Axes
IDL> !P.Multi = 0
```

You see the result in Figure 22. Notice the axes are overwritten by the images. Clearly, we want to include some extra room around the images to allow for the image annotation. `cgImage` has a `MultiMargin` keyword that can be used for exactly this purpose.

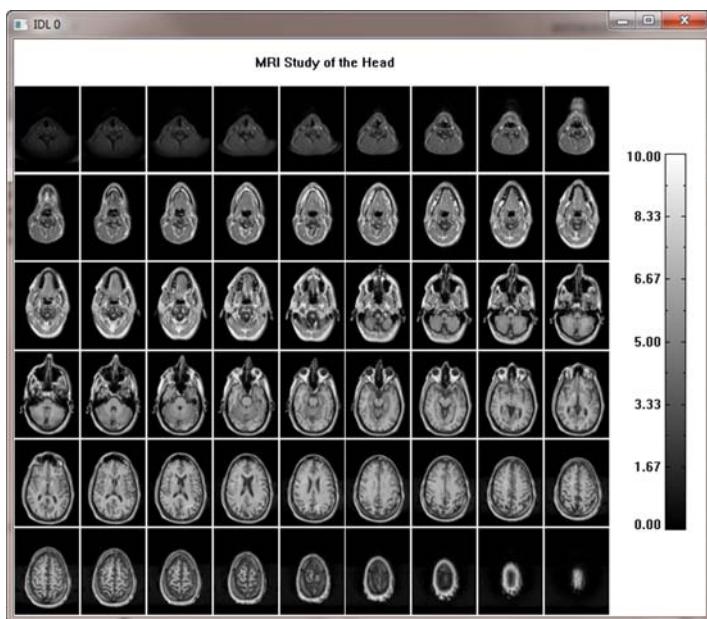


Figure 21: Multiple image plots can be displayed using a combination of `!P.Multi` and `cgImage`.

The `MultiMargin` keyword is a four-element array that specifies margins about the bottom, left, top, and right of the image, respectively. Margins in IDL are always specified in character size units, and that is the case here, except that it is generally `!P.Multi` that is controlling character size, not you. What this means in practice is that some experimentation is usually required to find the proper margins for any specific multi-plot layout. Here we need larger margins on the bottom and left of the image. The commands above can be modified like this.

```
IDL> LoadCT, 33, /Silent
IDL> !P.Multi = [0, 2, 2]
IDL> margin = [2, 2, 1, 0]
IDL> cgImage, cgDemoData(18), /Keep_Aspect, /Axes, $
      MultiMargin=margin
IDL> !P.Multi = 0
```

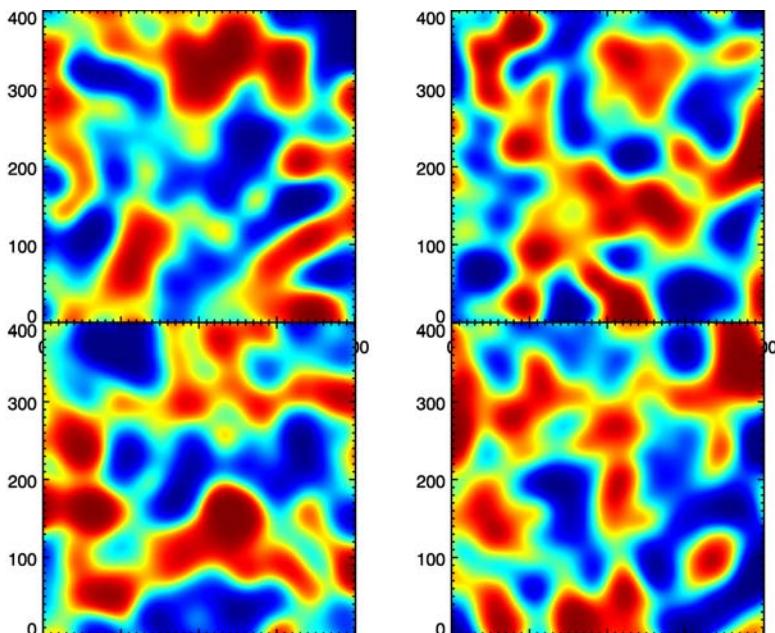


Figure 22: Normally, the image takes up the entire space devoted to the multiplot “position,” not leaving enough room for axis annotation.

You see the result of adding image margins in Figure 23.

Transparent Images

One of the significant advantages of `cgImage` as a TV alternative command is its ability to display transparent images. Transparent images have an alpha channel that indicates how the transparent image is to be “blended” with the information already on the display. Alpha channels are generally coded as values from 0 to 1 (or, in images, from 0 to 255), with 0 indicating total transparency (i.e., display the background pixel) and 1 indicating total opacity (i.e., display the foreground pixel). All other pixels indicate a blending of the background and foreground pixels in the final result.

To demonstrate, I have downloaded a 32-bit image I found on Stefan Schneider’s web page on Transparent PNG Images (www.libpng.org/pub/png/pngs-img.html). It is a small image of a toucan. You can also download the

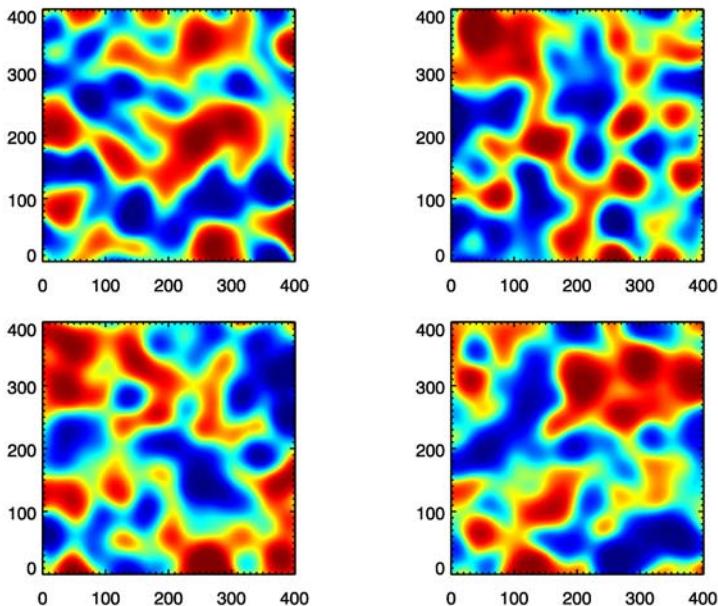


Figure 23: The images can be displayed with their annotations by using the **MultiMargin** keyword to [cgImage](#)

file on my web page (www.idlcoyote.com/books/tg/data/toucan.png). I saved the image file as *toucan.png* and read it like this.

```
IDL> toucan = Read_PNG('toucan.png')
IDL> Help, toucan
TOUCAN   BYTE = Array[4, 162, 150]
```

Transparent PNG files are always pixel interleaved like this. In this case, the alpha channel is scaled from 0 to 255. The actual PNG image can be displayed separately from the alpha channel, like this.

```
IDL> Window, XSize=162*2, YSize=150
IDL> !P.Multi = [ 0, 2, 1]
IDL> cgImage, toucan[0:2,*,*]
IDL> cgImage, Reform(toucan[3,*,*])
IDL> !P.Multi = 0
```

You see the result in Figure 24.

To see this transparent PNG image displayed correctly, let's first display it on a colored background, where the shadow is easily seen.



Figure 24: The PNG image on the left and its alpha channel on the right. The white pixels in the alpha channel are where the pixels are totally opaque, and the black pixels are where the pixels are totally transparent. You see gradations in the shadows of the alpha channel.

```
IDL> s = Size(toucan, /Dimensions)
IDL> Window, XSize=400, YSize=400
IDL> cgErase, Color='bisque'
IDL> cgImage, toucan, /Keep_Aspect, Margin=0.2
```

You see the result in the left hand side of Figure 25.

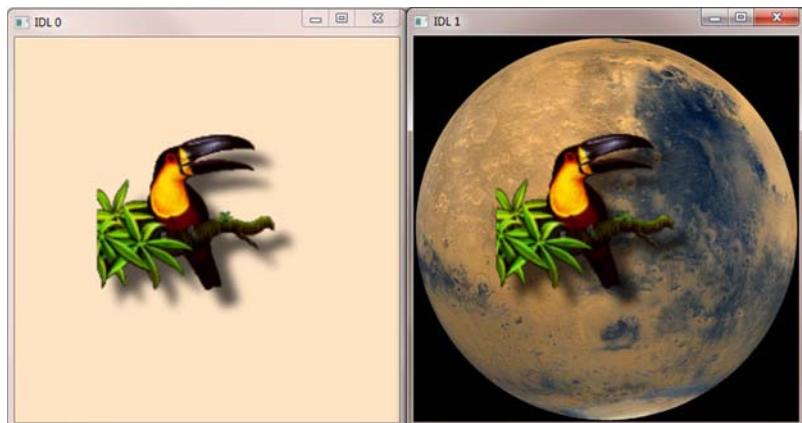


Figure 25: Alpha channel images can be displayed on any type of background image.

But, of course, this image does not have to be displayed on a colored background. It can be displayed on any image whatsoever. Try these commands to display it on an image of Mars.

```
IDL> marsfile = Filepath(Subdir=['examples', 'data'], $  
    'marsglobe.jpg')  
IDL> Read_JPEG, marsfile, mars  
IDL> Window, 1, XSize=400, YSize=400  
IDL> cgImage, mars  
IDL> cgImage, toucan, /Keep_Aspect, Margin=0.2
```

You see the result in the right hand side of Figure 25.

Creating Blended Images

A “blended” image is similar to a transparent image. To create a blended image, we need two 24-bit images. To see how this is done, let’s take two completely different images. One will be a 2D cat scan image of a skull, and the other will be this image of Mars we just used. We would like to blend the two images together to see a fused image.

We read the two image files like this.

```
IDL> scanfile = Filepath(Subdir=['examples','data'], $  
    'md1107g8a.jpg')  
IDL> Read_JPEG, scanfile, scan  
IDL> Help, scan  
    SCAN  BYTE = Array[250, 250]  
IDL> marsfile = Filepath(Subdir=['examples', 'data'], $  
    'marsglobe.jpg')  
IDL> Read_JPEG, marsfile, mars  
IDL> Help, mars  
    MARS  BYTE = Array[3, 400, 400]
```

The easiest way to make the scan file the correct size, *and* make it into the 24-bit image we need, is to display it in an IDL graphics window and take a snapshot of the window with the [Coyote Library](#) routine `cgSnapshot`.

```
IDL> Window, XSize=400, YSize=400  
IDL> LoadCT, 0  
IDL> cgImage, scan, /NoInterp  
IDL> snapshot = cgSnapshot()  
IDL> Help, snapshot  
    SNAPSHOT  BYTE = Array[3, 400, 400]
```

Now we have to decide which is the foreground image and which is the background. Let’s say the Mars image is the background image and the

scan image is the foreground. We select an alpha value between 0 and 1. Suppose we want 60% of the foreground image and 40% of the background image in the blended image. Then we can construct and display the blended image like this.

```
IDL> alpha = 0.6
IDL> blended = (snapshot * alpha) + (mars * (1-alpha))
```

We can display all three images like this.

```
IDL> Window, XSize=400*3, YSize=400
IDL> !P.Multi = [0,3,1]
IDL> cgImage, mars
IDL> cgImage, scan
IDL> cgImage, blended
IDL> !P.Multi = 0
```

You see the result in Figure 26.

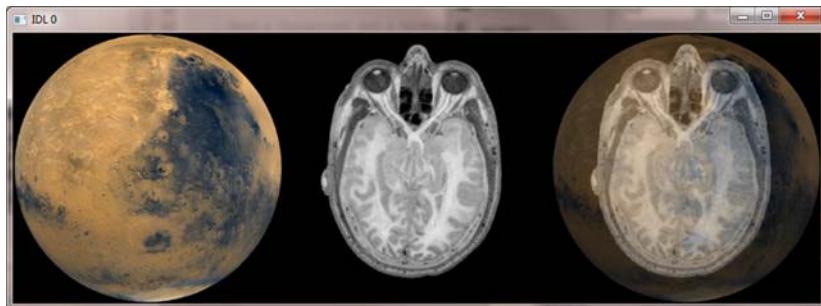


Figure 26: Blended images are easily created with any two true-color images. True-color images can be created from any graphics window with the `cgSnapshot` command.

The `cgBlendImage` program in the [Coyote Library](#) will blend and display any two matched size true-color images with a user-specified alpha value between 0 and 1. Any keyword appropriate for the `cgImage` command can be used with `cgBlendImage`.

```
IDL> Window, XSize=400, YSize=400
IDL> cgBlendImage, snapshot, mars, Alpha=0.6
```

Creating Transparent Images

While we can't create transparent images as fancy as the toucan image (well, maybe we can, but IDL is not Photoshop, after all), we can create transparent PNG images that often do the job for us.

For example, consider the IDL logo image that is distributed in the IDL data directory. This is a two-dimensional image that is distributed with its own color table vectors.

```
IDL> logfile = Filepath(Subdir=['examples','data'], $  
    'examples.tif')  
IDL> logo = Read_TIFF(logofile, r, g, b)  
IDL> Help, logo  
LOGO    BYTE = Array[375, 150]
```

To display this image, we have to first load the color table vectors. This is an image where the color table is specific for individual pixels. The color table is not a smooth gradient from one color to another, like most IDL color tables. If we resize an image like this one, we *must* use nearest neighbor sampling. We cannot use any kind of interpolation or the colors in the image will appear distorted.

TIFF is also a file format that treats the [0,0] pixel in an image as the pixel in the upper-left corner of the image. I have never read a TIFF image that wasn't upside-down when displayed in IDL, so my rule of thumb is to always reverse the Y direction of images after they are read and before they are displayed.

```
IDL> logo = Reverse(logo, 2)  
IDL> TVLCT, r, g, b  
IDL> Window, XSize=375, YSize=150  
IDL> cgImage, logo, /NoInterp
```

You see the result in Figure 27. The black color in this image is the color we want to make transparent.



Figure 27: The black color in the IDL logo is the color we want to make transparent.

To make a transparent image, we must create a pixel interleaved PNG image with an alpha channel. The alpha channel will be set to 0 in the loca-

tions where we have black pixels and to 255 everywhere else. We start by creating a pixel interleaved image from the logo image in the display window. We can simply use `cgSnapshot` to capture the image out of the window.

```
IDL> logo24 = cgSnapshot()
IDL> Help, logo24
LOGO24      BYTE = Array[3, 375, 150]
```

Next, we create a totally opaque alpha channel.

```
IDL> alpha = ByteArr(375, 150) + 255B
```

Then, we find the black pixels in the `logo24` image. This will be considerable easier if we separate the individual color bands of the image, which will, in turn, be easier if this were a band interleaved image rather than a pixel interleaved image.

Here is code, then, that converts this pixel interleaved image to a band interleaved image and then separates each band into its own color channel. Note, we could have just read the image out of the graphics window as a band interleaved image by setting the `TRUE=3` keyword to `cgSnapshot`, but knowing how to manipulate band interleaving with the `Transpose` command is a skill that comes in handy on many occasions. The vector in the second positional parameter in the `Transpose` command below specifies the new order of the bands. Band 1 becomes band 0, band 2 becomes band 1, and band 0 becomes band 2.

```
IDL> logo24 = Transpose(logo24, [1,2,0])
IDL> Help, logo24
LOGO24      BYTE = Array[375, 150, 3]
IDL> red = logo24[*,*,0]
IDL> grn = logo24[*,*,1]
IDL> blu = logo24[*,*,2]
```

We are now ready to locate the black pixel. A black pixel is one that contains the value 0 in the same location in all three colored bands. We can find it with the `Where` function, like this. If we find any black pixels, we set those indices to 0 in the alpha channel.

```
IDL> blackIndices = Where( (red EQ 0) AND (grn EQ 0) $ 
    AND (blu EQ 0), count)
IDL> IF count GT 0 THEN alpha[blackIndices] = 0B
```

Now, we simply put the pieces back together into a pixel interleaved 32-bit image that we save to a PNG file as a transparent image. (A 32-bit image is simply a 24-bit image with an alpha channel. The alpha channel is the

fourth “plane” in the image.) We start by making a band interleaved image that we transpose to a pixel interleaved image.

```
IDL> logo32 = [ [[red]], [[grn]], [[blu]], [[alpha]] ]
IDL> logo32 = Transpose(logo32, [2,0,1])
IDL> Write_PNG, 'idl_logo.png', logo32
```

All this is made quite a bit simpler by the [Coyote Library](#) program [Make_Transparent_Image](#), which can make and write a transparent PNG image from either a 24-bit image supplied as a parameter, or by reading a 24-bit image directly out of the current graphics window. In other words, we can create the same file like this.

```
IDL> image = Make_Transparent_Image(Color='black', $
Filename='idl_logo.png', /Save_PNG)
```

To see how this works, we can display the transparent logo image on the Mars image from before.

```
IDL> Window, XSize=400, YSize=400
IDL> cgImage, mars
IDL> cgImage, logo32, Position=[0.30,0.05,0.70,0.25], $
/Keep_Aspect
```

You see the result in Figure 28.



Figure 28: The transparent logo image created from a 2D image is superimposed on another image.

Images in Resizeable Graphics Windows

Images are a little harder to display in resizeable graphics windows than, say, a line plot, due to the fact that the *TV* command, which is always used to display images in IDL, does not erase the graphics window before displaying the image in it. If we resize a graphics window without first erasing what was in the old window, we can end up with quite a lot of window debris. Consider the *ImDisp* alternative *TV* command in this resizeable graphics window, for example.

```
IDL> image = cgDemoData(19)
IDL> cgWindow, 'ImDisp', image, Margin=0.2
```

If we make the window smaller, we will see something that looks like the graphics window in Figure 29. You can see the debris left over from the previous, larger image display.

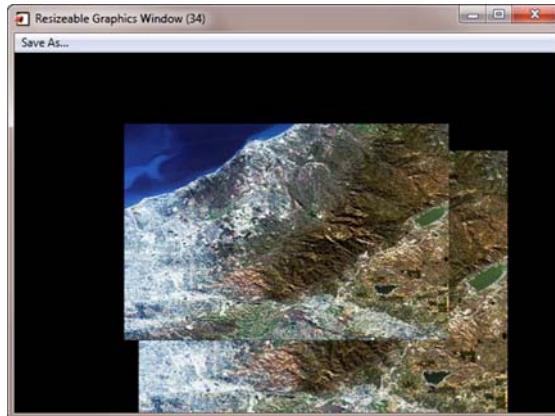


Figure 29: If the display is not erased before a graphic is redrawn in a display window, there is likely to be window debris still present in the graphics window.

There are a couple of ways to solve this particular problem for *ImDisp*. We could, for example, add an *Erase* command to *cgWindow* before we add the *ImDisp* command to it.

```
IDL> cgWindow, 'Erase'
IDL> cgWindow, 'ImDisp', image, Margin=0.2
```

Or, we could set the *Erase* keyword on *ImDisp* itself.

```
IDL> cgWindow, 'ImDisp', image, Margin=0.2, /Erase
```

The `cgImage` program is written in such a way that it solves this problem automatically, setting the *Erase* keyword as appropriate when it is added to `cgWindow`.

```
IDL> cgImage, image, Margin=0.2, /Window
```

The problem now is that sometimes you want to add an image to a graphics display without first erasing the window. Suppose, for example, you want to display an image inside a line plot. You would first display the line plot and then the image. But you don't want to erase the line plot. The solution here is to use the *AddCmd* keyword to the `cgImage` command to simply add the `cgImage` command to the `cgWindow` display without setting the *Erase* keyword. The code looks like this.

```
IDL> cgPlot, cgDemoData(1), YRange=[0, 80], /Window  
IDL> cgImage, image, Position=[0.7, 0.5, 0.9, 0.9], $  
    /Keep_Aspect, /AddCmd
```

Note that if your graphics display window is behind other windows, you can bring it forward on the display by tying the `cgSet` command without an argument.

```
IDL> cgSet
```

You see the result in Figure 30.

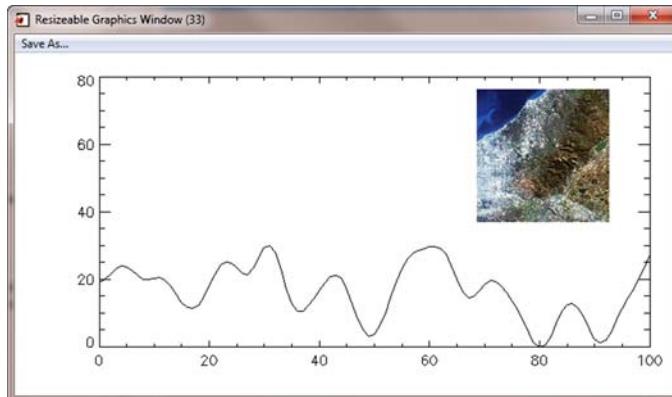


Figure 30: To add an image to `cgWindow`, simply set the *AddCmd* keyword, rather than the *Window* keyword to the command.

Chapter 8



Image Processing

Basic Image Processing

The purpose of image processing is to simply manipulate an image in some mathematical or graphical way that makes the information contained in the image more readily accessible. IDL was originally written as a program for working with and processing image data, so, naturally, image processing tools are part of the package of commands IDL provides.

This chapter does not do justice to all the possible ways you can process digital images. If this is your interest, I would refer you to *Digital Image Processing, Third Edition*, by Rafael C. Gonzalez and Richard E. Woods. This is the best digital image processing book I have ever read, and the only one in which I could immediately translate a section of the book into an IDL program. I spent a pleasant Christmas vacation once reading that book cover to cover, it was *that* good!

My goal for this chapter is to introduce you to some of the basic image processing commands in IDL and to give you a taste of how you can use these commands to build your own image processing programs.

Contrast Enhancement

The most basic image processing technique is to simply manipulate the contrast in an image to enhance parts of the image at the expense of other, less interesting parts. At its most basic level, this means knowing how to use the *BytScl* command and its keywords, which you learned how to use in the last chapter.

Consider this medical image. We will plot the pixel distribution next to the image itself.

```
IDL> ctscan = Reverse(cgDemoData(5), 2)
IDL> cgDisplay, 500, 250
IDL> !P.Multi = [0,2,1]
IDL> cgHistoplot, ctscan, Missing=0
IDL> cgLoadCT, 17, /Brewer
IDL> cgImage, ctscan
IDL> !P.Multi = 0
```

You see the result in Figure 1. Notice the vast majority of the pixels in this image have values between 50 and 100, resulting in a low-contrast image.

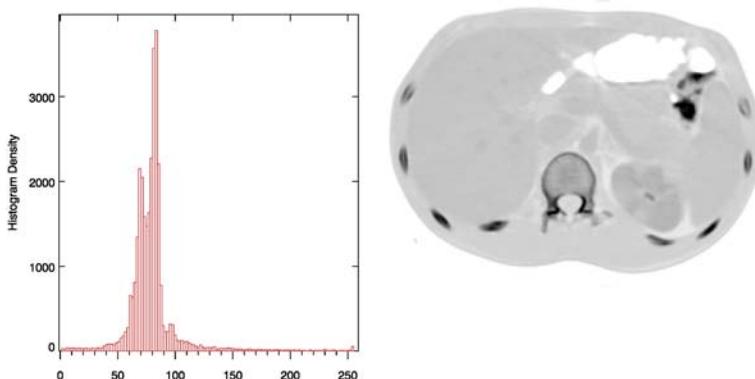


Figure 1: Most of the pixels in this medical image have values between 50 and 100, resulting in a low-contrast image.

We could see more detail in this image if we could “spread the contrast out” to show just this section of the image in the full range of 0 to 255. We can do this by setting the *Min* and *Max* keywords to the *BytSel* command to scale the data before we display it. The commands to display the original image next to a high contrast image look like this.

```
IDL> !P.Multi = [0,2,1]
IDL> cgImage, ctscan
IDL> cgImage, BytSel(ctscan, Min=50, Max=100)
IDL> !P.Multi = 0
```

You see the result in Figure 2.

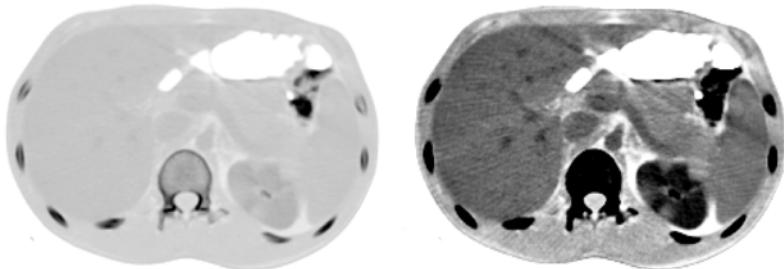


Figure 2: The original image on the left, and the enhanced contrast image on the right. Notice how much more detail can be seen in the image on the right.

In the example on the right, any pixel in the original image with a value less than 50 is set to 0 in the scaled image, and any pixel with a value greater than 100 is set to 255 in the scaled image. The pixel values between 50 and 100 in the original image are linearly scaled between 0 and 255 to produce this increased contrast image.

Window Leveling

Sometimes we call the range of values between the minimum and maximum that we set in the *BytScl* command the “window” of the image. And we call the value in the center of the range the “level” of the image. We talk, then, about “window leveling” an image. Sometimes you will hear the term “contrast” for the window and “bright-ness” for the level.

Here are two views of the same image that have been “window leveled” with the same window, but with different levels.

```
IDL> !P.Multi = [0,2,1]
IDL> window = 50
IDL> halfwin = window / 2
IDL> level = 65
IDL> cgImage, BytScl(ctscan, Min=level-halfwin, $
      Max=level+halfwin)
IDL> level = 110
IDL> cgImage, BytScl(ctscan, Min=level-halfwin, $
      Max=level+halfwin)
IDL> !P.Multi = 0
```

You see the result in Figure 3. You see that depending upon what window and level you choose, different information in the image can be emphasized.

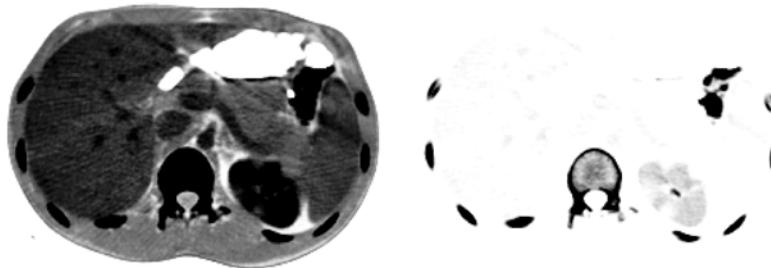


Figure 3: Depending on the window and level, different information will be emphasized in the image. The image on the left has a window of 50 and a level of 65. The image on the right has a window of 50, but a level of 110.

Normally, window leveling of an image is done interactively in a widget program. Generally speaking the size of the image window or the contrast is changed by moving the cursor vertically in the graphics window and the level or brightness is changed by moving the cursor horizontally in the graphics window. The [WindowImage](#) command in the [Coyote Library](#) is an example of an application that uses this technique to manipulate the contrast of an image.

```
IDL> WindowImage, ctscan
```

You see an example of [WindowImage](#) in Figure 4.

Histogram Equalization

Another method of improving contrast in images is to perform histogram equalization. Histogram equalization is a method that takes the image histogram and attempts to “spread the pixels out” so there are approximately the same number of pixels in each bin.

IDL provides two different commands for performing histogram equalization. The first is *Hist_Equal*, which performs a “normal” histogram



Figure 4: The `WindowImage` command allows the user to window and level an image interactively by moving the cursor in the image display. Moving the cursor horizontally sets the level, and moving the cursor vertically sets the window.

equalization. The second is `Adapt_Hist_Equal`, which is an adaptive histogram equalization method in which the equalization is based on the local region surrounding each pixel in the image. Each pixel is mapped to an intensity based on its rank within the surrounding neighborhood.

Here is code that shows you what these commands do. Each contrasted image is shown next to its histogram. The cumulative probability distribution function of the histogram is plotted in blue on each histogram.

```
IDL> cgDisplay, 250*2, 250*3
IDL> !P.Multi = [0,2,3]
IDL> cgImage, ctscan, Background='white'
IDL> cgHistoplot, ctscan, Missing=0, /OProbability
IDL> cgImage, Hist_Equal(ctscan)
IDL> cgHistoplot, Hist_Equal(ctscan), Missing=0, /OProb
IDL> cgImage, Adapt_Hist_Equal(ctscan)
IDL> cgHistoplot, Adapt_Hist_Equal(ctscan), Missing=0, $
    /OProbability
IDL> !P.Multi = 0
```

You see the result in Figure 5.

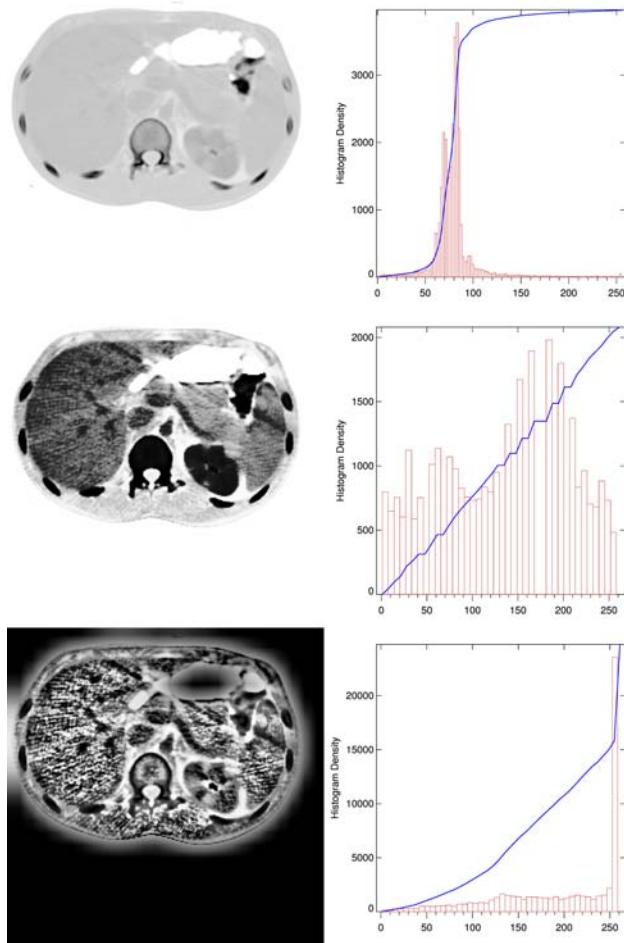


Figure 5: Histogram equalization techniques in IDL. The top figure is the unprocessed image. The middle figure is processed with Hist_Equal. The bottom figure is processed with Adapt_Hist_Equal.

You can see that histogram equalization does a good job of making the probability function more linear over the range of pixel values in the output image.

Histogram Matching

Histogram matching is similar in concept to histogram equalization, in which the pixel values of an image are manipulated in such a way as to distribute the pixel values in an approximately even distribution over the dynamic color range of the image.

The difference is that in histogram matching we don't use an even distribution, but a distribution that is supplied to us from another image or perhaps from a subset of the same image. In other words, we want to manipulate the pixel distribution of one image to mirror the pixel distribution of another image. The pixel distribution of an image is, of course, its histogram.

Suppose we want to distribute the pixels in the CT scan image we have been working with to match the pixel distribution of some other image entirely, say, the world elevation data set in the IDL examples directory. You can type these commands to see what this image and its histogram look like.

```
IDL> world = cgDemoData(7)
IDL> cgDisplay, 720, 360
IDL> !P.Multi = [0,2,1]
IDL> cgHistoplot, world
IDL> cgImage, world
IDL> !P.Multi = 0
```

You see the result in Figure 6. Compare this histogram plot with the histogram plot in Figure 1.

To develop a histogram matching algorithm, we must first calculate the histograms of both images. We use the *Histogram* command to do this.

```
IDL> h_to_match = Histogram(world, Min=0, Max=255)
IDL> h = Histogram(ctscan, Min=0, Max=255)
```

Note: We are not using the *Binsize* keyword to the *Histogram* function here because both of these images are byte images, so the bin size is automatically set to 1. Normally, however, the bin size must be set. It is extremely important to match the data type of the *Binsize* argument with the data type of the image. Very strange and incorrect results can be returned by the *Histogram* function if this detail is ignored! The [Convert_To_Type](#) program in the [Coyote Library](#) was written specifically to solve this problem, although it comes in handy in other situations, too.

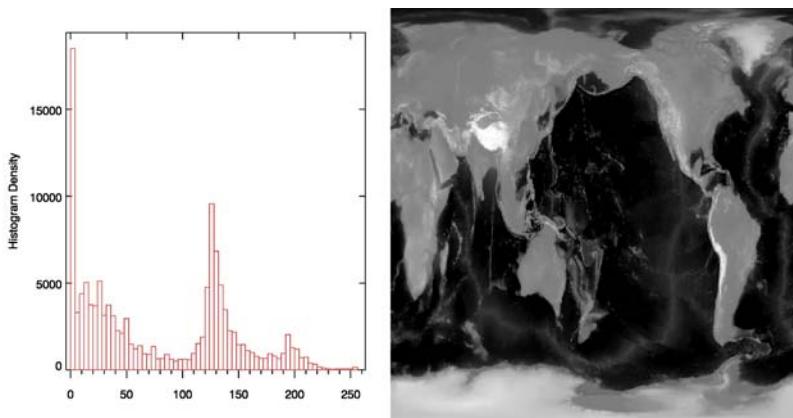


Figure 6: The Earth elevation image and its pixel distribution histogram. We would like to match the pixel distribution of the CT scan image to this pixel distribution.

The method we are going to use to match histograms requires that both images have the same number of values. This is not the case here, since our images are different sizes (*ctscan* is 256x256 and *world* is 360x360). Therefore, we need to correct the *h_to_match* variable so that it has the same number of values as the *h* variable, like this.

```
IDL> factor = Total(h) / Total(h_to_match)
IDL> h_to_match = h_to_match * factor
```

Efficient Programming Practices

The multiplication line above could also be written like this.

```
hist_to_match *= factor
```

Writing the code this way is more efficient because the multiplication is done “in place,” and doesn’t require allocating memory for a second array (a temporary one to do the multiplication on). It is identical to writing the command with the *Temporary* function like this.

```
hist_to_match = Temporary(hist_to_match) * factor
```

The meaning of this command is “make a temporary variable that points to this current variable’s location in memory, perform the multiplication there, then switch the permanent variable’s pointer to this memory location.” This is what we mean by “doing the multiplication in place.”

The alternative (the way we originally wrote the command) is to allocate memory the size of the *h_to_match* variable to a temporary memory loca-

tion, perform the multiplication in this new area of memory, then switch the *h_to_match* variable's pointer to this new area of memory, while allowing the old memory location to be used over again. This, in fact, requires two copies of the variable in memory (the original and the copy).

This is not a performance issue with this particular command, but it can be an issue when the same data variable is on both sides of the equal sign and the variable is large. Using the *Temporary* function, or performing the algebraic manipulations with the proper operators, can not only make your code more efficient, but it can prevent memory fragmentation issues, too, and is an important feature of good memory management practices. Unfortunately, this is a topic for another book, but it is worth mentioning here.

The next step requires that we create two cumulative total vectors from each of the histograms. We will call these vectors *s* and *v*.

```
IDL> s = Total(h, /Cumulative)
IDL> v = Total(h_to_match, /Cumulative)
```

Next, we construct a distribution function, *z*, the same length as *s* and *v*, by adding up values in *s* until we find approximately the same number of values as are in *v*. Note that the algorithm we use, like histogram equalization, is only approximate.

```
IDL> z = BytArr(N_Elements(s))
IDL> FOR j=0,N_Elements(z)-1 DO BEGIN & $
IDL>     i = Where(v LT s[j], count) & $
IDL>     IF count GT 0 THEN z[j] = (Reverse(i)) [0] & $
IDL> ENDFOR
```

Finally, we create the histogram matched image by simply looking up the new image values in the distribution function we just created. Here are commands to create the matched image and display its new pixel distribution or histogram. (The *Byte* function in the first command is superfluous, since the *ctscan* variable is already byte data, but I include it as a reminder that we need byte (or, at least integer) data in this index operation.)

```
IDL> matchedImage = z[Byte(ctscan)]
IDL> cgDisplay, 400, 400
IDL> !P.Multi = [0,2,1]
IDL> cgHistoplot, matchedImage, Max_Value=5000
IDL> cgLoadCT, 0, /Reverse
IDL> cgImage, matchedImage, /Scale
IDL> !P.Multi = 0
```

You see the result in Figure 7. Compare this histogram to the histogram shown in Figure 6.

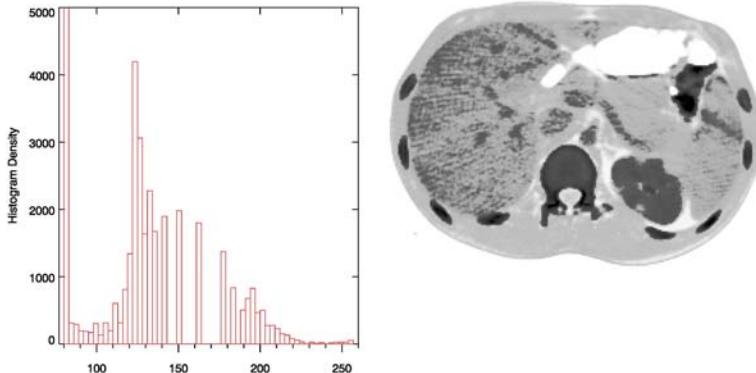


Figure 7: Histogram matching has been performed on the CT scan image. The histogram came from the world elevation data image.

This kind of histogram matching can be performed with the [HistoMatch](#) program from the [Coyote Library](#).

```
IDL> matchedImage = HistoMatch(ctscan, h_to_match)
```

The variable *h_to_match* in the above command can either be a histogram to match or it can be another image whose histogram will be matched.

Additional Methods of Contrast Stretching

Contrast stretching does not always have to be done in a linear fashion, as the *BytScl* function performs it. In fact, any number of mathematical transformations can be applied to digital images. IDL itself doesn't provide any of these other contrast transformations, but they are fairly easy to program in IDL.

For example, a common image transformation is to take the square root of the image. Normally this is used when an image has values that vary over several orders of magnitude. Here is how a simple square root transformation could be created in IDL.

```
IDL> cgImage, BytScl(Sqrt(ctscan))
```

A selection of other contrast stretching programs to implement other mathematical transformations is provided in the [Coyote Library](#). Here is a

list of programs and a short description of the contrast stretch each program performs.

[ASinhScl](#)

This is an inverse hyperbolic sine function intensity transformation. I think of this as a “tuned” gamma or power-log function. It is a transformation that allows for linear scaling of noise values and a logarithmic scaling of signal values. It is a transformation I have found particularly effective with astronomical images. The *ASinhScl* function uses the following formula.

```
output = ALog(Abs(input) + Sqrt(input^2 + 1.0))
```

[ClipScl](#)

This is a function that performs linear scaling of image arrays in a manner similar to *BytScl*. The only difference is that a user-specified percentage of pixels can be “clipped” from the image histogram prior to scaling. By default, two percent of the pixels are clipped, resulting in a “2% stretch” that you may recognize as the default contrast stretch for images in ENVI.

[GaussScl](#)

This function applies a Gaussian function to the image histogram, weighting pixels in the center of the Gaussian curve more than pixels on the outer edges. Parameters select the width and location of the Gaussian curve used in the stretch.

[GmaScl](#)

This function performs a gamma log scale transformation of the image pixels. This is also sometimes called a power-law scaling. A gamma log scale is similar to *BytScl* in that it generally has a long linear section in the middle of the curve, but trails off logarithmically at either end of the curve. Input parameters control the slope of the linear portion of the curve. A gamma transformation uses the formula.

```
output = inputγ
```

Gamma values can range from 0.04 to 25.0.

[LogScl](#)

This function performs a log scale transformation of the image pixels. Input parameters control the overall shape and center of the curve and how quickly the curve tails off at either end. A log transformation uses the formula.

```
output = ALog(1 + input)
```

Interactive Contrast Stretching

Because contrast stretching is fairly arbitrary, and most of the time we are only looking for an image that “looks good” in the context of appearing on the display, we often perform the contrast stretch interactively. The [Coyote Library](#) routine [XStretch](#) is used for this purpose. All of the contrast stretches discussed so far are available from within [XStretch](#). You see an example of [XStretch](#) in Figure 8. The red and blue bars can be moved interactively with the mouse to select the minimum and maximum values for the stretch. Parameters for the various stretches are set with widgets that appear below the image histogram when they are appropriate.

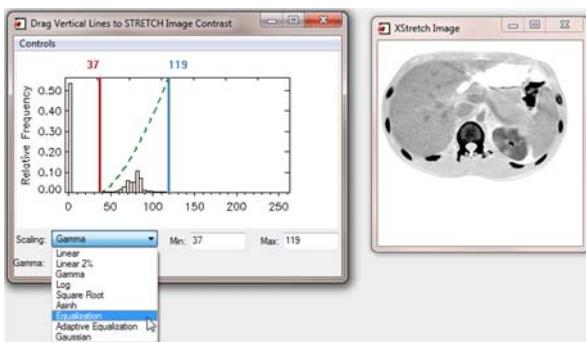


Figure 8: The [XStretch](#) program allows you to interactively apply nine different contrast stretch image transformations. The stretched image and/or the histogram window can be saved in a variety of file formats, or the image can be saved to the IDL command line.

From the pull-down *Controls* menu, you can save the stretched image or the image histogram in a variety of file formats, or you can just save the stretched image to the main IDL command line where you can continue to work with the stretched image as a variable.

Differential Image Scaling

Sometimes it is not enough to scale the entire image to increase contrast. Parts of the image have to be scaled differently than other parts of the image. This is called *differential scaling*. Differential scaling often plays a role when working with MODIS reflectance data to create true-color images, for example.

MODIS is an instrument that collects radiance imagery at different wavelengths or “channels.” Some of these channels are used to produce reflectance images that show the fraction of incident sunlight reflected from the Earth’s surface. The standard MODIS reflectance products contain reflectance values in the range of -0.01 to 1.10. A true-color image is constructed from channels 1, 4, and 3, for the red, green, and blue images, respectively.

You see on the left-hand side of Figure 9 what happens when these three channels are added together and byte scaled with *BytScl* with the *Min* keyword set to -0.01 and the *Max* keyword set to 1.10. The image appears dark and dingy compared to the image on the right, which has been differentially scaled. We are going to discuss how this differential scaling is accomplished. If you want to step right to the bottom line, however, you will find this algorithm implemented in the [ScaleModis](#) function in the [Coyote Library](#).

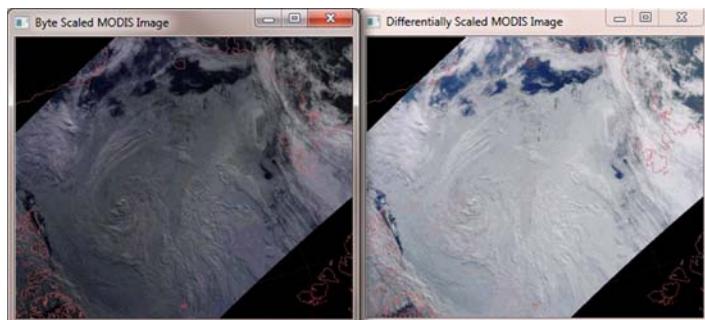


Figure 9: MODIS channels 1, 4, and 3 are combined to create a true-color image. The image on the left is simply byte scaled. The image on the right is differentially scaled.

You see in Table 1 representative input values from the MODIS image, what the resulting byte scaled value would be for that input value, and what we would like the desired byte scaled value to be. What this table tells you is that we would like to scale the image values between -0.01 and 0.121 between 0 and 110, not between 0 and 30. Or, we would like to scale the values between 0.251 and 0.512 between 210 and 240, not between 120 and 190. In other words, equally spaced differences in the image are scaled

into an unequal, variable number of “colors” upon output.” Essentially, we have five data bins and we would like to scale each bin in a different way.

Input Value	Byte Scaled	Desired Value
-0.01	0	0
0.121	30	110
0.251	60	160
0.512	120	210
0.817	190	240
1.10	255	255

Table 1: This table shows input MODIS image values, what the corresponding byte scaled value would be, and what we would like the byte scaled value to be to brighten up the MODIS image.

Image Partitioning

One way to separate the image values into five data bins is to simply use the *Where* function to locate the image pixels we are interested in.

Because of the difficulties associated with locating floating point values, sometimes floating point data are scaled to integers (or bytes) before the pixels are located, although we will not do that here.

If we assume the MODIS channel 1 image is in the variable *ch1*, then we might write code like this to scale the pixels differentially.

```
s = Size(ch1, /Dimensions)
outImg = BytArr(s[0], s[1])
b1 = Where((ch1 GE Min(ch1)) AND (ch1 LT 30), c1)
b2 = Where((ch1 GE 30) AND (ch1 LT 60), c2)
b3 = Where((ch1 GE 60) AND (ch1 LT 120), c3)
b4 = Where((ch1 GE 120) AND (ch1 LT 190), c4)
b5 = Where((ch1 GE 120), c5)
IF c1 GT 0 THEN outImg[b1] = BytScl(ch1[b1], $
    Top=110)
IF c2 GT 0 THEN outImg[b2] = BytScl(ch1[b2], $
    Top=40) + 110B
IF c3 GT 0 THEN outImg[b3] = BytScl(ch1[b3], $
    Top=50) + 160B
IF c4 GT 0 THEN outImg[b4] = BytScl(ch1[b4], $
    Top=30) + 210B
IF c5 GT 0 THEN outImg[b5] = BytScl(ch1[b5], $
    Top=15) + 240B
```

Not only is this code long, tedious to write, and prone to errors, it makes it almost impossibly difficult to change the code if the differential scaling we implemented here is not exactly what we want. The code can't be tweaked easily and is almost the antithesis of the *IDL Way*. (The *IDL Way* is the tongue-in-cheek way IDL newsgroup readers refer to code changes that dramatically speed up the performance of IDL programs, usually by taking advantage of array operations or the justifiably famous *Histogram* command.)

Value_Locate and Histogram for Fast Partitioning

As it happens, there is a better way. It involves the powerful routine *Value_Locate*, which is one of those routines in IDL, like the *Histogram* function, that always seems to be playing a role when an array operation needs to be done quickly and easily.

The stated purpose of the *Value_Locate* command is to “find the intervals within a give monotonic vector that brackets a given set of one or more search values”. If that is as unclear to you as it is to me, maybe this will help. If you pass *Value_Locate* a vector of increasing values and you also pass it a number, *Value_Locate* can tell you where that number would fall within the “bins” created by the vector. It will, essentially, return the bin number to you.

Consider a simple example. Suppose we make a vector of values from 0 to 99, and we ask which bin the number 33.4 is in.

```
IDL> vector = IndGen(100)
IDL> binNumber = Value_Locate(vector, 33.4)
IDL> Print, binNumber
      33
```

You see, in fact, that the number 33.4 falls into this bin. Here are the numbers that bracket our value.

```
IDL> Print, vector[binNumber], vector[binNumber+1]
      33      34
```

The number you pass to *Value_Locate* doesn't have to be a scalar, it can be a vector of values itself.

```
IDL> Print, Value_Locate(vector, [21.3, 89.1, 44.7])
      21      89      44
```

What about a vector made up of the values in Table 1 above?

```
IDL> vector = [0.121, 0.251, 0.512, 0.817]
```

Consider these three values: 0.05 (a value smaller the first number in the vector), 0.46 (a number contained with the vector values), and 0.98 (a number larger the last vector value).

```
IDL> Print, Value_Locate(vector, [0.05, 0.46, 0.98])
-1    1    3
```

What do these three numbers mean? Consider that the first “bin” created by the vector is the bin from 0.121 to 0.251. If there was a number in this bin, *Value_Locate* would return a 0, indicating the first bin number.

```
IDL> Print, Value_Locate(vector, 0.2)
0
```

So, a -1 means that the number is less than the number that starts the first bin.

We find the same thing on the other end. This vector contains three “bins”, labelled 0, 1, 2. If a number is larger than the limits of the last bin (0.817), then this number must be in the “next” bin, or bin 3.

If we add a 1 to each of the numbers returned from *Value_Locate*, then we have numbers that go from 0 to 4, identifying the indices of five bins: the three bins created by the vector values, and the one bin that is “less than” the vector values and the one bin that is “greater than” the vector values.

To partition the image pixels into five bins, we write code like this.

```
IDL> partitionedImage = Value_Locate(vector, ch1) + 1
IDL> Print, Min(partitionedImage), Max(partitionedImage)
0    4
IDL> cgHistoplot, partitionedImage, /Fill
```

You see a histogram plot of the partitioned image in Figure 10.

Do you see what has happened? All of the image pixels that are less than 0.121 have a value of 0 in the partitioned image. All of the image pixels that have a value between 0.121 and 0.251 have a value of 1, and so forth. All of the pixels with a value greater than 0.817 have a value of 4 in the partitioned image.

We still don’t know which particular pixels in the image have a value of, say, 2, so we can’t scale those pixels yet. But what we do have are integer values which can easily be manipulated by the *Histogram* command in IDL. And the *Histogram* command, by means of a *Reverse_Index* keyword, can identify those number “2” pixels for us!

```
IDL> h = Histogram(partitionedImage, Reverse_Indices=ri)
```

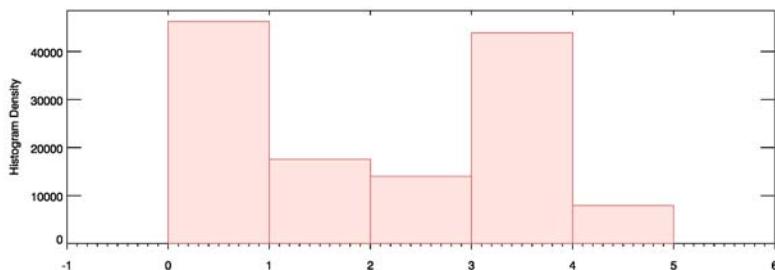


Figure 10: A histogram plot of the partitioned MODIS image.

```
IDL> pix2 = ri[ri[2]:ri[3]-1]
IDL> outImg[pix2] = BytScl(ch1[pix2], Top=50) + 160B
```

The other partitioned image values can be scaled in the same fashion, and as we did when we selected image pixels with the *Where* function.

The *Histogram* command is one of the most highly optimized commands in the IDL language. Partitioning pixels with the *Histogram* command is not only much easier than using repeated *Where* functions, it is several orders of magnitude faster, too.

Understanding Histogram Reverse Indices

One drawback of using the *Histogram* function is that the return value from the *Reverse_Indices* keyword is difficult for users to understand. It is actually simple. The return value consists of two vectors, concatenated end-to-end. JD Smith, in his famous Histogram Tutorial (www.idlcoyote.com/tips/histogram_tutorial.html), calls the first vector the *i-vector* or index vector. It contains indices, one for each bin of the histogram, that locate the starting position of the actual pixel indices that fall into that bin. The starting position is a position in the second vector, which JD calls the *o-vector* or output vector. The *o-vector*, then, contains the actual indices of the pixels that fall into each bin of the histogram.

To see how this works, consider the following vector. We want to know where the 2s are in this vector.

```
IDL> vector = [1,3,0,2,3,1,2,2,4,2]
```

We calculate the histogram of the vector, in the variable *h*, and also return the reverse indices from the *Reverse_Indices* output keyword in the variable *ri*.

```
IDL> h = Histogram(vector, Reverse_Indices=ri)
IDL> Print, h
1 2 4 2 1
```

We see from printing the histogram result (the variable *h*) that there is one 0, two 1s, four 2s, two 3s, and one 4 in the vector. Where are the 2s located? Let's print the *ri* vector.

```
IDL> Print, ri, Format='(16I3)'
6 7 9 13 15 16 2 0 5 3 6 7 9 1 4 8
```

The first five (or, actually, the number of elements in the variable *h*) elements are indices into the *ri* vector, the sixth element is the total number of elements in the *ri* variable. So, the first six (number of elements of *h* plus 1) elements constitute the *i-vector*. The next 10 elements are the indices of the elements in our original vector, but now rearranged by bin number. These 10 numbers are the *o-vector*.

We want to know the indices of the number 2s in the original vector. The starting location of the “2” indices is in the “2” slot in the *ri* vector.

```
IDL> start = ri[2] & Print, start
9
```

The ending location of the “2” indices is one less than the starting location of the “3” indices.

```
IDL> ending = ri[3]-1 & Print, ending
12
```

This tells us that the “2” indices start at index 9 and continue through index 12 in the *ri* vector.

```
IDL> indices_2 = ri[start:ending] & Print, indices_2
3 6 7 9
```

And, in fact, this is correct.

```
IDL> Print, vector[indices_2]
2 2 2 2
```

Normally, these indices are selected like this.

```
IDL> indices_2 = ri[ri[2]:ri[3]-1]
```

The only time this *doesn't* work, is when there are no values for that particular histogram bin. If there are no values in the histogram for a value *n*, then the *ri[n]* value is identical to the *ri[n+1]* value in the *ri* vector. If this happens, the expression above gives you a negative subscript range and an error is thrown.

We prevent that error by writing code like this that checks the *Histogram* result to be sure there are values in that bin. This is usually done in a FOR loop of some kind.

```
IDL> IF h[n] GT 0 THEN $  
      indices_n = ri[ri[n]:ri[n+1]-1]
```

If you like, you can use the `ReverseIndices` utility function to return these indices from a reverse index vector. For example, the indices for the “2” pixels above can be returned like this.

```
IDL> indices = ReverseIndices(ri, 2, Count=count)  
IDL> Help, indices, count  
INDICES      LONG      = Array [4]  
COUNT        LONG      = 4
```

Image Filtering

Images are filtered to smooth images and give a less pixelated appearance, to remove noise, to find the edges and other features of interest in images, or simply to sharpen the features of an image. Linear filter operations can be performed in the spatial domain of the image (i.e., on the image itself) or by transforming the image into the frequency domain by means of the Fast Fourier Transform (FFT) where a linear filter can be applied. We will talk about filtering images in the frequency domain in just a moment, but let’s start with IDL’s spatial filtering capability.

Entire books are written about the spatial filtering of images, so I can’t do justice to the topic in just a few pages. But I can introduce you to some of IDL’s image filtering functions and tools, and I can show you how you can build more sophisticated spatial filters for yourself using these tools. If you are looking for a more advanced and theoretical treatment of these ideas, the Gonzales and Wood book I mentioned on the first page of this chapter has an excellent introduction to this topic.

Image Smoothing

You often hear the terms *high-pass* filter or *low-pass* filter used when people are speaking about image filtering. If you are new to image processing it is sometimes difficult to know what people are talking about. Mostly they are talking about how image values change as you take a path, say, from one side of an image to another. This path is sometimes called an *image profile*.

Consider the mineral image in the IDL *examples* directory.

```

IDL> file = Filepath(SUBDIR=['examples','data'], $  

    'mineral.png')  

IDL> image = Read_PNG(file)  

IDL> Help, image  

IMAGE   BYTE = Array[288,216]

```

To see the image and a column profile across the image, type these commands.

```

IDL> !P.Multi = [0,2,1]  

IDL> cgLoadCT, 0  

IDL> cgImage, image, /Save, /White, /Keep_Aspect  

IDL> cgPlots, !X.CRange, [120,120], Color='red6'  

IDL> cgPlot, image[*,120], Color='red6', XStyle=1, $  

      XTitle='Column Position', YTitle='Image Value'  

IDL> !P.Multi = 0

```

You see the result in Figure 11. The image profile reminds me of the output of one of those pen recorders you sometimes see in a documentary film about the San Andres Fault in California. The pen sort of twitches about, until the big earthquake hits, when it moves wildly in some direction. The twitching of the “pen” as it moves from left to right along the image profile represents the high frequency components of the image. The low frequency components are represented by the overall shape of the curve in the plot. I find it easy to view low frequency components of an image or plot by simply taking off my reading glasses. Everything gets blurred, and all I can really see are general shapes. The general shapes *are* the low frequency components.

Image smoothing is a low-pass filter technique that removes the high frequency components of the image (the twitchy nature of the profile) while allowing the low frequency components to pass through the filter. Another way to say this is that low-pass filtering reduces the disparity between near-by image values. It does this by replacing each image pixel value with a value that depends on the pixel’s neighbors.

The IDL *Smooth* function, for example, performs a “boxcar” smoothing by centering a box with an odd-numbered width at each pixel and taking the average value of each of the pixels in the box. So, if the box is a 3 by 3 box, each pixel value is replaced by the average of it and its eight closest neighbors. If the box is a 5 by 5 box, each pixel value is replaced with the average of it and its 24 closest neighbors, and so on.

You can see the result graphically by smoothing the image with, say, a 9 by 9 box over the image.

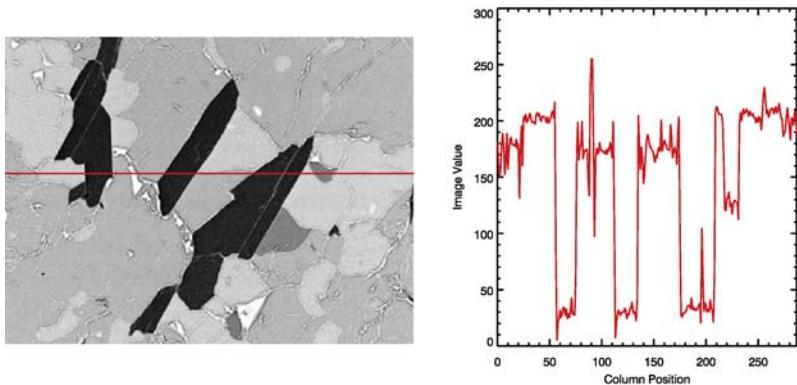


Figure 11: The mineral image with a column profile. The twitchy changes in value are the high frequency components of the image and the general shape of the plot, with its three large dips, reflects the low frequency components.

```
IDL> !P.Multi = [0,2,1]
IDL> simage = Smooth(image, 9)
IDL> cgImage, simage, /Save, /White, /Keep_Aspect
IDL> cgPlotS, !X.CRange, [120,120], Color='red6'
IDL> cgPlot, simage[*,120], Color='red6', XStyle=1, $
      XTitle='Column Position', YTitle='Image Value'
IDL> !P.Multi = 0
```

You see the result in Figure 12. Notice how the general shape of the profile curve is still there, but most of the high frequency components have been smoothed out. This image was chosen to show you that there are still some high frequency changes (quick changes in image value) in the image that cannot be eliminated entirely unless we use a very large smoothing window.

If you look closely at Figure 12 you may notice some pixelation about the edges of the image. This is because as the box is placed on these edge pixels, the box extends off the image. There is a question of what to do with the fewer pixels than normal that are inside the box. How should they be treated by the algorithm? In the default case we have used here, those pixels are just ignored and their values are not changed at all. Thus, these pixels do not get smoothed and that is the effect you are seeing in Figure 12.

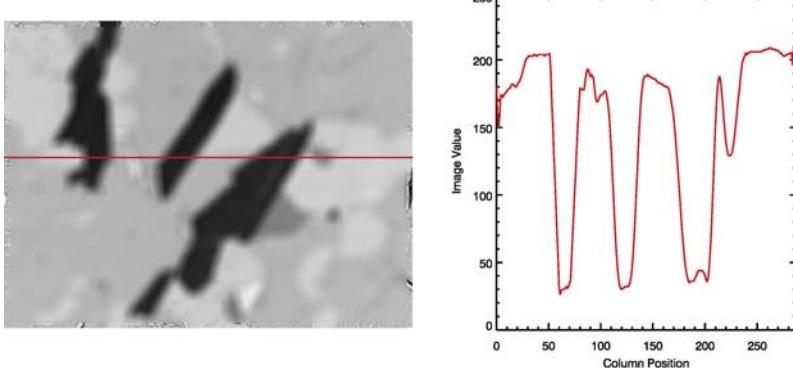


Figure 12: The image in Figure 11 smoothed with a 9 by 9 boxcar average smoothing. The high frequency components of the image have been nearly eliminated by this low-pass filtering technique, leaving only the general shape of the curve.

However, you can set the *Edge_Truncate* keyword to the *Smooth* function and the edge pixels will be replicated for those pixels that are outside the range of the image, but still in the smoothing box. In practice, this keyword is set most of the time.

```
IDL> simage = Smooth(image, 9, /Edge_Truncate)
```

Note that the box (or *kernel*, as you will learn in a moment) does not have to be square. You can set the *width* parameter in the *Smooth* command to an *n*-element array, where *n* is the number of dimensions of the image to be smoothed. Use a 1 for dimensions you don't want to smooth.

For example, to smooth this image with a 9-element smoothing in X, but with no smoothing in Y, you could type these commands.

```
IDL> !P.Multi = [0,2,1]
IDL> simage = Smooth(image, [9,1], /Edge_Truncate)
IDL> cgImage, simage, /Save, /White, /Keep_Aspect
IDL> cgPlotS, !X.CRange, [120,120], Color='red6'
IDL> cgPlot, simage[*,120], Color='red6', XStyle=1, $
      XTitle='Column Position', YTitle='Image Value'
IDL> !P.Multi = 0
```

You see the result in Figure 13.

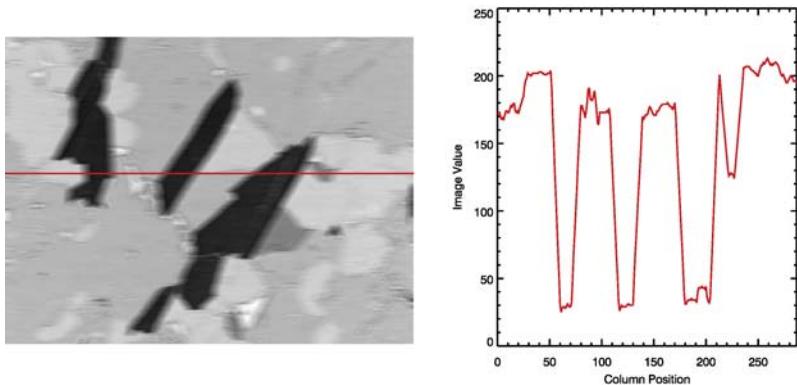


Figure 13: This image has been smoothed in the X direction, but not in the Y direction by using a rectangular smoothing window or kernel.

Occasionally, you want to smooth images that have missing data in them. The *Missing* and *NAN* keywords can be used with the *Smooth* function to handle this kind of data, too.

Filtering with Image Convolution

The *Smooth* function is a specific application of a more general image filtering technique called convolution, which is implemented in IDL with the *Convol* function. The “box” in the case of the *Smooth* function is called the *kernel* in convolution and it can be any rectangular size. The kernel is moved along the image and centered at each pixel. The kernel is then multiplied by the image pixels underlying it, and the center pixel is replaced by the sum of the products. The kernel in a 3 by 3 smoothing operation looks like this.

$$\text{kernel} = \begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

The only trick when doing image convolution is to be sure you are working with something other than integer (or byte) image data. Otherwise your

result is likely to be incomprehensible! Normally, image data is cast to floating point type to do the convolution and then cast to integer or byte at the end of the process to save memory space.

In IDL, you can create this kernel like this.

```
IDL> kernel = Replicate(1.0/9, 0, 3, 3)
```

These following two commands, then, are equivalent.

```
IDL> simage = Smooth(image, 3, /Edge_Truncate)
IDL> simage = Convol(Float(image), kernel, $
/Edge_Truncate)
```

Performing image filtering with the *Convol* function gives you many more options because you have complete control over how you construct the kernel. You will often find kernels constructed for various purposes in the appendix of image processing books. And you will see another application of image convolution in an image sharpening example in just a moment.

High-Pass Filtering

With image convolution it is easy to construct a kernel that will provide high-pass filtering rather than low-pass filtering. Simply give the center of the kernel a large positive value to accentuate that value relative to its neighbors, and give the neighbors small negative values. A 3 by 3 kernel for high-pass filtering might look like this. Such a kernel will accentuate the differences between pixel values and therefore show us the edges of images.

$$\text{kernel} = \begin{bmatrix} -1/9 & -1/9 & -1/9 \\ -1/9 & 8/9 & -1/9 \\ -1/9 & -1/9 & -1/9 \end{bmatrix}$$

To perform high-pass filtering, type code like this.

```
IDL> !P.Multi = [0,2,1]
IDL> kernel = Replicate(-1.0, 3, 3) / 9.0
IDL> kernel[1,1] = 8.0/9.0
IDL> cimage = Convol(Float(image), kernel, Center=1, $
/Edge_Truncate)
```

This kind of high-pass filtering will tend to “center” the image values about 0. You can see this if you plot a histogram of the convolved image. This is the result of removing most low frequency image values.

```
IDL> cgHistoplot, cimage
```

You see the result in Figure 14.

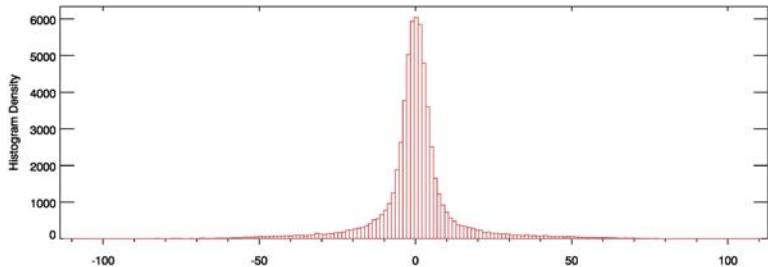


Figure 14: A histogram plot of the convolved image shows that the values get “centered”. This is the result of removing low frequency image values.

This means that you need to byte scale the convolved image in order to display it properly. I usually have a quick look at the image histogram in order to know how to set the *Min* and *Max* keywords to the *BytScl* command for best effect.

```
IDL> simage = BytScl(cimage, Min=-40, Max=40)
IDL> cgImage, simage, /Save, /White, /Keep_Aspect
IDL> cgPlots, !X.CRange, [120,120], Color='red6'
IDL> cgPlot, simage[*,120], Color='red6', XStyle=1, $
      XTitle='Column Position', YTitle='Image Value'
IDL> !P.Multi = 0
```

You see the result in Figure 15.

Image Sharpening

A kernel of this type is sometimes called a Laplacian kernel and it is particularly good at discovering the fine detail in images. In fact, it is often used to sharpen images.

Consider the NASA Galileo mission picture of the moon in the file *galileo_moon.jpg*. It is among the data files you downloaded to use with

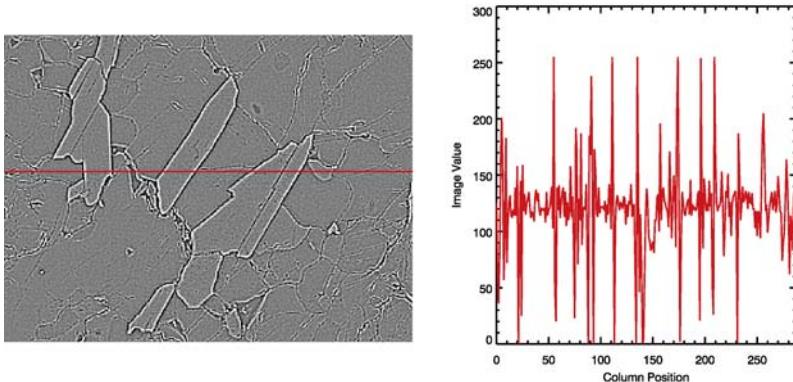


Figure 15: The `Convol` command can perform high-pass filtering as well as low-pass filtering of images. Notice here how all the low frequency values are gone in the image profile.

this book. If you put the file in a directory that is located somewhere on your IDL path, you can open the file like this, using the `Find_Resource_File` function from the [Coyote Library](#).

```
IDL> file = Find_Resource_File('galileo_moon.jpg')
IDL> Read_JPEG, file, moon
IDL> Help, moon
      MOON  BYTE = Array[703, 813]
```

You can see the original moon image beside the image convolved with the Laplacian sharpening kernel by typing these commands. The scaling values for the `BytScl` command were found empirically, by looking at the histogram of the convolved image output.

```
IDL> cgDisplay, 940, 540, Title='NASA Moon Mission'
IDL> !P.Multi = [0,2,1]
IDL> cgLoadCT, 0, /Reverse
IDL> cgImage, moon, /NoInterp
IDL> kernel = [[-1,-1,-1], [-1,8,-1], [-1,-1,-1]] / 9.0
IDL> cimage = Convol(Float(moon), kernel, Center=1, $
      /Edge_Truncate)
IDL> simage = BytScl(cimage, Min=-10, Max=10)
IDL> cgImage, simage, /NoInterp
IDL> !P.Multi = 0
```

You see the result in Figure 16.

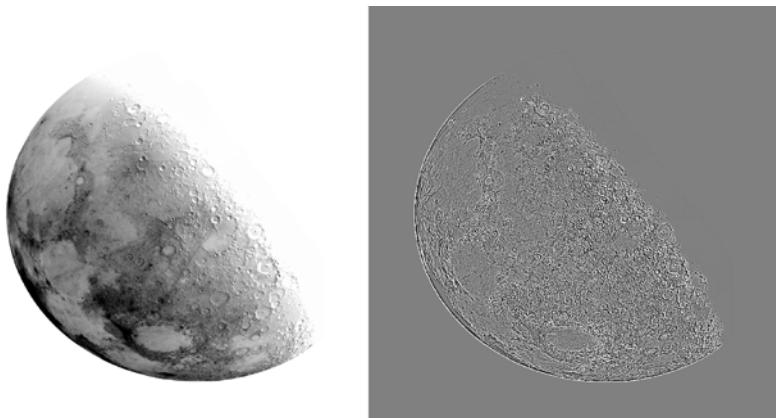


Figure 16: The original moon image on the left and the image convolved with a Laplacian sharpening kernel on the right.

To sharpen the image, you must add the convolved image (or a portion of it) to the original image, then scale the image into the range 0 to 255. Here is the result with its accompanying image histogram.

```
IDL> sharpImage = BytScl(Float(moon) + (simage/4.0))
IDL> cgImage, sharpImage, /White
IDL> cgHistoplot, sharpImage
```

You see the result in Figure 17.

The sharp peaks in the histogram between about 20 and 30 are due mostly to background pixels. These can be removed by scaling the final sharpened image with a minimum value of 30. The code here displays the final image next to the original image, so you can see the difference.

```
IDL> cgDisplay, 940, 540, Title='NASA Moon Mission'
IDL> !P.Multi=[0,2,1]
IDL> cgImage, moon, /White
IDL> cgImage, sharpImage, /Scale, MinValue=30
```

You see the result in Figure 18.

The [Coyote Library](#) routine [Sharpen](#) can be used to sharpen any 2D image using this method. If you set the *Display* keyword, you will see a display of all the intervening steps in creating the Laplacian filtered image. You can pass your own kernel to the program with the *Kernel* keyword. Here is the program working with the Hurricane Gilbert image in the IDL examples directory.

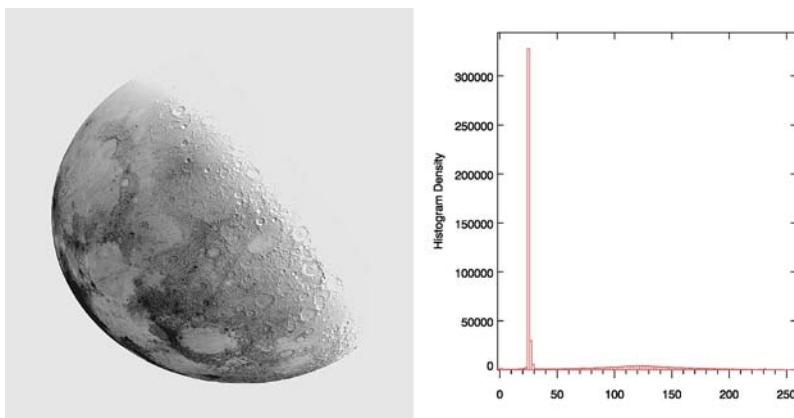


Figure 17: The original image with the convolved image added back to it is on the left. On the right is the histogram of the result. The sharp peaks between 20 and 30 are from the background pixels.



Figure 18: The original image on the left and the Laplacian sharpened image on the right.

```
IDL> sharpenImage = Sharpen(LoadData(13), /Display)
```

You see the result in Figure 19.

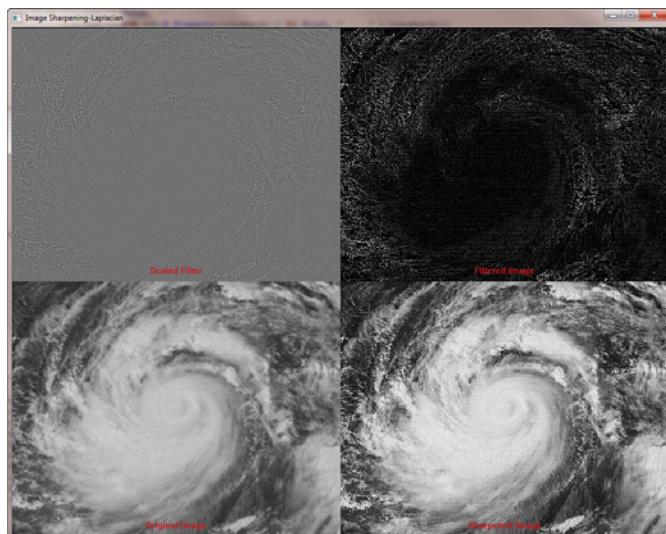


Figure 19: The **Sharpen** program can perform Laplacian sharpening on any 2D image.

Unsharp Masking

Another type of image sharpening is called *unsharp masking*. In this case, a smoothed (unsharp) image is subtracted from the original image to create the mask. Then, this mask is added back to the original image. If the entire mask is added back to the image, this is called unsharp masking. If just a portion of the mask is added back to the original image, this is called *high boost filtering*. Unsharp masking has been used extensively in the printing industry to improve the look of type on images, since it tends to enhance and smooth the edges of type faces.

Consider the mineral image we were working with previously.

```
IDL> file = Filepath(SUBDIR=['examples','data'], $  
           'mineral.png')  
IDL> image = Read_PNG(file)  
IDL> Help, image  
IMAGE      BYTE = Array[288,216]
```

Here is how you can show the original image next to the unsharp masked image and a 50 percent high boost filtered image.

```
IDL> cgDisplay, 288*3, 216  
IDL> cgLoadCT, 0
```

```
IDL> !P.Multi = [0,3,1]
IDL> cgImage, image
IDL> mask = Float(image) - Smooth(Float(image), 3)
IDL> cgImage, image + mask, /Scale
IDL> cgImage, image + (0.5 * mask), /Scale
IDL> !P.Multi = 0
```

You see the result in Figure 20.

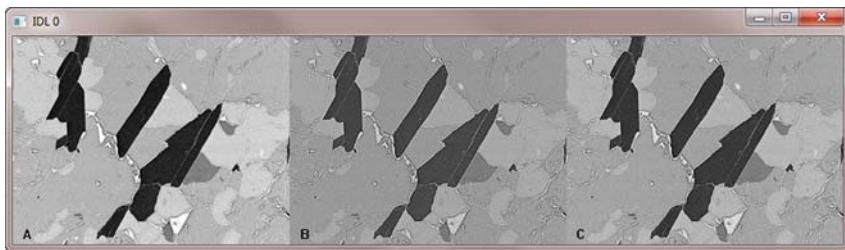


Figure 20: (A) The original image. (B) The unsharp masked image. (C) A image with a 50% high boost filter applied.

Image Filtering in the Frequency Domain

Linear filters of the sort we are using here so far in the spatial domain can also be applied to an image in the frequency domain. There are three basics steps to frequency domain filtering.

1. The image must be transformed from the spatial domain to the frequency domain using the Fast Fourier Transform (FFT) function.
2. The resulting complex image must be multiplied by a filter (that usually only has real values).
3. The filtered image must be transformed back to the spatial domain for viewing.

These steps are implemented with the IDL *FFT* function. A -1 argument causes the image to be transformed from the spatial domain to the frequency domain, and a 1 argument does the opposite. In general, then, frequency domain filtering is implemented like this.

```
filteredImage = FFT(FFT(image, -1) * filter), 1)
```

Building a Butterworth Low-Pass Filter

IDL makes it easy to build many types of image filters with its array-oriented operators and functions. Many common filters take advantage of what is called a frequency image or Euclidean distance map. A Euclidean distance map for a two-dimensional image is an array the same size as the image, in which each pixel in the distance map is given a value that is equal to its distance from the nearest corner of the two-dimensional array. The *Dist* function is used to create this Euclidean distance map in IDL.

You can view the Euclidean distance map as a surface. Here is a distance map for a 41x41 image.

```
IDL> cgSurf, Dist(41,41)
```

You see the result in Figure 21.

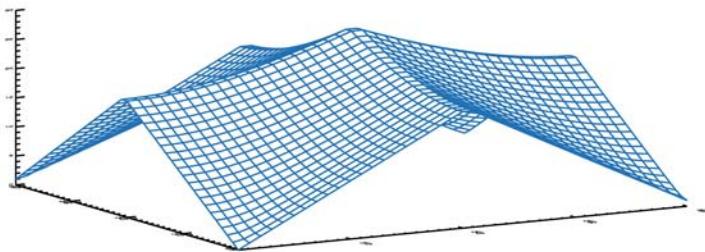


Figure 21: An Euclidean distance map made with the *Dist* function and used in frequency filtering.

A common frequency filter is a Butterworth frequency filter. The general form of a low-pass or smoothing filter is given by the following equation:

$$\frac{1}{1 + c \left(\frac{R}{R_0}\right)^{2n}}$$

In this equation, c is a constant that defines the magnitude of the filter at the point where $R = R_0$ and is either set to 1 or to the square root of 2 (0.414). R is the Euclidean distance image. R_0 is the nominal filter cutoff frequency and represents a pixel distance from the origin of the filter to a point where the filter is equal to approximately half power. It is normally

set in the range of 5-15 for most low-pass filters. And n is the order of the Butterworth filter and is usually set to 1.

To see how this works, let's use the Earth mantle convection image from the IDL examples directory.

```
IDL> convec = cgDemoData(11)
IDL> Help, convec
CONVEC    BYTE = Array[248, 248]
```

The first step in building an image filter is to transform the image into the frequency domain with the *FFT* function.

```
IDL> freqImage = FFT(convec, -1)
```

Viewing the Power Spectrum

Viewing the frequency image is not normally instructive, but it is sometimes useful to view the *power spectrum* of the frequency image. The power spectrum is a plot (usually an image) of the magnitude of the frequency components of the frequency image. Each frequency component is represented at a different distance from the center of the plot or image. The directions of the frequencies in the power spectrum represent the different orientations of image features in the original image. We are using a symmetrical image, so we are not concerned with power spectrum directional properties here.

The power at each location in the power spectrum shows how much of that frequency and orientation is present in the image. The power spectrum is particularly useful for isolating periodic structure or noise in an image. The power spectrum magnitude is usually represented on a log scale, since power can vary dramatically from one frequency to the next.

Normally, the origin of power spectrum plot or image is placed in the center of the plot. We do this in IDL by shifting the power spectrum image by half its width in both the X and Y directions. Here is code that displays the original image next to its power spectrum.

```
IDL> LoadCT, 33
IDL> cgDisplay, 248*2, 248
IDL> !P.Multi = [0,2,1]
IDL> cgImage, convec
IDL> power = Shift(Alog(Abs(freqImage)), 124, 124)
IDL> cgImage, LogScl(power)
IDL> !P.Multi = 0
```

You see the result in Figure 22.

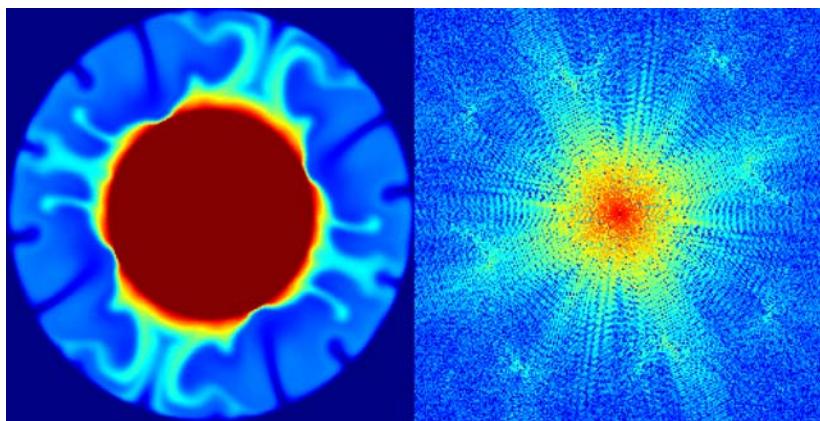


Figure 22: The original convection image on the left and the power spectrum of the frequency image on the right. The highest frequencies are in the center of the power spectrum.

Note: A new `Center` keyword was added to the `FFT` function in IDL 7.1. If this keyword is set, the shifting that goes on to properly view the power spectrum, or to create filters later in the chapter, is done automatically by the `FFT` function. I continue to use the `Shift` function here because not everyone has IDL 7.1 or higher and because the `Center` keyword has been reported to work incorrectly in IDL 8.0.1. But be advised, it might also be possible to create the power spectrum like this when the keyword is finally working properly.

```
freqImage = FFT(convec, -1, /Center)
power = Alog(Abs(freqImage))
```

The next step is to construct the frequency filter. The filter will be constructed using double precision values, and we will use a cut-off frequency of 10 in this filter. It may be helpful to think of the cut-off value as a pixel distance from the center of the power spectrum. As you move further from the center of the power spectrum you are allowing more and more frequencies or power to pass through the filter. You should experiment with the filter cut-off to see how the filter works. Here are some commands to show you what the filter looks like with a cut-off of 10 and a cut-off of 50.

```
IDL> cgLoadCT, 4, /Brewer, /Reverse
IDL> !P.Multi = [0,2,1]
IDL> r = Dist(248,248)
IDL> r0 = 10.0
```

```

IDL> filter = 1.0D / (1.0D + r / r0)^2
IDL> cgSurf, Shift(filter, 124, 124), /Shaded
IDL> r0 = 50.0
IDL> bigfilter = 1.0D / (1.0D + r / r0)^2
IDL> cgSurf, Shift(bigfilter, 124, 124), /Shaded
IDL> !P.Multi = 0

```

You see the result in Figure 23.

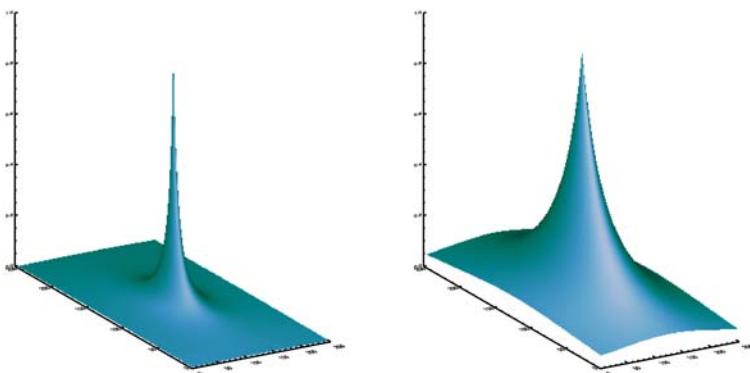


Figure 23: A low-pass Butterworth filter with a cut-off of 10 on the left and with a cut-off of 50 on the right. The wider the filter the more frequencies passed by the filter.

The next step is to apply the filter and transform the frequency image with the filter applied back to the spatial domain where it can be viewed. Here is code to apply the filter and look at the filtered image next to its power spectrum. Note that the filtered image will be returned as a complex array. The amplitude of the image is in the real part of the complex array. The phase is in the imaginary part of the complex array. We are only interested in the amplitude here, so we extract the real part of the filtered image.

```

IDL> filtered = Real_Part(FFT(freqImage * filter, 1))
IDL> LoadCT, 33
IDL> !P.Multi = [0,2,1]
IDL> cgImage, filtered, /Scale
IDL> ffreqImage = FFT(filtered, -1)
IDL> fpower = Shift(Alog(Abs(ffreqImage)), 124, 124)
IDL> cgImage, LogScl(fpower)
IDL> !P.Multi = 0

```

You see the result in Figure 24.

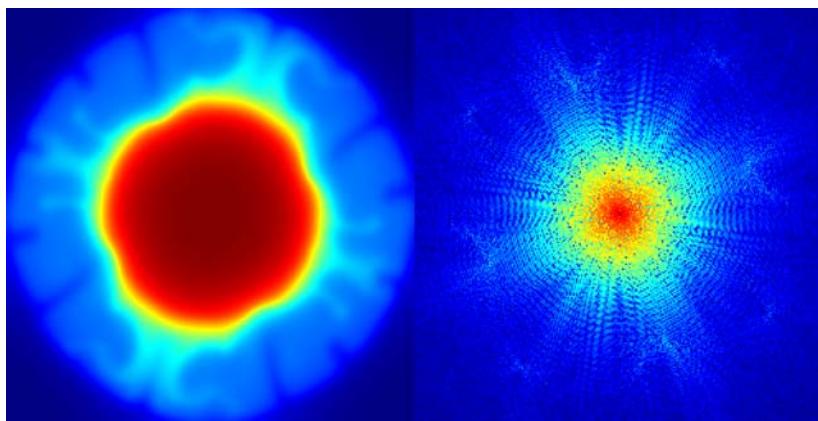


Figure 24: The image and its power spectrum filtered with a Butterworth low-pass filter. You see far fewer frequencies in the power spectrum after filtering.

Building a Gaussian High-Pass Filter

A Gaussian high-pass filter is given by this equation, in which R is the Euclidian distance function, and R_0 is the frequency cut-off:

$$\frac{e^{-\frac{R^2}{(2R_0^2)}}}{1 - e}$$

Let's consider the galaxy image in the IDL *examples* directory for this high pass filter. In high pass filtering, areas of constant intensity in the input image will be zero in the output image. These zero areas will be a uniform color in the middle of the color range when the output image is scaled. Areas of pronounced intensity change in the input image will have negative or positive values in the output image. Thus, a high-pass filter will tend to “dampen” the contrast intensity of low-frequency values in the input image and enhance the intensity of high-frequency values.

```
IDL> image = cgDemoData(4)
IDL> Help, image
IMAGE  BYTE = Array[256,256]
```

In IDL, the filter for this image can be constructed like this.

```
IDL> s = Size(image, /Dimensions)
IDL> r = Dist(s[0],s[1])
IDL> r0 = 10.0
```

```
IDL> filter = 1.0D - Exp(-r^2/(2.0D*r^2))
IDL> cgSurf, Shift(filter, s[0]/2, s[1]/2), /Shaded
```

You see what this Gaussian high-pass filter looks like in Figure 25.

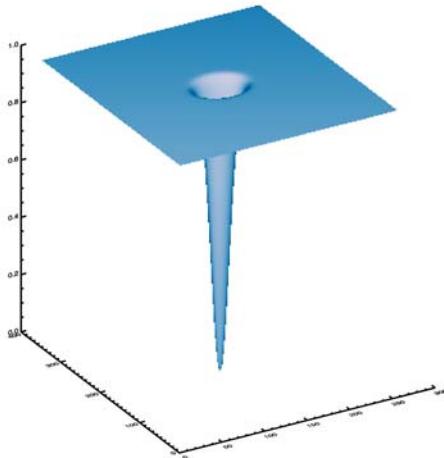


Figure 25: A Gaussian high-pass filter of width 10.

As before, we apply the filter to the frequency image and back transform to view the filtered image. Here is the code to perform the filtering operation and display the original image next to the high-pass filtered image. Notice how the bright area in the very center of this image is reduced in intensity by the filter.

```
IDL> freqImage = FFT(image, -1)
IDL> filtered = Real_Part(FFT(freqImage * filter, 1))
IDL> !P.Multi=[0,2,1]
IDL> cgLoadCT, 0, /Reverse
IDL> cgDisplay, s[0]*2, s[1]
IDL> cgImage, image
IDL> cgImage, filtered, /Scale
IDL> !P.Multi = 0
```

You see the result in Figure 26.

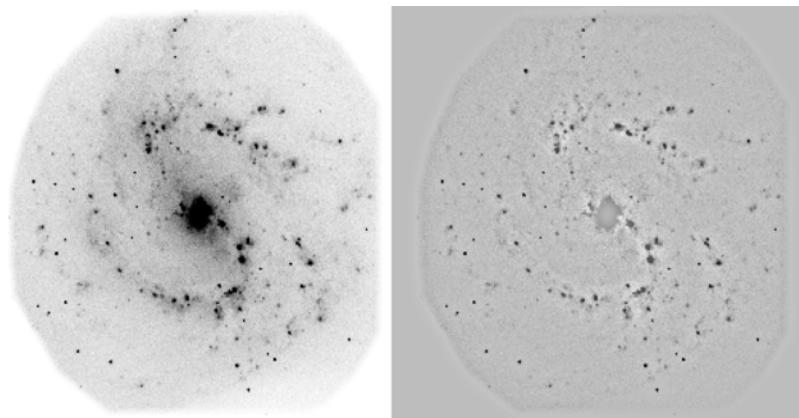


Figure 26: The original image on the left, and the Gaussian high-pass filtered image on the right. Note that areas of constant intensity in the input image are filtered to neutral grays in the output image.

Image Edge Enhancement

We have already discussed how a Laplacian kernel convolved with the image can be used to sharpen an image. There are other kernels that can be used to perform similar edge finding or edge enhancing operations on images. These are often called *gradient operators* since they are used to enhance the first derivative, or rate of change, of image values. The Laplacian kernel we used before is a second derivative operator.

When we speak of the “rate of change” and “derivatives”, we are, of course, referring to how pixel values change as one traverses the image in a particular dimension. Values that change dramatically in a short distance (several pixels) typically represent the edge of a feature in the image. It is these areas that edge operators try to enhance or identify in an image.

Nearly any high-pass filter will find image edges, but two common edge enhancement operators are implemented in IDL. These are the Sobel and Roberts edge enhancement operators, implemented in IDL as the *Sobel* and *Roberts* commands. These functions convolve directional filters with the image in both the X and Y directions, then take the magnitude of the result. (This is the definition of the gradient and is why they are called gradient filters.) The Sobel and Roberts operators differ only in the kernel that is convolved with the image.

Consider the mineral image we worked with previously in this chapter.

```
IDL> file = Filepath(SUBDIR=['examples', 'data'], $  
IDL>      'mineral.png')  
IDL> image = Read_PNG(file)  
IDL> Help, image  
IMAGE    BYTE = Array[288, 216]
```

Here is how you can display the original image next to a Sobel and Roberts edge enhanced image. Note that the result of these two functions will *not* be byte scaled data. You will need to byte scale the edge enhanced results for display.

```
IDL> s = Size(image, /Dimensions)  
IDL> cgDisplay, s[0]*3, s[1]  
IDL> cgLoadCT, 0, /Reverse  
IDL> !P.Multi = [0,3,1]  
IDL> cgImage, image  
IDL> cgImage, Sobel(image), /Scale  
IDL> cgImage, Roberts(image), /Scale  
IDL> !P.Multi = 0
```

You see the result in Figure 27.

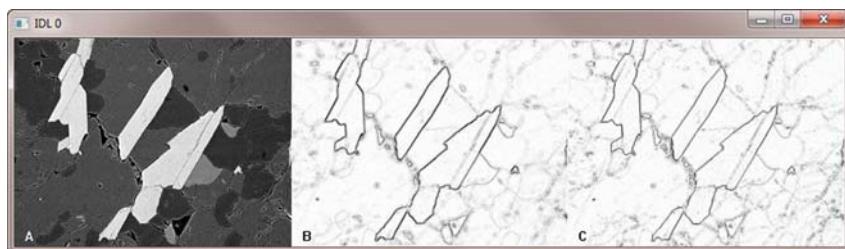


Figure 27: (A) The original image. (B) The Sobel edge enhanced image. (C) The Roberts edge enhanced image.

Image Noise Reduction

A common image processing technique is to remove noise from an image. Noise can accumulate from many sources and often acts to degrade the image. One common form of noise is salt and pepper noise, in which random pixels throughout the image have extreme values. Cosmic rays, for example, passing through a CCD camera during image capture can cause this kind of random pixel noise.

To see how this works, let's create an image with random salt and pepper noise. (Such noise is sometimes called *speckle noise*.) Let's use the image of David Stern, the creator of IDL, as an example. Type these commands to create 3600 random pixels with salt and pepper noise.

```
IDL> stern = (cgDemoData(10)) [*,* ,1]
IDL> s = Size(stern, /Dimensions)
IDL> noisy = stern
IDL> blackPts = RandomU(seed, 1800) * s[0] * s[1]
IDL> whitePts = RandomU(seed, 1800) * s[0] * s[1]
IDL> noisy[blackPts] = 0
IDL> noisy[whitePts] = 255
```

One effective noise reduction filter is the median filter, implemented in IDL as the *Median* command. The median filter is an example of a non-linear filter than can only be applied in the spatial domain of an image. It is similar to the *Smooth* function in that you give the filter a *width* parameter that describes the width of a square window of neighboring pixels.

However, unlike the *Smooth* function which replaces the center pixel with the average value of pixels within the neighborhood, the *Median* function replaces the center pixel with the median value of the pixels within the neighborhood. The median value does not give undue weight to extreme values in the neighborhood the way the average value does, and will better represent the “true” value of the image pixel.

Type these commands to see the original image, the noisy image we created, and the noisy image filtered with a 3 by 3 median filter.

```
IDL> cgDisplay, s[0]*3, s[1]
IDL> !P.Multi = [0,3,1]
IDL> cgImage, stern
IDL> cgImage, noisy
IDL> cgImage, Median(noisy, 3)
IDL> !P.Multi = 0
```

You see the result in Figure 28.

A program that is similar to the *Median* command is the Lee filter, implemented in IDL as the *LeeFilt* command. The Lee filter changes the central image pixel according to the variance of the pixel neighborhood. The Lee filter is often used with radar images to remove speckle noise while preserving edges in the image.



Figure 28: The original image on the left, the image with salt and pepper noise added in the middle, and the image filtered with a 3 by 3 median filter on the right.

Type these commands to use a 3 by 3 neighborhood and produce the images in Figure 29. It looks almost identical to the previous example using the *Median* command.

```
IDL> cgDisplay, s[0]*3, s[1]
IDL> !P.Multi = [0,3,1]
IDL> cgImage, stern
IDL> cgImage, noisy
IDL> cgImage, LeeFilt(noisy, 1)
IDL> !P.Multi = 0
```

Note that the *width* parameter in the *LeeFilt* command is specified differently than it is for the *Smooth* and *Median* commands. The *width* here is multiplied by 2 and 1 is added to it to produce the same width used by the other commands.

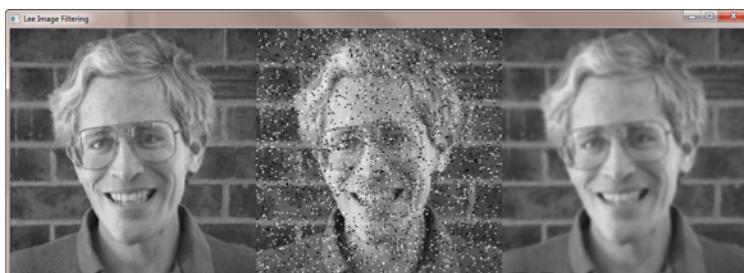


Figure 29: The *Lee* filter is an adaptive filter that uses local statistics to filter an image. It is similar to a median filter, but does a better job of preserving high-frequency components of the image.

Noise Reduction in the Frequency Domain

Noise reduction can also be done in the frequency domain. Consider a rather noisy image from a gated blood pool study.

```
IDL> blood = (cgDemoData(6)) [*,* ,0]
```

Let's look at this image next to its power spectrum.

```
IDL> s = Size(blood, /Dimensions)
IDL> cgDisplay, s[0]*8, s[1]*4
IDL> cgLoadCT, 13, /Brewer, /Reverse
IDL> !P.Multi = [0,2,1]
IDL> cgImage, blood
IDL> freqImage = FFT(blood, -1)
IDL> power = Shift(Alog(Abs(freqImage)), s[0]/2, s[1]/2)
IDL> cgImage, LogScl(power)
```

You see the result in Figure 30.

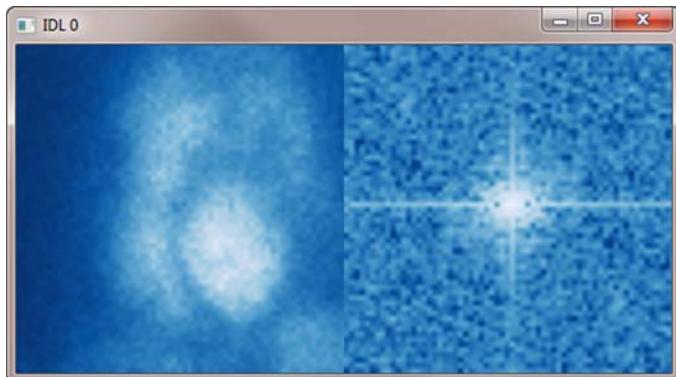


Figure 30: The original gated blood pool study image is on the left and its power spectrum is on the right.

It is instructive to see the power spectrum displayed as a surface, but first let's scale the power spectrum so that all of its values are positive.

```
IDL> power = power + Abs(Min(power))
IDL> cgSurface, power, /Shaded
```

You see the result in Figure 31. Note that with the exception of the central peak, most of the peaks in the power spectrum fall below the value of six. It is these peaks that contain most of the noise in the image.

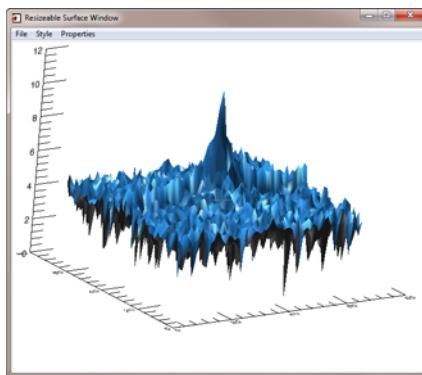


Figure 31: The power spectrum displayed as a surface after scaling the power spectrum so all values are positive.

The idea here is that we can create a frequency image mask that will allow peaks (or frequencies) above a certain threshold to pass, while suppressing frequencies below a particular threshold. This is really no different from the frequency filtering we did earlier in this chapter; we are just creating the frequency filter in a different way. The method we are using here is called *frequency windowing*.

The mask will have a value of 1 above a certain cut-off and a value of 0 everywhere else. The cut-off we will use is the value 6 in this power spectrum.

```
IDL> mask = power GT 6.0
```

The mask, since it was created from the shifted power spectrum, has to be shifted back to its original shape before it is applied as a filter to the frequency image. Here is code to display the mask and the filtered image side by side.

```
IDL> filter = Shift(mask, -s[0]/2, -s[1]/2)
IDL> filteredImg = FFT(freqImage * filter, 1)
IDL> filteredImg = Real_Part(filteredImg)
IDL> cgDisplay, 512, 256
IDL> !P.Multi = [0,2,1]
IDL> cgSurf, mask, /Shaded, RotX=60, XStyle=1
IDL> cgImage, filteredImg, /Scale
IDL> !P.Multi = 0
```

You see the result in Figure 32.

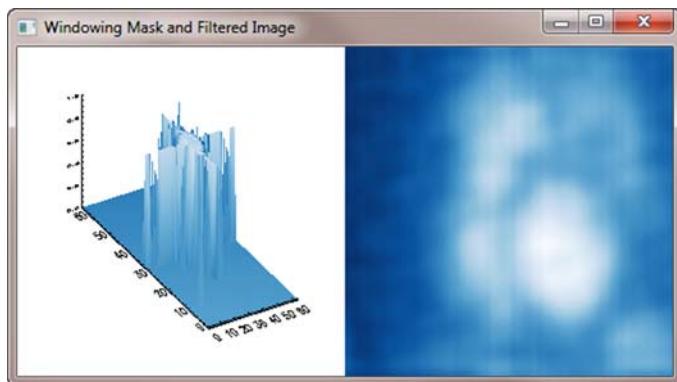


Figure 32: The windowing mask used to filter the image is on the left and the filtered image is on the right.

You can do a similar kind of noise reduction filtering with the *Hanning* command in IDL, which is designed to reduce the high frequency components of an image. The *Hanning* command can implement a hanning filter by setting the *Alpha* keyword to 0.5 (the default) or a hamming filter by setting the *Alpha* keyword to 0.54. The function creates a frequency filter that must be applied in the frequency domain.

The odd thing about the *Hanning* command is that it creates a filter that must be applied to a shifted frequency image, which means the filtered image must be shifted in the opposite direction. I always find this hard to remember, so I tend to shift the hanning filter so it applies directly to the frequency image.

Here are commands to display this blood study image with hanning and hamming filtering side by side.

```
IDL> cgDisplay, 512, 256
IDL> !P.Multi = [0,2,1]
IDL> hanning = Shift(Hanning(s[0], s[1], Alpha=0.50), $
   -s[0]/2, -s[1]/2)
IDL> filteredImg = FFT(freqImage * hanning, 1)
IDL> filteredImg = Real_Part(filteredImg)
IDL> cgImage, filteredImg, /Scale
IDL> hamming = Shift(Hanning(s[0], s[1], Alpha=0.54), $
   -s[0]/2, -s[1]/2)
IDL> filteredImg = FFT(freqImage * hamming, 1)
IDL> filteredImg = Real_Part(filteredImg)
IDL> cgImage, filteredImg, /Scale
IDL> !P.Multi = 0
```

You see the result in Figure 33.

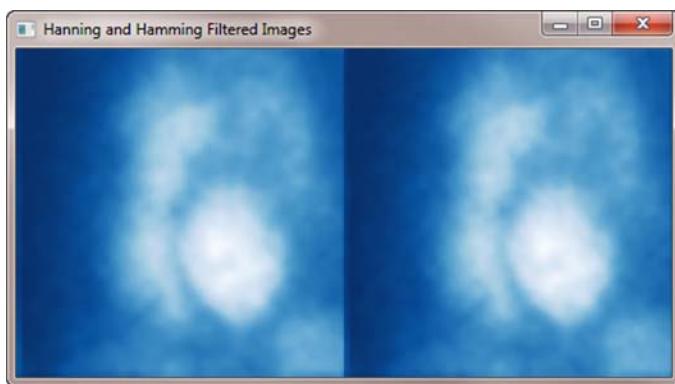


Figure 33: A hanning filtered image on the left and a hamming filtered image on the right.

You can see the effect of a hanning or hamming filter by examining the power spectrum of the filtered image. For example, here is code that will display the power spectrum of the hamming filtered image above.

```
IDL> freqImage = FFT(filteredImage, -1)
IDL> power = Shift(Alog(Abs(freqImage)), s[0]/2, s[1]/2)
IDL> power = power + Abs(Min(power))
IDL> cgSurface, power, /Shaded
```

You see the result in Figure 34. Compare this power spectrum surface to the power spectrum surface in Figure 31.

Regions of Interest in Images

It is common in image applications to define and analyze some feature or region in an image. These features are often called *regions of interest* or ROIs. Often the term *segmentation* is used, too. There are many ways to define regions of interest, but in this section I will introduce you to some ways you can create and analyze ROIs both interactively and programmatically.

Let's consider the mineral image we used previously in this chapter. This is a byte scaled image with values ranging from 0 to 255.

```
IDL> file = Filepath(Subdirectory=['examples','data'], $
'mineral.png')
```

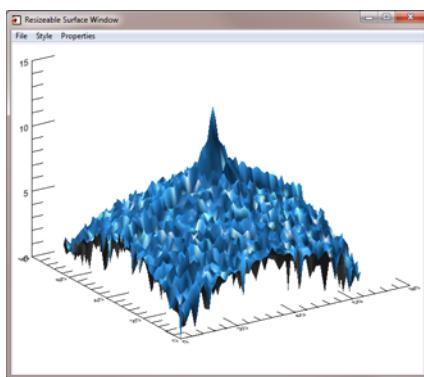


Figure 34: The power spectrum of the hamming filtered image.

```
IDL> image = Read_PNG(file)
```

As shown in Figure 35, there are at least three prominent mineral sections in this image, displayed in dark colors, that it would be reasonable to want to select and characterize.

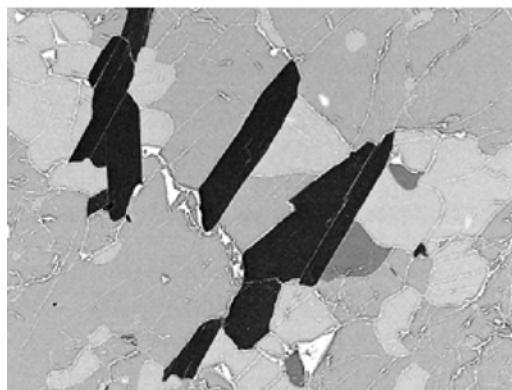


Figure 35: The three dark minerals are candidates for analyzing as regions of interest.

Image Preparation

Most images require some kind of preparation or pre-processing before an ROI can be selected. Quite often this involves a contrast enhancement or smoothing of the sort discussed earlier in this chapter. My general

approach is to view an image like this with the interactive stretching tool, [XStretch](#), where I can view the image histogram while stretching the image in various ways. My goal is to isolate the ROIs visually as much as possible with a contrast stretch, so I can segment them by parameters obtained from [XStretch](#).

These particular minerals are fairly easy to isolate with a linear contrast stretch, as shown in Figure 36. Once I have isolated the image, I either make note of the scaling parameters (if I want to continue programmatically) or I save the stretched image to the IDL command line (from the *Controls* menu), where I can work with it as a local variable.

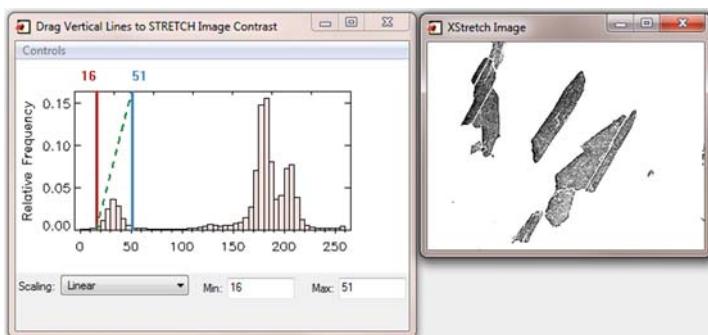


Figure 36: The first step is generally some kind of pre-processing to isolate the ROI from its background. Here a simple contrast stretch does the job nicely.

In this case, I'll note the minimum stretch value is 16 and the maximum is 51 and I will scale the image accordingly.

```
IDL> workImg = BytScl(image, Min=16, Max=51)
```

Image Thresholding

I still have a grayscale image here (values between 0 and 255), and what I would prefer is a bi-level image with just two values: a 0 for the background pixels and a 1 for the ROIs I am interested in working with. I can obtain such a bi-level image by *image thresholding* with IDL boolean operators. Such an image is also called a *binary* image.

In this case, I want any value in the image less than 255 to have a value of 1, and any pixel having a value of 255 to have a value of 0.

```
IDL> bilevel = workImg LT 255
IDL> cgImage, bilevel, /Scale, /Keep_Aspect
```

You see the result in Figure 37.



Figure 37: A binary or bi-level image created by image thresholding.

Another way I can create the bi-level image is to use Boolean operators directly on the original image.

```
IDL> bilevel = image GE 16 AND image LE 51
```

This is going in the right direction, but if you look carefully, you can find small white areas around the image edges and, of course, there are quite a few black pixels in the interior of the minerals I want to select. I would like to clean this image up even more by eliminating the tiny areas of both black and white pixels that I am not interested in.

Using Morphological Operators

Processing digital images with morphological operators is a standard way of cleaning up images of the type we have now. Mathematical morphology is a method of processing images based on shape. IDL comes with the standard erosion (*Erode*) and dilation (*Dilate*) operators, and a number other routines that are based on the application of erosion and dilation to produce desired effects. These commands are described briefly below.

Morphological operators are non-linear and so don't have some of the undesired effects of linear operators, such as smoothing effects at the edges of features. This makes them ideal for cleaning operations. Morphological operators can be considered non-linear convolution operators, in that a structuring element (the equivalent of the convolution kernel) applies a mathematical operation to each pixel as the structuring element is moved over the image. The mathematical operation acts to either minimize or maximize the value underlying the structuring element. The structuring element is discussed in more detail in just a moment.

Erode Erosion replaces each pixel by the minimum of all pixels under the structuring element. This will erode border pixels and shrink the perimeter of image features. It will also cause small objects (in terms of the size of the structuring element) to be deleted.

Dilate Dilation is the complement of erosion. It replaces each pixel by the maximum of all pixels under the structuring element. With binary images, dilation adds pixels to the perimeter of image features and connects small disconnected areas in the image.

Morph_Close This operator simply implements a dilation followed by an erosion. The idea is to close small holes or gaps in the image, without changing the size of the original image features.

Morph_Open This operator simply implements an erosion followed by a dilation. The idea is to remove small features in the image without changing the size of the original image features.

Morph_Open and *Morph_Close* are idempotent, non-linear smoothing operators. (Idempotent means that when a binary operation, like multiplication, is applied to itself, it equals itself.) In contrast to linear filtering operations, repeated application of these operators does not cause further change in the image.

Morph_Distance This operator allows the user to specify a particular distance, N , in a feature. It is as if the feature was eroded with a disk of radius N . Another way of thinking about this is to say that *Morph_Distance* is the sum of all erosions with a circular structuring element of increasing radius, out to the distance N . This operator forms the basis for the partitioning of connected objects with the *Watershed* operator, discussed below.

Morph_Gradient This operator is a non-linear gradient operator (essentially an edge enhancement operator). It subtracts an eroded image from the same dilated image to produce an “edge” the width of the structuring element. It mimics the “difference”

used in a normal gradient operation.

Morph_HitOrMiss This operator is a set operator that works on two results. First, a “hit” operator is used to identify pixels that are “hit” or selected. Then a “miss” operator is used to select pixels that are “missed” by the operation. The result is a set containing pixels that are found in both the “hit” and the “miss” operations. You might use this operator, for example, to identify features that were bigger than 4 pixels in size, but less than 10 pixels in size.

Morph_Thin This operator performs a “thinning” operation on binary images. Thinning is performed by first performing a hit or miss operator to the original image and then subtracting that result from the original image.

Morph_Tophat This operator applies an opening operator (smoothing) to the original image, and then subtracts the result from the original image. It is used to identify the brightest pixels (those sticking up like a top-hat) in an image. The structuring element should be larger than the features to be detected in the image.

Watershed This operator represents a sequential thinning operation. If we assume light intensities are high and dark intensities are low, this routine calculates “watersheds” or “basins” in the image. A watershed has its common meaning of a boundary in which a drop of water placed anywhere inside the boundary will run downhill to a common drainage location. A watershed is the boundary and the basin is the area enclosed by a watershed. The basins in a watershed image are labeled sequentially by number, as they are in the *Label_Region* command, which is described shortly.

There is no “right” way for morphological analysis to proceed. Typically, based on what you know about the image, you “noodle around” at the IDL command line and try various structuring elements and morphological

operators until you find something that gives you the desired result. In this case, I wanted to eliminate small regions of both black and white pixels.

I decided to try first to eliminate the “holes” in my ROIs by using *Morph_Close* with a 3 by 3 structuring element of all 1s. Then, I wanted to eliminate small regions in my image, so I was only dealing with ROIs of reasonable size. I decided to use a 5 by 5 structuring element of 1s with *Morph_Open* to achieve this goal. Here is the code I used, shown in a reversed color table for better clarity.

```
IDL> cgLoadCT, 0, /Reverse
IDL> s = Size(image, /Dimensions)
IDL> cgDisplay, s[0]*3, s[1]
IDL> !P.Multi = [0,3,1]
IDL> cgImage, bilevel, /Scale, Multimargin = 1
IDL> closeImg = Morph_Close(bilevel, Replicate(1,3,3))
IDL> cgImage, closeImg, /Scale, Multimargin = 1
IDL> openImg = Morph_Open(closeImg, Replicate(1,5,5))
IDL> cgImage, openImg, /Scale, Multimargin = 1
IDL> !P.Multi=0
```

You see the result in Figure 38.

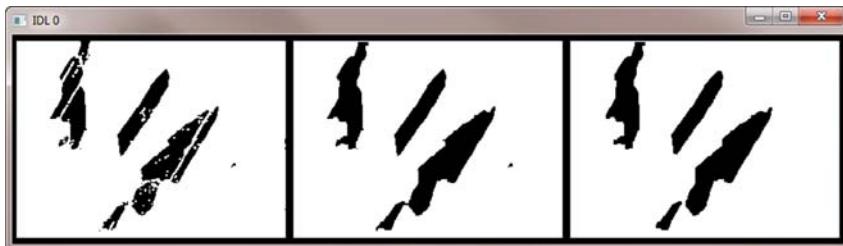


Figure 38: The original bi-level image is on the left. The image transformed with *Morph_Close* to eliminate gaps and holes is in the middle. The image transformed with *Morph_Open* to eliminate small, uninteresting artifacts is on the right.

Identifying Image Features

The next step is to identify the different “objects” or “features” in the image. Sometimes image features are called “blobs” because of the way they appear in a bi-level image. Because I have done a good job cleaning this image up and separating the blobs or features, I can use *Label_Region* for this purpose.

The *Label_Region* function consecutively labels all the connected regions or blobs in a bi-level image with an unique long integer value. Sometimes this process is called “blob coloring.” A region is a set of connected non-zero pixels, where the pixel neighborhood searched for connections is either the four neighbors contiguous to the fours sides of a pixel (the default), or the eight neighbors actually touching a pixel (selected by setting the *All_Neighbors* keyword).

Looking at the right-hand image in Figure 38, I would expect *Label_Region* to find four regions, labeled 1 to 4. But be careful, *Label_Region* is extremely thorough and it might find more regions than you expect. Especially if you did not do a particularly good job of image cleaning. Small regions of a few pixels, or small areas, are often found by *Label_Region* and then eliminated in subsequent processing steps as not worthy of attention. You have to examine the results closely and decide what is best for your application.

Potential Problem with *Label_Region*

There is one potential problem with *Label_Region* you need to know about. *Label_Region* gets things very wrong when a feature is adjacent to an image edge. If there is any chance that an image feature is touching an image edge (and there is *always* the chance, no matter what you think now!), then you need to put the image you pass to *Label_Region* into a slightly bigger image with a one pixel border around the real image. The code looks like this.

```
IDL> bigImage = BytArr(s[0]+2, s[1]+2)
IDL> bigImage[1,1] = openImg
```

To label the regions by searching for connections in the eight pixel neighborhood surrounding each pixel, type this.

```
IDL> labelImage = Label_Region(bigImage, /All_Neighbors)
IDL> MaxMin, labelImage
MaxMin:        4          0
```

The return value from *Label_Region* is an unsigned integer array, unless the *ULong* keyword was set, in which case it will be an unsigned long integer array.

Note: Once you have created the label image, you need to remove the border you place around the image before you called the *Label_Region* command. If you do not do this, then the indices you find for your ROIs in the next step will not point to the correct indices in your original

image. If you plan to calculate statistics or do anything else that involves the original image values, this step is critical.

```
IDL> labelImage = labelImage[1:s[0], 1:s[1]]
```

Generally, at this point, each region or ROI is processed separately. I will only process a single ROI in this example, but the same process is used for each of the ROIs identified by *Label_Region* unless there is some reason to exclude a region. For example, an ROI might be excluded from further processing by virtue of having too few pixels or too small an area.

We can quickly get a feel for how many pixels are in each region by looking at a histogram plot of the output image.

```
IDL> cgDisplay, 400, 400
IDL> cgHistoplot, labelImage, MinInput=1, /Fill
```

You see the result in Figure 39.

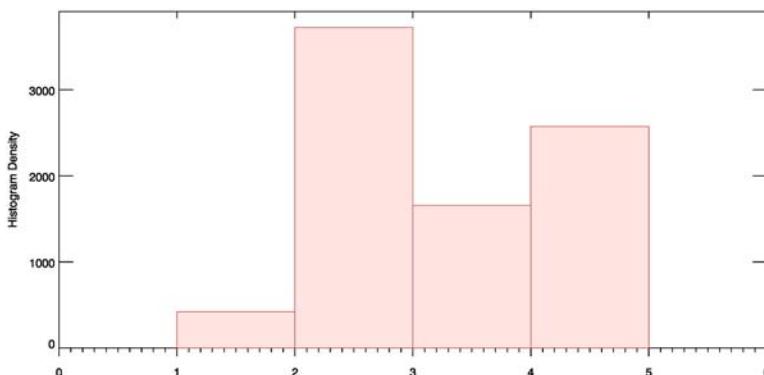


Figure 39: A histogram plot showing the number of pixels in each of the four labeled regions.

Processing the Image ROI

There are several possible ways to get pixels of each ROI for further processing. Let's use the pixels labeled in this image with the number 2 for this example. One way we can obtain the pixels is with the *Where* function.

```
IDL> roi_2_indices = Where(labelImage EQ 2, count)
```

Another way to obtain the indices is to calculate the histogram of the output image and use the *Reverse_Indices* keyword to locate the indices for

each labeled region. (See “Understanding Histogram Reverse Indices” on page 283 for additional information about reverse indices and how they are used.)

```
IDL> h = Histogram(labelImage, Binsize=1UL, $  
    Reverse_Indices=ri)  
IDL> roi_2_indices = ri[ri[2]:ri[3]-1]
```

In an image with only four regions, the *Where* function will be fast enough, even if we process all the ROIs in a loop. But the *Where* function can be slow if there are many regions to process. In one image I processed recently, there were over 11,000 regions and it took nearly 45 minutes to process this image when I located the ROI pixels with the *Where* function.

However, when I processed the same image, in exactly the same way, using the *Histogram* technique demonstrated above, the program ran in just 10 seconds! In other words, locating the ROI indices with the histogram technique was *270 times faster*.

ROI Statistics

What you do with the ROI indices is completely up to you. One of the things you might do is calculate statistics for the ROI. For example, you might want to know the average value of the pixels in the ROI and maybe their standard deviation. You can calculate these and other statistical properties using the *Image_Statistics* command.

```
IDL> Image_Statistics, image[roi_2_indices], $  
    Count=count, Maximum=max, Minimum=min, $  
    Mean=mean, StdDev=stddev  
IDL> Print, 'Count: ', count  
Count: 3723  
IDL> Print, 'Min Max Mean: ', min, max, mean  
Min Max Mean: 0.000000 200.000 37.2106  
IDL> Print, 'Standard Deviation: ', stddev  
Standard Deviation: 13.4288
```

ROI Morphological Properties

Other information you might want to know about an ROI are general shape properties such as center of mass, area, and the length of its perimeter. To do this, you need to create a polygon that describes the boundary of this ROI. There are several ways to obtain this boundary.

One method is to contour an empty image with just this ROI present.

```
IDL> blank = BytArr(s[0], s[1])  
IDL> blank[roi_2_indices] = 1  
IDL> IsoContour, blank, C_Value=1, vertices, connect
```

In the *IsoContour* call, *vertices* is an output parameter that contains the vertices of the contour and *connect* is an output parameter that contains the connectivity information about the vertices. Even though we will not use the connectivity information here, the parameter *must* be present in the call.

The boundary vertices are in the first two columns of the *vertices* parameter.

```
IDL> b1_x = Reform(vertices[0,*])
IDL> b1_y = Reform(vertices[1,*])
```

Another method is to use the [Find_Boundary](#) command from the [Coyote Library](#). This command uses a chain-code algorithm to find the boundary by moving from pixel to pixel along the boundary. This is probably a more accurate, but time consuming, method than the contouring method.

The [Find_Boundary](#) program can find the area, center of mass, and perimeter length of the ROI at the same time as it is finding the boundary, so it precludes the need to make an ROI object as described below.

```
IDL> vertices = Find_Boundary(roi_2_indices, $
      XSize=s[0], YSize=s[1], Area=area, $
      Center=cm, Perimeter=length)
IDL> b2_x = Reform(vertices[0,*])
IDL> b2_y = Reform(vertices[1,*])
IDL> Print, 'Area: ', area
      Area: 3723
IDL> Print, 'Center of Mass: ', cm
      Center of Mass: 165.861     84.4582
IDL> Print, 'Perimeter Length: ', length
      Perimeter Length: 352.33304
```

If you use the contouring method to obtain the ROI boundary, you will want to create an IDLanROI object to analyze the shape properties. This object has methods (procedures and functions that act on the object's data) to obtain the same information obtained above with [Find_Boundary](#). For example, you can use the *ComputeGeometry* method to determine the area, center of mass, and perimeter length of the ROI.

```
IDL> roi_2 = Obj_New('IDLanROI', b2_x, b2_y)
IDL> void = roi_2 -> ComputeGeometry(Area=area, $
      Centroid=cm, Perimeter=length)
IDL> Print, 'Area: ', area
      Area: 3511.0000
IDL> Print, 'Center of Mass: ', cm
      Center of Mass: 165.73274     84.356641
```

```
IDL> Print, 'Perimeter Length: ', length  
Perimeter Length: 426.14214
```

When you are finished with the object, be sure to destroy it to avoid unnecessary memory leakage. (IDL keeps track of object and pointer memory itself starting in IDL 8 and will clean up after itself when an object reference is no longer valid. But it is still necessary to clean up object memory yourself in versions of IDL prior to IDL 8.)

```
IDL> Obj_Destroy, roi_2
```

You will notice you get slightly different results using the two different methods to obtain the boundary of the ROI. This is normal and there is no “correct” answer. Both methods work “correctly.” The discrepancy arises from different interpretations of how to calculate the distance and area when moving from one pixel to the next. (Do you follow the pixel edges, take the shortcut route across a pixel, perform a statistical fudge, etc.?) The discrepancy is less important than choosing one method and handling all your analyses consistently. I do note, however, that the area reported by [Find_Boundary](#) is identical to the pixel count, as it should be using the chain code algorithm. There is a short discussion and reference in the [Find_Boundary](#) code header about this difference in computing area and perimeter length.

In any case, the boundaries are virtually identical, as you can see by typing these commands.

```
IDL> cgDisplay, s[0]*2, s[1]  
IDL> cgLoadCT, 0, /Reverse  
IDL> !P.Multi = [0,2,1]  
IDL> cgImage, openImage, /Scale, /Save  
IDL> cgPlots, b1_x, b1_y, Color='red', Thick=2  
IDL> cgImage, openImage, /Scale, /Save  
IDL> cgPlots, b2_x, b2_y, Color='red', Thick=2  
IDL> !P.Multi = 0
```

You see the result in Figure 40.

Fitting An Ellipse to the ROI

Another shape property that is often used with ROIs is to fit an ellipse to the shape. The ratio of the two principal axes of the ellipse, or sometimes the area of the ellipse, provides a handy proxy for comparing one ROI with another. For example, you might want to compute the ellipsoidal area of a number of blobs in an image and then perform a histogram analysis to compare the number of small, medium, and large blobs in the image.



Figure 40: The ROI boundary constructed with a contouring method on the left and with the chain code algorithm on the right. Both do an adequate job, but produce slightly different results for the ROI area and perimeter length.

There is a handy function in the [Coyote Library](#), named `Fit_Ellipse`, that performs this kind of fitting for you. Information about the fitted ellipse is returned by output keywords.

```
IDL> xyPts = Fit_Ellipse(roi_2_indices, XSize=s[0], $  
IDL> YSize=s[1], Center=center, Orientation=orient, $  
IDL> Axes=axes)  
IDL> Print, 'Center: ', center  
Center: 165.861 84.4583  
IDL> Print, 'Orientation: ', orient  
Orientation: -128.305  
IDL> Print, 'Axes Lengths: ', axes  
Axes Lengths: 143.247 35.5608
```

You can display the ROI with its fitted ellipse like this.

```
IDL> cgDisplay, s[0], s[1]  
IDL> cgImage, openImg, /Save, /Scale  
IDL> cgPlotS, xyPts, Color='red', Thick=2
```

You see the result in Figure 41.

Creating an Image Mask

An image mask is any 2D array the same dimensions as the image and filled with either 0s or 1s. The image we have been using, `openImage`, can, for example, be used as an image mask. Masks can be created in a variety of ways. One popular method is to use a Boolean operator like `GT` to cre-

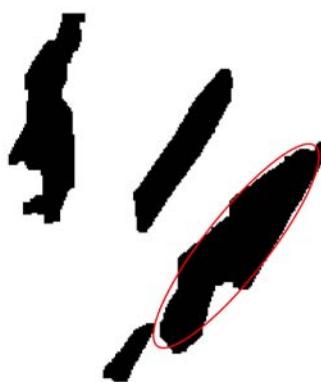


Figure 41: An ellipse is fitted to the ROI with Fit_Ellipse.

ate an image mask by image thresholding. Here are two different image masks used to mask pixels from the original image.

```
IDL> cgDisplay, s[0]*2, s[1]
IDL> !P.Multi = [0,2,1]
IDL> cgImage, image * openImage, /Scale
IDL> mask = image LT 45
IDL> cgImage, image * mask, /Scale
IDL> !P.Multi = 0
```

You see the result in Figure 42.



Figure 42: The original image masked with the openImage variable on the left, and the original image masked with a threshold operator mask on the right.

Using the Blob Analyzer

The kind of blob analysis we have been discussing is made easier by the [Coyote Library](#) program [Blob_Analyzer](#). Input to the program is a bi-level image. The program will automatically find the blobs in the image (using [Label_Region](#)) and calculate statistics for each of the blobs (using [Find_Boundary](#) and [Fit_Ellipse](#)).

The only trick to using the program is to create the bi-level image. Let's use another image in the IDL *examples* directory to see how this works. The *Grayscale* keyword in the *Read_JPEG* command below converts the RGB true-color image to an 8-bit grayscale image.

```
IDL> file = Filepath(SubDir=['examples','data'], $  
           'r_seeberi.jpg')  
IDL> Read_JPEG, file, image, /Grayscale
```

Next, we create a disk-shaped structuring element 11 pixels wide and apply a grayscale opening operation with *Morph_Open*. We will display the original image, the morphologically opened image, and the masked image next to one another.

```
IDL> rad = 5  
IDL> structElem = Shift(Dist(2*rad+1), rad, rad) LE rad  
IDL> openImage = Morph_Open(image, structElem, /Gray)  
IDL> s = Size(image, /Dimensions)  
IDL> cgDisplay, s[0]*3, s[1]  
IDL> cgLoadCT, 0, /Reverse  
IDL> !P.Multi = [0,3,1]  
IDL> cgImage, image  
IDL> cgImage, openImage  
IDL> mask = openImage GE 150  
IDL> bilevel = (openImage * mask) GT 0  
IDL> cgImage, bilevel, /Scale  
IDL> !P.Multi = 0
```

You see the result in Figure 43.

The bi-level image can be passed to the [Blob_Analyzer](#) program, which is written in the form of an object.

```
IDL> blobs = Obj_New('blob_analyzer', bilevel)
```

To create an image with blobs masked out of the original image, you can type this.

```
IDL> blobCnt = blobs -> NumberOfBlobs()  
IDL> blankImage = BytArr(s[0], s[1])
```

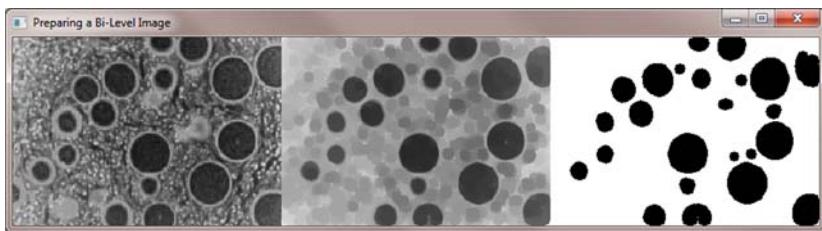


Figure 43: The original grayscale image on the left, the image transformed with Morph_Open in the middle, and the masked, bi-level image on the right.

```
IDL> FOR j=0,blobCnt-1 DO BEGIN & $  
IDL>     blobIndices = blobs -> GetIndices(j) & $  
IDL>     blankImage[blobIndices] = image[blobIndices] & $  
IDL> ENDFOR
```

To see this image next to the original image with the blobs outlined and labeled type these commands.

```
IDL> cgDisplay, s[0]*2, s[1]  
IDL> !P.Multi = [0,2,1]  
IDL> cgImage, blankimage  
IDL> cgImage, image, /Save  
IDL> FOR j=0,blobCnt-1 DO BEGIN & $  
IDL>     stats = blobs -> GetStats(j) & $  
IDL>     cgPlots, stats.perimeter_pts, COLOR='red' & $  
IDL>     cgText, stats.center[0], stats.center[1]-5, $  
           StrTrim(j+1,2), Alignment=0.5, $  
           CharSize=0.75, Color='yellow' & $  
IDL> ENDFOR  
IDL> !P.Multi = 0
```

You see the result in Figure 44.

You can return the statistics from the blob analyzer object with the *GetStats* method, but you can also print a report to the display screen with the *ReportStats* method.

```
IDL> blobs -> ReportStats
```

You see the result in Figure 45.

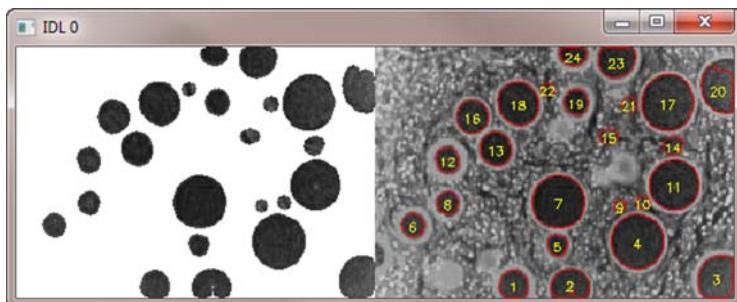


Figure 44: Using the *Blob Analyzer* object to view and work with ROIs.

INDEX	NUM_PIXELS	CENTER_X	CENTER_Y	PIXEL_AREA	PERIMETER_AREA	PERIMETER_LENGTH	MAJOR_AXIS	MINOR_AXIS	ANGLE
0	426	107.89	9.78	426.00	392.00	75.11	24.20	22.58	-8.13
1	580	151.97	10.22	580.00	537.00	98.43	32.57	23.50	0.50
2	612	266.29	15.36	612.00	763.00	105.11	35.77	29.31	72.52
3	1438	204.53	43.29	1438.00	1376.50	140.47	43.37	42.25	-256.47
4	231	142.10	40.58	231.00	205.50	56.04	17.99	16.48	-249.08
5	278	29.38	56.06	278.00	251.50	60.59	19.37	18.30	-253.57
6	1391	142.79	74.15	1391.00	1330.00	138.23	42.22	41.98	73.42
7	261	56.45	75.29	261.00	235.00	59.11	18.89	17.65	83.20
8	91	191.05	71.00	91.00	66.00	32.17	10.39	10.19	-90.00
9	100	208.64	73.64	100.00	83.50	36.38	11.59	11.04	-225.00
10	1233	233.13	87.28	1233.00	1176.00	129.40	39.94	39.33	65.57
11	311	56.26	106.61	311.00	292.00	65.11	21.24	18.69	-256.65
12	548	93.97	115.23	548.00	510.00	85.60	27.60	25.29	-282.87
13	200	232.00	117.79	200.00	177.00	51.46	16.00	15.23	39.25
14	155	181.97	125.07	155.00	134.00	46.63	16.25	12.19	1.35
15	540	75.62	140.30	540.00	503.00	86.08	27.00	25.49	-269.02
16	1441	227.99	151.28	1441.00	1379.00	141.05	44.74	41.05	82.17
17	893	111.45	150.51	893.00	845.00	109.74	35.51	32.05	-246.11
18	328	156.61	151.55	328.00	299.00	66.77	21.30	19.63	-265.27
19	772	267.46	161.49	772.00	721.50	111.04	30.40	26.04	-254.79
20	121	198.26	150.00	121.00	103.00	39.80	12.46	12.41	-90.00
21	100	134.36	161.64	100.00	83.50	36.38	11.59	11.04	45.00
22	615	180.07	103.32	615.00	573.00	91.94	31.20	25.49	7.67
23	270	154.14	187.57	270.00	242.00	61.46	22.51	15.42	-1.55

Figure 45: Statistics for all the blobs found by the blob analyzer can be printed out for display with the *ReportStats* method.

Principal Component Analysis of Images

Principal Component Analysis (PCA) is a technique for analyzing the variances or differences (also called the *covariances*) in two or more data sets or images. It is often used as a filter to eliminate noise in a collection of images. The purpose of PCA is to obtain a set of orthogonal (completely uncorrelated) vectors that describe the differences in a set of images. These orthogonal, uncorrelated vectors are called the *eigenvectors* of the data. There will be as many eigenvectors as there are images or data sets that are being compared.

In addition to the eigenvectors, PCA will provide a matching set of numbers, called the *eigenvalues*. These numbers can be ordered from largest to smallest (along with their corresponding eigenvectors) to give you a measure (by their absolute magnitude) of how much of the differences in your data sets can be “explained” by the corresponding eigenvector. These corresponding eigenvectors and eigenvalues are called the *principal components* of the differences in the data.

Note: Sometimes Principal Component Analysis goes by the name of Empirical Orthogonal Function, or EOF, analysis. EOF analysis is identical to PCA. The empirical orthogonal functions are, simply, the eigenvectors and eigenvalues of PCA.

To learn how PCA can be used with images, we are going to consider images from a gated blood pool study in the IDL *examples* directory. The data are enlarged by the *Congrid* command just to make the images easier to view.

```
IDL> data = cgDemoData(6)
IDL> data = Congrid(data, 256, 256, 15)
IDL> cgDisplay, 1000, 600, Title='Gated Blood Study'
IDL> !P.Multi = [0,5,3]
IDL> FOR j=0,14 DO cgImage, data[*,*,j]
IDL> !P.Multi = 0
```

You can see the result in Figure 46.

To perform PCA, the data we want to compare (these images) must be placed in a column array. That is to say, each of these 15 images must be placed in a single column of a 15 by N array. We can reorder the elements of each of these 256x256 images with the *Reform* command in IDL, and swap columns and rows with the *Transpose* command. Finally, we convert the byte array to floating point values to do the calculations.

```
IDL> data = Transpose(Reform(data, 256L*256L, 15))
IDL> data = Float(data)
IDL> Help, data
      DATA      FLOAT = Array[4, 65536]
```

The next step is to create the covariance matrix. This is done with the *Correlate* command in IDL with the *Covariance* keyword set. Be sure to set the *Double* keyword as well to do this kind of analysis in double precision math. Note that we subtract the mean value of images from images as we pass them to *Correlate*. Because PCA deals with the *differences* in data, it

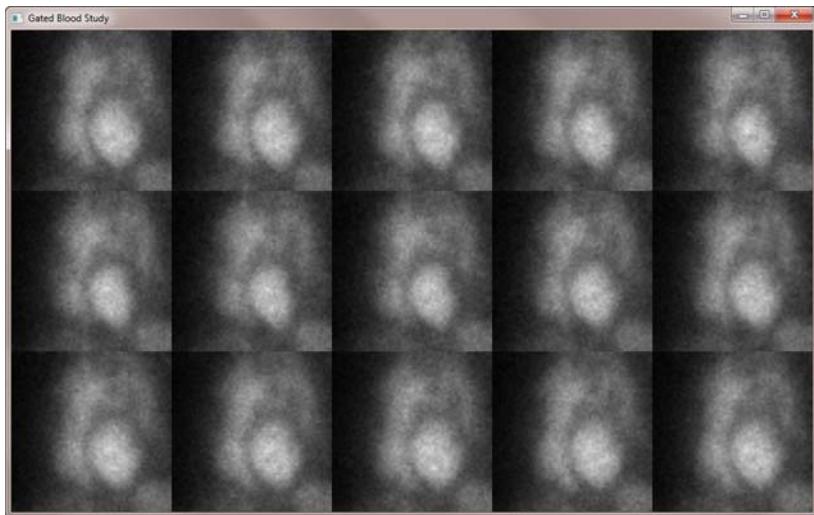


Figure 46: The images from the gated blood pool study which were selected for Principal Component Analysis.

is standard practice to work with the difference of the data from the mean value.

```
IDL> covMatrix = Correlate(data - Mean(data), $  
/Covariance, /Double)
```

Once we have the covariance matrix, the next step is to obtain the eigenvectors and eigenvalues. This is done with the *EigenQL* command in IDL.

```
IDL> eigenValues = EigenQL(covMatrix, /Double, $  
/Absolute, Eigenvectors=eigenVectors)
```

The *Absolute* keyword in the *EigenQL* command assures that the eigenvectors and eigenvalues are ordered largest to smallest in terms of their absolute magnitude, rather than their signed values.

Note: Note that the last two steps can also be performed with the *PComp* command in IDL. The two methods are identical, although they don't appear so on first inspection of the eigenvectors that are produced by the two methods. It turns out that the *PComp* output is multiplied by the square root of the corresponding eigenvalue, a fairly common practice in PCA analysis. *PComp* eigenvectors are identical to the eigenvectors produced here, if the eigenvectors are divided by the square root of their corresponding eigenvalues.

We can print the eigenvalues.

```
IDL> Print, eigenValues, Format='(3(F8.2, 2x))'  
30564.05    185.40    30.40  
   28.74    26.48    25.97  
   25.04    25.02    24.47  
   23.46    23.39    22.74  
   22.48    22.30    21.00
```

You can see that the first eigenvalue, with its corresponding eigenvector, “explains” or predicts about 98 percent of differences in these images.

```
IDL> percent = eigenValues[0] / Total(eigenValues) * 100  
IDL> Print, 'Percent Variance Explained: ', percent  
Percent Variance Explained: 98.368549
```

In other words, we could use this eigenvector to “predict” what a typical image is going to look like. Or, another way to say this, is that if we used this eigenvector to reconstruct the image we would have an image that contained most of the information from these series of images, but without the noise that is confined to the other principal components. We would have a smoothed image, in other words.

You can see this illustrated by plotting the eigenvector coefficients. Note that the eigenvectors are in the *rows* of this matrix, not the columns! You can think of these vectors as being the temporal components of the spatial (image) data.

```
IDL> cgDisplay, 1000, 600, Title='PC Coefficients'  
IDL> !P.Multi = [0,5,3]  
IDL> FOR j=0,14 DO cgPlot, eigenvectors[*,j]  
IDL> !P.Multi = 0
```

You see the result in Figure 47. The first plot in the upper left of the figure shows the smoothed image, without the noise. The plot to its right shows, essentially, the heart beating, and the rest of the plots show noise components.

The next step is to apply the eigenvectors to the image data set to create the principal component arrays. Note that the ## operator in IDL multiplies the rows of the array on the left of the operator with the columns of the array on the right of the operator. This means the *data* array needs to be transposed to perform the operation. The principal component array is then reformed to the original image shape.

```
IDL> pc = eigenVectors ## Transpose(data)  
IDL> pc = Reform(Temporary(pc), 256, 256, 15)
```

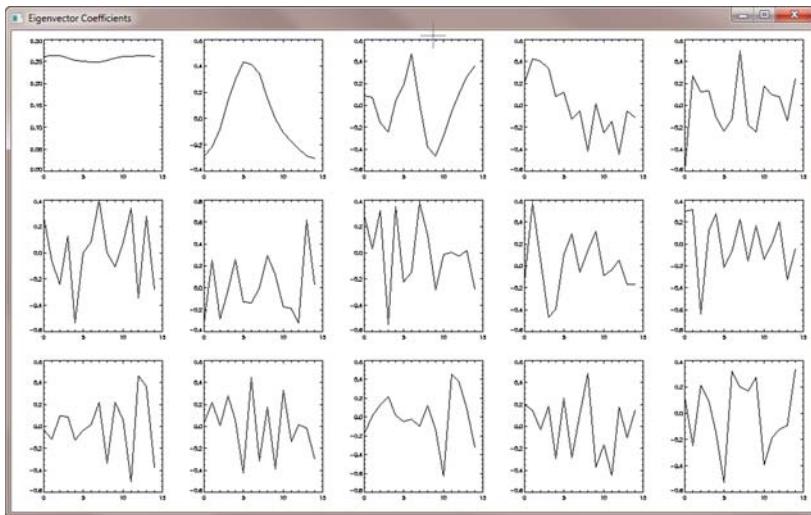


Figure 47: Plots of the eigenvector coefficients. The first plot in the upper left has isolated the basic features of the image, while the plot next to it shows the temporal components of the heart beating. The rest of the plots isolate noise in the images. Compare this figure to Figure 48.

Type these commands to view the images as transformed by the principal component vectors.

```
IDL> cgDisplay, 1000, 600, Title='Transformed Images'
IDL> !P.Multi = [0, 5, 3]
IDL> FOR j=0,14 DO BEGIN $
IDL> cgImage, pc[*,*,j], /Scale, /Save & $
IDL> cgText, 25, 25, 'PC Image ' + StrTrim(j,2), $
      Color='yellow', Font=0 & ENDFOR
IDL> !P.Multi = 0
```

You see the result in Figure 48.

The smoothed image in the upper left corner of the figure is the image transformed by the first principal component. The image to its right, transformed by the second principal component, has isolated the beating heart. The other principal components are primarily isolating noise in the images.

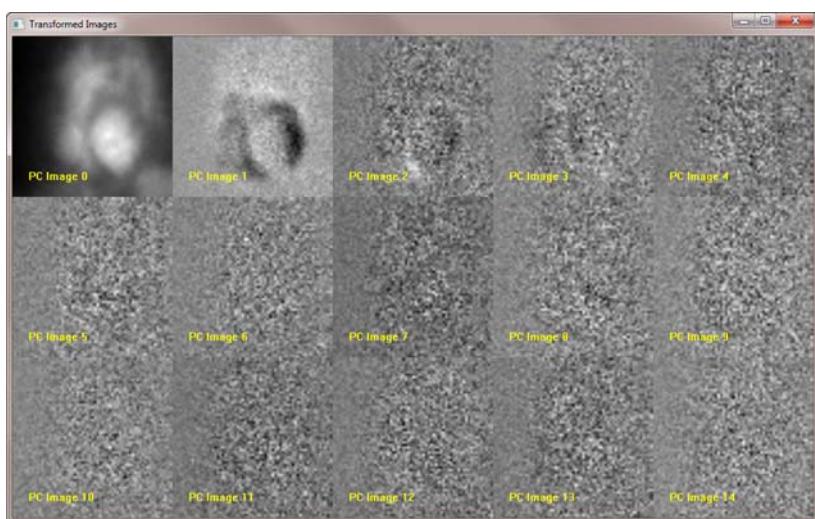


Figure 48: The images transformed by their principal components.

Chapter 9



Creating PostScript Output

Why PostScript Output is Essential

I don't think it is overstating the facts to say that the ability to produce PostScript output is essential to IDL's usefulness in scientific programming. IDL has never excelled at producing JPEG, PNG, or TIFF files for presentations or web display, although it is certainly possible to produce these kinds of files in IDL. But, the Hershey character fonts used by traditional graphics commands, which show up adequately on most computer displays, are not as adequate when they are used on web pages or projected onto the screen of a presentation.

And, yet, PostScript output is high resolution, vectorized, and more than adequate for journal articles, reports, and—more and more these days—conversion to PDF files, which can be read on the computer and on other electronic reading devices. Even better, PostScript output can use PostScript and TrueType fonts, which are both scalable to any size and a pleasure to look at. Wouldn't it be nice if we could somehow leverage PostScript output to create better looking and completely adequate graphical output for both presentations and web display?

In fact, we can. But to do so, we need to use programs other than IDL. I will show you some specific ways it is possible to do this in "Creating Raster Output" on page 405. Most of these methods start with our ability to create PostScript files, which can be transformed to other types of presentation quality files. We can e-mail these presentation files to colleagues, display them on a web page, include them in presentations, or use them as illustrations in books or journal articles. In short, if it wasn't possible to

produce PostScript output quickly and easily in IDL, we would probably all be looking for another program to produce presentation quality output.

But, I also don't think it is overstating the facts to say that the PostScript driver supplied with IDL for its traditional graphics commands seems to have as its goal to completely confuse and confound the new user. Nearly everyone learning IDL for the first time finds it a challenge to produce adequate PostScript file output from the graphics programs they write in IDL.

It doesn't have to be that hard. It *shouldn't* be that hard! The purpose of this chapter is to (1) explain how the IDL PostScript device works, and (2) introduce you to tools I have developed over the years that make creating PostScript output so easy I seldom give it a second thought. I believe this: Every IDL program you write should work identically on the display and with the PostScript device. The purpose of this chapter is to illustrate how this is done. And then, later, to show you how you can use the PostScript files you create to produce high quality graphical output files you can use for other purposes.

How Does the PostScript Device Work?

The PostScript device or driver is what we call a *graphics device*. IDL comes with a number of standard graphics devices. Some of these devices, such as the CGM, HP and PCL drivers, are hardly used any more. These drivers produced output for printing devices which were popular from about 1975 to 1995 and which now reside in landfills the world over.

Of the graphics devices supplied with IDL, I only know of three that are used with any frequency. These are your normal graphics display device on your computer (*WIN* or *X*, typically, depending on whether you are running IDL on a Windows or UNIX machine, but possibly *MAC* if you have an older Macintosh computer), the Z-graphics buffer device, *Z*, and the PostScript device, *PS*.

(The *PRINTER* device was sometimes used in the past to send graphics output directly to a printer attached to the computer, but this device was so perverse in the way it handled anything but black and white graphical output (all colors have to be loaded in the color table *before* you call the printer device!) that it has mostly been abandoned.)

The PostScript and Z-graphics devices have both been slow to move into the 21st century, where 24-bit graphics displays are standard. The Z-graphics driver was an 8-bit graphics device until IDL 6.4 and the PostScript

driver was an 8-bit graphics device until IDL 7.1, although it didn't work correctly as a 24-bit graphics device until IDL 7.1.1.

The fact that the PostScript device was an 8-bit device, which required graphics routines to be written using the indexed color mode, while our computer graphics displays were 24-bit devices, was the primary reason so many IDL programmers crippled their computer graphics devices by sticking that damnable *Device, Decomposed=0* command into their IDL start-up files, and thereby limited themselves to just 256 colors!

I'm hopeful those days are over, at least for readers of this book. But I know the wheels of change roll slowly. This is the primary reason I try to write IDL graphics programs that are immune to the color model used in IDL. I want everyone to be able to use them. We have discussed and used a number of these programs already in this book, but without much discussion of *why* I use those programs and not the ones supplied by IDL that do similar jobs.

The reason, simply, is that those programs work on the display and produce PostScript output without the user having to dance on the top of a rolling ball to make it happen, a *huge* advantage over the normal IDL traditional graphics commands.

The idea is that graphical output (i.e., the output of graphical commands such as *Contour, Plot, Surface*, etc., or the alternative commands `cgContour, cgPlot, cgSurf`, etc.) is sent directly to the *current graphics device*. By default, the current graphics device is set to what we often call the *display device*. In other words, the device that displays graphical output on your computer. If you are running IDL on a Windows machine, that device is named *WIN*. If you are running IDL on a UNIX machine, that device is named *X*.

The name of the current graphics device is always stored in the *!D.Name* system variable. This is *WIN*, in my case, because I am writing this book and running IDL on a Windows computer.

```
IDL> Print, !D.Name  
WIN
```

If I want to send graphical output to some other graphics device, I need to make that other graphics device the current graphics device. I do that with the *Set_Plot* command in IDL. So, if I want to send my graphical output to a PostScript file, I make the PostScript driver the current graphics device.

```
IDL> Set_Plot, 'PS'
```

Note: The name of the device is always a string. It doesn't have to be set in uppercase letters (although it usually is), but it is always stored in the `ID.Name` system variable as uppercase letters no matter how it is set. This is an important fact if you have to write the occasional line of device-specific code in an IDL program.

Once you have changed graphics devices, any graphical output will be sent to the new graphics device. In the case of the PostScript device, graphical output will be written into a PostScript file and will no longer appear in a graphics window on your computer. By default, the PostScript file that is created is named `idl.ps`, but as you will see in a moment, the file name can be changed with the `Filename` keyword to the `Device` command.

You can issue as many graphics commands as you like. If you issue a graphics command that on your computer would cause the graphics window to erase itself (e.g., `Plot`, `Contour`, `Surface`, etc.), you will simply generate a new page of PostScript output with that command. (The only exception is the first graphics command written to the file, which always goes on the first PostScript page.) In fact, an `Erase` command will simply move you to the next page of PostScript output. Commands that don't erase the display (e.g., `Oplot`, `XYOutS`, `PlotS`, etc.) will simply go into the same PostScript "page."

```
IDL> Plot, [2, 5, 4, 8, 7, 10]
```

When you have completed sending graphics commands to the PostScript device, you have to close the open file before you can print the file or do anything else with it. This is done with the `Device` command and by setting the `Close_File` keyword.

```
IDL> Device, /Close_File
```

Closing the PostScript file does two things that are essential for PostScript output. It causes the graphics buffer in IDL to be flushed, so that all the graphics output is written to the PostScript file, and it inserts a PostScript `showpage` command into the file. The `showpage` command is responsible for ejecting the PostScript output from a PostScript printer. Files that don't have a `showpage` command will process in the printer (you will see the printer "busy" light blinking), but nothing will ever come out of the printer.

Note: Be absolutely sure you always close the file before you print a PostScript file or do anything else with it. Changing to another graphics device does not close the PostScript file, although exiting from IDL does.

Once you have closed the PostScript file you can either write another file (you will have to change the name of the file with the *Filename* keyword, discussed in the PostScript configuration section below), or you can switch to another graphics device. Typically, you switch back to your display device. In my case, since I am working on a Windows machine, my display device is WIN. If you are on a UNIX machine, your display device is probably X. (Print *!D.Name* at the IDL prompt when you first start IDL up if you are unsure of the name of your display device.)

```
IDL> Set_Plot, 'WIN'
```

Naturally, if I write this line of code it will work perfectly on Windows machines and fail miserably on every other computer ever made. So, I encourage you to write programs that do *not* assume every person running your program will be using the same computer you are!

Normally, when we work with PostScript output, we save the name of the current graphics device before we switch to the PostScript device, and then we restore that device when we are finished with the PostScript device. The sequence of commands to produce machine independent programs looks something like this.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Plot, [2, 5, 4, 8, 7, 10]
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

Configuring the PostScript Device

I have left out one important step in creating PostScript output. I have not mentioned that after making the PostScript driver the current graphics device you have to configure it for output. It is this step that confounds most new users of IDL. The primary reason it is confusing is that the default configuration of the PostScript device is, uh, ... Well, let's just say the defaults were probably state of the art in 1985. But they are archaic today and no one producing PostScript output on modern computers (produced after, say, 1990) would want to use them.

Much of the information in this chapter is explaining what these defaults are and offering strategies and tools that will make configuring the PostScript device as easy as possible, so you can use the device without going to a great deal of extra effort.

The PostScript device, like all graphics devices in IDL, is controlled with the *Device* command and various, appropriate keywords. The keywords that can be used with a graphics device depend entirely on which graphics device you are using, but there are on the order of 50 or so keywords that are appropriate for the PostScript device, although only 15 or 20 are used with any frequency.

Here are nine *Device* keywords, with short descriptions, that are nearly always used to configure the PostScript device.

Filename	This keyword is used to set the name of the PostScript output file. The default filename is <i>idl.ps</i> .
Close_File	Setting this keyword flushes the graphics output buffer to assure all the graphic output is written into the PostScript file and closes the PostScript file to further input. It also places a PostScript <i>showpage</i> command into the PostScript file which is used to eject the page from a PostScript printer.
Inches	Setting this keyword indicates the <i>XSize</i> , <i>YSize</i> , <i>XOffset</i> , and <i>YOffset</i> keywords are specified in inches rather than in the default unit of centimeters. To return to centimeters after setting this keyword, set the keyword to 0.
Landscape	Setting this keyword positions the PostScript page in landscape mode in which the horizontal direction is longer than the vertical direction. As you will see in a moment, IDL landscape mode is often called “seascape” mode in other software and IDL landscape files usually appear upside-down when viewed in PostScript file viewers. This can be corrected with FixPS , which I will demonstrate in just a moment.
Portrait	Setting this keyword positions the PostScript page in portrait mode in which the vertical direction is longer than the horizontal direction. This is the default page mode.

The following four keywords set up the PostScript “window” on the PostScript page. If the PostScript page is equivalent to your monitor, then the PostScript window is equivalent to an IDL graphics window on your monitor. It is the area on a page of output where the graphics will appear. Graphics are written into the PostScript window in exactly the same way

they are written into the graphics window on your monitor, with a couple of exceptions involving images, which will be discussed below.

XSize	The X size of the PostScript “window” where graphics are displayed on the PostScript page. The X size is expressed in centimeters or inches.
YSize	The Y size of the PostScript “window” where graphics are displayed on the PostScript page. The Y size is expressed in centimeters or inches.
XOffset	The distance, in centimeters or inches, to the lower-left corner of the PostScript window in the “X” direction, which you will see in a moment depends on the orientation of the PostScript page.
YOffset	The distance, in centimeters or inches, to the lower-left corner of the PostScript window in the “Y” direction, which you will see in a moment depends on the orientation of the PostScript page.

You can learn how the PostScript device (or any current graphics device, for that matter) is currently configured by using the *Help* keyword to the *Device* command. You must first be in the PostScript device to see how the PostScript device is configured.

```
IDL> Set_Plot, 'PS'  
IDL> Help, /Device
```

```
Available Graphics Devices: CGM HP METAFILE NULL PCL PRINTER PS WIN Z  
Current graphics device: PS  
File: <none>  
Mode: Portrait, Non-Encapsulated, EPSI Preview Disabled, Color Disabled  
Input Color Mode: Gray/Indexed  
Output Color Model: RGB  
Offset (X,Y): (1.905,12.7) cm., (0.75,5) in.  
Size (X,Y): (17.78,12.7) cm., (7,5) in.  
Scale Factor: 1  
Preview Size (X,Y): (4.51556,4.51556) cm., (1.77778,1.77778) in.  
Preview Depth: 8 bits per pixel  
Font Size: 12  
Font Encoding: AdobeStandard  
Font: HelveticaTrueType Font: <default>  
# bits per image pixel: 4  
Font Mapping:  
    (13) Helvetica          (!4) Helvetica-Bold  
    (!5) Helvetica-Narrow   (!6) Helvetica-Narrow-BoldOblique  
    (!7) Times-Roman        (!8) Times-BoldItalic  
    (!9) Symbol             (!10) ZapfDingbats  
    (!11) Courier            (!12) Courier-Oblique  
    (!13) Palatino-Roman    (!14) Palatino-Italic  
    (!15) Palatino-Bold      (!16) Palatino-BoldItalic  
    (!17) AvantGarde-Book   (!18) NewCenturySchlbk-Roman  
    (!19) NewCenturySchlbk-Bold  (!20) <Undefined-User-Font>
```

Producing Color (or Grayscale) Output

There are several things to notice in this default configuration, all of which will be discussed in detail below. But the two most confusing defaults are that color output is disabled and that only 4 bits of information are stored for each 8-bit image pixel. In effect, this is telling you that you can get black and white graphical output or images expressed in no more than 16 shades of gray. This is parsimony carried to the extreme!

So, no matter what else you do, at the very *least*, you want to turn color on and save all the information contained in an image pixel.

```
IDL> Device, Color=1, Bits_Per_Pixel=8
```

Why this isn't the standard default behavior is a deep and enduring mystery. But, because PostScript configurations are "sticky," meaning that once they are set they stay set until they are actively changed, it would not be at all inappropriate to see commands like the following in an IDL start-up file so that the PostScript device is configured properly every time IDL starts up.

```
thisDevice = !D.Name
Set_Plot, 'PS'
Device, Color=1, Bits_Per_Pixel=8
Set_Plot, thisDevice
```

Setting the *Color* keyword automatically copies the current color table vectors into the PostScript file, although colors can be changed at any time in PostScript files by loading colors and color tables in the normal manner. You can also copy the current color table vectors into the PostScript file at the time you set the PostScript device by using the *Copy* keyword to *Set_Plot*.

```
IDL> Set_Plot, 'PS', /Copy
```

To return to the configuration listing, note that the default color mode (listed as the "Input Color Mode" in the configuration output from the *Help* command) is to use the indexed color model. In other words, the PostScript device is set up to act like an 8-bit indexed color graphics device.

This fact has always made it challenging to write IDL programs that worked identically on a 24-bit display device and in an 8-bit PostScript device. Nevertheless, the [Coyote Library](#) graphics routines ([cgPlot](#), [cgContour](#), [cgSurf](#), [cgImage](#), etc.) have always managed to do just that, although users do have to remember to load color tables just before they are needed when working with images in 8-bit devices, as local drawing

colors, which *must* be loaded into the local color table to be used, can easily contaminate the color tables used by images.

If you are using IDL 7.1 and higher I think there is no reason anymore to *ever* work in indexed color mode. Naturally, you have to use or write graphical programs (like those from the [Coyote Library](#)) that allow you to do this.

Configuring 24-Bit Color

The PostScript device was upgraded to allow 24-bit graphics in IDL 7.1, although the device didn't work correctly with all the 24-bit keywords until IDL 7.1.1. Naturally, it is *much* easier to write programs that work correctly on the display and in PostScript when both environments share the ability to specify drawing and annotation colors directly, without the need to load these colors into color tables, where they can contaminate the color table for images.

To put the PostScript device into a 24-bit color mode, you set the *Decomposed* keyword to 1, as you do normally. But, if you need to write programs that work everywhere, you need to make sure the version of IDL running the program supports the functionality.

```
IDL> IF Float(!Version.Release) GE 7.1 THEN $  
      Device, Decomposed=1
```

To learn if the PostScript device is in 24-bit color mode, you use the *Get_Decomposed* keyword to the *Device* command, but this keyword was broken in IDL 7.1 and wasn't fixed until IDL 7.1.1

```
IDL> IF (!Version.Release EQ '7.1.1') THEN $  
      Device, Get_Decomposed=colorState
```

Similar restrictions apply in the Z-graphics buffer, which was upgraded to support 24-bit graphics in IDL 6.4 and didn't support the use of the *Get_Decomposed* keyword until then.

Because I never know which version of IDL people running my programs are using, I need to include in my programs a great deal of version specific (and sometimes machine specific) code to sort all these problems out. Because I don't want to write pages and pages of code every time I need to do this, I usually write one program that isolates the machine and version specific code in one location.

To this end, I can learn the current color decomposition state of any device, in any version of IDL, by using the [GetDecomposedState](#) command, and I can set the color decomposition state of any device, in any version of IDL, using the [SetDecomposedState](#) command.

```
IDL> SetDecomposedState, 1
IDL> currentState = GetDecomposedState()
IDL> Print, 'Decomposition State: ', currentState
      Decomposition State:          1
```

The [Coyote Library](#) routines use these routines to work in a 24-bit color decomposition mode whenever possible to avoid loading drawing colors in the local color table. The color decomposition state is always restored to the input decomposition state when these routines exit, so the routines don't force you to work any differently than you do now. Naturally, these routines must use a color routine like [cgColor](#) to create the proper 8-bit or 24-bit color value, as needed.

Color Reversal in PostScript Output

Another color issue that makes it hard to write IDL programs that work identically on the display and in PostScript is that normally display programs use a white foreground color and a black background, whereas PostScript reverses these colors to use a black foreground color on a white background.

In fact, PostScript switches the system variables *!P.Color* and *!P.Background*, and then ignores the *!P.Background* color. The joke has always been that you can have any color you like as a PostScript background color, as long as it is white.

The foreground and background colors on the display are often represented as the color table indices 255 and 0, respectively. Since the meaning of these color indices change when sending graphical output to a PostScript file, if we want to use colors for a particular purpose, say as a drawing color, and have them be the same color everywhere, it is imperative that we always load such colors at any color table index *except* 0 and 255. Of course, if you choose to use the decomposed color mode for your drawing colors, and I hope you do, then none of this matters because decomposed colors can be specified directly and do not need to be loaded into the color table indices at all.

Even so, it is hard to pick drawing or annotation colors that look good on both a black *and* a white background. This is the reason I almost always display graphics on my computer monitor using a white background. Then I don't have to make color changes when I send the graphical output to a PostScript file. You will notice that the [Coyote Library](#) graphics routines like [cgPlot](#), [cgContour](#), etc. all use white backgrounds as their default color scheme.

One thing you do *not* want to do, however, if you want to use a black foreground color on a white background on your computer display is switch the *!P.Color* and *!P.Background* values. If you do, you are likely to see absolutely nothing in your PostScript output, since you will be drawing white axes and labels on a white background! This is never a problem if you just follow this simple rule: If you want to use a drawing color, load it *anywhere* in the color table, *except* at indices 0 or 255. Or, even better, specify your drawing colors by using the decomposed color model, so the colors do not have to be loaded in the color table at all. This is the surest way to get the colors you are expecting.

What if All My PostScript Colors are Red!?

Occasionally people will complain about their plot and axis colors all coming out in shades of red. This is simply a mis-match between the color model you have selected for your PostScript device (decomposed color) and the way you are specifying your colors (indexed color).

Remember, PostScript configurations are “sticky.” Once they are set, they remain in effect throughout that IDL session. To fix the “everything is red!” problem, simply choose the indexed color model for your PostScript device.

```
IDL> Set_Plot, 'PS'  
IDL> Device, Decomposed=0
```

You will have much better luck with colors if you use `cgColor` to specify your colors. It is designed to give you the correct color no matter which color model you happen to be using.

What if I Don't See Any Colors At All!?

If you don't see any colors (or anything else!) in your PostScript file, there is an excellent chance (assuming you remembered to actually write the graphics command *to* the PostScript file and you remembered to close the file) that you are drawing your graphics in a white color on a white background.

This can also be the result of a mis-match between the color model of the PostScript device and the way you are specifying your drawing colors, although in this case it is likely you are using an indexed color model for your PostScript file, but are specifying your colors as 24-bit colors. This is easy to test by specifying one of your colors using `cgColor` and see if anything changes. If it does, this is the problem and you probably need to change your PostScript color model to match the way you are specifying the colors.

```
IDL> Set_Plot, 'PS'
IDL> Device, Decomposed=1
```

Color Backgrounds in PostScript

Despite the joke about color backgrounds in PostScript always being white, it *is* possible to have a background color other than white in a PostScript file. It is just that the background color has to be specified in a different way than you would specify it normally. On a display monitor, which is a raster device, you just use the *Background* keyword on a graphics command to specify the background color.

```
IDL> Plot, cgDemoData(1), Background=cgColor('wheat'), $
      Color=cgColor('opposite')
```

You see the result in Figure 1.

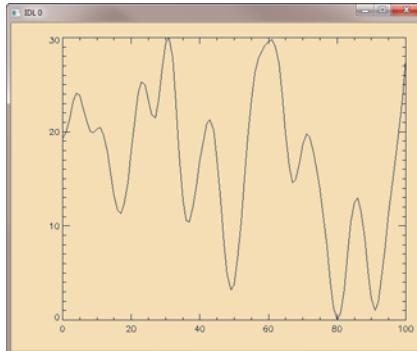


Figure 1: On a raster device, the background color is set by setting the *Background* keyword on a graphics command.

This same command would result in a black plot on a white background if sent to a PostScript file. However, if you want this color background, you can create it by “filling” the PostScript window with a background color before drawing the graphic. For example the background color can be drawn with the *Polyfill* command. Be sure to use a *NoErase* keyword on the *Plot* command so the background is not “erased” by the *Plot* command.

```
IDL> PolyFill, [0,1,1,0,0], [0,0,1,1,0], /Normal, $
      Color=cgColor('wheat')
IDL> Plot, cgDemoData(1), Color=cgColor('opposite'), $
      /NoErase
```

This is the technique all the [Coyote Library](#) graphics commands use to honor the *Background* keyword color both on the display and in a Post-

Script file. In other words, this command looks exactly the same on the display and in a PostScript file.

```
IDL> cgPlot, cgDemoData(1), Background='wheat'
```

Representing Color on Grayscale Printers

Naturally, there is not much point in specifying PostScript colors if you plan to print your output on a grayscale PostScript printer. Your printer must be able to create colors to see colors. What you will see if you do print a color PostScript file on a grayscale PostScript printer is that colors are represented as dithered lines, which often appear as dotted or “broken” lines. If you plan to print on a grayscale printer, it is probably best to use the color black for your axes and annotation color. And, of course, you have to set the *Color* keyword to be able to display *any* color, including grayscale, in a PostScript file, no matter what kind of PostScript printer you are using.

CMYK Color Output

The PostScript device is capable of converting the normal RGB color output to a CMYK color model if the CMYK keyword is set.

```
IDL> Device, /CMYK
```

The CMYK color model is a subtractive color model that is used primarily in the color printing industry. If the PostScript device is set to produce CMYK output, the RGB colors are converted to CMYK colors as the last step in the process. (See the on-line help for the IDL command *CMYK_Convert* for details of how this is done.)

To use CMYK colors, you must have a printer capable of interpreting PostScript Language Level 2 or higher. Therefore, setting this keyword will also set the *Language_Level* keyword to 2.

Setting Up the PostScript “Window”

The *XSize* and *YSize* keywords are used to set up a PostScript “window” or location on the PostScript page where graphics will be drawn. Drawing graphics into this window is essentially identical to drawing graphics into a graphics window on your display. The only real difference is the way images are displayed in a PostScript window, which will be discussed in detail below. The *XOffset* and *YOffset* keywords are used to position or locate this window on the page.

The default offset for PostScript files displayed in portrait mode is 0.75 inches in the X direction and 5 inches in the Y direction. This puts the Post-

Script window (and graphics output) in the upper half of the page. It is easy to see, therefore, that the offsets are calculated from the lower-left corner of the page (see Figure 2).

When you place the PostScript page into landscape mode, however, the entire page is rotated 90 degrees, including the lower-left corner of the page, which now becomes the lower right-hand corner of the page. It is from this position that landscape offsets are calculated! You can see an illustration of this non-intuitive situation in Figure 2.

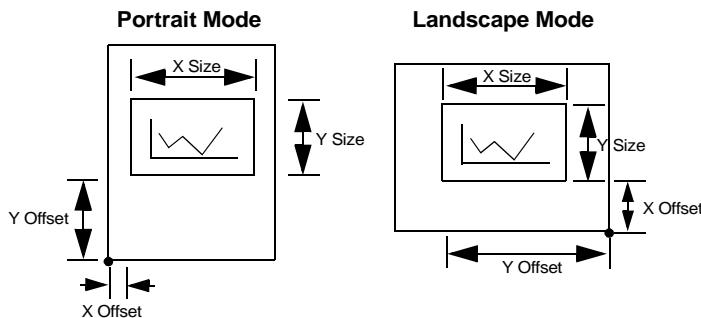


Figure 2: The window sizes and offsets for PostScript output. Notice that in landscape mode the entire page is rotated 90 degrees, causing the X and Y offsets to be rotated 90 degrees from the X and Y sizes. This can cause great confusion for unwary users.

If you didn't realize that the offset point actually rotated with the page, you might set the offsets so that the graphic was rotated off the page. For example, suppose you want both your X and Y offsets set at one inch and you did this.

```
IDL> Set_Plot, 'ps'
IDL> Device, XOffset=1.0, YOffset=1.0, /Inches, $
      /Landscape
IDL> Plot, data
```

You can see in Figure 3 the figure you *think* you are producing with these offsets and what you actually produce. Be sure you understand how offsets work in landscape mode or it is possible to rotate the graphic completely off the PostScript page! If this happens, you will not see anything at all in your PostScript file.

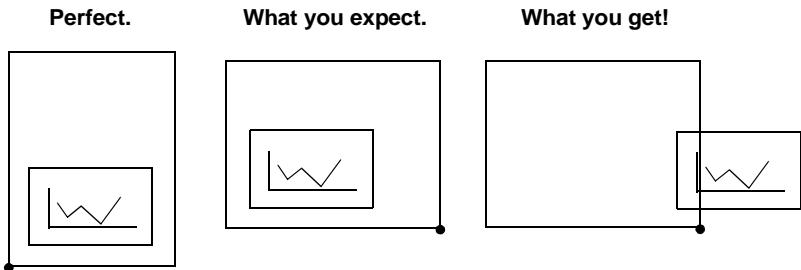


Figure 3: If you don't understand the direction and origin of offsets in landscape mode, it is possible to rotate the graphic completely off the PostScript page.

My Landscape Files Are Upside Down!

You will notice if you create PostScript output in landscape mode, and then open those files in almost any other program that can read PostScript files, that your files are upside down. In GhostView such files are called “seascape” files, which I think is a polite word for “foobar.” In any case, it is massively annoying.

Fortunately, PostScript files are ASCII text files, so they can be opened up, read by humans, edited if need be, and even fixed if they have been written with incorrect information. And that is exactly what needs to be done with landscape files. (If you are interested, you can read an article that describes all the details on my web page: www.idlcoyote.com/ps_tips/landscapeup.html).

For our purposes here, you can simply use the [Coyote Library](#) routine [FixPS](#) to edit the landscape files created in IDL so that they are truly in “landscape orientation” to the rest of the world. If you don’t trust [FixPS](#) entirely, you can save the input file to an output file of some other name.

```
IDL> FixPS, 'test.ps', 'test_fixed.ps'
```

Or, if you are like me and believe the world is not as dangerous as everyone says it is, you can just let [FixPS](#) change the file on the fly. [FixPS](#) is smart enough to ignore a file that is already in the correct landscape mode.

```
IDL> FixPS, 'test.ps'
```

I call [FixPS](#) automatically for all PostScript files I create in landscape mode (via [PS_End](#), which is discussed in “The Role of PS_End” on page 414). I’ve never known it to fail on a file, and I don’t worry about upside-down landscape files any more.

Matching the Window Aspect Ratio

If you are trying to write a program that looks the same both on your computer display and in PostScript file output, then you will want to keep the aspect ratio (the ratio of height to width) the same for the graphics window and for the PostScript “window” you create for the PostScript file. And most people like to center the window on the page.

Since graphics windows on your display are sized in pixel units, and PostScript windows are sized in centimeter or inch units, and since a PostScript page comes in different sizes (e.g., US letter or metric A4 size), this can become an exercise in tedious arithmetic. In other words, a task perfectly suited for a computer.

The [Coyote Library](#) contains a function, named [PSWindow](#), that can “match” the aspect ratio of the current graphics window with a centered PostScript window having the same aspect ratio. The resulting PostScript window will be as large as possible to maintain a window with this aspect ratio. This is convenient for me because I “test” programs on the display and then “send” the programs to PostScript (usually with a single command, as you will see below). My PostScript output will have the same aspect ratio as the “test” window on the display.

Consider a graphics window that is 600 by 400. What is returned by [PSWindow](#) is a structure of PostScript *Device* keywords.

```
IDL> Window, XSize=600, YSize=400
IDL> output = PSWindow()
IDL> Help, output, /Structure
** Structure PSWINDOW_STRUCT, 7 tags:
      XSIZE          FLOAT          9.90000
      YSIZE          FLOAT          6.60000
      XOFFSET        FLOAT          0.950000
      YOFFSET        FLOAT          10.4500
      INCHES         INT            1
      PORTRAIT       INT            0
      LANDSCAPE      INT            1
```

In this particular case, because the graphics window has a “landscape” aspect ratio (its horizontal dimension is greater than its vertical dimension), the return values from [PSWindow](#) set up a landscape PostScript page with a “window” that has an aspect ratio of 6/4. (Keywords to [PSWindow](#) allow other configurations, but here we are talking about default behavior.)

Note that the sizes and offsets are in inches and that the *Inches* field of the structure is set to 1. You will also note, after some investigation no doubt, that the sizes are appropriate for a US letter sized page.

All this can be changed by setting the *Metric* keyword, in which case the sizes are expressed in centimeters and the “window” is sized to an A4 size PostScript page.

```
IDL> Window, XSize=600, YSize=400
IDL> output = PSWindow(/Metric)
IDL> Help, output, /Structure
** Structure PSWINDOW_STRUCT, 7 tags:
      XSIZE          FLOAT        26.7462
      YSIZE          FLOAT        17.8308
      XOFFSET        FLOAT        1.58750
      YOFFSET        FLOAT        28.2321
      INCHES         INT          0
      PORTRAIT       INT          0
      LANDSCAPE      INT          1
```

You notice that the fields of the return structure are identical to keywords you can use with the *Device* command to configure the PostScript device. Rather than using individual fields to set individual keywords, the entire structure is passed to the *Device* command using the keyword inheritance method in IDL. In other words, to configure the PostScript device with this graphics “window”, we simply type commands like this.

```
IDL> Set_Plot, 'PS'
IDL> Device, _Extra=output
```

Once the output structure has been passed to the *Device* command like this, the PostScript device is configured properly, and you can start typing your graphics commands, just as you would normally.

Setting the Window Aspect Ratio

At other times you don’t necessarily want to match the window aspect ratio so much as you want to *set* the window aspect ratio. Maybe, for example, you want to create a plot that the user can use a ruler with to measure a particular quantity. In this case, the aspect ratio of the plot window might be critical.

The **PSWindow** command can be used to create a centered, large as possible, window with a specified aspect ratio by using the *AspectRatio* keyword. For example, if you would like to have a PostScript window that was 2 units high and three units wide, you can type this command.

```
IDL> output = PSWindow(AspectRatio=2./3)
```

```
IDL> Help, output, /Structure
** Structure PSWINDOW_STRUCT, 7 tags:
    XSIZE          FLOAT      9.90000
    YSIZE          FLOAT      6.60000
    XOFFSET        FLOAT      0.950000
    YOFFSET        FLOAT      10.4500
    INCHES         INT       1
    PORTRAIT       INT       0
    LANDSCAPE      INT       1
```

Another way to set a plot aspect ratio is to use the [Aspect](#) command from the [Coyote Library](#). The [Aspect](#) command produces a four-element normalized array that can be used with the *Position* keyword on any graphics command to produce a graphic with a particular aspect ratio in any sized graphics window, including PostScript windows.

```
IDL> Plot, [4,3,8,6,2], Position=Aspect(2.0/3)
```

These methods are used by [cgDisplay](#) to create the properly sized “window” for the PostScript device. For example, the following command produces a 500x500 pixel window on the display device, or sets the resolution of the Z-graphics device to 500x500 pixels, or centers a large window with a square aspect ratio on the PostScript page, depending on the value of *!D.Name* when the command is issued.

```
IDL> cgDisplay, 500, 500
```

By using [cgDisplay](#) to “open” a graphics window, my graphics output always looks as identical as it is possible to look on three different graphics output devices.

Producing Encapsulated PostScript Output

To produce PostScript output that can be inserted directly into other PostScript documents, such as journal articles or books, you must select the encapsulated option before you send graphics output to the PostScript file.

```
IDL> Set_Plot, 'PS'
IDL> Device, /Encapsulated
```

IDL encapsulated PostScript output can be placed into LaTeX, Microsoft Word, FrameMaker, and other word processing documents.

Note that encapsulated PostScript files will not, generally, print on a PostScript printer by themselves, although the printer will churn away and otherwise act as if it were processing the file. Encapsulated PostScript files have traditionally lacked the PostScript *showpage* command, which is

used to eject the page from the printer. (More recent versions of the PostScript specifications actually allow the *showpage* command in encapsulated files.) The files must be included or “encapsulated” within another document to print properly. It is not unusual these days to find PostScript previewers, such as GhostView, that will allow you to print encapsulated PostScript files. But the previewer is providing the wrapper for you, rather than you having to provide it explicitly.

Note also that when you import the encapsulated file into the other document, you probably will not be able to see the graphic until it is printed, unless you have a PostScript previewer or you use the *Preview* keyword described below.

To turn encapsulated PostScript off, set the *Encapsulated* keyword to 0, like this.

```
IDL> Device, Encapsulated=0
```

Encapsulated PostScript Graphic Preview

Normally, encapsulated PostScript files are not displayed in the document into which they have been imported. That is to say, the frame that holds the imported file usually is blank or is grayed out. However, the PostScript graphic will print properly when the document is sent to a PostScript printer.

If you would like to be able to see a representation of the graphic in the document into which you imported the file, you must specify a value for the *Preview* keyword. A *Preview* keyword value of 1 will cause the PostScript driver to include a bit-mapped image of the graphic along with its PostScript description. The bit-mapped image will be shown in the document frame and the PostScript description will be used when the document is printed.

```
IDL> Device, /Encapsulated, Preview=1
```

Note that not all word processing programs are capable of displaying a bit-mapped preview image. Experiment with your word processing software to see how well it displays on your machine. UNIX-based word processing software will often display a bit-mapped preview image, whereas Windows word processing software will more often display a TIFF preview image (see below).

If you would prefer to use a TIFF preview image, instead of a bitmap ped image, set the *Preview* keyword to 2.

```
IDL> Device, /Encapsulated, Preview=2
```

To turn the preview option off, set the *Preview* keyword to 0.

```
IDL> Device, Preview=0
```

Not all software is capable of showing a preview image and some software may show TIFF images better than bit-mapped images. Also, in the past IDL's preview images have been low-resolution and not very attractive. Many users found it was easier to produce better looking preview images in PostScript previewers, such as GhostView or GhostScript. It pays to experiment with the software you use to see what is possible and whether it meets your needs.

Setting the Language Level

There are three basic versions of PostScript. Level 1 is the basic PostScript language. Level 2, released in 1992, has better support for color printing and for some graphics functionality. Level 3 was released in 1997 and supports more fonts, better graphics handling, and includes several features to speed up PostScript printing. IDL does not support Level 3 PostScript as of IDL 8.0.

IDL does support PostScript Level 1 and 2. You choose which language level you want to support by means of a *Language_Level* keyword. There are only two reasons that I know to choose language level 2. One is if you want to convert PostScript RGB output to CMYK output for better support of color printers. The other is if you want to fill polygons with textures instead of colors.

```
IDL> Device, Language_Level=2
```

Using PostScript Fonts

There is a bit more involved in using fonts correctly in PostScript files than just configuring the PostScript device with the proper keywords. But this seems as good a place as any to discuss the situation, since users are often thinking about which fonts to use when they configure the PostScript device.

One thing you can do, however, with respect to PostScript configuration, is turn Isolatin encoding on for the PostScript device. This allows you to use certain characters that are not found in a Western character set, or have ASCII values above 127, as well as allowing you to use Unicode characters in PostScript. This is one of those defaults that I set every time I set up a PostScript device, similar to the way I set the *Color* and *Bits_Per_Pixel* keyword. (Note the spelling of the *IsoLatin1* keyword.

That is not a typo or extra letter at the end of the word, it is the number 1. I have no idea why it is there.)

```
Device, /IsoLatin1
```

The default font type in IDL are the vector fonts we call Hershey fonts. These are the fonts that are used if we simply use the *Plot* command to create a graphics display, for example. Hershey fonts, as vector fonts, can be rotated in 3D space and seem to be perfectly adequate for looking at graphical displays on the computer display. They are created as vectors of various pen “widths.”

But Hershey fonts sometimes seem a bit thin and inadequate to be used in PostScript output. Most of the time we prefer to use either PostScript hardware fonts, which I will discuss in more detail in a moment, or TrueType fonts, which are like Hershey fonts in that they can easily be rotated in 3D space. They are unlike Hershey fonts, however, in being “filled” fonts, rather than vector fonts. In other words, TrueType fonts are made from filled polygons rather than lines of various thickness, and so have a more substantial and “full” appearance.

The font type is typically selected with the *!P.Font* system variable, although it is also possible to use the *Font* keyword on most traditional IDL commands. A value of -1 selects Hershey fonts, a value of 0 selects hardware fonts, and a value of 1 selects TrueType fonts.

TrueType fonts almost always appear slightly smaller than comparable Hershey or hardware fonts, so when TrueType fonts are used, the font size is often increased a bit with either the *!P.Charsize* system variable or the *Charsize* keyword on a traditional graphics command.

Here is code that displays the same plot in a PostScript file using the three different font systems available.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, _Extra=PSWindow(AspectRatio=1.0/3)
IDL> !P.Multi = [0,3,1]
IDL> data = cgDemoData(1)
IDL> cgPlot, data, Font=-1, Title='Hershey Fonts'
IDL> cgPlot, data, Font=0, Title='Hardware Fonts'
IDL> cgPlot, data, Font=1, Title='TrueType Fonts'
IDL> !P.Multi = 0
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

You see the result in Figure 4.

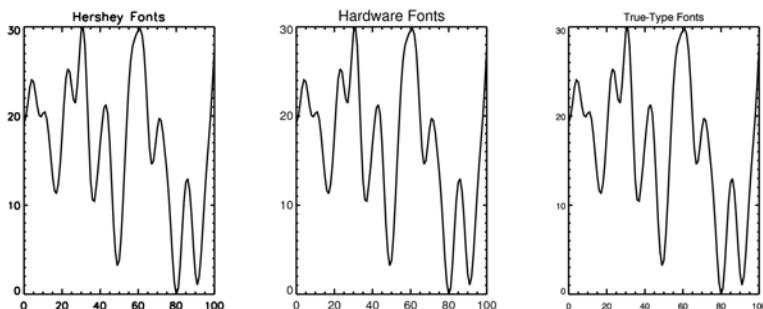


Figure 4: The same plot drawn in a PostScript file with Hershey fonts (left), Hardware fonts (middle), and TrueType fonts (right).

If you elect to use hardware fonts as you create the PostScript file, then there is a default mapping from Hershey fonts to PostScript hardware fonts. The mapping is displayed in the output from the *Help* command with the *Device* keyword set, but you can see it also in Table 1.

The default Hershey font is Simplex Roman (selected with a !3 character encoding) and that is mapped to the Helvetica PostScript font. If you want to use a Times-Roman PostScript font, you could place a !7 in front to the text you want to display in this font. Normally, you end a text string with !X to return the text to the default encoding.

Number	Hershey Font	PostScript Font
!3	Simplex Roman	Helvetica
!4	Simplex Greek	Helvetica-Bold
!5	Duplex Roman	Helvetica-Narrow
!6	Complex Roman	Helvetica-Narrow-BoldOblique
!7	Complex Greek	Times-Roman
!8	Complex Italic	Times-BoldItalic
!9	Math and Special	Symbol

Table 1: The default mapping from Hershey fonts to PostScript fonts.

Number	Hershey Font	PostScript Font
!10	Special	ZapfDingbats
!11	Gothic English	Courier
!12	Simplex Script	Courier-Oblique
!13	Complex Script	Palatino-Roman
!14	Gothic Italian	Palatino-Italic
!15	Gothic German	Palatino-Bold
!16	Cyrillic	Palatino-BoldItalic
!17	Triplex Roman	AvantGarde-Book
!18	Triplex Italic	NewCenturySchlbk-Roman
!19	Undefined	NewCenturySchlbk-Bold
!20	Miscellaneous	Undefined

Table 1: The default mapping from Hershey fonts to PostScript fonts.

```
IDL> cgPlot, data, Title='!7Times Roman Font!X'
```

Another way to choose a PostScript hardware font is with keywords to the *Device* command after you have switched to the PostScript device. Most PostScript printers come with 5 to 15 PostScript fonts pre-installed. The most common of these fonts can be selected by means of keywords to the *Device* command. You can also select the weight and style of the fonts using this method.

For example, if you want to create a plot with a ZapfChancery Bold font, you can create such a plot like this. Remember, though, that hardware font selection is *not* done by configuring the PostScript device. It is done either by setting the *Font* keyword to 0 on the graphics command or by setting the *!P.Font* system variable to 0 so that all graphics commands use hardware fonts.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, /Bold, /ZapfChancery, Filename='zaph.ps'
IDL> cgPlot, data, Font=0, title='ZapfChancery Font', $
      XTitle='Time', YTitle='Signal Strength'
```

```
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

You see the result in Figure 5. Remember that PostScript configurations are “sticky,” so you will be using the ZapfChancery font for all your PostScript hardcopy font output in this session, until you change it from within IDL.

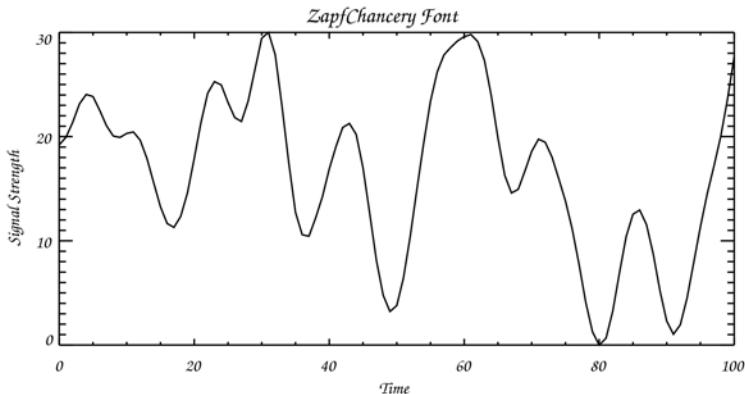


Figure 5: Selecting a PostScript hardware font for a plot with Device keywords. In this case, a ZapfChancery font in a bold typeface.

Choosing TrueType Fonts

TrueType fonts are selected in PostScript the way they are anywhere else in IDL. You set the *!P.Font* system variable or the *Font* keyword on a graphic command to 1. The TrueType font actually used for the plot is determined by the value of the *Set_Font* keyword. You must also set the *TT_Font* keyword when you set the TrueType font.

For example, to set the TrueType font to a Times font, you would do this.

```
IDL> Set_Plot, 'PS'
IDL> Device, Set_Font='Times', /TT_Font
IDL> !P.Font = 1
```

You may use any TrueType font you have installed on your computer. However, if your goal is to write machine-portable code, it is a good idea to stick to the four TrueType font families distributed with IDL. These are Helvetica, Times, Courier, and Symbol. The first three of these come in Bold and Italic versions.

```
IDL> Device, Set_Font='Times*Bold', /TT_Font  
IDL> Device, Set_Font='Times*Bold*Italic', /TT_Font  
IDL> Device, Set_Font='Times*Italic', /TT_Font
```

The default TrueType font family is Helvetica.

Choosing Rotating Fonts in PostScript

Not all fonts can be rotated. For example, if you choose hardware fonts for a plot on the computer display, such fonts cannot be rotated at all.

```
IDL> Set_Plot, thisDevice  
IDL> cgPlot, data, Font=0, XTitle='Good Fonts Here', $  
YTitle='Bad Fonts Here'
```

You see the result in Figure 6. Notice how the Y axis title is not rotated properly.

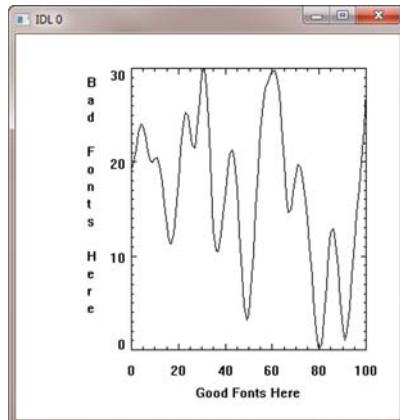


Figure 6: Computer hardware fonts cannot rotate.

PostScript hardware fonts can be rotated, but only in a flat plane, not in 3D space. Here is code that produces a line plot and a surface plot in a PostScript file.

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'PS'  
IDL> Device, _Extra=PSWindow(AspectRatio=1.0/2)  
IDL> !P.Multi = [0,2,1]  
IDL> cgPlot, cgDemoData(1), Font=0, $  
XTitle='Good Fonts', YTitle='And Good Fonts'  
IDL> cgSurf, cgDemoData(2), XTitle='X Fonts', $  
YTitle='Y Fonts', ZTitle='Z Fonts', Font=0
```

```
IDL> !P.Multi = 0
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

Note the problem with the titles on the surface plot in Figure 7.

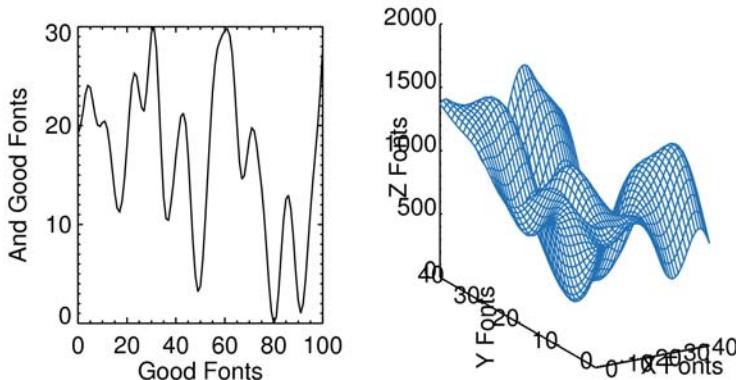


Figure 7: PostScript hardware fonts rotate in the XY plane, but not in 3D space.

The problem with surfaces can be solved by using Hershey fonts or, even better, TrueType fonts.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, _Extra=PSWindow(AspectRatio=1.0/2)
IDL> !P.Multi = [0,2,1]
IDL> cgSurf, cgDemoData(2), XTitle='X Fonts', $ 
      YTitle='Y Fonts', ZTitle='Z Fonts', $ 
      Font=-1, Title='Hershey Fonts'
IDL> cgSurf, cgDemoData(2), XTitle='X Fonts', $ 
      YTitle='Y Fonts', ZTitle='Z Fonts', $ 
      Font=1, Title='TrueType Fonts'
IDL> !P.Multi = 0
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

You see the result in Figure 8. Compare this surface result with the result in Figure 7.

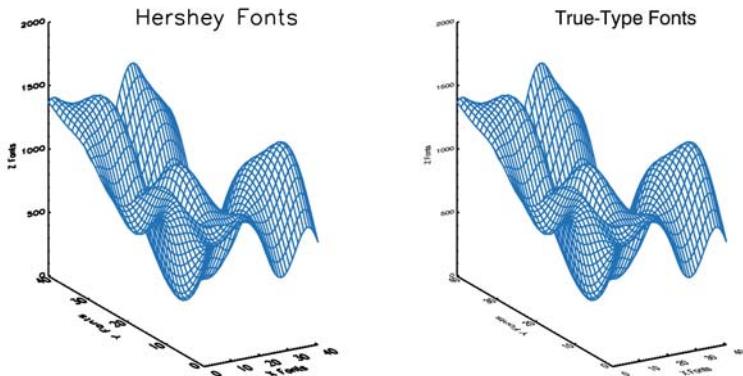


Figure 8: To rotate text in 3D in PostScript, you must use either Hershey or TrueType fonts.

Using Greek Characters in PostScript

Using Greek characters in PostScript is also a bit more complicated than you probably want it to be. Suppose, for example, you want to use the Greek character *mu* in a title. Perhaps you want the title to read “Wavelength (μm)” How would you go about doing this?

If you want to use the standard Hershey font, you would probably find the *mu* character in one of the octal tables that represent the Hershey character fonts. A likely place to find such a character is in the Greek Simplex character font, font number 4. You see a representation of this font table in Figure 9. (Font tables are notoriously hard to find in the IDL on-line help, even when you know they are there. Try searching the index for the word “fonts,” then look for anything that says “Hershey fonts.” They will be somewhere around there. Don’t search for “font table,” because that never works.)

The way the table is used is to find the Greek letter you are interested in and read the number in the first column of the row the letter is in. This is 14x in the case of *mu*. Multiply this number by 10, then take the number at the top of the column the letter is in and add it to the first number. We have $14*10+14=154$. This number is an octal number and a byte value. (The fact that it is a byte value is extremely important.) In IDL we write an octal number with a double quote in front of it like this. The B at the right end of the number indicates this is a byte type number.

```
IDL> value = "154B
```

Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17
04x	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
06x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
10x	@	A	B	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
12x	Π	Ρ	Σ	Τ	Τ	Φ	Χ	Ψ	Ω	∞	Ζ	[\]	^	_
14x	'	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
16x	π	ρ	σ	τ	υ	φ	χ	ψ	ω	∞	Ζ	[\]	^	_

Figure 9: The Simplex Greek font table. One of the Hershey vector character fonts available in IDL.

We need to make a string out of this number, so we have to cast it.

```
IDL> mu = '!4' + String(value) + '!X'
```

The “!4” in front of the string value selects the Simplex Greek font, font 4, and the “!X” behind the string value reverts the string to whatever font was in effect before we changed it to font 4. We can make a string using this letter and display it in a PostScript file (or, on the display for that matter), if we are using Hershey character fonts.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> mu = '!4' + String("154B") + '!X'
IDL> cgPlot, cgDemoData(1), Font=-1, $
      XTitle='Wavelength (' + mu + 'm)'
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

You see the result in Figure 10.

But what if you prefer to use hardware or TrueType fonts in PostScript instead of Hershey fonts. Does this technique still work?

Unfortunately, no. The problem is that you can't use the Simplex Greek font, which is a Hershey font. You have to use either the PostScript hardware Symbol font or a TrueType Symbol font (these are often the same, although I hear rumors that some versions of Macintosh computers are using a Symbol font that is different from these.) In any case, you see the Symbol font table for the TrueType font distributed with IDL in Figure 11.

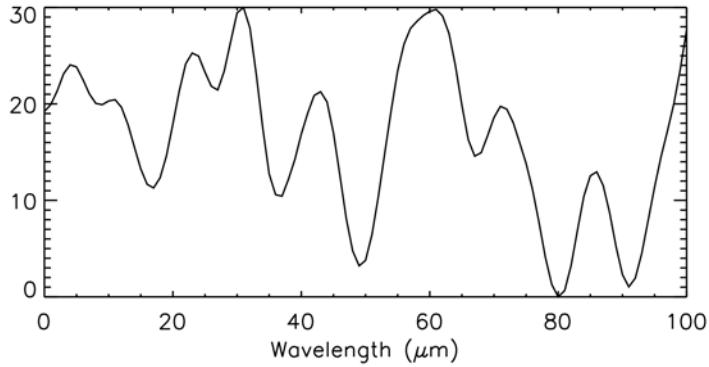


Figure 10: Greek letters can be used in PostScript using the Greek Simplex font and selecting Hershey fonts.

Font Symbol																	
Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17	
04x	!	∀	#	Ξ	%	&	Ξ	()	*	+	,	-	.	/		
05x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	
10x	≡	A	B	X	Δ	E	Φ	↳	H	I	∅	K	Λ	M	N	O	
12x	Π	Θ	P	Σ	T	Y	s	Ω	Ξ	Ψ	Z	[..]	⊥	-	
14x	α	β	χ	δ	ε	φ	γ	η	ι	φ	κ	λ	μ	ν	ο		
16x	π	θ	ρ	σ	τ	v	w	ω	ξ	ψ	ζ	{		}	~	□	
20x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	
22x	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	
24x	□	Y	'	≤	/	∞	f	♣	♦	♥	♠	↔	↑	→	↓		
26x	°	±	"	≥	×	×	∂	•	÷	≠	≡	≈	...		—	↔	
30x	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
32x	△	▽	®	©	™	∏	√	·	¬	∧	∨	↔	⇒	↑	⇒	↓	
34x	◊	{	®	©	™	Σ	(\					}		□	
36x	apple)	∫		J	\)			J			}	J	□	

Figure 11: The font table for the TrueType Symbol font distributed with IDL.

If you perform the same octal arithmetic for the *mu* letter in this table, you will find that the octal value you want is 155, not 154 as it was before with the Simplex Greek Hershey font. And, a “19” must be inserted in front of the letter string to select the Symbol font, rather than the “!4” we used before.

So, to create this same plot using TrueType fonts, we have to type this code.

```
IDL> thisDevice = !D.Name
```

```

IDL> Set_Plot, 'PS'
IDL> mu = '!9' + String("155B") + '!X'
IDL> cgPlot, cgDemoData(1), Font=1, $
      XTitle='Wavelength (' + mu + 'm)'
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice

```

You see the result in Figure 12.

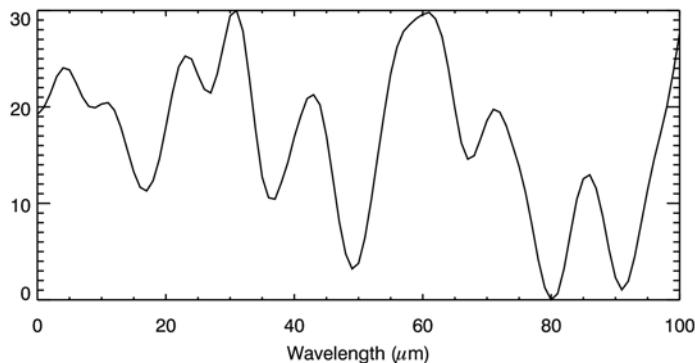


Figure 12: With *hardcopy* and *TrueType* fonts, Greek characters are created differently, using the *Symbol* font and different octal values.

The switch necessary to create Greek characters one way for Hershey fonts and another way for PostScript hardware and TrueType fonts is unfortunate, to say the least, because it makes it very difficult to write device-independent graphics programs. But, don't give up hope, yet.

The [Coyote Library](#) contains a program named [Greek](#) that will make this switch for you, on the fly, depending upon the setting of the *!P.Font* system variable. If this system variable is set to -1, the Simplex Greek version of the Greek letter will be returned. If it is anything else, the Symbol version of the Greek letter will be returned.

You can select lowercase Greek letters by specifying the Greek letter wanted in lowercase letters. You can select uppercase Greek letters either by setting the *Capital* keyword or by making the first letter of the Greek letter name a capital letter. To see a list of the Greek letters available, with their names, set the *Example* keyword.

```
IDL> xtitle = 'Wavelength (' + Greek('mu') + 'm)'
IDL> cgPlot, cgDemoData(1), XTitle=xtitle
IDL> void = Greek(/Example)
```

You see the result in Figure 13.

alpha:	α A	nu:	vN
beta:	β B	xi:	ξ E
gamma:	γ G	omicron:	oO
delta:	δ D	pi:	π I
epsilon:	ϵ E	rho:	ρ P
zeta:	ζ Z	sigma:	σ S
eta:	η H	tau:	τ T
theta:	θ O	upsilon:	υ Y
iota:	ι I	phi:	φ ø
kappa:	κ K	chi:	χ X
lambda:	λ L	psi:	ψ Ψ
mu:	μ M	omega:	ω Ω

Figure 13: The Greek letters, both lowercase and capital, that are available with the Greek command. Select a Greek letter by using its name. Capital letters can be selected by capitalizing the first letter of the Greek name.

If you run into a situation where you need to use Unicode character encoding for Greek letters (I have heard rumors that the Symbol font supplied with new Macintosh computers will require this), you just need to set the *Unicode* keyword and the Greek letters are returned as a Unicode string.

```
IDL> text = 'Wavelength (' + Greek('mu', /Unicode) + 'm')
```

Using Special Characters in PostScript

The situation with special characters in PostScript is similar to that described for Greek characters above. Namely, you create a special character one way for Hershey fonts and some other way if you want to use PostScript or TrueType fonts. Let's use as an example of a special character the less-than-or-equal sign (\leq).

For Hershey fonts, we need to look at the font table for Font 9, the Math and Symbol font. You see a replication of the table in Figure 14. The less-than-or-equal sign is represented as the octal value $14*10+14=154$. A total coincidence I can assure you!

As before, we can use this value directly in some text.

```
IDL> le_sign = '!9' + String("154B") + '!X'
```

Font 9, Math and Special																	
Octal	00	01	02	03	04	05	06	07	10	11	12	13	14	15	16	17	
04x		"		∞	°	§	,	()	*	±	*	≠	*	÷		
06x	≤	≥	≈	↖	↙	↘	↗	↑		≡		{	≠	}	∞		
10x	✗	~	□	✓	∂	Ǝ	ℐ	∇	∫	∅			=	†			
12x	φ		√	·	♪	◊	⊕	×		[]	^	--				
14x	'	∠	≥	≈	∂	∈	♀	∅	∫	ʃ	≤	σ	○	‡			
16x	ρ	q	√	ς	ϑ	♡	⊕	₩	⊥						^	--	

Figure 14: The Hershey Math and Symbol font, Font 9.

```
IDL> XYOutS, 0.5, 0.5, Alignment=0.5, /Normal, $
      'This is the LESS-THEN-OR-EQUAL (' + $_
       le_sign + ') sign.', Color=cgColor('opposite')
```

For PostScript and TrueType fonts, the situation is slightly more complicated than with Greek characters. With special characters we pretty much have to use Unicode character representations of special characters. This means that we must turn on Isolatin encoding in the PostScript device, or these Unicode characters will not be recognized.

```
IDL> Set_Plot, 'PS'
IDL> Device, /IsoLatin1
```

I must caution you that not all fonts have Unicode special characters, and even those fonts that do have Unicode special characters do not always code the same special character in the same way. The Unicode “standard” is an evolving standard, apparently. In any case, you must check to be sure the font you are using supports Unicode special characters and to find the proper Unicode “value” for the character you want to use. You can find Unicode charts on the Internet (<http://www.unicode.org/>).

Checking the Unicode web pages under the *Mathematical Operators* section, I find that the less-than-or-equal sign should be represented by the Unicode value 2264. Note that this value is a hexadecimal value, not an octal value or a decimal value!

Unicode values are represented in IDL with the !Z(*unicode*) syntax, where *unicode* is the four-digit hexadecimal Unicode value. You may have more than one Unicode value inside the parentheses if the values are separated by commas.

Let's use this value with the Times TrueType font in a PostScript file. Note that the Symbol TrueType font distributed with IDL is *not* a Unicode encoded font! If you attempt to draw a Unicode symbol with a non-Unicode font, the "symbol" usually shows up as a small box.

Here is code that displays the same text string in a PostScript file as a Hershey font and as a Times TrueType font.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, /IsoLatin1, Set_Font='Times', /TT_Font, $
      XSize=5, YSize=1, /Inches
IDL> le_sign = '!9' + String("154B") + '!X'
IDL> XYOutS, 0.5, 0.66, Alignment=0.5, /Normal, $
      'This is the LESS-THEN-OR-EQUAL (' + $
      le_sign + ') sign.', Color=cgColor('opposite')
IDL> XYOutS, 0.5, 0.33, Alignment=0.5, /Normal, $
      'This is the LESS-THEN-OR-EQUAL (!Z(2264)' + $
      ') sign.', Color=cgColor('opposite'), Font=1
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

You see the result in Figure 15.

This is the LESS-THEN-OR-EQUAL (\leq) sign.

This is the LESS-THEN-OR-EQUAL (\leq) sign.

Figure 15: Special characters in PostScript using the Hershey fonts (above) and TrueType fonts (below).

Configuring the PostScript Device Interactively

To solve nearly all of the PostScript configuration problems mentioned so far, I configure the PostScript device with [PSConfig](#). This program is actually a wrapper program for the first IDL object program I ever wrote, named [FSC_PSConfig](#). The point is, the [PSConfig](#) program can return all the keywords needed to configure the PostScript device in a single structure variable and, more to the point, the keywords are set to default values that actually make the PostScript device useful to you, as opposed to the out-of-box alternatives.

Naturally, you don't have to use [PSConfig](#) to configure your PostScript device, but you probably want something like it.

The sequence of commands for interactively configuring the PostScript device with [PSConfig](#) looks like this.

```
keywords = PSConfig(Cancel=cancelled)
IF cancelled THEN RETURN
thisDevice = !D.Name
Set_Plot, 'PS'
Device, _Extra=keywords
```

By default, [PSConfig](#) presents the user with a graphical user interface by which the PostScript device can be configured. You see an example of [PSConfig](#) in Figure 16.

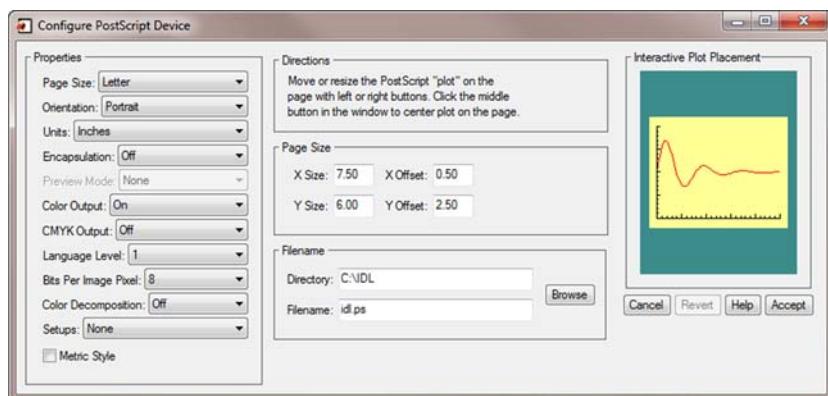


Figure 16: The basic [PSConfig](#) program for configuring the PostScript device.

Basic properties of the PostScript device can be set via pull-down dropdowns on the left-hand side of the program. You can change the page size, specify the units of measure, set up to do encapsulated PostScript (with or without previews), and turn color decomposition and other features on or off. There is a button at the bottom of the Properties section of the program that allows metric style operation (centimeters instead of inches, A4 page size, etc.) with a single click.

The right-hand side of the program shows you the window (yellow box) on the PostScript page (green box) where graphics will be drawn. The size and placement of the box can be changed using the text fields in the middle section of the program (which will be expressed in inches or

centimeters, depending upon the setting of the Units droplist), or by just dragging the window around and resizing the yellow box with the cursor. A click of the middle mouse button will center the yellow box in the green box. Grab any corner or side of the yellow box to resize it.

There is a pull-down menu named Setups near the bottom of the Properties box on the left. Here you can find named “set-ups” that configure the PostScript device in particular ways. It is a trivial job to add your own favorite set-ups to this list, so that your PostScript device can literally be configured with a click of a button.

You will notice that if you change the page orientation to Landscape that the green box will flip to landscape mode, but that the offsets will continue to be measured from the lower-left corner of the box. This is incorrect, of course, from the IDL PostScript device’s point of view, but it presents a consistent (and understandable) interface to the user. The proper (strange?) offsets will be filled out correctly behind the scenes, so that the PostScript device receives what it needs and the result is what the user expects.

PSConfig makes sure that color is always turned on, the keyword *Bits_Per_Pixel* is always set to 8, Isolatin encoding is always turned on, and the plot is centered in the PostScript window and made as large as possible.

You can, of course, set all of the **PSConfig** properties by means of keywords when the program is called, so the initial configuration can be anything you would like it to be. Many people will specify an initial file name, for example, and people outside the United States will probably set the *Metric* keyword to give the interface a non-American look.

```
IDL> keywords = PSConfig(Filename='coyote.eps', $  
/Metric, /Landscape, /Encapsulated)
```

If **PSConfig** is called with the *Match* keyword set, the initial window configuration will match the aspect ratio of the current graphics window. This makes it much easier to create PostScript output that appears nearly identical to what you see in a display window.

```
IDL> cgDisplay, 600, 300  
IDL> keywords = PSConfig(/Match)
```

You see the result in Figure 17.

When the *Accept* button is clicked, the program returns an IDL structure variable whose fields are PostScript keywords for the *Device* command.

```
IDL> Help, keywords, /Structure
```

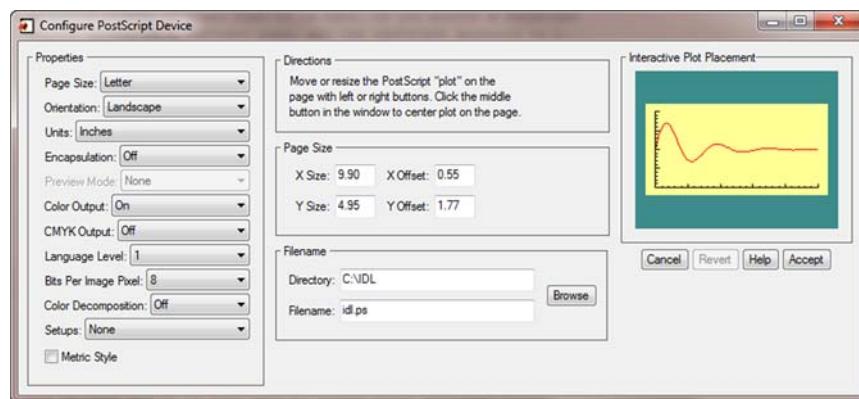


Figure 17: By setting the **Match** keyword when **PSConfig** is called, the initial graphics window will match the aspect ratio of the current graphics window, making it easier to create PostScript files that look nearly identical to what you see on the display.

```
** Structure <7234310>, 28 tags:
BITS_PER_PIXEL    INT          8
CMYK              INT          0
COLOR              INT          1
ENCAPSULATED      INT          0
FILENAME           STRING       'C:\IDL\idl.ps'
FONT_SIZE          INT          12
INCHES              INT          1
ISOLATIN1          INT          1
LANGUAGE_LEVEL     INT          1
PREVIEW             INT          0
TT_FONT              INT          0
XOFFSET             FLOAT        0.500000
XSIZE               FLOAT        7.50000
YOFFSET             FLOAT        2.50000
YSIZE               FLOAT        6.00000
PAGETYPE            STRING       'LETTER'
FONNTYPE             INT          -1
DECOMPOSED          INT          0
PORTRAIT             INT          1
LANDSCAPE            INT          0
HELVETICA            INT          1
BOLD                 INT          0
BOOK                 INT          0
DEMI                 INT          0
ITALIC                INT          0
```

LIGHT	INT	0
MEDIUM	INT	0
NARROW	INT	0
OBLIQUE	INT	0

These keywords are passed along to the PostScript device using the “keyword inheritance” mechanism.

```
IDL> Set_Plot, 'PS'
IDL> Device, _Extra=keywords
```

Notice the “extra” keywords *PageType* and *FontType*. These are returned as informational keywords and do no harm when passed to the *Device* command, as they are completely ignored.

Collecting Font Information with PSConfig

Font information can also be collected with [PSConfig](#), although as you have learned there is more to it than just configuring the PostScript device. Suppose, for example, you want to use PostScript hardware fonts. You could call [PSConfig](#) like this, setting the *FontInfo* keyword to add font information to the interface, and the *FontType* keyword to select PostScript hardware fonts as the starting type.

```
IDL> keywords = PSConfig(/FontInfo, FontType=0)
```

You see the result in Figure 18.



Figure 18: The **PSConfig** program interface with font information.

To use the font information successfully in the PostScript file, you have to set the *!P.Font* system variable to the *keywords.fonttype* value, in addition to passing the information along to the *Device* command. Normally, you

save the current `!P.Font` value, so it can be restored after you close the PostScript file. The sequence of commands looks like this.

```
thisDevice = !D.Name
thisFont = !P.Font
keywords = PSCconfig(Cancel=cancel, /FontInfo, FontType=1)
IF cancel THEN RETURN
!P.Font = keywords.fontType
Set_Plot, 'PS'
Device, _Extra=keywords
.... Graphics commands here...
Device, /Close_File
Set_Plot, thisDevice
!P.Font = thisFont
```

Using **PSCconfig** Without Its Interface

If you prefer not to use the graphical user interface to **PSCconfig** you simply have to set the `NoGUI` keyword. With this keyword set, the function returns the set-up keyword structure immediately after processing the input keywords. This makes it extremely easy to set up the PostScript device with sensible default values without having to bother with a lot of different keywords yourself.

```
IDL> Set_Plot, 'PS'
IDL> Device, _Extra=PSCconfig(Filename='coyote.ps', $
/Match, /NoGUI)
```

Maintaining the Current PostScript Configuration

I mentioned that **PSCconfig** is a wrapper to an **FSC_PSCconfig** object program. Objects are, of course, persistent in the IDL session, so it is possible to use the object underlying **PSCconfig** to keep a record of the current PostScript configuration state and to populate the **PSCconfig** interface at any time with the current PostScript configuration.

This is especially useful in a widget program where the user might be sending multiple files off to the PostScript device. You won't have to annoy the user by asking him to configure the PostScript device to his specifications each time. The **PSCconfig** interface will appear in the same state it was in when he last used it. All the user need do is perhaps change the name of the file (or you could do this for him), hit the *Accept* button, and the PostScript file is created.

Here is how you create the configuration object.

```
IDL> psConfigObject = Obj_New('FSC_PSCconfig')
```

And here is how you use the object whenever it is needed to populate the `PSConfig` interface with the current settings of the object.

```
IDL> keywords = PSConfig(psConfigObject)
```

Be sure you destroy the configuration object when you are finished with it. (Generally this is done in the `Cleanup` callback routine of a widget program.)

```
IDL> Obj_Destroy, psConfigObject
```

Displaying Images in PostScript

One difference between the display device on the computer and the PostScript device is that images are displayed differently. In particular, the display device has fixed-sized pixels and the PostScript device has scalable pixels. In other words, in PostScript a single pixel can be virtually *any* rectangular size. This affects the way images are output to the PostScript file.

What PostScript does is use the size of the PostScript “window” and the aspect ratio of the image to determine how big an image should be in the PostScript file. For example, if the PostScript drawing window is two-inches by two-inches, and the image to be output is 360 by 360 pixels, then a simple `TV` command results in a PostScript image of exactly two-inches by two-inches.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=2, YSize=2, /Inches, Decomposed=1
IDL> x = [0, 1, 1, 0, 0]
IDL> y = [0, 0, 1, 1, 0]
IDL> TV, cgDemoData(7)
IDL> cgPlots, x, y, Color='red', /Normal, Thick=2
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

You see the result in Figure 19. The PostScript “window” is the box drawn in red.

However, if the output window size does not have the same aspect ratio as the image itself, the image is sized so that it maintains its aspect ratio, and one dimension of the image completely fills the output window. For example, using the same image as above, here is an output box that is 2 inches in the X direction and 1 inch in the Y direction.

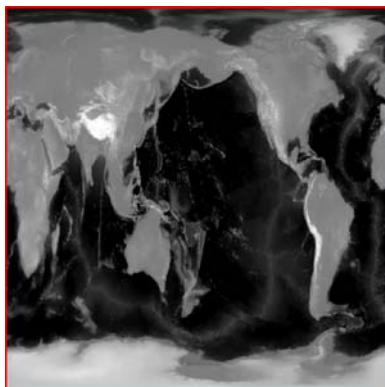


Figure 19: When the PostScript window and the image have the same aspect ratio, the PostScript window is completely filled with the image.

```
IDL> Set_Plot, 'PS'  
IDL> Device, XSize=2, YSize=1, /Inches, Decomposed=1  
IDL> TV, cgDemoData(7)  
IDL> cgPlotS, x, y, Color='red', /Normal, Thick=2  
IDL> Device, /Close_File
```

You see the result in Figure 20.

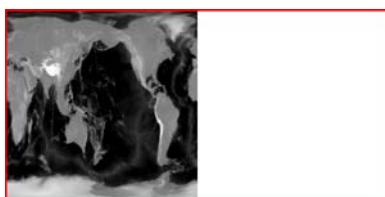


Figure 20: When the PostScript window has a different aspect ratio than the image, the image is placed in the window in such a way as to preserve its aspect ratio.

The fact that images are always sized according to the size of the output window and the aspect ratio of the image can cause difficulties. For example, suppose you have a 500x500 pixel display window and you want to center an image that was 400x400 pixels in the window. Suppose further that you want to draw an outline around the image. You might use these commands to display and position the image in the window.

```

IDL> image = Congrid(cgDemoData(7), 400, 400)
IDL> cgDisplay, 500, 500
IDL> TV, image, 0.1, 0.1, /Normal
IDL> cgPlot, Findgen(100), /NoData, /NoErase, $
      Position=[0.1, 0.1, 0.9, 0.9], Font=0

```

You see the result in Figure 21.

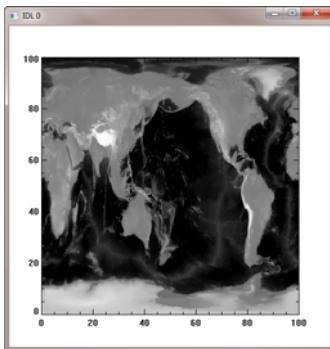


Figure 21: Fitting a set of axis around an image on the display.

But if you execute exactly the same commands in a PostScript file with a window size of 5 by 5 inches, you see something completely different.

```

IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> image = Congrid(cgDemoData(7), 400, 400)
IDL> Device, XSize=5, YSize=5, /Inches, Decomposed=1
IDL> TV, image, 0.1, 0.1, /Normal
IDL> cgPlot, Findgen(100), /NoData, /NoErase, $
      Position=[0.1, 0.1, 0.9, 0.9], Font=0
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice

```

You see the result in Figure 22.

Sizing PostScript Images Correctly

The solution to this problem is to not allow the PostScript file to size the images for you. Rather, you want to do it yourself. And you do it not with *Rebin* or *Congrid* or by sizing the image variable at all. In fact, you size images by using the *XSize* and *YSize* keywords available on the *TV* com-

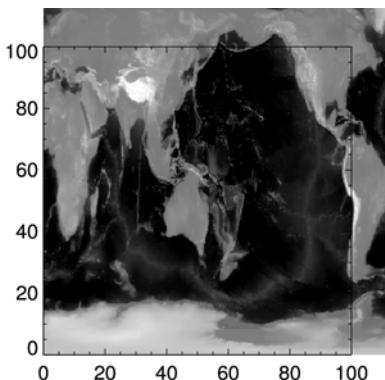


Figure 22: The image overflows its bounds in the PostScript file.

mand. (Of course, we no longer use the *TV* command, but you know what I mean.)

In other words, we can rescue the plot above by using these commands, which size the image with the *XSize* and *YSize* keywords to the *TV* command.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
IDL> Device, XSize=5, YSize=5, /Inches, Decomposed=1
IDL> TV, image, 0.5, 0.5, XSize=4, YSize=4, /Inches
IDL> cgPlot, Findgen(100), /NoData, /NoErase, $
      Position=[0.1, 0.1, 0.9, 0.9], Font=0
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice
```

You see the result in Figure 23.

Suppose, then, that you decide you would like to write a program, call it *ImageAxes*, that is able to display an image in a specific location in a graphics plot and wrap the image with a pair of axes. And, suppose you want it to work on the display, as in Figure 21, and in a PostScript file, as in Figure 23. You might write this program.

```
PRO ImageAxes, image, Position=position
IF N_Params() EQ 0 THEN $
  Message, 'Must pass image argument.'
IF N_Elements(position) EQ 0 THEN $
  position = [0.15, 0.15, 0.95, 0.95]
```

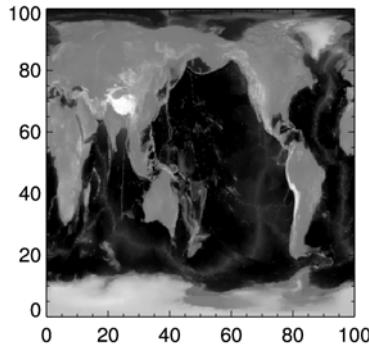


Figure 23: The image can be positioned properly if it is sized with the *TV* command.

```
; Calculate the size and starting locations in pixels.  
xsize = (position[2] - position[0]) * !D.X_VSize  
ysize = (position[3] - position[1]) * !D.Y_VSize  
xbeg = position[0] * !D.X_VSize  
ybeg = position[1] * !D.Y_VSize  
  
; Size the image differently in PostScript.  
IF (!D.Name EQ 'PS') $  
    THEN TV, image, xbeg, ybeg, XSize=xsize, YSize=ysize $  
    ELSE TV, Congrid(image, xsize, ysize), xbeg, ybeg  
  
; Draw the axes around the image.  
Plot, Findgen(100), /NoData, /NoErase, Position=position  
END
```

And there you have it, the germ of an idea that turned into [cgImage](#) over 15 years ago and later a whole bevy of other *TV* alternative commands (see “Alternative Image Display Commands” on page 241). All designed to solve this one particular PostScript problem, and then all gone on to bigger and better things. It is not too much of a stretch to say that the idea embodied in that one little program eventually turned into this book and the desire to write IDL programs that were device and color model independent.

Displaying Color Images in PostScript

Well, there was one more step to a fully functioning [cgImage](#) command. And that was to figure out how to display a color image in the same way on the display and in a PostScript file.

There has never been any problem displaying a 2D image in a PostScript file. Simply load the color table you want to use, and display the image.

The only thing you have to remember is to always load the color table just before you display the image, so you are sure the image colors are not contaminated with other drawing colors. (If you use the color decomposed model for your drawing colors in PostScript in IDL 7.1 and higher, you don't have to worry about this, of course.)

In the old days, it was possible to display a 24-bit color image with an 8-bit PostScript driver, but you would only be able to create a grayscale PostScript image. I am not sure which version of IDL fixed this in the PostScript driver but I haven't noticed the problem for a long time. If you have an old version of IDL, you may have to do something like this to get a color version of a 24-bit image. Consider the 24-bit rose image in the IDL *examples* directory. We use *Color_Quan* to turn the 24-bit image into an 8-bit image and color vectors, which we load before displaying the image.

```
IDL> rose = cgDemoData(16)
IDL> Help, rose
      ROSE    BYTE = Array[3, 227, 149]
IDL> Set_Plot, 'PS'
IDL> image2d = Color_Quan(rose, 1, r, g, b)
IDL> TVLCT, r, g, b
IDL> TV, image2d
IDL> Device, /Close_File
```

But in most modern versions of the PostScript driver, you can display the 24-bit image directly without having to create a color table for it with *Color_Quan*.

However, you *do* have to be certain when you display a 24-bit image that you have a grayscale color table loaded. The PostScript driver will incorrectly pass true-color values through the local color table if you are using an indexed color model to display the 24-bit image. Let me show you what I mean.

Remember, indexed color is the default color model for the PostScript driver. It is only since IDL 7.1 that you can use the decomposed color model in PostScript, but you have to turn color decomposition on by setting the *Decomposed* keyword on the *Device* command to 1.

So, using the default indexed color mode, here is code that will display the rose image correctly (because a grayscale color table is loaded at the time the image is display) and incorrectly (because another color table is loaded at the time the image is displayed).

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'PS'
```

```

IDL> Device, XSize=5.0, YSize=1.5, Decomposed=0
IDL> LoadCT, 0
IDL> TV, rose, True=1, XSize=2.5, YSize=1.5, /Inches
IDL> LoadCT, 5
IDL> TV, rose, 2.5, 0.0, True=1, XSize=2.5, $
      YSize=1.5, /Inches
IDL> Device, /Close_File
IDL> Set_Plot, thisDevice

```

You see the result in Figure 24.

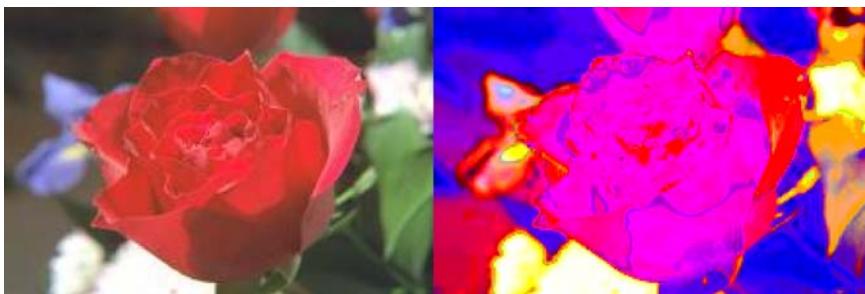


Figure 24: To display a true-color image correctly when the PostScript device is in indexed color mode (the default mode), you must load a grayscale color table before you display the image.

This is another detail that `cgImage` and other alternative `TV` commands handle for you.

Writing Device Independent Programs

The way to achieve efficiency in IDL is to make sure you write IDL programs that work everywhere. Programs that have to be tweaked to work on this or that device eat into your time and are nearly impossible to maintain. And, in particular, your goal should be to write every program so that it works identically on the display and in a PostScript file. This way you can test the program on the display and know you can make a high-quality graphic output file (e.g., PostScript, PNG, JPEG, TIFF, even PDF) any time you need to.

There are, however, several things you need to pay attention to when writing IDL programs to make this happen. And the first, of course, is windows. It may seem obvious, but when you are writing graphics com-

mands into a file there is really no such thing as a window, even for all our talk about the PostScript “window.” The PostScript “window” is a concept more than a physical reality. In any case, the *Window* command, as well as any command pertaining to windows (*WSet*, *WShow*, etc.) are prohibited in the PostScript device.

But we would be hard pressed to create graphics programs if we couldn’t open graphics windows of a particular size and shape when we need to. Does the goal of device independence mean we have to give up creating windows?

No, but it does mean we have to learn to work with windows a little more judiciously, and it means that when we do open a window, or use a window command, we have to “protect” it from the PostScript device.

Protecting Window Commands in PostScript

Protecting a window command simply means we only issue a *Window* command, or commands pertaining to windows, on devices that support graphics windows. So the real question is, how do we know if the current graphics device supports windows?

Fortunately, IDL can tell us. All graphics device information is gathered together for us in the *!D* system variable (“D” for device). Recall that the name of the current graphics device (*!D.Name*) is stored here. The field we are looking for in this system variable structure is the *Flags* field. The field is a bit map, whose bits are described in Table 2.

Bit	Value	Function
0	1	Device has scalable pixel size (e.g., PostScript).
1	2	Device can output text at an arbitrary angle using hardware.
2	4	Device can control line thickness using hardware.
3	8	Device can display images.
4	16	Device supports color.
5	32	Device supports polygon filling with hardware.
6	64	Device hardware characters are monospace.

*Table 2: The *!D.Flags* bit definitions. Devices that support windows have bit 8 set. We check for this bit by comparing *!D.Flags* with the value 256.*

Bit	Value	Function
7	128	Device can read pixels (i.e. supports TVRD).
8	256	Device supports windows.
9	512	Device prints black on white background (e.g., printers).
10	1024	Device has no hardware characters.
11	2048	Device does line-fill style polygon filling in hardware
12	4096	Device allows embedded text formatting commands.
13	8192	Device is a pen plotter.
14	16384	Device can transfer 16-bit pixels.
15	32768	Device supports Kanji characters.
16	65536	Device supports widgets/
17	131072	Device has a Z-graphics buffer.
18	262144	Device supports TrueType fonts.

Table 2: The !D.Flags bit definitions. Devices that support windows have bit 8 set. We check for this bit by comparing !D.Flags with the value 256.

The particular bit we are interested in at the moment is bit 8, represented by the value 256, which can tell us if the current graphics device supports windows.

Those of you who are not computer scientists may not remember what a bit map is. We can use the [Coyote Library](#) command [Binary](#) to see a binary representation of this number, with the highest bits on the left and the lowest bits on the right. Here I am showing the first 24 bits of the number, separated every 8 bits.

```
IDL> Set_Plot, 'WIN' ; Or 'X' if in UNIX.
IDL> Print, !D.Flags
      328124
IDL> Print, Binary(!D.Flags, /Color, /Separate)
      0 0 0 0 0 1 0 1   0 0 0 0 0 0 0 1   1 0 1 1 1 1 0 0
```

We are interested in knowing whether bit 8, which is the first bit on the right, in the middle bit group (we start counting from 0), is set to 1. (Which

it is in this case.) To determine this, we are going to compare the *!D.Flags* number with the number 256, which is written like this in binary format.

```
IDL> Print, Binary(256, /Color, /Separate)
      0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 1   0 0 0 0 0 0 0 0
```

We are going to perform a bit-wise AND operation on these two numbers. The AND operation returns a 1 if both numbers are 1 at that bit position and a zero otherwise. Here is the operation in binary format.

```
IDL> Print, Binary(!D.Flags AND 256, /Color, /Separate)
      0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 1   0 0 0 0 0 0 0 0
```

You see we get a number, which in this case is the number 256, if the bit is set, and would be 0 if the number were not set. Another way to say this is that if we AND the *!D.Flags* number with *any* of the values in Table 2, and the resulting value is *not* zero, then the bit representing that value must be set. In IDL we write this notion like this.

```
IDL> IF ((!D.Flags AND 256) NE 0) $
    THEN Print, 'Bit is set.' $
    ELSE Print, 'Bit is not set.'
      Bit is set.
```

But, what happens in the PostScript device?

```
IDL> Set_Plot, 'PS'
IDL> IF ((!D.Flags AND 256) NE 0) $
    THEN Print, 'Bit is set.' $
    ELSE Print, 'Bit is not set.'
      Bit is not set.
IDL> Print, Binary(!D.Flags, /Color, /Separate)
      0 0 0 0 0 1 0 0   0 0 0 1 0 0 1 0   0 0 1 1 0 1 1 1
```

As you can see, bit 8 is not set in the PostScript device, because the PostScript device does not support windows. We use this knowledge to “protect” the *Window* command from the PostScript device. In other words, we might write code like this to set up a “window” for graphics output on the display, in a PostScript file, and in the Z-graphics buffer.

```
xsize = 640
ysize = 512
IF ((!D.Flags AND 256) NE 0) THEN BEGIN
    Window, XSize=xsize, YSize=ysize, /Free
ENDIF ELSE BEGIN
    CASE !D.Name OF
        'PS': BEGIN
            aspect = Float(ysize) / xsize
            Device, _Extra=PSWindow(AspectRatio=aspect)
        END
```

```
'Z': Device, Set_Resolution=[xsize, ysize]
ENDCASE
ENDELSE
```

This, as it happens, is *exactly* what [cgDisplay](#) does. So, rather than type those dozen lines in every IDL program we write, we can simply substitute a single line that accomplishes the same thing.

```
IDL> cgDisplay, 640, 512
```

This allows us to create a “window” with the same aspect ratio in every graphics device. We write one graphics program, and it works everywhere in the same way.

Occasionally, however, we still have to protect a particular command, such as this [WSet](#) command.

```
IDL> IF ((!D.Flags AND 256) NE 0) THEN WSet, windowIndex
```

Work in Decomposed Color Space

Another secret for writing device-independent programs is to work whenever possible in decomposed color space. We have 24-bit color monitors for a reason. If you are lucky enough to be running a version of IDL that has 24-bit color support for PostScript (IDL 7.1 and higher) take advantage of it and use it.

I use the routines [GetDecomposedState](#) and [SetDecomposedState](#) to get and set the color decomposition state in programs. All of the IDL version and machine dependencies are encapsulated in these programs. They allow me to work in a color decomposition state whenever possible. This avoids all kinds of complications that occur when you work in indexed color mode. (The main problem is contamination of the color tables used to display images.) Try to avoid working in indexed color mode at all costs. The graphics display routines in the [Coyote Library](#) are all written this way.

I generally write code like this.

```
SetDecomposedState, 1, CurrentState=currentState
.... Graphics commands here.
SetDecomposedState, currentState
```

If the graphics commands I am using *must* be issued in indexed color mode, then I know I am going to have to contaminate the current color table. If so, then I generally save the current color table vectors before I contaminate them, then restore them when I am finished with the color table.

```

TVLCT, r, g, b, /Get
SetDecomposedState, 0, CurrentState=currentState
.... Load graphics colors here.
.... Graphics commands in indexed color here.
SetDecomposedState, currentState
TVLCT, r, g, b ; Restore the input color vectors.

```

Note: If you are writing programs that are meant to work in the Z-graphics buffer, too, note that you cannot restore the color table vectors when you are finished drawing in indexed color. If you do, the colors will be incorrect when you take the “snapshot” of the Z-graphics buffer. This is another reason to use SetDecomposedState, since it will set the Z-graphics buffer up to work in a 24-bit color space, too.

Position Graphics in Normalized or Data Coordinates

Another key to writing device-independent programs is to always position text and graphics using normalized or data coordinates. Do not use device coordinates, and especially do not use device coordinates in the PostScript device.

The reason is simply this. There are approximately 35 times as many pixels per centimeter in the PostScript device as there are pixels per centimeter on your display monitor. Text or graphics positioned in device coordinates in the display window will wind up somewhere else entirely when positioned with the same coordinates in PostScript.

```

IDL> Set_Plot, 'WIN' ; Or 'X' if in UNIX.
IDL> Print, !D.X_PX_CM, !D.Y_PX_CM
      28.3186      28.3657
IDL> Set_Plot, 'PS'
IDL> Print, !D.X_PX_CM, !D.Y_PX_CM
      1000.00      1000.00

```

And, because fonts are generally different on your display and in the PostScript file, it is best, if you are positioning text, to center the text about a point, if at all possible. In practice, this means setting the *Alignment* keyword to *XYOutS*, or *cgText* to 0.5.

```

IDL> XYOutS, 0.5, 0.5, Alignment=0.5, /Normal, $
      'Position text in NORMAL or DATA coordinates'

```

Printing PostScript Files

This seems to me to be self-evident (although I am always surprised at the e-mails I receive on this subject), but a PostScript file can only be printed

on a printer that is set up to interpret the PostScript language. That is to say, on a PostScript printer. And if you hope to see your PostScript output in color, it must be printed on a color PostScript printer.

(It is possible, however, to run the PostScript file through some kind of converter that will prepare the file to be printed on non-PostScript printers. I'll say more about this in just a moment.)

Printing from UNIX Computers

Many people, however, use IDL on a UNIX network, and the network printer is some kind of a PostScript printer. It is usually a color PostScript printer these days. Most of these people can use the normal UNIX *lpr* command to send a PostScript file to the printer queue. Often, this is spawned from the IDL command line. And even more often, the *Spawn* command is shortened to the dollar sign character, which is the short-hand way of spawning a UNIX command from the IDL command line on a UNIX machine.

```
IDL> $ lpr 'myplot.ps'
```

The only trick here is to be *sure* you have closed the PostScript file before you try to print it. If you don't, the file will be processed by the printer, but nothing will ever come out of it, because the file will lack the PostScript *showpage* command that closing the file inserts into the file.

Printing from Windows Computers

It is a little more difficult to print PostScript files on Windows machines, especially from within IDL. Depending on your operating system, you might have to install some type of third-party print spooler. I have always used the small freeware program named *PrintFile* on my Windows machines.

PrintFile (<http://www.lerup.com/printfile>) is written by Peter Lerup, the author of the well-known *ps2eps* program for Macintosh computers. This program works well with both stand-alone and networked PostScript printers. One of its best features is that it allows you to print encapsulated PostScript files without first embedding the file in another PostScript document.

On a Windows 7 operating system, you can just right click on a file and choose *Print* from the pull-down menu to send the file to the default printer.

Printing from Macintosh Computers

Macintosh users can print PostScript files from the *Preview* program.

Converting PostScript for Non-PostScript Printers

If you don't have a PostScript printer, then you will have to install a PostScript conversion program to convert the PostScript into something that can be printed on your printer. UNIX users often install the free GhostScript program (<http://pages.cs.wisc.edu/~ghost/>) that can do the conversion for you. I have used GSView (<http://pages.cs.wisc.edu/~ghost/>), which is the Windows version of GhostScript, on my Windows computers for years and have always been happy with it.

These programs can also convert PostScript files to JPEG, TIFF, PNG, and PDF files, among other formats, before sending the resulting file to a non-PostScript printer.

Macintosh users find the *Preview* application useful for viewing and converting PostScript files. It automatically converts a PostScript file to PDF format for viewing and/or printing.

Viewing PostScript Files

Both GhostView (<http://pages.cs.wisc.edu/~ghost/>) and GSView (<http://pages.cs.wisc.edu/~ghost/>) are PostScript previewers and allow you to open and view your PostScript files, as well as print them. You can even add preview images to your PostScript files with these extremely useful applications. You can install GhostView on a Macintosh computer, but the Macintosh has its own applications for viewing PostScript files. Adobe Illustrator is another popular cross-platform choice for viewing (and editing!) PostScript files.

Be careful, however, when viewing PostScript files in previewers. A great many of the “problems” associated with PostScript output are not problems at all, but are artifacts introduced by the previewer. One common “problem” is to see white stripes that appear almost like scratches in your PostScript preview, as shown in Figure 25.

In fact, these stripes do not exist in the PostScript file. Rather, they are an artifact of the previewer anti-aliasing the file input. If you turn anti-aliasing off in the previewer, these stripes will disappear.

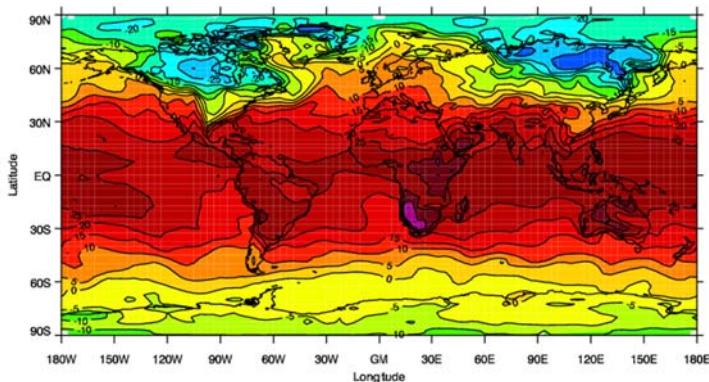


Figure 25: White stripes can be introduced into your PostScript file. These are artifacts from the PostScript previewer anti-aliasing the input. They do not exist in the PostScript file itself.

In GSView, for example, on Windows computers, you change the anti-aliasing property by selecting the *Media -> Display Settings* menu. Find the *Text Alpha* setting, and change the value from 4 bits (use anti-aliasing) to 1 bit (turn anti-aliasing off). In Adobe Illustrator, you select the *Edit -> Preferences -> General -> Anti-Aliased Artwork* menu item.

If you are using ImageMagick to convert PostScript to other formats, use the *+antialias* switch on the *convert* command. If you are using the Macintosh Preview application, choose the *Preview -> Disable Smooth Text and Graphics* button. By turning off anti-aliasing, you will see the output as it actually exists in the file. See Figure 26 for an example of how the file will look with anti-aliasing turned off.

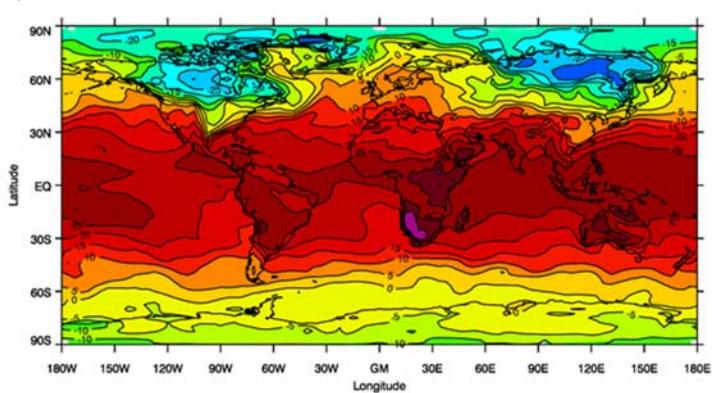


Figure 26: The PostScript output will appear correct in the pre-viewer if anti-aliasing is turned off.

Chapter 10



The Z-Graphics Buffer

Purpose of the Z-Graphics Buffer

One of the problems with traditional graphics commands in IDL is that they are not truly three-dimensional. They represent three-dimensional space by projecting onto a flat 2D surface in what is sometimes called a “two and a half D” perspective. You can see this particularly well with the *Surface* command. Surfaces rendered with the *Surface* command cannot be rotated around all three orthogonal space directions simultaneously. (There is no *AY* rotation keyword, for example, to rotate about the Y axis.)

The absence of true three-dimensionality in the traditional graphics commands was, in fact, what motivated IDL developers to create the object graphics system in IDL. The object graphics system *is* a true 3D graphics system. To create a true 3D graphics system, you need some way to specify “depth” in a 3D coordinate system. In a 3D system with depth, you have a notion that some objects are behind other objects, and that the objects in front might well obstruct your view of objects behind. (This is how we visually understand the meaning of the word “depth.”)

Visual clues about depth can be put into effect in such a system. We can, for example, implement a vanishing view perspective where objects of the same size will appear smaller as they recede in the distance. Or, lines creating a object (a box, for example) might diminish in color or intensity as the lines fade into the distance. Or, in the simplest way to represent depth, objects that are in front of other objects simply prevent us from viewing the objects behind. We call this method of representing depth the method of hidden line or hidden surface removal. We can even introduce the

notion of an object being “transparent,” so that we can see through a transparent object to an object behind.

In computer graphics, depth is normally represented in a Z-graphics buffer. (The Z is meant to represent the third dimension.) The object graphics system in IDL *is* a Z-graphics buffer, which is one of the reasons rendering graphics there is slower than in the traditional, 2D raster graphics system. But, the disadvantages of a Z-graphics buffer are often outweighed by the advantages for visualizing certain kinds of scientific data (e.g., volumetric data, typically).

For example, you could construct a cut-away visualization of the Earth’s core, or you could construct a 3D representation of a beating heart or molecule, or you could construct a terrain model that the user could “fly-through” with airplane-like controls.

What I am about to describe to you is a 3D system in which these kinds of visualizations can be created. But, it is not a 3D system that approaches the sophistication or power of the object graphics 3D system. It is a relatively simple system that allows us to work in a three-dimensional space, with depth information, using traditional graphics commands. Namely, it is the Z-graphics device.

I should caution you that the Z-graphics device (also called the Z-graphics buffer) is not used as much as it once was. If you have a current version of IDL, and you have a 3D computer visualization in mind, you are often better served thinking about an object graphics solution first, which can do full justice to three-dimensional data. But, you will run into IDL code that uses the Z-graphics device, and the Z-buffer is still used for a variety of IDL programming tasks. It is worth understanding exactly how it works.

Configuring the Z-Graphics Device

The Z-graphics device (also called the Z-graphics buffer) is a graphics output device like the PostScript device. You make it the current graphics device with the *Set_Plot* command and control its properties with the *Device* command and various keywords that are appropriate to it. You can think of it as a graphics window, if you like, with special properties. If you want to create a graphics window in the Z-graphics device that was 500x400 pixels in size, you can type commands like this.

```
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'Z'  
IDL> Device, Set_Resolution=[500,400]  
IDL> Set_Plot, thisDevice
```

Like the PostScript device, the Z-graphics device settings are “sticky,” in that once they are set they are set for the rest of that IDL session or until they are changed again by the user.

I like to think of the Z-graphics device as a three-dimensional box in which 2D and 3D objects can be deposited without regard to their “solidity” (see the illustration in Figure 1). The box has the ability to keep track of the “depth” of an object in a 16-bit depth buffer. This depth buffer is also referred to as the “Z-graphics buffer.” One side of the box is a projection plane. Think of rays of light going through each pixel in the projection plane and eventually encountering a solid object in the box. The pixel value that the light ray encounters on its way through the box is the value that is “projected” back onto the projection plane. Objects that are “behind” other objects will not be encountered or projected back onto the projection plane.

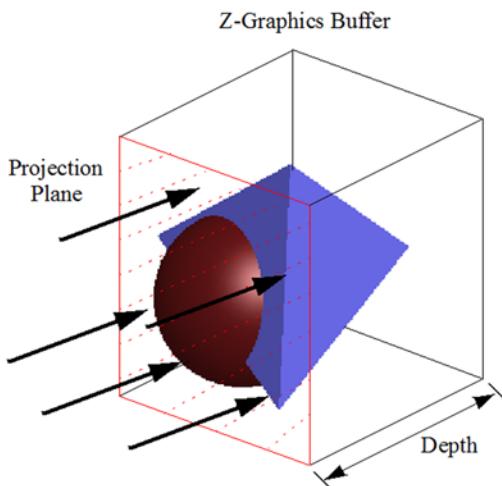


Figure 1: The Z-graphics device can be thought of as a 3D box that keeps track of depth information. Light rays strike the box contents and the pixel values the rays encounter are projected back to the projection plane. A snapshot or pixel dump of the projection plane produces an array of pixel values representing the 3D scene in the box.

The idea is that once you load the “solid” objects in the 3D box, you can take a “snapshot” or pixel dump of the projection plane to create an image of the contents of the box. In this way, the Z-graphics buffer is used to

implement hidden surface and line removal automatically. The snapshot of the projection plane is accomplished with the *TVRD* command in IDL, or sometimes with easier to use alternative commands, such as [cgSnapshot](#) from the [Coyote Library](#).

The Z-graphics device is implemented in software, which makes it an attractive alternative for creating graphics windows when it is inconvenient or impossible to open graphics windows on the machine running IDL (e.g., in *cron* jobs). The default resolution of the Z-graphics device “window” is 640x480 pixels. Note that this is almost certainly different from the default window size on your display. If you are trying to match output from the Z-graphics device to your normal graphics window display size (as is commonly done), you will have to set the Z-graphics window size to the correct size with the *Set_Resolution* keyword.

```
IDL> Window, Title='Z-Buffer Example'
IDL> xsize = !D.X_Size
IDL> ysize = !D.Y_Size
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'Z'
IDL> Device, Set_Resolution=[xsize,ysize]
```

Note that you can obtain current information about the Z-graphics device by making it the current graphics device and using the *Help* keyword to the *Device* command.

```
IDL> Help, /Device
Available Graphics Devices: CGM HP METAFILE NULL PCL
PRINTER PS WIN Z
Current graphics device: Z
Resolution: 640 x 512
Buffers allocated: (8-bit) Pixels and (16-bit) Z
(983040 total bytes).Z Buffering: On.
Graphics function: 3 (Copy), Write Mask: 255 (0xff).
Current TrueType Font: <default>
```

Note, in particular, the help line that starts with “Buffers allocated.” There are two completely different “buffers” in the Z-graphics device and it is critical that you understand the difference and how to use the information in the buffers.

The Depth Buffer Versus the Frame Buffer

The Z-buffer is the depth buffer. It is a 16-bit buffer, which means depth can be specified to a resolution of one part in 2^{16} or 65536. Most of the time we use the information in the depth buffer indirectly by working with

the projection plane, but it is possible to obtain the depth information directly should you want to use it. You see from the on-line help that this depth buffer is turned on (“`Z Buffering: On.`”). However, you can turn the depth buffer off and use the Z-graphics device as a normal IDL graphics window by setting the `Z_Buffer` keyword to zero. This is especially helpful if you are running IDL in an environment (e.g., a cron job) in which a display device isn’t available.

```
IDL> Device, Z_Buffer=0
```

To illustrate, let’s turn the Z-buffer back on and let’s “load” the Z-graphics device with colored surfaces we will call *peak* and *saddle*.

```
IDL> Device, Z_Buffer=1
IDL> peak = Shift(Dist(20,16), 10, 8)
IDL> peak = Exp( -(peak / 5.0)^2)
IDL> saddle = Shift(peak,6,0) + Shift(peak,-6, 0) / 2
```

We are going to display the peak with a red color table and the saddle with a blue color table. We will allocate 100 blue colors to *peak* and 100 red colors to *saddle*.

```
IDL> LoadCT, 1, NColors=100, Bottom=1
IDL> LoadCT, 3, NColors=100, Bottom=101
IDL> black = cgColor('black', 253)
IDL> white = cgColor('white', 254)
```

Finally, we load the surfaces into the Z-buffer.

```
IDL> Set_Shading, Values=[1,100]
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2], Color=black, $
      Background=white, Charsize=2.0
IDL> Set_Shading, Values=[101,200]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase, $
      Color=black, Charsize=2.0
```

To view these two surfaces as an image, we have to read what I have called the projection plane, but which is more accurately called the *frame buffer*. The frame buffer has traditionally been an 8-bit buffer, but starting in IDL 6.4, it has also been possible make the frame buffer a 24-bit buffer. In essence, this is the same as the difference between an 8-bit graphics window and a 24-bit graphics window.

Reading the Frame Buffer

The standard way to take a “snapshot” of a graphics window is to use the `TVRD` command in IDL. This command requires you to know the depth of the graphics window you are copying so you can set the `True` keyword to

the proper value. (The [Coyote Library](#) command `cgSnapshot`, which is often used as a *TVRD* replacement command can determine the setting of the *True* keyword automatically, so the *True* keyword does not need to be set with this command.)

By default, the Z-graphics device is configured with an 8-bit frame buffer. That is the meaning of “(8-bit) Pixels” in the “Buffers allocated” line of the *Help* command above.

To read this 8-bit buffer and display the image correctly in a graphics window, we must capture not only the 8-bit buffer, but the current color table vectors, because an 8-bit image is an image that must use the indexed color model to be displayed properly.

To capture the image from the frame buffer and display it, we type commands like this.

```
IDL> TVLCT, r, g, b, /Get
IDL> snapshot = TVRD()
IDL> Help, snapshot
      SNAPSHOT   BYTE = Array[640, 512]
IDL> Set_Plot, thisDevice
IDL> cgImage, snapshot, /NoInterp
```

You see the result in Figure 2.

Note that I didn’t load the color table vectors before I displayed the image. There is no need for that here, since the color table vectors are copied from the Z-graphics device into the current graphics device when I issue the *Set_Plot* command. But if I don’t *immediately* display the image, I will need to load the color table vectors before I do so. And, I will need the color table vectors if I want to make a colored image file for output. For example, to make a color PNG file of the output to send to a colleague, I would create the file like this.

```
IDL> Write_PNG, 'example.png', snapshot, r, g, b
```

If I want to create a color JPEG file of the output, I will have to use the color table vectors to make a 24-bit image to pass to the *Write_JPEG* command, like this.

```
IDL> image24 = [[[r[snapshot]]], [[g[snapshot]]], $
      [[b[snapshot]]]]
IDL> Write_JPEG, 'example.jpg', image24, True=3
```

In this case, it is easier to read a 24-bit color image directly out of the Z-graphics device by using a 24-bit frame buffer. Recall that to configure the PostScript device to become a 24-bit device, you simply used the *Device*

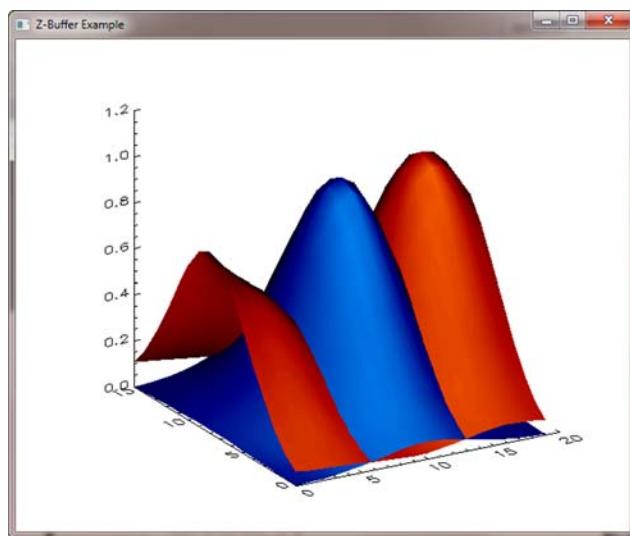


Figure 2: The output image from the Z-graphics device.

command and set the *Decomposed* keyword to 1. You must do the same thing for the Z-graphics device, but this alone is not sufficient. You must also use the *Set_Pixel_Depth* keyword to set the frame buffer depth to 24.

But, even so, things are a *little* more complicated than they seem!

Suppose, for example, that we try to produce our image with code like this, in which we set the *Decomposed* keyword to 1 and the *Set_Pixel_Depth* keyword to 24 before we load the images into the Z-buffer.

```
IDL> Set_Plot, 'Z'  
IDL> Device, Decomposed=1, Set_Pixel_Depth=24  
IDL> Help, /Device  
Available Graphics Devices: CGM HP METAFILE NULL PCL  
PRINTER PS WIN Z  
Current graphics device: Z  
Resolution: 640 x 512  
Buffers allocated: (24-bit) Pixels and (16-bit) Z  
(983040 total bytes).Z Buffering: On.  
Graphics function: 3 (Copy), Write Mask: 255 (0xff).  
Current TrueType Font: <default>  
IDL> Set_Shading, Values=[1,100]  
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2], Color=black, $  
Background=white, Charsize=2.0
```

```
IDL> Set_Shading, Values=[101,200]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase, $
      Color=black, CharSize=2.0
IDL> image24 = TVRD(True=3)
IDL> Help, image24
      IMAGE24   BYTE = Array[640, 512, 3]
IDL> Set_Plot, thisDevice
IDL> cgImage, image24
```

You see the result in Figure 3. What happened to the color!?

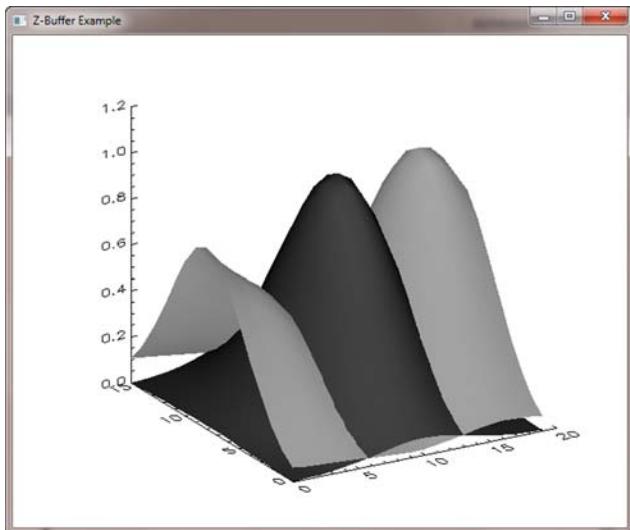


Figure 3: Setting up the Z-graphics device as a 24-bit device caused color to disappear. What happened!?

The problem here is the same problem we have with 24-bit graphics windows: they are sensitive to the color model being used to load the graphics into the window. In this case, the *Shade_Surf* command is an IDL traditional command that can only be used with the indexed color model. The proper way to create a color image in this case, is to use a 24-bit frame buffer, but to use the indexed color model by setting the *Decomposed* keyword to 0. These are the commands we should use.

```
IDL> Set_Plot, 'Z'
IDL> Device, Decomposed=0, Set_Pixel_Depth=24
IDL> Set_Shading, Values=[1,100]
```

```

IDL> Shade_Surf, peak, ZRange=[0.0, 1.2], Color=black, $
      Background=white, Charsize=2.0
IDL> Set_Shading, Values=[101,200]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase, $
      Color=black, Charsize=2.0
IDL> image24 = TVRD(True=3)
IDL> Help, image24
      IMAGE24  BYTE = Array[640, 512, 3]
IDL> Set_Plot, thisDevice
IDL> cgImage, image24

```

Now the 24-bit image displays in color and the output looks identical to the output in Figure 2. This image can be made into a JPEG file directly.

```
IDL> Write_JPEG, 'example_direct.jpg', image24, True=3
```

Reading the Depth Buffer

Reading the Z-buffer directly is done differently, depending upon the depth of the frame buffer. In general, reading the Z-buffer (also called the *depth buffer*) is done with *TVRD* by setting the *Channel* keyword to the proper channel number and setting the *Words* keyword.

For example, to read the Z-buffer when the Z-graphics device is set up with an 8-bit frame buffer, you would set the *Channel* keyword to 1, like this.

```

IDL> Set_Plot, 'Z'
IDL> Device, Decomposed=0, Set_Pixel_Depth=8
IDL> Set_Shading, Values=[1,100]
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2], Color=black, $
      Background=white, Charsize=2.0
IDL> Set_Shading, Values=[101,200]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase, $
      Color=black, Charsize=2.0
IDL> depth = TVRD(Channel=1, /Words)
IDL> Set_Plot, thisDevice
IDL> Help, depth
      DEPTH  INT = Array[640, 512]

```

To do the same thing when the Z-graphics buffer is set up with a 24-bit frame buffer, you would set the *Channel* keyword to 4, like this.

```

IDL> Set_Plot, 'Z'
IDL> Device, Decomposed=0, Set_Pixel_Depth=24
IDL> Set_Shading, Values=[1,100]
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2], Color=black, $
      Background=white, Charsize=2.0
IDL> Set_Shading, Values=[101,200]

```

```
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase, $  
      Color=black, Charsize=2.0  
IDL> depth = TVRD(Channel=4, /Words)  
IDL> Set_Plot, thisDevice  
IDL> Help, depth  
DEPTH   INT = Array[640, 512]
```

Character Size in the Z-Graphics Device

The Z-graphics device uses a slightly different character size than the character size for an IDL graphics window. Normally, this makes almost no difference, but occasionally you are trying to match up a graphic you created in the Z-buffer with something you are doing in a graphics window, and the matching just doesn't *quite* work.

This mismatch of character sizes causes slight differences in the locations of axes, since plot margins (the regions outside of the axes) are calculated in units of character size. Suppose, for example, that you want to display the *peak-saddle* output in a PostScript file. It is likely that you would *not* want to simply pass the snapshot of the Z-buffer to the PostScript file, both because the axes are rendered using Hershey fonts and because the axes will be rendered in PostScript in screen resolution (96 pixels per inch) rather than PostScript resolution (1000 pixels per inch).

You would probably want to add the axis to the image that is created in the Z-graphics buffer. The code you write might look like this.

```
IDL> Set_Plot, 'Z'  
IDL> Device, Decomposed=0, Set_Pixel_Depth=24  
IDL> Set_Shading, Values=[1,100]  
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2], Color=black, $  
      Background=white, Charsize=2.0  
IDL> Set_Shading, Values=[101,200]  
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase, $  
      Color=black, Charsize=2.0  
IDL> image24 = TVRD(True=3)  
IDL> PS_Start, Filename='peak_saddle_z.ps', Font=1  
IDL> cgImage, image24  
IDL> Surface, peak, ZRange=[0.0, 1.2], /NoData, $  
      Charsize=2.0  
IDL> PS_End  
IDL> Set_Plot, thisDevice
```

You see the result in Figure 4. Compare this figure to the output in Figure 2. If you look closely, you will notice that the axes are not positioned correctly in Figure 4. They are too far away from the surface. This is

because the *Surface* command above, which was used to draw the axes, used a different default character size, which changed the plot margins, which, in turn, moved the axes. (You will learn about the *PS_Start* and *PS_End* commands, which here take care of the details of configuring the PostScript device, in the next chapter.)

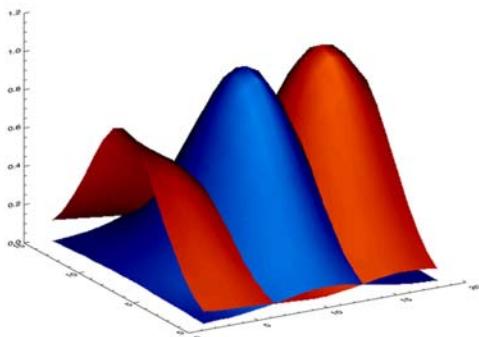


Figure 4: The axes were added in the PostScript device, which uses a different character size than the Z-graphics device, which moves the axes locations so that they are not exactly in the right place. Notice how far the surfaces are from the X and Y axes. Compare this figure to Figure 2.

The way to “fix” this problem is to be sure to locate the axes in both devices in normalized window coordinates by using the *Position* keyword to position the axes in the “window.” In other words, don’t allow the position of the axes to be located in character units. The code you should write is this.

```
IDL> Set_Plot, 'Z'
IDL> Device, Decomposed=0, Set_Pixel_Depth=24
IDL> Set_Shading, Values=[1,100]
IDL> pos = [0.1, 0.1, 0.9, 0.9, 0.1, 0.9]
IDL> Shade_Surf, peak, ZRange=[0.0, 1.2], Color=black, $
      Background=white, CharSize=2.0, Position=pos
IDL> Set_Shading, Values=[101,200]
IDL> Shade_Surf, saddle, ZRange=[0.0, 1.2], /NoErase, $
      Color=black, CharSize=2.0, Position=pos
IDL> image24 = TVRD(True=3)
IDL> PS_Start, Filename='peak_saddle_z.ps', Font=1
IDL> cgImage, image24
```

```
IDL> Surface, peak, ZRange=[0.0, 1.2], /NoData, $
      CharSize=2.0, Position=pos
IDL> PS_End
IDL> Set_Plot, thisDevice
```

You see the result in Figure 5. Compare this result to the results in Figure 2 and Figure 4. This figure is virtually identical to Figure 2.

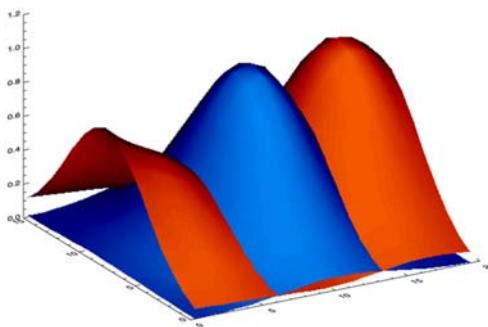


Figure 5: The axes locations are specified in normalized units instead of character units, which permits the two graphic inputs to align properly. This figure is virtually identical to Figure 2.

Warping Images in the Z-Buffer

Another useful property of the Z-graphics device is its ability to warp images into a 3D space. Suppose, for example, you have a cube of data and you want to view slices of that cube in relation to one another. The Z-graphics buffer would be a good place to start with this kind of visualization.

As an example, let's use the MRI data in the IDL *examples* directory. This is MRI data of a head. This cube of data is 80 by 100 by 57. Suppose we want to visualize the sagittal (x), coronal (y) and transverse (z) planes at the location (60,64,20) in this cube of data. Here is how we can proceed.

First, read the data cube and pull the proper image slices out of the data cube.

```
IDL> head = cgDemoData(8)
IDL> xpt = 60 & ypt = 64 & zpt = 20
```

```
IDL> ximage = Reform(head[xpt, *, *])
IDL> yimage = Reform(head[*, ypt, *])
IDL> zimage = Reform(head[*, *, zpt])
```

Next, we want to create polygons that describe the three planes of data we are interested in visualizing in terms of the image corner locations in a 3D data space equal to the size of the image cube. (We could use any arbitrary data coordinate space, but this one is convenient.) Notice that the polygons are closing on themselves (the last vertex must be the same as the first). This is not a requirement for the *PolyFill* command we will use shortly, but it is a requirement for the *PlotS* command, which we will use to draw a boundary around the polygon after we fill it.

```
IDL> s = Size(head, /Dimensions)
IDL> xs = s[0]-1 & ys = s[1]-1 & zs = s[2]-1
IDL> xplane = [ [xpt, 0, 0], [xpt, 0, zs], $ 
    [xpt, ys, zs], [xpt, ys, 0], [xpt, 0, 0] ]
IDL> yplane = [ [0, ypt, 0], [0, ypt, zs], $ 
    [xs, ypt, zs], [xs, ypt, 0], [0, ypt, 0] ]
IDL> zplane = [ [0, 0, zpt], [xs, 0, zpt], $ 
    [xs, ys, zpt], [0, ys, zpt], [0, 0, zpt] ]
```

This particular data has a maximum value of 232.

```
IDL> Print, Max(head)
232
```

This means we can use color indices 0 to 232 to display this data, so we will limit the number of colors to 233. We set up the Z-graphics device and create a 3D data space by typing these commands.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'Z'
IDL> Device, Decomposed=0, Set_Pixel_Depth=24
IDL> cgDisplay, 500, 500
IDL> Erase, cgColor('white')
IDL> Scale3, XRange=[0,xs], YRange=[0,ys], ZRange=[0,zs]
```

Next, we warp the three images into the three polygons we have created by using the *PolyFill* command. The key to warping images in the Z-graphics device are the two *PolyFill* keywords *Image_Coord* and *Pattern*. These keywords have special meaning in the Z-graphics device. (The *Image_Coord* keyword can only be used in the Z-graphics device.) *Image_Coord* is used to identify the corners of the image in 3D space. *Pattern* is used to pass the image to the *PolyFill* command.

We are going to display the three image planes with three different color tables, so we can see the planes clearly in the final result. This can only be done this way, using a 24-bit frame buffer and the indexed color model, which is why we configured the Z-graphics device the way we did above.

```
IDL> ncolors = 233
IDL> cgLoadCT, 3, /BREWER, NColors=ncolors
IDL> PolyFill, xplane[*,:3], /T3D, Pattern=ximage, $
      Image_Coord=[[0,0], [0,zs], [ys,zs], [ys,0]], $
      /Image_Interp
IDL> cgLoadCT, 6, /BREWER, NColors=ncolors
IDL> PolyFill, yplane[*,:3], /T3D, Pattern=yimage, $
      Image_Coord=[[0,0], [0,zs], [xs,zs], [xs,0]], $
      /Image_Interp
IDL> cgLoadCT, 9, /BREWER, NColors=ncolors
IDL> PolyFill, zplane[*,:3], /T3D, Pattern=zimage, $
      Image_Coord=[[0,0], [xs,0], [xs,ys], [0,ys]], $
      /Image_Interp
```

Next, we draw lines around the polygons. If these lines are going to show up later, we have to make them fairly thick now.

```
IDL> thick=4
IDL> PlotS, xplane, /T3D, Thick=thick,
      COLOR=cgColor('black')
IDL> PlotS, yplane, /T3D, Thick=thick,
      COLOR=cgColor('black')
IDL> PlotS, zplane, /T3D, Thick=thick,
      COLOR=cgColor('black')
```

Finally, we can take a snapshot of the Z-buffer and display the resulting image on the display.

```
IDL> snapshot = cgSnapshot()
IDL> Set_Plot, thisDevice
IDL> cgDisplay, 500, 500, Title='Image Warping'
IDL> cgImage, snapshot
```

You see the result in Figure 6.

Here is code that allows you to see all three images individually, as well as in relation to one another.

```
IDL> cgDisplay, 1000, 250, Title='Image Warping'
IDL> !P.Multi=[0,4,1]
IDL> cgLoadCT, 3, /Brewer
IDL> cgImage, Congrid(ximage, 100, 100), /White, $
      MultiMargin=[5, 2, 2, 2]
```

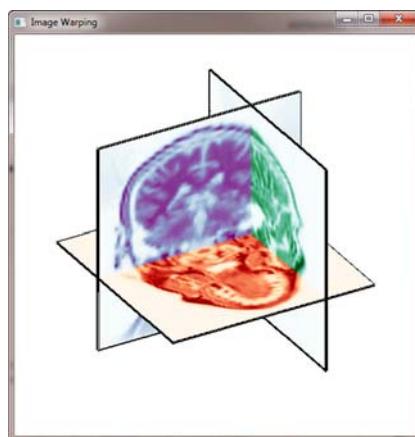


Figure 6: Image warping can be done in the Z-graphics device.

```
IDL> cgLoadCT, 6, /Brewer
IDL> cgImage, Congrid(yimage, 100, 100), $
      MultiMargin=[5, 2, 2, 2]
IDL> cgLoadCT, 9, /Brewer
IDL> cgImage, Congrid(zimage, 100, 100), $
      MultiMargin=[5, 2, 2, 2]
IDL> cgImage, snapshot, /Keep_Aspect
IDL> !P.Multi=0
IDL> cgText, 0.125, 0.05, 'Sagittal', /Normal, $
      Alignment=0.5, Font=0
IDL> cgText, 0.375, 0.05, 'Coronal', /Normal, $
      Alignment=0.5, Font=0
IDL> cgText, 0.625, 0.05, 'Transverse', /Normal, $
      Alignment=0.5, Font=0
```

You see the result in Figure 7.

Transparency Effects in the Z-Buffer

The *PolyFill* command can also be used to add transparency effects in the Z-graphics device. We can continue to use the data from the previous section as an example. This time, let's create an isosurface from the head data and combine that with a transparent polygon. An isosurface is a surface that has the same value everywhere. It is like a three-dimensional contour plot of the data, where the data is contoured at a single level.

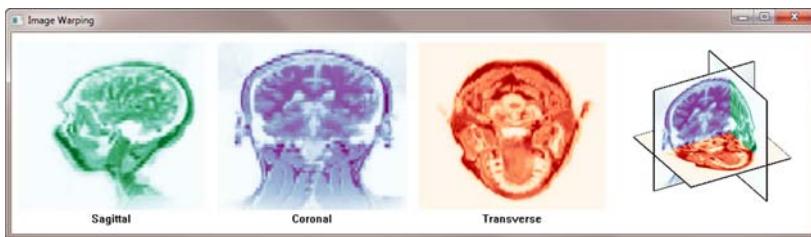


Figure 7: An image strip showing the three image planes and the composite image with the individual images in their proper relationship to one another.

We create an isosurface with the *Shade_Volume* command. This command constructs a list of vertices and polygons that describes a surface at a particular value or level. In this example, we set the isosurface value to 40. We also set the *Low* keyword. By setting this keyword, the low side of the isosurface is shown and the isosurface encloses high values. If the *Low* keyword is not used, the high side of the isosurface is shown and the isosurface encloses low values.

The *vertices* and *polygons* parameters are output variables that contain the vertex and polygon list, respectively. (Note that there are no syntax differences between input and output variables in IDL. You simply have to read the command documentation to know which they are.)

```
IDL> Shade_Volume, head, 40, vertices, polygons, /Low
```

The isosurface is rendered with the *PolyShade* command. We can see the isosurface alone, before we render it in the Z-graphics buffer, with these commands.

```
IDL> cgDisplay, 500, 500, Title='Isosurface Display'  
IDL> Scale3, XRange=[0,xs], YRange=[0,ys], ZRange=[0,zs]  
IDL> isosurface = PolyShade(vertices, polygons, /T3D)  
IDL> cgLoadCT, 5, /Brewer  
IDL> cgImage, isosurface
```

You see the result in Figure 8.

The next step is to use only a portion of the head data (up to the Z plane we identified earlier) for the isosurface, and to lay the Z-plane of data (*zimage*) over the top of the isosurface. We set the *Transparency* keyword of the *PolyFill* command equal to 20, so that any image pixel with a value of 20 or less is transparent. (The image is passed to *PolyFill* with the *Pattern* keyword.) Note that the *Transparency* keyword *only* works with 8-bit



Figure 8: An isosurface rendered with *Shade_Volume* and *PolyShade*.

images, which means you *must* be using the indexed color model when the *PolyFill* command is issued. The commands look like this.

```
IDL> Shade_Volume, head[*,*,0:zpt], 40, vertices, $  
      polygons, /Low  
IDL> isosurface = PolyShade(vertices, polygons, /T3D)  
IDL> thisDevice = !D.Name  
IDL> Set_Plot, 'Z'  
IDL> Device, Decomposed=1, Set_Pixel_Depth=24  
IDL> cgDisplay, 500, 500  
IDL> image24 = [[[r[isosurface]]], [[g[isosurface]]], $  
      [[b[isosurface]]]]  
IDL> cgImage, image24  
IDL> Device, Decomposed=0  
IDL> cgLoadCT, 9, /Brewer  
IDL> PolyFill, zplane[*,0:3], /T3D, Pattern=zimage, $  
      Image_Coord=[[0,0], [xs,0], [xs,ys], [0,ys]], $  
      /Image_Interp, Transparent=20  
IDL> snapshot = cgSnapshot()  
IDL> Set_Plot, thisDevice  
IDL> cgDisplay, 500, 500, WID=1  
IDL> cgImage, snapshot
```

You see the result in Figure 9.

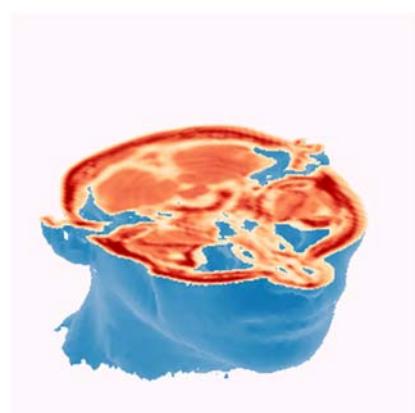


Figure 9: An isosurface combined with an image slice in the Z-graphics buffer. Transparency has been turned on for this image slice.

Chapter 11



Creating Raster Output

Presentation Quality by Leveraging PostScript

In the past, the only option for creating presentation quality graphics output in IDL was to create a PostScript file. And, PostScript output is still the preferred choice for journal articles and many types of reports. But, increasingly, we need the ability to create high quality graphical output for web pages, presentations, and other methods of graphical communication.

This has never been one of IDL's strengths, and especially so for traditional graphics commands. The most glaring weakness in traditional graphics output is the lack of support for presentation quality fonts. But, if you leverage IDL's ability to produce PostScript output with its ability to link to external software programs, the lack of font support becomes almost irrelevant. I routinely produce presentation quality raster graphics output for my web page, for presentations, for sharing with colleagues, and even for the illustrations in this book directly from IDL using nothing but the traditional graphics commands explained in this book.

The secret is to be able to write IDL graphics routines in a device-independent and color-model independent fashion. In other words, to write programs that take the lessons outlined in these pages to heart. It is essential, in my work, to be able to see and test ideas on my computer, before quickly making a PostScript file that I can convert to a high quality PNG, JPEG, or TIFF file. This image conversion is done automatically, from within IDL, by the powerful software program, ImageMagick (<http://www.imagemagick.org/>).

The world is changing rapidly in the presentation graphics realm and, as usual, Apple and its Macintosh computer are out in front of the pack. So,

there are some extremely good, platform specific ways to create presentation quality graphics from PostScript files. The UNIX command *pstoimg* comes immediately to mind. There are even commercial software packages, such as Adobe Distiller, that do the job extremely well across different operating systems.

But I tend to use the open-source, freely available, cross-platform ImageMagick to do the file conversion for me, and that is what I am going to describe here. There are binary and source code versions of ImageMagick available for UNIX, Mac OS X, and Windows computers. It is easy to install, easy to use from within IDL, and there is enough on-line support and documentation for how to use ImageMagick itself (beyond the basic information I am going to provide here) to make it a viable alternative for nearly everyone using IDL.

ImageMagick is described as “a software suite to create, edit, and compose bitmap images.” It can read, convert, and write raster images in over 100 different formats, including PNG, JPEG, TIFF and other common formats. It does not specialize in vector image formats like PostScript, but it can convert vector formats like PostScript to raster image formats nicely.

(It would be a mistake to use ImageMagick to convert a vector image format like PostScript to another vector image format like PDF. There are other freeware and commercial programs that do a better job of this. Adobe Distiller, for example, is a popular commercial alternative for converting vector PostScript output to PDF files. The Preview program on the Macintosh does the conversion automatically, and the UNIX program *epstopdf* is another popular alternative. There are numerous freeware and shareware packages for Windows computers (e.g., CutePDF) that will do this conversion easily, too.)

Although ImageMagick doesn’t specialize in vector formats like PostScript, it does the job well enough I have stopped complaining about how “ugly” IDL traditional fonts are in presentations. PostScript text is automatically anti-aliased during the conversion process to produce raster file output that I think is suitable for any display purpose, including web and Powerpoint presentations, and nearly all the illustrations in this book.

Creating Raster Output in IDL

Before I show you how I automatically produce high-quality raster output with ImageMagick from within IDL, let me show you a couple of ways you might try to obtain similar results from within IDL without using third-party software.

The simplest approach is to simply take a “snapshot” of the graphics window and write the resulting color image to an image file of your choice. To see how this works, let’s create a graphics display in an IDL graphics window. Here we will create a histogram plot of an image.

```
IDL> cgHistoplot, cgDemoData(7), /Fill
```

The standard way of creating a snapshot or screen dump of a graphics window is to use the *TVRD* command in IDL. On a 24-bit display, the command will be used as below. The *True* keyword will indicate the kind of pixel interleaving we will use. In this case, we will create a pixel-interleaved output image by setting the *True* keyword equal to 1.

```
IDL> snapshot = TVRD(True=1)
IDL> Help, snapshot
      SNAPSHOT      BYTE = Array [3, 640, 512]
```

Creating Raster Files

The resulting image can be sent to any appropriate output file type. For example, here is how you could write the snapshot to JPEG, PNG, and TIFF files. (TIFF images almost always need to be reversed in the vertical direction if they are to be displayed with other software properly.)

```
IDL> Write_JPEG, 'test.jpg', snapshot, True=1
IDL> Write_PNG, 'test.png', snapshot
IDL> Write_TIFF, 'test.tif', Reverse(snapshot), 1
```

You can see the PNG output that results from this sequence of commands in Figure 1. Notice in particular that the quality of the text in this figure is significantly reduced from other figures in this book. This is because this is a screen dump of Hershey text characters.

It is even possible to write this snapshot output to a PostScript file, although this is rarely done, since the image will have “screen” resolution, rather than true “PostScript” resolution. It is always better to produce the graphic in a PostScript file rather than producing the graphics in a graphics window and then sending it to a PostScript file using this method.

Color Vectors in Raster Output

Sometimes we need to create a color image of a graphics display on an 8-bit device. Sometimes this is because we are running IDL on an old machine with an 8-bit graphics driver, but more often this is because we have used an 8-bit Z-graphics buffer as the graphics display window.

If this is the case, then we need to capture and store the color table vectors in the image file, if such a thing is supported. For example, we could write

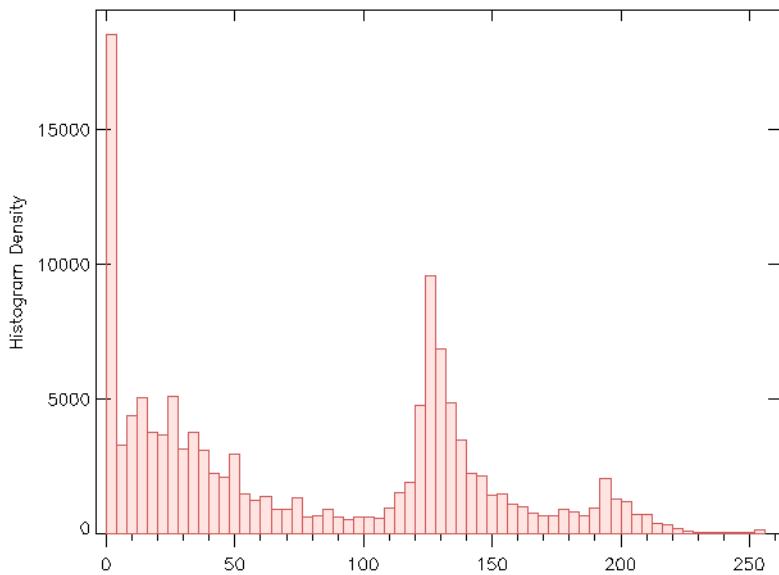


Figure 1: The standard output from taking a screen dump or “snapshot” of a graphics display window with TVRD.

code like this to read an image from the Z-graphics device and make an 8-bit PNG file with color vectors.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'Z'
IDL> cgHistoplot, cgDemoData(7), /Fill
IDL> snapshot = TVRD()
IDL> Help, snapshot
      SNAPSHOT      BYTE = Array[640, 480]
IDL> TVLCT, r, g, b, /Get
IDL> Set_Plot, thisDevice
IDL> Write_PNG, 'test8.png', snapshot, r, g, b
```

Sometimes it is not possible to write the color table vectors into the file. For example, it is not possible to write a JPEG image using color table vectors. In this case, the color table vectors are used to create a true-color image that can be written into the file. Here is how a JPEG file can be created from an 8-bit display.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'Z'
IDL> cgHistoplot, cgDemoData(7), /Fill
```

```
IDL> snapshot = TVRD()
IDL> Help, snapshot
      SNAPSHOT      BYTE = Array[640, 480]
IDL> TVLCT, r, g, b, /Get
IDL> Set_Plot, thisDevice
IDL> image24 = [[[r[snapshot]]], [[g[snapshot]]], $ 
      [[b[snapshot]]]]
IDL> Help, image24
      IMAGE24      BYTE = Array[640, 480, 3]
IDL> Write_JPEG, 'test8.jpg', image24, True=3
```

All-in-One Raster Output Command

Because it is hard to remember what has to be done if you are on a 24-bit versus 8-bit device, and because the syntax of each image output command is always slightly different, I have put all these complications into a single alternative command that I use to create display window “snapshots” which I can write to output image files. I use the [Coyote Library](#) command `cgSnapshot` for this purpose. You don’t have to worry about which `True` keyword to set or whether the graphics window you are reading is on an 8-bit or 24-bit device, or even how to write the output file. This is all computed for you.

Here, for example, is how you would write the JPEG, PNG, and TIFF files at the start of this section with `cgSnapshot`.

```
IDL> void = cgSnapshot(Filename='test', /JPEG)
IDL> void = cgSnapshot(Filename='test', /PNG)
IDL> void = cgSnapshot(Filename='test', /TIFF)
```

Called in this manner, a dialog will appear asking if the user wants to change the name of the file. Note that the file extension of the appropriate file type is appended automatically. If you want to write the file without the dialog appearing, simply set the `NoDialog` keyword.

```
IDL> void = cgSnapshot(Filename='test', /JPEG, $
      /NoDialog)
```

The `cgSnapshot` command can write BMP, GIF, JPEG, PICT, PNG, and TIFF image output. Just set the appropriate keyword. The image is collected from the current graphics window unless another window identifier is given with the `WID` keyword.

Using the Z-Buffer for Raster Output

Another tick that is sometimes used to create better looking graphical output is to perform anti-aliasing by making a much larger image than you

intend to use, and then shrinking the image down to the correct size. Here, for example, is how we can make two small images of a surface.

First, we create a normal surface, using Hershey fonts in a 300x300 window.

```
IDL> data = cgDemoData(2)
IDL> cgDisplay, 300, 300, Title='Normal Picture'
IDL> cgSurf, data, Font=-1, XTitle='X Title', $
      YTitle='Y Title', ZTitle='Z Title', $
      Position=[0.25, 0.15, 0.90, 0.90, 0.15, 0.95]
```

You can see the result in the left-hand side of Figure 2.

Next, we create the same figure in the Z-graphics buffer, except this time we make the figure four times its normal size, and we make the lines and axes four times as thick as normal. We also use TrueType fonts for this surface plot. We take a snapshot of this large graphics display, and then display it as an image in a window that is the same size as the first window. Note how much better this second plot appears in the right-hand side of Figure 2.

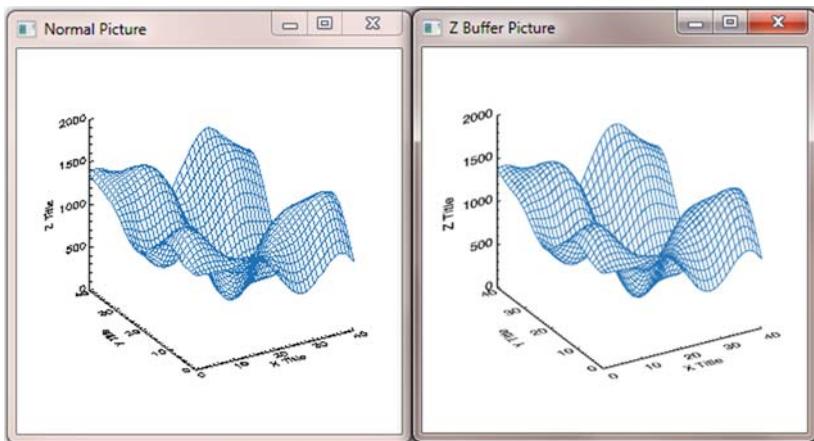


Figure 2: A normal surface plot with Hershey fonts on the left, and a “Z-graphics buffer enhanced” plot on the right, created by making the original plot four times as large and four times as thick, then displaying the result as an image.

```
IDL> thisDevice = !D.Name
IDL> Set_Plot, 'Z'
IDL> Device, Set_Resolution=[1200, 1200]
```

```
IDL> cgSurf, data, Font=1, XTitle='X Title', $  
      YTitle='Y Title', ZTitle='Z Title', Charsize=6, $  
      XThick=4, YThick=4, ZThick=4, Thick=4, $  
      Position=[0.25, 0.15, 0.90, 0.90, 0.15, 0.95]  
IDL> snapshot = cgSnapshot()  
IDL> Set_Plot, thisDevice  
IDL> cgDisplay, 300, 300, WID=2, $  
      Title='Z Buffer Picture'  
IDL> cgImage, Rebin(snapshot, 3, 300, 300)
```

We have displayed this “shrunken” image in a display window, but you could just as easily write it to a file.

```
IDL> image24 = Rebin(snapshot, 3, 300, 300)  
IDL> Write_JPEG, 'test2.jpg', image24, True=1
```

Using ImageMagick for Raster Output

ImageMagick is a command driven program and is most often used via some type of program interface. For example, programming interfaces exists for C, Perl, Python, and C++ languages, among many others. Unfortunately, there is no programming interface for IDL. What we typically do from within IDL is simply execute an ImageMagick command using the *Spawn* command in IDL.

The ImageMagick command we use is the *convert* command, which allows us to convert images from one raster or vector format into another raster format. For example, to turn the TIFF file we created earlier in this chapter into a BMP file, we can issue a command like this from within IDL. ImageMagick uses the file extensions of the input and output file names to know what to do.

```
IDL> Spawn, 'convert test.tif test.bmp'
```

Naturally, for this to work correctly, ImageMagick must be installed on your machine, and the *convert* command must be recognized as an ImageMagick command by your operating system. For additional help and information on installing ImageMagick on your machine, see the ImageMagick web page (www.imagemagick.org).

The procedure I use to create graphical output that I want to share with colleagues is this. I write a program to perform some analysis and display some graphical output. I write the program using the techniques and programs outlined in this book. I “test” the program on my display as I write it. When I am happy with it, I create a PostScript file and write an image file

(typically a PNG file) directly from the program I have been testing on my display.

Doing all this requires three commands, two from the [Coyote Library](#). `PS_Start` opens and configures the PostScript device. Then I run my graphics command, whatever it is. Finally, I type `PS_End` to close the PostScript file, flip the IDL seascape output to real landscape output, if necessary, and make a PNG file automatically. Here are the commands I would use to make a histogram plot for inclusion in this book.

```
IDL> PS_Start, Filename='histoplot.ps'  
IDL> cgHistoplot, cgDemoData(7), /Fill  
IDL> PS_End, Density=300, Resize=100, /PNG
```

You see the result in Figure 3.

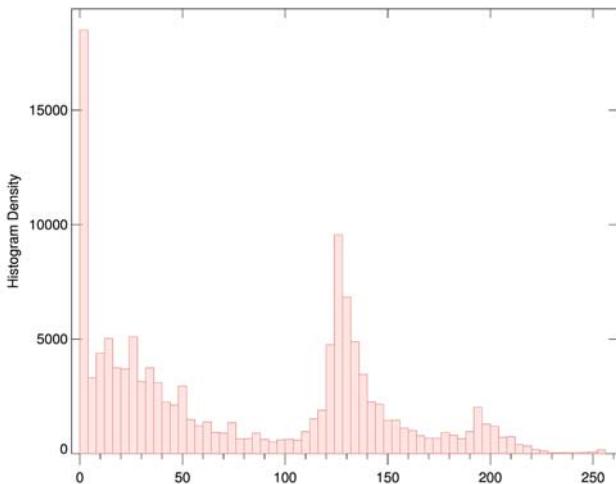


Figure 3: This figure output was created directly from within IDL with just three IDL commands.

The Role of `PS_Start`

The purpose of `PS_Start` is to configure the PostScript device for the graphical command to come. It does several things automatically. First, it creates a common block so that it can share data with `PS_End`. Among the information stored in the common block is the name of the PostScript file that will be created and the name of the current graphics device.

The PostScript device is opened and configured by `PS_Start`. The configuration is actually accomplished by calling `PSConfig` with the *NoGUI* keyword set. `PSConfig` is called in such a way as to create a PostScript “window” with the same aspect ratio as the current graphics window where I have been testing my program.

`PS_Start` doesn’t have to match the aspect ratio of the current graphics window, since I can use any `PSConfig` keyword in `PS_Start`, including the size and offset keywords. The keywords I use are simply collected by `PS_Start` and passed directly to `PSConfig` using keyword inheritance. I do have to set the *NoMatch* keyword, however, if I want to make a PostScript window that has a size or aspect ratio different from the current graphics window. This seldom happens in practice.

```
IDL> PS_Start, XSize=5, YSize=4, /Inches, /NoMatch
```

If I like, I can configure the PostScript device interactively, simply by setting the *GUI* keyword. This will present the `PSConfig` interface to the user (see “Configuring the PostScript Device Interactively” on page 365 for information).

```
IDL> PS_Start, /GUI
```

Setting Fonts and Thicknesses

`PS_Start` also sets a number of system variables that affect graphical output. Remember that system variable values can be over-written by similar keyword values for graphics commands. For this reason, I tend *not* to set the font or the plot thicknesses using keywords on graphics commands. Rather I let `PS_Start` set these values for me with system variables.

The system variables that get set are the following:

```
IDL> !P.Thick = 2
IDL> !P.CharThick = 2
IDL> !X.Thick = 2
IDL> !Y.Thick = 2
IDL> !Z.Thick = 2
IDL> !P.Font = 0 ; PostScript hardware fonts.
```

Note: A system variable is set by `PS_Start` only if that system variable is set to its default value. (The default value is 0 in the case of most system variables and -1 for the font system variable.)

The `!P.Charsize` system variable is also set (from the value of the `Charsize` keyword, if it is used) if it also has a default value of 0. The value of the `!P.Charsize` system variable is determined by the type of machine (Win-

dows versus UNIX), by whether the *!P.Multi* system variable is also being used to produce multiple plots, and by whether the font in use is the PostScript hardware font (*!P.Font=0*) or a TrueType font (*!P.Font=1*). The details of this can be found in the program [cgDefCharSize](#).

In general, larger fonts sizes are used for UNIX computers than Windows computers, and larger font sizes are used for single plots than for multiple plots. TrueType fonts require larger font sizes again. The *!P.Charsize* values are designed to meet my usual requirements. Any of these values can be changed to suit the way you typically produce your PostScript files simply by modifying the code in [PS_Start](#).

These defaults all work well except when I am producing a surface plot. Then I must remember to set the *Font* keyword to 1 to select rotating TrueType fonts for the surface.

```
IDL> PS_Start, Font=1, Filename='surface_plot.ps'
```

[PS_Start](#) stores the values of any system variable it changes, the name of the current graphics device, and the name of the PostScript file that will be created in a common block where it will be accessed by [PS_End](#).

The Role of [PS_End](#)

After [PS_Start](#) configures the PostScript device, and I have issued whatever graphics commands I plan to issue, I call [PS_End](#) to complete the process. [PS_End](#) does the following:

1. Closes the PostScript device.
2. Returns control of graphics output to the previous graphics device.
3. Fixes any PostScript output files that need to be flipped from sea-scape to landscape orientation by calling [FixPS](#). (This can be turned off by setting the *NoFix* keyword.)
4. Restores the original values of any system variables that [PS_Start](#) may have changed.
5. If needed, constructs and issues an ImageMagick command to convert the PostScript file to a raster image file.

Constructing the ImageMagick Command

The essential point here is how the ImageMagick command is constructed. ImageMagick is a software program, like IDL itself, that tends to change fairly rapidly from one version to the next. In fact, commands that work

perfectly in one version will often not work in another version. I point this out because at the time you read this, things may well have changed. You can always find the latest version of ImageMagick on its web page, as well as help and information about the various parameters you can set on an ImageMagick *convert* command.

You can determine the version of the *convert* command you are using by typing the following command at a command prompt (Windows users will have to open a Command Prompt window to issue this command). This is the version of ImageMagick I am using as I write this book.

```
%> convert -version  
Version: ImageMagick 6.5.8-5 2009-12-06 Q16
```

ImageMagick converts input files to output files based on the file extensions of the input and output file parameters. [PS_End](#) has keywords (*BMP*, *GIF*, *JPEG*, *PNG*, and *TIFF*) to create BMP, GIF, JPEG, PNG, and TIFF output files. Other file formats (ImageMagick supports over 100 formats) are possible.

The ImageMagick parameters I use every time are *resize*, *density*, and *alpha*. These parameters can be changed from their default values with the *Resize*, *Density*, and *Allow_Transparent* keywords, respectively.

The *resize* parameter allows me to change the size of the output raster image. I normally create PostScript files in as large a “window” as possible on the PostScript page. I find that by resizing the PostScript output to 25 percent of its original size I get image files that are of sufficient size for me to view them easily. Therefore, the default value for this parameter is 25. (The value must be between 0 and 100 percent of the original size.) Note that you cannot set the pixel size of the output image with this parameter. The default value can be changed by using the *Resize* keyword to [PS_End](#).

```
IDL> PS_End, /PNG, Resize=50
```

The *density* parameter specifies the image resolution of the output image when reading a vector format image. It is in units of dots per inch (DPI). The default density is 300 DPI, which is completely oversampled for the display (the reason the *resize* parameter is set to 25), which normally supports a resolution of between 72 and 96 DPI. I use the 300 DPI as the default because it creates reasonable output if I want to include the image in a PDF file, or print it on a printer. For this book, I normally make my images with a density of 600 DPI for sharper printing, although the images are typically resampled (by Adobe Distiller) to 300 DPI for inclusion in the PDF version of the book.

There is an interplay between the density and the resize parameters, and you must set these according to your purpose in creating the image file. If I want to create images for viewing on my display, which has a pixel resolution of 96 DPI, for example, I might specify these parameters like this.

```
IDL> PS_End, /PNG, Resize=100, Density=96
```

Several years ago, ImageMagick produced images started showing up with transparent pixel backgrounds. Sometimes this was desired, but often it was not. As a default, I have decided to produce images with (typically) white backgrounds, as these show up better on web pages and in presentations. To turn transparent pixels off, I set the *alpha* parameter to "off". To turn transparent pixels on, you must set the *Allow_Transparent* keyword.

```
IDL> PS_End, /PNG, /Allow_Transparent
```

Since the *alpha* parameter is a recent addition to ImageMagick, the *PS_End* program checks the version number and only includes the *alpha* parameter if the parameter is allowed for your version of ImageMagick.

The default *convert* command looks something like this.

```
%convert inputFile.ps -resize 50% -density 300  
-alpha off outputFile.png
```

There are many other ImageMagick optional parameters you may want to use in your ImageMagick command. You can create a string of these options and pass them to the *convert* command with the *IM_Options* keyword. For example, if you want to flip the resulting image and create a mirror image, you can set the *flip* parameter like this.

```
IDL> PS_End, /PNG, IM_OPTIONS=' -flip '
```

The actual ImageMagick command used to create the image file is written to the command window for you to review.

The ability to write device and color model independent graphics programs that can be displayed on my computer and then turned into high quality PostScript, PNG, JPEG, and TIFF output files with just two additional commands has completely changed the way I work with IDL. I routinely make PostScript and PNG files of every graphics output command that is important to me. Not only is my work better documented, but it is ready for presentations, web page development, and even figures for books without any additional effort on my part.

Chapter 12



The Coyote Graphics System

Do We Need Another Graphics System?

With IDL seemingly introducing another graphics system with every major release of IDL (e.g., Insight, Live Tools, iTools, function graphics), who in the world needs another graphics system!? Well, you do, if I may be so bold.

Each of these new graphics systems has been based on the object graphics system in IDL. And I agree that the object graphics system is often a powerful alternative to traditional graphics routines. But there are two problems with it. First, because the object graphics system is powerful, it is also low-level. You have to work close to the bone, as it were, to create graphical displays. It helps, frankly, to have some formal training in computer graphics programming to use it, and the learning curve for most IDL programmers is extremely steep. Too steep to be of practical use.

Second, attempts to shield the user from the steep learning curve almost always result in programs of mind-numbing complexity under the hood. This makes the programs themselves difficult to understand and virtually impossible to write from scratch. In my opinion, all the “new” graphics systems introduced in the past 10 years fall into this category. I think this is the primary reason most of these systems are no longer used.

The second problem is the more serious problem. To make complex programs “simple to use,” you must inevitably make choices that close off alternatives for end users. What we end up with, then, are a handful of extremely complex programs that do what they do extremely well. There

is no question, for example, that the new *Plot* function in IDL 8 can display a line plot reliably (albeit slowly).

What the *Plot* function cannot do is be used to build another, completely different graphics display routine for a particular purpose. It can *only* do what it can do. And because of this limitation, it has been loaded up with all *kinds* of functionality that it would never occur to most people to use! This is why every time you put your cursor in a *Plot* function graphics window, the plot slides all over the window. What!?

Please, this is *way* too much functionality. It reminds me of a certain word processing application from a well-known software vendor.

Give me some simple tools I can use to build graphical displays exactly the way I want them to appear. If I want user interactions, I can program simple user interactions in a widget program. I don't need my plots sliding and zooming and disappearing in my graphics windows, thank you very much!

Okay, enough ranting. But, I *do* need graphics programs that don't look like they were written by programmers still wearing wide paisley ties and bell-bottomed pants from the 1970s. I need something simple that works in a modern computer environment.

This is where the Coyote Graphics System fits in. It uses simple tools that work together. You can build things with it. It conforms to your ideas, rather than making you conform to its. I think of it as a stop-gap graphics system that works until there is a better alternative that can be understood and programmed by people who do something other than write computer programs for a living. In other words, I think of it as a graphics system for people who are still craftsmen, who still have ideas for graphical displays that fall outside the box. This chapter explains how the Coyote Graphics System works.

Using Resizeable Graphics Windows

You have already been introduced to bits and pieces of the Coyote Graphics System in previous chapters. The Coyote Graphics System started out as just a pretentious name for graphics programs in the [Coyote Library](#). To tell you the truth, I didn't even realize it *was* a system until I was nearly finished writing this book. I thought it was just a collection of pretty good programs I was putting together from principles outlined in the book. I was writing these programs in a “proof of the pudding is in the eating” sort of way.

I wrote the final program in this series, [cgWindow](#), in two long days of programming after I finished a draft of the book and while I was taking a break from writing. While I was putting around with it, fixing bugs, I suddenly had a Big Idea. I could use [cgWindow](#) and the other programs I was writing for the book to write programs in resizeable graphics windows in almost *exactly* the same way I was currently using normal IDL graphics windows! One week of frantic, around-the-clock programming later, and I had a “system” that was, quite frankly, larger than the sum of its parts.

The resizeable graphics window, [cgWindow](#), is the heart of the system. You will be better able to write programs that take advantage of [cgWindow](#) functionality if you understand exactly how it works.

How Does a Resizeable Graphics Window Work?

The [cgWindow](#) module itself is a small, almost insignificant wrapper to an underlying object named *FSC_CmdWindow*. This underlying object is also a small widget program that consists of a top-level base, a few buttons in a pull-down menu, and a draw widget that is the graphics window. Widget resize events are turned on for the top-level base and there is a method (*Resize*) that responds to these widget events by making the draw widget the correct new size and then re-drawing whatever graphics commands happen to be in the window. The essential feature of this resizeable graphics window is the process by which these graphics commands “happen to be in the window.”

Requirements for Loading a Graphics Command

The fundamental principle of [cgWindow](#) is that graphics commands can be “loaded” into the program. There are several requirements a graphics command has to meet to be successfully loaded into the window.

1. The graphics command must be an IDL procedure or an object method. It does not have to be a built-in routine (e.g., *Plot*, *Contour*, etc.). It can be any IDL routine you care to write that meets the requirements.
2. The graphics command must have no more than three positional parameters. None of these positional parameters is required. It can have an unlimited number of keyword parameters associated with it. If the graphics command is an object method, the first positional parameter *is* required and must be a reference to a valid object to call the method on. (Three is an arbitrary number. It just happened that none of the traditional IDL graphics commands I

wanted to use had more than three. Nor did any of my graphics routines in my library.)

3. The graphics command must have an *_Extra* keyword defined for it on the procedure or method definition statement.
4. The graphics command must be written in such a way that it will execute properly in the PostScript device. In practice this means “protecting” the PostScript device from commands that are not appropriate to use in the PostScript device (see “Protecting Window Commands in PostScript” on page 378 for details).
5. The graphics command must not open other graphics windows. In other words, it must display itself in any currently open graphics window.

To load a graphics command into `cgWindow`, you pass the name of the graphics command as its first positional parameter and then pass the positional and keyword parameters exactly as if you were executing the command itself. If the graphics command is a method, you will also have to set the *Method* keyword to alert the program to that fact.

```
IDL> !P.Background = cgColor('white', 254)
IDL> !P.Color = cgColor('black', 253)
IDL> cgWindow, 'Plot', cgDemoData(1), XTitle='Time', $
      YTitle='Signal', CharSize=1.5
```

You see the result in Figure 1.

Called in this way, a new `cgWindow` is created with the graphics command displayed in the window. The window is resizeable and you can use the button controls in the upper-left of the display to create a PostScript file or to save the contents of the graphics window in any of five different raster file formats. If ImageMagick is found on your machine, there will also be a button to make raster image files from PostScript output using the ImageMagick *convert* command. (See “Using ImageMagick for Raster Output” on page 411 for details.) If ImageMagick is not found, this button will not be available.

There are also buttons available to save the current visualization (i.e., what you see on the display) in a file so you can e-mail it to a colleague or review it again at some later time.

Packaging the Graphics Command

Internally, `cgWindow` “packages” the graphics command in another internal object named *FSC_Window_Command*. By “packages” I mean the

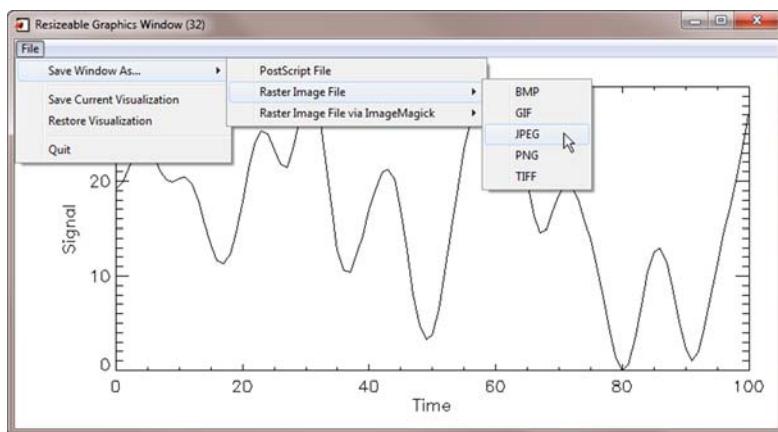


Figure 1: A basic `cgWindow` with a `Plot` command displayed in it. The command can be sent directly to a PostScript file, or the window contents can be saved in any of five raster file formats.

command name and all the data and keyword values are stored so the command can be played back or recreated at a later time.

This object is very simple. It has a `List` method that can print a simplified representation of the command. And it has a `Draw` method that can “execute” the command by calling `Call_Procedure`, if the graphics command is a procedure, or by calling `Call_Method`, if the graphics command is an object method.

Note: The command keywords are passed to `Call_Procedure` or `Call_Method` using the keyword inheritance mechanism. Because these routines do not use a pass by reference method of passing keywords, it is currently not possible to obtain output keyword values from commands that are entered into a `cgWindow` application. This is a limitation I hope to overcome in the next generation of Coyote Graphics System commands.

After the graphics command is packaged properly, it is added to a “command list” that is maintained by the `FSC_CmdWindow` object. The command list is simply a linked list (created with the `LinkedList` object). Objects are persistent in the IDL session, so this object with its command list will persist as long as the `cgWindow` application remains on the display. When the `cgWindow` is removed from the display (we say “destroyed” in widget terminology), the underlying object is destroyed,

which in turn destroys the internal command objects, releasing any memory that was required to store data.

Adding Graphics Commands

There is no limit to the number of command objects that can be stored in the command list. To add an additional command to the current `cgWindow` application, you simple call `cgWindow` again, but this time you set either the `AddCmd` or `LoadCmd` keyword. The difference is that if `AddCmd` is set, all the commands in the command list are immediately executed as soon as a new command is loaded. If `LoadCmd` is set, the command is loaded, but the commands are not executed. This gives you the opportunity to load a number of commands into the window without a great deal of window flashing as commands are loaded and executed. You can execute the commands on the command list at any time by calling `cgWindow` with no arguments and with the `ExecuteCmd` keyword set.

So, for example, we could add a second command to the current window by using the `AddCmd` keyword, like this. In this case, both commands are immediately executed and shown in the window.

```
IDL> cgWindow, 'PlotS', IndGen(101), cgDemoData(1), $  
      PSym=2, SymSize=1.5, /AddCmd
```

Or, we can load the same command, but not execute it immediately, and then execute both commands when we are ready, like this.

```
IDL> cgWindow, 'PlotS', IndGen(101), cgDemoData(1), $  
      PSym=2, SymSize=1.5, /LoadCmd  
IDL> cgWindow, /ExecuteCmd
```

You see the result in Figure 2.

Care should be taken to only load commands that “go together” or create a single graphical visualization. It would be a mistake, for example, to load a *Plot* command and then load a *Surface* command as the second command. When the commands were executed by the window, the *Plot* command would appear and be immediately overwritten by the *Surface* command, since the *Surface* command erases what is currently in the graphics window before it displays itself. (You can, of course, display multiple plots in a `cgWindow`, but this will be discussed in “Displaying Multiple Commands” on page 425.)

Listing Graphics Commands

To see a listing of the graphics commands currently in a window, call `cgWindow` without any arguments and set the `ListCmd` keyword. The

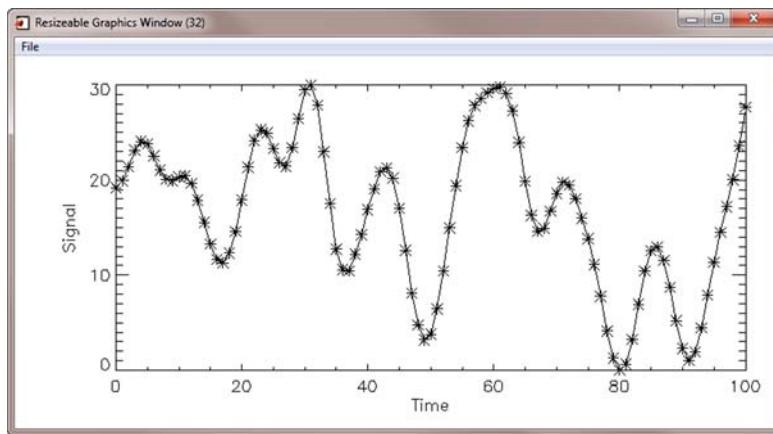


Figure 2: The `PlotS` command is added to the window, creating symbols on the plot.

commands are listed with placeholder arguments. The listing is just meant to give you an idea of which commands are in the window.

```
IDL> cgWindow, /ListCmd
      0. Plot, p1, CHARSIZE=value, XTITLE=value, $
          YTITLE=value
      1. PlotS, p1, p2, PSYM=value, SYMSIZE=value
```

Notice that the commands are numbered with their “command index number,” starting with command 0. Commands are often identified by their command index number when they are being manipulated by other commands. The command index number is typically specified with the `CmdIndex` keyword.

Replacing Graphics Commands

One use of the command index number is to identify which command in a command list we would like to replace. Suppose, for example, we meant to display the symbols in Figure 2 in a red color. We can replace the command in the command window, using the `ReplaceCmd` keyword to `cgWindow`. We identify the command we want to replace with the `CmdIndex` keyword.

```
IDL> cgWindow, 'PlotS', IndGen(101), cgDemoData(1), $
      PSym=2, SymSize=1.5, Color=cgColor('red'), $
      /ReplaceCmd, CmdIndex=1
```

You see the result in Figure 3.

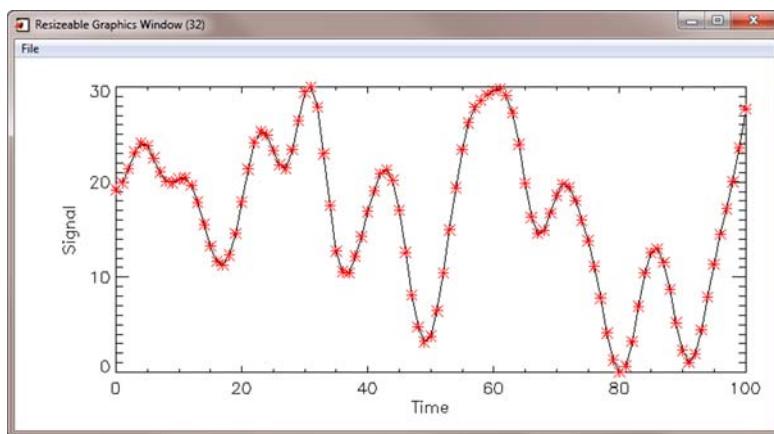


Figure 3: A command can be replaced in the command list by setting the *ReplaceCmd* keyword and specifying the command index number with the *CmdIndex* keyword.

You have to be careful replacing a command because one of the features of `cgWindow` is that if you use the *ReplaceCmd* keyword and you do not specify a specific command with the *CmdIndex* keyword, all the commands in the window are replaced by this single new command. This is basically how normal IDL graphics windows work and, as you will see shortly, it makes it possible to treat `cgWindow` as a normal graphics window when using other programs in the Coyote Graphics System.

To replace the current two commands with a *Surface* command, you would type this.

```
IDL> cgWindow, 'Surface', cgDemoData(2), /ReplaceCmd
```

You see the result in Figure 4.

Deleting Graphics Commands

It is possible to delete a command in the command list by using the *DeleteCmd* keyword. If you have a specific command in mind, it should be identified by its command index with the *CmdIndex* keyword. Unlike replacing a command, however, when you don't specify a command index, it is the command with the largest command index number that is deleted. This is typically the last command added to the command list, which gives you the opportunity to experiment with a command and if you don't like the result, simply delete it.

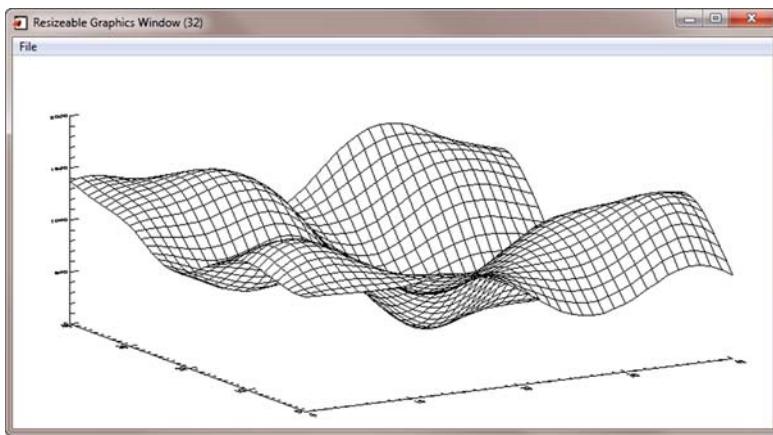


Figure 4: All of the commands in the window are replaced with the new command if you use the `ReplaceCmd` keyword without specifying a command index.

You can delete the `Surface` command, which is the only command left in this `cgWindow` application now, and create an empty graphics window with this command.

```
IDL> cgWindow, /DeleteCmd
```

If you have multiple commands in the command window and you want to delete them all, and *not* replace them with a new command, then you can use the `cgControl` command to do so. This command will be discussed in detail in the section “Configuring Resizeable Window Properties” on page 435.

Displaying Multiple Commands

`cgWindow` has the ability to display multiple commands in its window. One way to set the program up to display multiple commands is to use the `WMulti` keyword when you first create the `cgWindow` application. The `WMulti` keyword is a five-element integer array identical to the `!P.Multi` system variable you have used throughout this book. In fact, the value of `WMulti` is used to configure the `!P.Multi` system variable in `cgWindow` before the commands are executed.

Once a window has been created, however, you must change its properties by using the `cgControl` command. This command has various keywords to allow you to control or configure properties of the `cgWindow`. The prop-

erty we are interested in at the moment is controlled with the *Multi* keyword.

The `cgControl` command has a selection parameter that allows the user to specify which window is being controlled. If no selection is made, the current `cgWindow` application is used. This is typically the last `cgWindow` created. Since we only have one window, that window is the current window. Since the window on the display has already been created, we must change its multiple plot property with `cgControl`. Here we set the window up to display side-by-side plots.

```
IDL> cgControl, Multi=[0,2,1]
```

Now can we add, for example, a line plot next to a surface plot.

```
IDL> cgWindow, 'Plot', cgDemoData(1), XTitle='Time', $  
      YTitle='Signal', CharSize=1.5, WMulti=[0,2,1], $  
      /AddCmd  
IDL> cgWindow, 'Surface', cgDemoData(2), /AddCmd
```

You see the result in Figure 5.

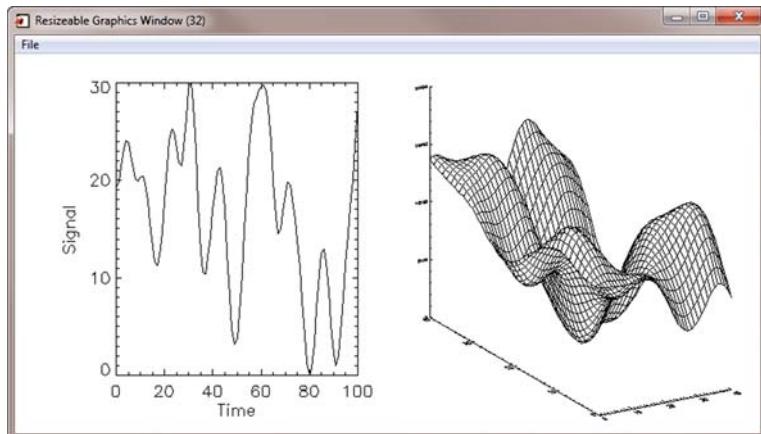


Figure 5: The `cgWindow` command is capable of displaying multiple plots in the window. Set the *Multi* property of the window in the same way you would set the `!P.Multi` system variable.

Commands that you add to `cgWindow` when you are doing multiple plots are no different from the same commands you would use if you were doing multiple plots in a normal window with `!P.Multi`.

Erasing the Resizeable Window

In an normal graphics window you use the *Erase* command to erase the contents of the window. We basically do the same thing with `cgWindow` applications, although in this case “erase” means delete all the current commands from the command list. The `cgErase` command, with the *Window* keyword set, will do this for the current `cgWindow`.

```
IDL> cgErase, /Window
```

Destroying the Resizeable Window

To completely destroy a `cgWindow` programmatically, you either use `cgControl` with the *Destroy* keyword set, like this.

```
IDL> cgControl, /Destroy
```

Or, you can use `cgDelete`, which we will discuss in the next section.

Working with Multiple Resizeable Windows

There is no limit to the number of resizeable `cgWindow` applications you can have on the display at the same time. Each one is completely autonomous and maintains its own command list and internal settings. Like normal graphics windows, if you create multiple `cgWindow` applications, the last one you create is the “current” window. Unless specifically addressed to a particular window, all commands, operations, or interactions intended for a `cgWindow` will be directed to the current window.

To make a window the current window, you use the `cgSet` command. If called without any arguments, this command will pull the current window forward of other windows on the display. That is to say, it will act like a `WShow` command. (Unfortunately, this extremely useful functionality has disappeared in IDL 8.0 and no one I have talked to knows if this functionality can or will be restored.)

Selecting a Window

Normally, a graphics window is selected to be the current graphics window (with the `WSet` command) on the basis of its window index number. `cgWindow` applications have window index numbers, too, and that is one way they can be identified and selected. To help with this, the default title of `cgWindow` contains the window index number, in parentheses, in the title bar of the window. This is similar to how normal IDL graphics windows are identified in their titles.

To see how this works, let’s create three separate `cgWindow` applications.

```
IDL> cgWindow  
IDL> cgWindow  
IDL> cgWindow
```

You see the result in Figure 6. You may have completely different window index numbers than those shown in the figure. But, you should see the three windows slightly offset from one another in the upper-left corner of your display.

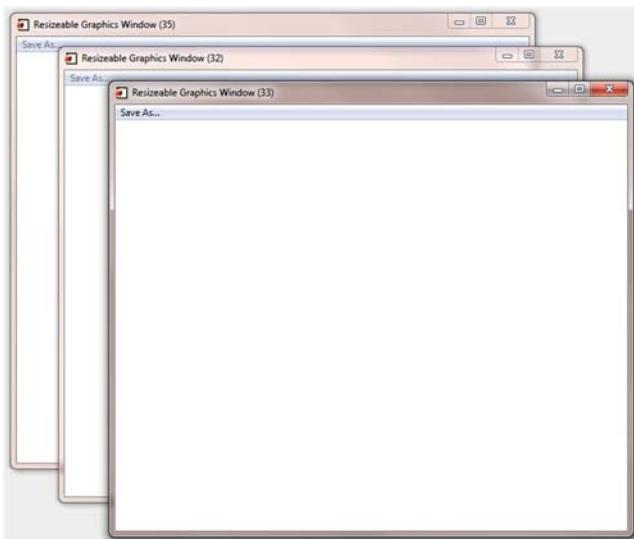


Figure 6: The default title of these resizable windows contains their window index numbers. Windows can be made the current window by selecting the window with `cgSet` and using the window index number as the selection criteria.

In the example in Figure 6, I can make window 35 the current `cgWindow` by typing this.

```
IDL> cgSet, 35
```

Notice that this window is also brought forward on the display so you can see it clearly. Window commands can now be directed to this “current” `cgWindow` application. For example, you can load a line plot in this window.

```
IDL> cgWindow, 'Plot', cgDemoData(1), XTitle='Time', $  
YTitle='Signal', CharSize=1.5, /AddCmd
```

Now you can switch to another window and load a surface plot, and so forth.

```
IDL> cgSet, 32  
IDL> cgWindow, 'Surface', cgDemoData(2), /AddCmd
```

Note: Resizeable graphics windows can be brought forward on the display, without making them the current window, with the `cgShow` command. This is the equivalent to the `WShow` command for normal IDL windows.

Let's delete these three windows, and then I will show you another way to select the current window. You can, of course, delete the three windows with your mouse, but another way you can delete a window is with the `cgDelete` command. Simply identify the window you want to delete. For example, to delete the window with window index number 35, you type this.

```
IDL> cgDelete, 35
```

If you want to delete all the `cgWindow` applications on the display, you can set the `All` keyword, like this.

```
IDL> cgDelete, /All
```

Another way to select `cgWindow` applications is by means of their titles. Let's create three windows again, but give them different titles. You will notice their window index numbers now appear to be unknown. (You will learn how to discover what they are in just a moment.)

```
IDL> cgWindow, WTitle='Plot Window'  
IDL> cgWindow, WTitle='Surface Window'  
IDL> cgWindow, WTitle='Contour Window'
```

Suppose you now want to put a plot in the plot window. You can still do the selection with `cgSet`, but it is easier to use the title of the window as the selection criteria. To alert `cgSet` to this fact, the `Title` keyword must be set.

```
IDL> cgSet, 'Plot Window', /Title  
IDL> cgWindow, 'Plot', cgDemoData(1), XTitle='Time', $  
    YTitle='Signal', CharSize=1.5, /AddCmd  
IDL> cgSet, 'Surface Window', /Title  
IDL> cgWindow, 'Surface', cgDemoData(2), /AddCmd
```

When the window title is used as the selection criteria, the title comparison is *not* case sensitive.

Windows can also be selected on the basis of their top-level base widget identifiers or their underlying object reference identifiers. You must use the appropriate keyword to `cgSet` to indicate which selection criteria you are using. The window index number is the default selection criteria.

Obtaining Information about Windows

But, how would you know what *any* of these identifiers are? And, how could you select them if you didn't know!?

You can obtain information about the `cgWindow` applications that are currently on the display with the `cgQuery` command. This command is a function that returns the window index numbers of all the `cgWindow` applications currently on the display. Keywords can be set to obtain the corresponding window titles, window top-level base identifiers, object references, and dimensions of the window. To obtain all this information at once, including a count of how many windows are currently on the display, type this. Notice that all five of these keywords are output keywords.

```
IDL> winIDs = cgQuery(Title=winTitles, $  
WidgetID=winTLBs, ObjectRef=winObjRefs, $  
Dimensions=dims, Count=count)
```

We can use the information in whatever way is appropriate. For example, we can print it to the command log.

```
IDL> FOR j=0,count-1 DO Print, winIDs[j], ' ', $  
winTitles[j], ' ', winTLBs[j], ' ', $  
winObjRefs[j], dims[*,j]  
32 Plot Window 560 <ObjHeapVar701> 640 512  
33 Surface Window 579 <ObjHeapVar711> 500 480  
34 Contour Window 598 <ObjHeapVar720> 620 512
```

Note that your output will probably look different from what is shown here.

If you want to obtain this information for the current `cgWindow` only, you simply set the *Current* keyword. The program works exactly the same, except that the *Title*, *WidgetID*, *ObjectRef*, and *Dimensions* output keywords will return just the information for the current window. Only the window index number of the current window is returned as the function result. (This is like asking for the value of *!D.Window*.) The *Count* keyword always returns the total number of `cgWindow` applications on the display, even when the *Current* keyword is used.

```
IDL> winID = cgQuery(/Current, Title=winTitle, $  
WidgetID=winTLB, ObjectRef=winObjRef)
```

If you just want to know the window index number of the current `cgWindow` program, for example, you would obtain it like this.

```
IDL> windowIndex = cgQuery(/Current)
```

Let's delete these three windows before moving on to the next topic.

```
IDL> cgControl, /Destroy, /All
```

Let me explain what just happened in this command. The `cgControl` program used `cgQuery` to find the top-level base widget identifiers of all the windows currently on the display. Then, it destroyed each of them with a call to *Widget_Control* with the *Destroy* keyword set. You can imagine other scenarios in which having the widget identifier or object reference of all the windows would be useful. For example, you might want to change the display colors for all the windows on your display. It would take perhaps five minutes to write a routine to do this.

Changing Colors in Resizeable Windows

The `cgWindow` program is designed to have its own color table. If a color table is not assigned to it with the *Palette* keyword from either of the property setting functions (discussed in “Configuring Resizeable Window Properties” on page 435), then it will use as its color table the color table in effect when the window is opened. So, for example, if you want to create a resizeable window for displaying an image you may want to start by loading a color table for the image.

```
IDL> cgLoadCT, 5, /Brewer, /Reverse  
IDL> cgWindow  
IDL> cgImage, cgDemoData(7), /Axes, /Window
```

You see the result in Figure 7.

Once a color table has been assigned to the window, these colors are “protected.” This simply means that before the graphics commands are executed, the window color vectors are loaded into the system color table. In other words, the image will always display in the proper colors, no matter what has happened to the system color table in the world outside the graphics window.

But, this is also a problem, because what if you want to change the colors the image is being displayed in? One way you can change the window color vectors is with the property setting command, `cgControl`. A new color palette is created with `cgLoadCT` and then loaded with `cgControl` into the current resizeable window.

```
IDL> cgLoadCT, 33, RGB_Table=palette  
IDL> cgControl, Palette=palette
```

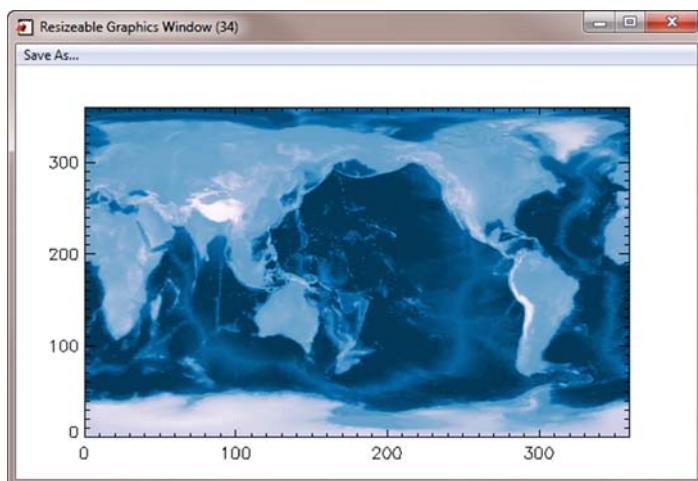


Figure 7: Color table vectors are loaded at the time the resizable graphics window is created unless a color palette has been loaded with either of the property setting commands.

This works, and the image colors are immediately updated in the window, but it is awkward. Certainly it is more awkward than just loading a new color table and re-displaying the image, as we would do in a normal graphics window. But, don't worry.

It turns out both `cgLoadCT` and `XColors` have been built with the ability to communicate with objects. (This functionality is lacking in both `LoadCT` and `XLoadCT`, the normal color table loading programs in IDL.) And, as it happens, `cgWindow` is, at heart, an object. Ergo, something that seemed hard to do is now simple. Simply setting the `Window` keyword on either of these two color table loading routines, will send the color table information directly to the current window object, so the colors can be updated immediately in the window.

So, all you need to do to change color tables for the current resizable graphics window is this.

```
IDL> cgLoadCT, 5, /Window
```

Or, this.

```
IDL> XColors, /Window
```

The colors in the graphics window are updated immediately with either routine. In this case, changing colors in resizable graphics windows turns out to be even *easier* than using normal graphics windows!

Using Command Delay

Although I don't recommend using `cgWindow` as an animation tool, the fact that it can execute commands in a command list does give it possibilities that don't exist in other tools. In fact, the first out-of-the-box use of `cgWindow` I heard about was from someone trying to use it to animate a short series of weather images. The novelty of the idea appealed to me, so I decided to add a command delay capacity in the program to facilitate such uses. But, did I mention I do *not* recommend using `cgWindow` as an animation tool? Please use good sense in using this feature and do *short* animations only, as once an animation sequence starts there is no way to stop it until it finishes.

Just to give you a sense of how this could work, here is code to create a short animated series of images.

```
IDL> image = cgDemoData(18)
IDL> cgLoadCT, 2, /Brewer
IDL> cgWindow, CmdDelay=0.25
IDL> cgImage, image, /Axes, Margin=0.1, /Window
IDL> FOR j=0,20 DO cgWindow, 'cgImage', cgDemoData(18), $
      /Axes, Margin=0.1, /LoadCmd
IDL> cgControl, /Execute
```

If you want to change the delay in the window, you can use `cgControl` with the *Delay* keyword. For example, to make the animation slower, you could type this.

```
IDL> cgControl, Delay=0.5, /Execute
```

To delete the commands in the window and use the window for another purpose, you can use the `cgErase` command.

```
IDL> cgErase, /Window
```

Saving and Restoring Window Visualizations

The `cgWindow` program allows you to save and restore visualizations you create in the graphics window. Whenever you have something you like, you can use the Save Current Visualization button under the File pull-down menu in the upper-left corner of the window to save the visualization to a file. The file is normally given a `*.cgs` extension, but you have the opportunity to name the file anything you like.

To show you how this works here is a visualization containing four different plot types.

```

IDL> cgLoadCT, 4, /Brewer, /Reverse
IDL> cgWindow, WMulti=[0,2,2]
IDL> cgPlot, cgDemoData(1), Color='red', $
    SymColor='grn5', PSym=-2, $
    YTitle='Signal', XTitle = Time, /AddCmd
IDL> cgHistogram, cgDemoData(7), /Fill, /AddCmd
IDL> cgImage, cgDemoData(16), /Axes, $
    MultiMargin=[4,8,2,2], /AddCmd
IDL> cgContour, cgDemoData(2), /Fill, NLevels=12, $
    /AddCmd
IDL> cgContour, cgDemoData(2), Color='charcoal', $
    NLevels=12, C_Charsize=0.75, /Overplot, /AddCmd

```

Once you have the plots loaded into the window, select the Save Current Visualization button. Give the visualization file a name like *example.cgs*. You see the result in Figure 8.

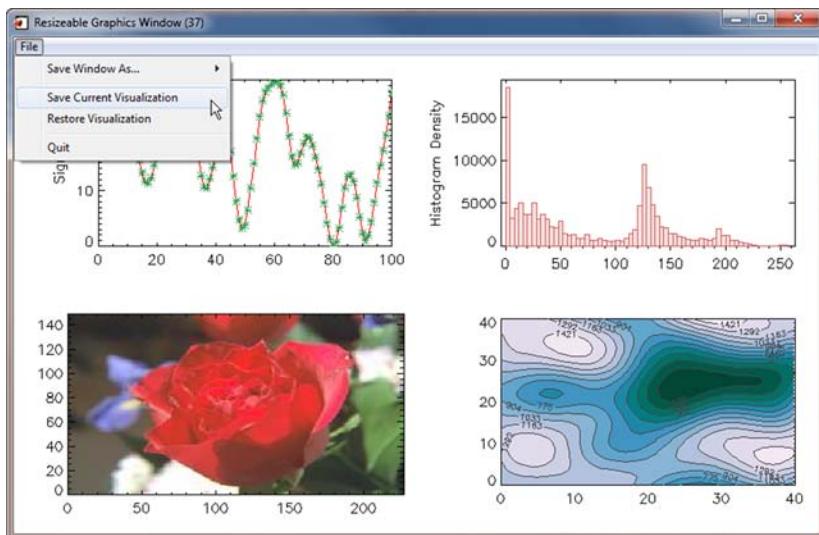


Figure 8: It is possible to save the current visualization. You can send the visualization to a colleague to view the same display you are viewing, or you can re-open the visualization and view it again at any time.

The visualization is stored in an IDL save file, which is a transportable binary format. You can e-mail such a file to a colleague, for example, and they can open the file and see the same data visualization you are viewing.

on your display. (Assuming, of course, your colleague has also installed the [Coyote Library](#) programs.)

To restore a previously saved visualization, you simply open a [cgWindow](#) program and use the Restore Visualization button, which is also in the pull-down File menu, to select a visualization file to restore. Any current commands in the window are deleted, and the commands from the saved visualization are loaded into the window.

If you would like to see the visualization you just saved, open a new [cgWindow](#) program, select the Restore Visualization from the pull-down menu, and find and select the *example.cgs* file you just saved. The saved commands now appear in this new window. The window has all the functionality it had before with the restored commands.

Configuring Resizeable Window Properties

You can configure [cgWindow](#) properties in either of two ways. First, the properties can be configured globally, so that the properties apply to any future [cgWindow](#) created after the properties are set. Or, alternatively, the properties can be configured for a particular [cgWindow](#) application *after* the window is created.

Note that I am talking here about *default* properties. Many properties of [cgWindow](#) can be set with keywords to [cgWindow](#) itself (e.g., the initial size of the window, the background color, the multiple plot layout, etc.) and these keyword properties always take precedence over default properties, as they always do in IDL graphics programs.

Global properties are set with [cgWindow_SetDefs](#) and window properties are set with [cgControl](#). Properties that can be set fall into three general categories: window properties, PostScript properties, and ImageMagick *convert* command properties. All properties are set by passing keyword arguments to the property setting routine.

Window Properties

Properties that can be set with the property setting functions will undoubtedly change over time, so you should be sure to check the documentation headers of these routines for the latest information. But, currently, the following window properties can be set by either routine.

Background

The background color of the window. Depending upon which commands are loaded into the

window, you may or may not see the window background color. The default background color is “white.”

Delay	The amount of delay between command execution. The default is 0.0.
EraseIt	This keyword must be set to erase the window in the background color. The default is 0.
Multi	This keyword sets the multiple plot layout. It is identical to the <i>!P.Multi</i> system variable. The default is to do single plot layouts.
Palette	This is a N by 3 or 3 by N byte array of color table vectors. These are the colors that will be loaded in the color table before window commands are executed.
XOMargin	If multiple plots are in effect, this keyword value sets the <i>!X.OMargin</i> system variable.
YOMargin	If multiple plots are in effect, this keyword value sets the <i>!Y.OMargin</i> system variable.

In addition, the `cgWindow_SetDefs` routine can set the following properties, which are otherwise set on the `cgWindow` command itself (although by attaching a “W” to the front of these keywords to avoid conflict with graphics command keywords).

Title	The default title for the window. The default is “Resizeable Graphics Window.”
XPos	The X offset from the upper-left corner of the display. This is used to position windows on the display. The default is -1, which indicates to the program that windows should be slightly offset and tiled as they are created.
XSize	The initial X size of the window in device units.
YPos	The Y offset of the window. (See the <i>XPos</i> explanation for details.)
YSize	The initial Y size of the window in device units.

I recommend you put a `cgWindow_SetDefs` call into your IDL start-up file to configure the default `cgWindow` routines for your IDL session. For

example, if you want your windows to be smaller and to have light gray backgrounds, and you want to use the rainbow color table as the default color table for image display, you can add commands like this to your IDL start-up file.

```
IDL> cgLoadCT, 33, RGB_Table=pal  
IDL> cgWindow_SetDefs, XSize=650, YSize=375, $  
    Background='light gray', Palette=pal
```

Then, all you have to do to display an image in the window configured like this is type this command.

```
IDL> cgImage, cgDemoData(7), /Margin, /Window, /NoErase
```

You see the result in Figure 9.

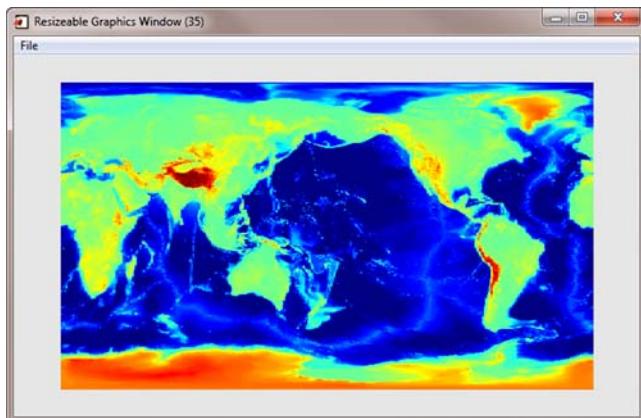


Figure 9: Windows can be configured to your personal specifications by setting window properties either globally for all windows or individually for a specific window.

If you want to change the background color and color table for this specific window, you can type something like this.

```
IDL> cgLoadCT, 5, RGB_Table=pal  
IDL> cgControl, Background='charcoal', Palette=pal
```

The window immediately updates itself, and you see the result in Figure 10.

You can obtain global window properties by setting comparable keywords to the `cgWindow_GetDefs` command. For example, if you want to know the setting for the global background color, you can type this.

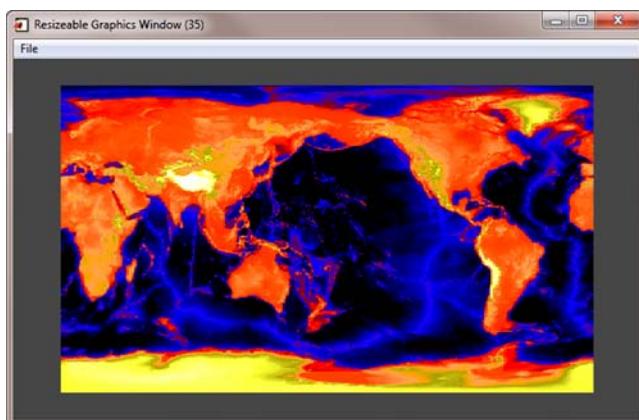


Figure 10: The window is immediately updated when the window properties are changed.

```
IDL> cgWindow_GetDefs, Background=bg  
IDL> Print, bg  
      light gray
```

To obtain the current setting for this particular window, you have to use the command object's *GetProperty* method. The commands you would use look like this.

```
IDL> void = cgQuery(ObjectRef=obj, /Current)  
IDL> obj -> GetProperty, Background=bg  
IDL> Print, bg  
      charcoal
```

You can change global properties at any time in the IDL session, but the properties will apply only to newly created `cgWindow` applications. The ones currently on the display will not be affected by the changes. To reset global properties to default values, use the *Reset* keyword.

```
IDL> cgWindow_SetDefs, /Reset
```

PostScript Properties

The `cgWindow` program can automatically send the command list to a PostScript file to create genuine PostScript output. (The output is not just copied from the graphics window.) The PostScript device is configured with `PS_Start`, which in turn calls `PSCConfig` to create the necessary `Device` keywords for the PostScript device. (See “The Role of `PS_Start`” on page 412 and “Configuring the PostScript Device Interactively” on page 365 for additional information.)

Users will have a chance to specify exactly how they want the PostScript file created from the [PSConfig](#) interface, but it is always a good idea if they don't have to make too many changes and can just hit the *Accept* button to create the PostScript file.

PostScript properties deal with how this interface is presented to the user and with some of the details of how the PostScript file is created. The following keywords can be used to set these properties either globally or locally.

PS_Charsize	The character size for PostScript output. The default is set to 0.0, which means use the “normal” way of determining character size. For Coyote Graphics routines, this means use the function cgDefCharsize to determine character size, but for most routines it means set the character size to 1.0. Note that this keyword only has an effect if the <i>!P.Charsize</i> system variable is set to its default value of 0.0.
PS_Encapsulated	This sets the encapsulated PostScript output button on the PSConfig interface. Default is 0.
PS_Font	This sets the type of font used in the PostScript device. The default is to use hardware PostScript fonts, so the default is 0. If you are going to display a 3D plot in a resizable graphics window, you should use either Hershey (-1) or TrueType (1) fonts.
PS_Metric	This keyword determines if the PSConfig interface appears in US style with inches and a standard US page size, or in metric units with centimeters and an A4 standard page size. The default (by a slim majority of users, I'm afraid) is 0, US style.
PS_Scale_Factor	This keyword sets the PostScript scale factor. The default is 1.0.
PS_TT_Font	This keyword sets the name of a TrueType font to use in creating PostScript output. It only has effect if <i>PS_Font</i> is set to 1 to create TrueType font output. The default TrueType font is “Helvetica.”

To display all `cgWindow` PostScript output with a Times TrueType font on A4 paper, you can set your global defaults like this.

```
IDL> cgWindow_SetDefs, PS_Metric=1, PS_Font=1, $  
      PS_TT_Font='Times'
```

The only way to access PostScript settings is via `cgControl` or `cgWindow_SetDefs`. These properties cannot be set from the `cgWindow` command itself.

ImageMagick Properties

If ImageMagick is installed on your machine, `cgWindow` gives you the option to create high-quality raster image files from PostScript intermediate files. (See “Using ImageMagick for Raster Output” on page 411 for detailed information.) The way these files are created with the ImageMagick *convert* command is controlled by keywords to `PS_End`. (See “The Role of `PS_End`” on page 414 for additional information.) These keywords can be set by setting the following ImageMagick properties in `cgWindow`.

IM_Transparent	Some versions of ImageMagick can create raster images with transparent background colors. This feature is turned off for ImageMagick versions that support the feature unless this keyword is set.
IM_Density	This keyword controls the density at which the PostScript file is sampled to create the image raster file. The default is 300 dpi, which is the recommended density for high-quality printing.
IM_Raster	This keyword controls whether raster files are programmatically created directly in IDL or by using ImageMagick to create the raster file via a PostScript intermediary. The default is to use ImageMagick, if it is available.
IM_Resize	This keyword controls the amount of resizing ImageMagick does when it creates the raster image file. The default is to shrink the sampled PostScript file to 25 percent of its current size. If you sample at a reduced density, then you don't need to shrink the output as much to produce a reasonably sized image for display. The <i>IM_Density</i> and <i>IM_Resize</i> parameters are

adjusted to create an output image of sufficient size and resolution to meet your requirements for the image.

IM_Options

This keyword allows you to pass other ImageMagick options to the *convert* command. Anything you want to include on the command can be specified here.

PS_Delete

This is actually a [PS_End](#) keyword, but it applies to ImageMagick. Normally, the intermediate PostScript file is deleted when the raster image file is created. Setting this keyword to 0 will prevent this from happening.

Suppose you were creating very high resolution PNG files for a book you were writing. You might want your ImageMagick properties to be set like this.

```
IDL> cgWindow_SetDefs, IM_Resize=100, IM_Density=300
```

The [cgControl](#) routine is rapidly acquiring the ability to control many aspects of [cgWindow](#). Be sure to check the latest version of [cgControl](#) to see what it can do. Between the time this chapter of the book was written and the book went to print, [cgControl](#) was able to programmatically create PostScript and raster files, turn automatic command execution on or off, and find the value of command keywords, among other useful things. More functionality will have certainly been added by the time you read this.

Using Coyote Graphics Commands

The [cgWindow](#) resizable graphics window has been written to work with any traditional IDL graphics command or program that conforms to a few simple rules (see “Requirements for Loading a Graphics Command” on page 419). These are, more or less, simply the rules outlined in this book for writing device independent programs. No surprise there.

And there is no surprise that other Coyote Graphics routines also follow these rules. I’ve been writing such routines for at least the past 10 years. What *did* come as a surprise to me, however, was how easily these Coyote Graphics routines could be adapted to work with resizable graphics windows in a natural and easy-to-use manner. It is almost as easy as working with normal graphics windows.

I was shaken when I realized that almost *anyone* could learn to write IDL graphics programs that worked with resizeable graphics windows. It isn't that it was just easy. The disorienting thing to me is that it was *dead easy!* Let me explain.

Preparing Commands for Resizeable Windows

The essential property of a graphics routine that is going to be displayed in [cgWindow](#) is that it work identically with both the display device (X or WIN, and preferably both!) and with the PostScript device. This is already true for any graphics program in the Coyote Graphics System, so this part of the preparation is dead simple.

You can either use these routines to build your graphics programs, in which case you don't have to give much thought to this issue, or you can use the information in this book to write your own device independent programs. If fact, you will probably have to. I don't expect the [Coyote Library](#) will ever contain all the graphics programs people need or want to build.

Once the program is written and stands on its own, you can think about making it compatible with [cgWindow](#). In the case of Coyote Graphics routines, "making it compatible" has meant adding *Window* and/or *AddCmd* keywords, as needed. I want to explain the difference between them.

Understanding the Window and AddCmd Keywords

When you are working with normal graphics windows you issue a command (say, a normal *Plot* command) and it appears in the window. You might add information to the graphics window by issuing an *OPlot* or *PlotS* command. Eventually, you obtain the result you want. Then you decide you want to look at something else, and maybe you issue a *Surface* command. The plot that was previously in the graphics window disappears, and the surface plot appears.

This is *exactly* how I want to work with resizeable graphics windows. I want to issue a command and have it appear in the window. Then I want to issue more commands and have them appear. Eventually, I tire of the plot and I want to see a surface, and I want *that* to appear in the same window.

To make this happen for Coyote Graphics routines I have decided that the *Window* keyword makes the command work as a normal *Plot*, *Surface*, or *Contour* command would work. That is to say, it will act to replace whatever is currently in the resizeable graphics window with the new command. Let's use this plot and surface example to show you what I mean.

First, I would like to see a line plot.

```
IDL> data = cgDemoData(1)  
IDL> cgPlot, data, /Window
```

You see the result in Figure 11.

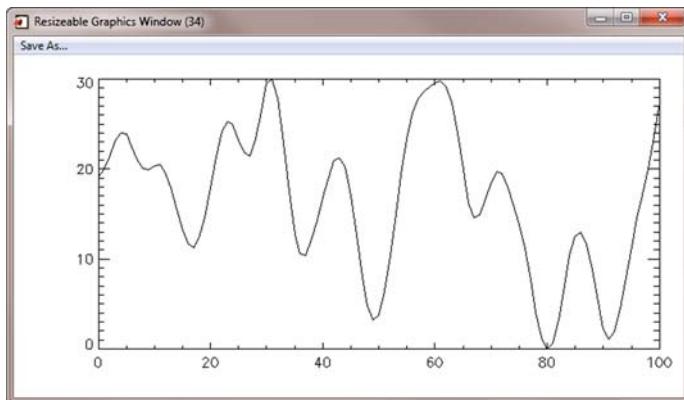


Figure 11: Coyote Graphics routines act like normal IDL graphics routines when used with resizable graphics windows.

Just as a normal *Plot* command will either create a window for itself or use the one that is the current graphics window on the display, this *cgPlot* command will either create a new *cgWindow* to display itself in, or it will appear in whatever *cgWindow* is the “current” window and displace whatever commands were previously in the window with this one.

The algorithm *cgPlot* uses to determine what to do is simple. First, it calls *cgQuery* to determine if there are any *cgWindow* applications currently on the display. If there are none, it creates one.

```
IDL> void = cgQuery(Count=count)  
IDL> IF count EQ 0 THEN cgWindow
```

It then loads itself into the current *cgWindow*. This might well be the window it just created because, remember, when a *cgWindow* is created, it becomes the current *cgWindow*, just as normal graphics windows become the current graphics window when created with the *Window* command.

The “loading” is done by calling *cgWindow* with the *ReplaceCmd* keyword set. Recall (page 423) that if you use the *ReplaceCmd* keyword without also specifying a specific command to replace, all the commands currently

in the window's command list are replaced. Here is the command that is used internally in the `cgPlot` code, but you can type it at the IDL command line just as well.

```
IDL> cgWindow, 'cgPlot', data, /ReplaceCmd
```

Suppose now we want to add symbols to the plot in a different color. We can do this with the `cgPlotS` command. In this case, we want to add this command to the previous command that is already in the graphics window, so we set the `AddCmd` keyword instead of the `Window` keyword.

```
IDL> cgPlotS, data, PSym=4, Color='red', /AddCmd
```

We add a line across the plot in the same way.

```
IDL> cgPlotS, !X.CRange, [15, 15], Color='grn5', $  
Thick=2, LineStyle=2, /AddCmd
```

You see the result in Figure 12.

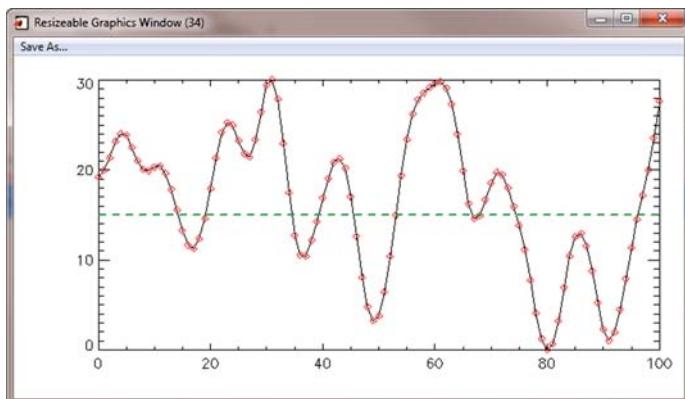


Figure 12: Commands are added to the current resizable graphics window by setting the `AddCmd` keyword to the Coyote Graphics command.

Internally, the `cgPlotS` commands simply call `cgWindow`, but instead of using the `ReplaceCmd` keyword in the call, it uses the `AddCmd` keyword. Don't type this command, but this is what is done inside the `cgPlotS` code.

```
IDL> cgWindow, 'cgPlotS', data, PSym=4, $  
Color='red', /AddCmd
```

Now, if you want to see a surface plot in the same window, you simply call `cgSurf` and set the `Window` keyword again.

```
IDL> surfdata = cgDemoData(2)
IDL> cgSurf, surfdata, /Window
```

A surface plot now appears in the window. You see the result in Figure 13.

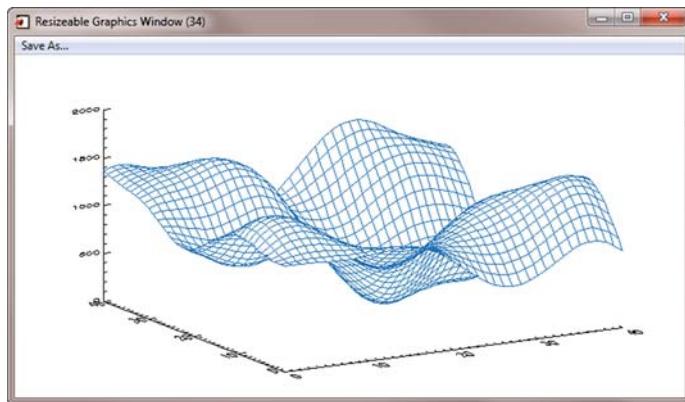


Figure 13: Whenever the `Window` keyword is set, the Coyote Graphic appears in the current `cgWindow` application.

Naturally, if you want to see both the surface plot and the line plot at the same time, you can display them in two different resizable graphics windows. Simply create another `cgWindow` and display the line plot in that.

```
IDL> cgWindow
IDL> cgPlot, data, /Window
IDL> cgPlots, data, PSym=4, Color='red', /AddCmd
IDL> cgPlots, !X.CRange, [15, 15], Color='grn5', $
    Thick=2, LineStyle=2, /AddCmd
```

You can use `cgSet` or `cgShow` to flip back and forth between them (see page 427), just as you would normally with the `WSet` or `WShow` commands.

Note: Typing `Window` and `AddCmd` keywords is not as burdensome as it would appear. Remember that keywords can be shortened to whatever length makes them unique keyword parameters. For most of the Coyote Graphic routines, `Window` can be shortened to "W" or "Win" and `AddCmd` can always be shortened to "Add."

Multiple Plot Layouts with Coyote Commands

Many of the Coyote Graphics commands are defined with *Layout* keys that allow you to position the graphic in a display window. The keyword is meant to work exactly like the *Layout* keyword available in IDL 8 function graphics commands. The layout is specified as a three-element array, $[n\text{columns}, n\text{rows}, \text{location}]$, where the first two elements indicate the number of columns and the number or rows in an imaginary grid in the window, and the third element identifies the location in the grid. Grid locations start in the upper-left hand corner of the display window and proceed across the window and down in what is conventionally called “row order” in IDL. Grid locations start with the number 1 and continue until the number of grid cells in the window.

You see in Figure 14 a three-column by two-row grid, with the grid location numbers shown inside each grid cell.

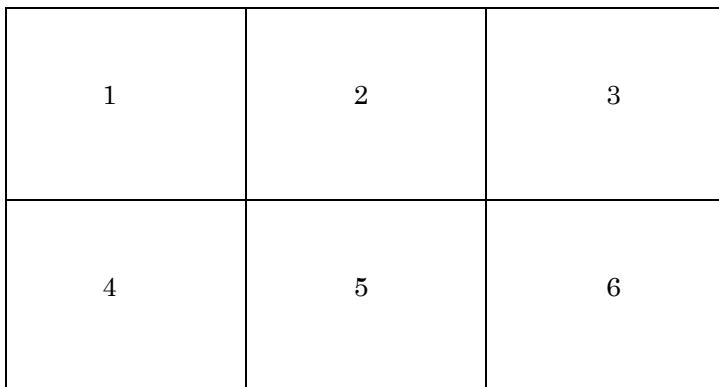


Figure 14: Grid layout locations for a three-column by two-row grid.

If you want to display a line plot in location 5, for example, you can type commands like this.

```
IDL> cgDisplay, 700, 450  
IDL> cgPlot, cgDemoData(1), Layout=[3,2,5]
```

You see the result in Figure 15.

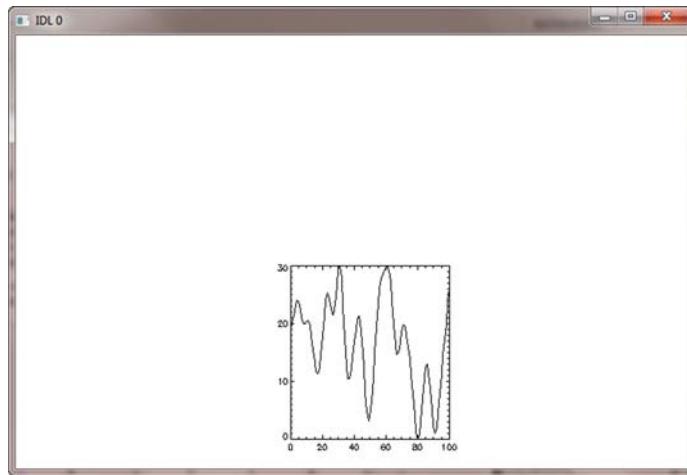


Figure 15: A line plot positioned with the `Layout` keyword set to `[3,2,5]`.

Note: If you choose to open a normal IDL graphics window to display these plots, rather than using `cgDisplay`, you will notice that the plots are created with white annotations on a black background. This is a consequence of not being able to “erase” the display. If you want to use a normal IDL graphics window, you may want to erase it with a background color before you draw into it. You can use the `cgErase` command for this purpose.

Other commands can be placed in other window positions.

```
IDL> cgLoadCT, 4, /Brewer, /Reverse
IDL> cgContour, cgDemoData(2), NLevels=10, /Fill, $
      Layout=[3,2,1]
IDL> cgContour, cgDemoData(2), NLevels=10, $
      Color='black', /Overplot, Layout=[3,2,1], $
      C_Charsize=0.65
IDL> cgHistoplot, cgDemoData(2), /Fill, Layout=[3,2,3]
```

You see the result in Figure 16.

By necessity, when you are laying out graphics with the `Layout` keyword the current window content cannot be erased. (Otherwise, you could never get two commands to show up in the same window.) In practice, this usually means the routines you use will have a `NoErase` keyword defined

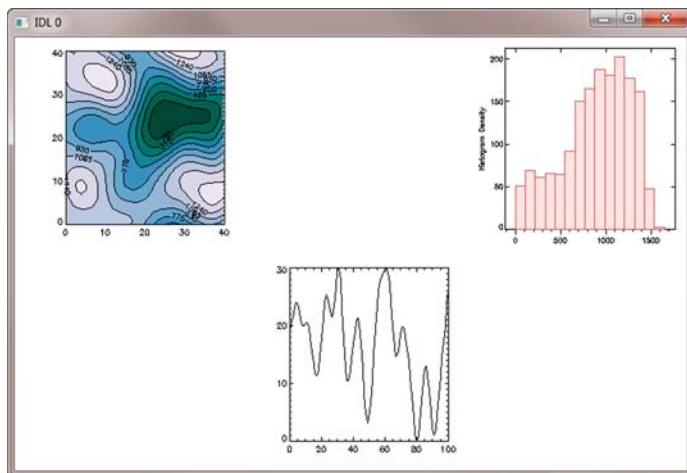


Figure 16: Most Coyote Graphic commands have *Layout* keywords to allow graphic positioning. Graphics can be positioned in any order or in any location.

for them, and that this keyword will be automatically set by the *Layout* keyword, so that you don't have to worry about this detail yourself.

As a consequence of this limitation, it is not possible to automatically “erase” a plot in a particular layout position and replace it with a second plot in the same position. The second plot will simply draw itself on top of the first plot. To work around this limitation, the `cgErase` command has also been provided with a *Layout* keyword. If this keyword is used, the layout position is “erased” by filling it with a color you can specify in `cgErase`. The default color is “white.”

For example, if you want to erase the line plot in the current window and substitute an image plot, you can type these commands.

```
IDL> cgErase, Layout=[3,2,5]
IDL> cgImage, cgDemoData(7), /Axes, MultiMargin=4, $
      Layout=[3,2,5]
```

You see the result in Figure 17.

These commands can just as easily be displayed in a resizeable graphics window by creating a window and adding the commands to it, like this.

```
IDL> cgLoadCT, 4, /Brewer, /Reverse
IDL> cgWindow
```

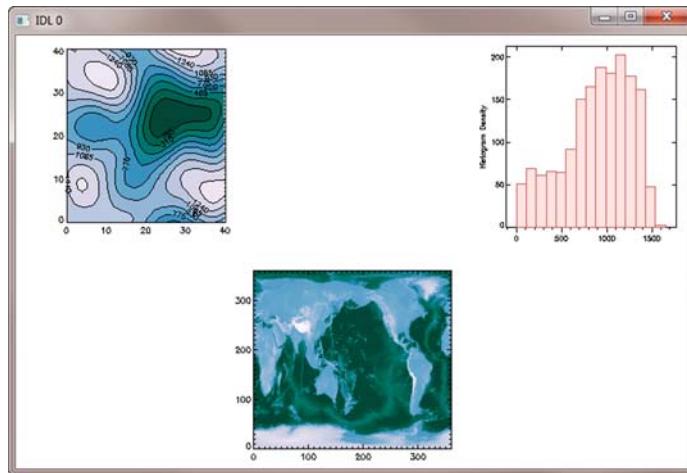


Figure 17: A graphic layout can be erased with `cgErase` before another graphic is drawn in its place.

```
IDL> cgContour, cgDemoData(2), NLevels=10, /Fill, $  
      Layout=[3,2,1], /AddCmd  
IDL> cgContour, cgDemoData(2), NLevels=10, $  
      Color='black', /Overplot, Layout=[3,2,1], $  
      C_Charsize=0.65, /AddCmd  
IDL> cgHistoplot, cgDemoData(2), /Fill, $  
      Layout=[3,2,3], /AddCmd  
IDL> cgImage, cgDemoData(7), /Axes, MultiMargin=4, $  
      Layout=[3,2,5], /AddCmd
```

You see the result in Figure 18.

Controlling Window Output Programmatically

It is possible to programmatically control `cgWindow` to produce PostScript or raster file output. This is done by using the appropriate keywords with `cgControl`. For example, suppose you had displayed a line plot in a `cgWindow` application, and you now want to create a PostScript file from the window from another program. (You might have created the `cgWindow` application from a widget program, for example, and given it the window title “Line Plot Display.”.)

The first step would be to make sure this `cgWindow` is the current window by selecting it. In this case, you could use the window title.

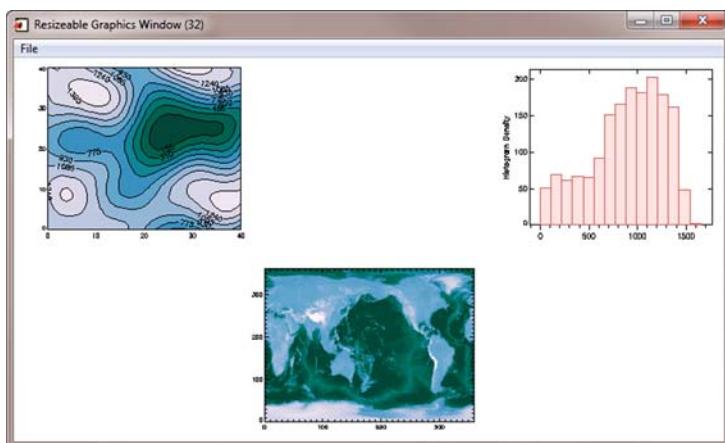


Figure 18: Layout keywords work as well in resizable graphics windows as they do in normal graphics windows.

```
IDL> cgSet, 'Line Plot Display', /Title
```

Then, you simply use the *Create_PS* keyword to [cgControl](#) to specify the name of a PostScript file. The file is created automatically.

```
IDL> cgControl, Create PS='line plot.ps'
```

You use similar keywords to create raster file output. The only difference is that the current setting of the *Raster_IM* property for [cgWindow](#) is used to choose whether the raster file is created with IDL or it is created by ImageMagick via a PostScript file intermediate. It is possible to set the *Raster_IM* property from [cgControl](#) at the same time you create the raster file.

For example, here is how you would create a PNG file from this window using ImageMagick.

```
IDL> cgSet, 'Line Plot Display', /Title  
IDL> cgControl, Create_PNG='line_plot.png', Raster_IM=1
```

If you wanted to allow IDL to create the PNG file, you would set the *Raster_IM* keyword to 0, like this.

```
IDL> cgSet, 'Line Plot Display', /Title  
IDL> cgControl, Create PNG='line plot.png', Raster IM=0
```

Similar keywords exist for creating BMP, GIF, JPEG, and TIFF raster files.

Index



Index

Symbols

!D.Flags system variable 378
 !D.Name system variable 335
 !D.X_Size system variable 24
 !D.Y_Size system variable 24
 !Order system variable 227
 !P.Background system variable 68
 !P.Charsize system variable 77, 413
 !P.Color system variable 68
 !P.Font system variable 197
 !P.Multi system variable 94
 !Values.D_NaN system variable 114
 !Values.F_NaN system variable 114
 !X.CRange system variable 107
 !X.OMargin system variable 96
 !X.S system variable 89
 !Y.CRange system variable 107
 !Y.OMargin system variable 96
 !Y.S system variable 89
 < operator 221
 > operator 221

A

Adapt_Hist_Equal command 270
 adaptive histogram equalization 270
 AddToPath command 6
 area of ROI 320
 arguments. *See* parameters
 ASinhScl command 277
 Aspect command 350
 aspect ratio
 of images 243
 of line plots 118

Astronomy Library 215

axes
 adding to plot 105
 auto-scaling 99, 124
 calculated range of 107
 calendar dates as labels 83
 carriage return in title of 92
 formatting with functions 81
 irregular spacing of labels 83
 linestyles of 84
 logarithmic 103
 reversing 102
 setting range of 99
 style of 102
 suppressing labels of 81
 turning off 102
 Axis command 105, 196
 axis labels
 defining own 79
 formatting 80

B

big endian 216
 bi-level image 312
 Binary command 379
 binary output 379
 blob analysis 316, 324
 blob coloring 317
 Blob_Analyzer command 324
 Brewer color tables 147
 Brewer color table file 55
 loading 56
 Butterworth low-pass filter 297
 byte swapping 216

ByteOrder command 216
BytSel command 220

C

calendar dates
 formatting with Label_Date 83
 labeling axes with 83
capitalizing commands 7
carriage return in plot labels 92
CDF file 214
center of mass of ROI 320
centroid of ROI 320
cgBlendImage command 261
cgColor command 41
cgColorbar command 139
cgColorFill command 106
cgContour command 162
cgDCBar command 161
cgDefCharSize command 79, 414
cgDemoData command 7
cgDisplay command 27, 160, 350
 in PostScript 27
 in Z graphics buffer 27
cgErase command 29, 247
cgHistoplot command 111
cgImage command 51
cgImageInfo command 249
cgLoadCT command 56
cgPlot command 116
cgPlotS command 106, 118
cgSnapshot command 260, 391, 409
cgSurf command 198, 203
cgSurface command 175
cgText command 106
cgWindow
 contour plot in 162
 controlling programmatically 449
 image plot in 265
 line plot in 117
 See resizable graphics windows
 surface plots in 201
cgWindow command 30, 120, 265
cgWindow_GetDefs command 437
cgWindow_SetDefs command 435
chain-code algorithm 320
character size
 defining 79
character size in Z-buffer 396
CIndex command 44
ClipSel command 277

code comments 7

color

 DirectColor visual class 34
 PseudoColor visual class 35
 TrueColor visual class 35
 warning for UNIX users 34
 working with in IDL 33

color bar

 discrete colors in 161
 in contour plots 139
 loading Brewer colors 56

color decomposition 37

color models 36

 decomposed color model 36
 determining current state of 53, 53
 diagram of decomposed model 38
 diagram of indexed color model 39
 indexed color model 36
 problems with mismatch 40
 setting 52

color palettes

 in cgWindow 431
 in images 248
 obtaining with cgLoadCT 165

color systems

 HLS 64

color systems used in IDL 63

color tables

 adding to color table file 62
 clipping ends of 135
 creating your own 56
 exchanging 62
 loading 53
 loading from file 55
 loading restricted colors 54
 restricting number of colors in 54
 reversing 56
 saving 61
 supplied with IDL 53
 viewing current 44

Color24 command 43

Color_Quan command 376

colors

 black on white scheme 71
 bug in UNIX 46
 cgColor command 41
 color model independent 40, 41
 color represented as color triple 36
 creating 24-bit value 43
 device independent 40, 41

- displaying images in correct 51
- drawing colors 70
- in contour plots 134
- loading Brewer color tables 56
- loading into color table 44
- names of 42
- of line plots 67
- selecting with PickColorName 42
- specifying with cgColor 41
- specifying with keywords 72
- viewing current color table 44
- where to load 70
- with TV command 47
- command continuation character 8
- commands
 - adding comments to 7
 - as functions 12
 - as procedures 12
 - capitalizing 7
 - continuing on next line 8
 - journal of 15
 - keyword parameters of 11
 - on-line help for 13
 - parts of 10
 - positional parameters of 10
 - saving 15
 - style conventions 7
- comment character 8
- comments 7
 - in IDL code 8
- Congrid command 233
- Contour command 121
- contour levels
 - labeling 124
- contour plots
 - annotating contour levels 131
 - axis auto-scaling 124
 - basic 121, 164
 - color bar with 139
 - color filled 138, 165
 - color palette with 165
 - colors in 166
 - creating 121
 - customizing 127
 - filled 138, 165
 - gridding with 149
 - hole in 163
 - hole in filled 140
 - in resizeable graphics window 167
 - keywords to 125
- labeling contour levels 124, 128, 163, 166
- linestyles in 132
- missing data in 152
- on map projections 146
- problems with Fill 145
- selecting contour levels 129
- setting color model 137
- using Cell_Fill instead of Fill 145
- using color with 134
- using decomposed color 144, 163
- using indexed color model 136
- with irregular data 147
- contrast enhancement 267
- contrast stretching
 - 2% stretch 277
 - clip colors 277
 - differential scaling 278
 - gamma log scale function 277
 - Gaussian function 277
 - interactive 278
 - inverse hyperbolic sine function 277
 - log scale function 277
- Convert_Coord command 251
- convex hull 152
- coordinate systems 108
 - converting from/to 251
- Correlate command 327
- covariance matrix 327
- Coyote Graphics System
 - cgColorbar command 139
 - cgColorFill command 106
 - cgContour command 162
 - cgControl command 425
 - cgDCBar command 161
 - cgHistoplot command 111
 - cgImage command 265
 - cgLoadCT command 56
 - cgPlot command 116
 - cgPlotS command 118
 - cgSurf command 198
 - cgSurface command 175
 - cgText command 106
 - cgWindow command 419
 - cgWindow_GetDefs command 437
 - cgWindow_SetDefs command 435
- Coyote Library
 - adding to IDL path 6
 - current version of 4
 - downloading 4

installing 4
Subversion repository 5
Coyote's Guide to IDL Programming 9
current graphics device 335
cursor
 to add text interactively 109
Cursor command 109, 251

D

data
 creating random 149
 gridding with GridData 155
 HDF5 example 157
 irregular on contour plots 147
 latent heat flux 157
 locating files used in book 7
 opening demo data files 7
data coordinate system 108
DataViewer command 215
decomposed color
 setting 52
decomposed color model
 diagram of 38
 setting 52
Delaunay triangulation 149, 155
Device command 25
Decomposed keyword 41
Get_Decomposed keyword 43, 53
Get_Screen_Size keyword 25
Get_Visual_Depth keyword 34
Get_Visual_Name keyword 34
Set_Character_Size keyword 78
Set_Font keyword 356
True_Color keyword 35
TT_Font keyword 356
device coordinate system 108
Dialog_Pickfile bug in file selector 210
Dialog_Pickfile command 210, 211
differential contrast stretching 278
digital elevation model 207
Dilate command 314
discrete color bar 161
documentation
 on-line 13
drawing colors
 Coyote's rule for 70

E

edge enhancement of images 303

EigenQL command 328
eigenvalues 327
eigenvectors 327
elevation shading 186
ellipse
 fit to ROI 321
Erase command 23, 247
erasing graphics windows 23
Erode command 314
error messages
 reading 14
Euclidean distance map 297

F

Fanning Software Consulting
 contacting 9
Fast Fourier Transform 296
FFT command 296
file name
 selecting 212
 selecting with file selection tool 210
 specifying 209, 209
Filepath command 209, 209
files
 locating on path 56
File_Which command 14
Find_Boundary command 320
Find_Resource_File command 56, 292
Fit_Ellipse command 322
FITS image files 215
FixPS command 347
frequency domain image filtering 296
frequency filtering 308
frequency windowing 308
FSC_Pickfile command 212
FSC_Resize_Image command 239
FSC_ZImage command 254
FSC_ZPlot command 101
function graphics 2, 2, 418
functions
 calling 12

G

Gaussian high-pass filter 301
GaussScl command 277
GetDecomposedState command 53, 138,
 186
GetImage command 217
Get_Screen_Size command 25

- GhostView PostScript previewer 351
GmaSel command 277
Gouraud shading 191
gradient operators 303
graphical displays
 including images in 249
graphics device
 current 335
 setting current 335
graphics display
 determining size of 25
 MaxWindowSize command 27
 multiple images in 254
 positioning image in 232
 red on black color scheme 40
 resizeable 167
 size is platform dependent 26
graphics displays
 positioning output in 91
graphics driver
 PostScript (PS) 334
 Z 387
graphics output
 red on black color scheme 40
graphics systems
 function graphics 2
 object graphics 2
graphics windows
 current size of 24
 device independent 27
 erasing 23, 29
 erasing with background color 23
 maximum size of 24
 resizeable 29
greater than operator 221
Greek characters 359
GridData command 155
gridding
 triangulation method 149
 with GridData 155
- H**
- hardcopy output
 closing the file 338
 landscape mode 338
 portrait mode 338
 selecting a file name 338
 specifying sizes of 338
HDF file 214
help
- contacting the author 9
on-line 13
resolving name conflicts 14
Help command 13
Hershey Math and Symbol font, 364
high boost image filtering 295
high-pass filter 285
high-pass image filtering 290
Hist_Equal command 270
histogram bin size
 Convert_To_Type command 273
 matching to data type 273
Histogram command 110, 111, 273, 273, 281, 319
histogram equalization 270
histogram matching 273
histogram reverse indices 283
Histogram tutorial 110
HistoMatch command 276
HLS color system 64
HSI color system 64
HSV color system 65
- I**
- IDL
 version required 4
image convolution 289
image filtering 285
 in frequency domain 296
image mask 322
image partitioning 280
image plots 209
 adding axes 236
 adding axes to 245
 color backgrounds 247
 combining with other plots 249
 image value at cursor location 250
 interacting with 249
 multiple images in 254
 zooming into 252
image power spectrum 298
image processing
 closing operator 314
 contrast enhancement 267
 convolution with kernel 289
 creating image mask 322
 dilation 314
 distance operator 314
 edge enhancement 303
 EOF analysis 326

- erosion 314
- filtering 285
- frequency filtering 308
- gradient operator 314
- high-pass filter 301
- high-pass filtering 290
- histogram equalization 270
- histogram matching 273
- hit or miss operator 315
- image partitioning 280
- labeling regions 316
- Lee filter 305
- low-pass filtering 286
- median filter 305
- morphological operators 313
- morphological properties 319
- noise reduction 304
- opening operator 314
- PCA analysis 326
- power spectrum 298
- recommended book for 267
- Roberts edge enhancement 303
- sharpening 291
- smoothing 285
- Sobel edge enhancement 303
- statistics of 319
- thinning operator 315
- thresholding 312
- top hat operator 315
- unsharp masking 295
- watershed operator 315
- window leveling 269
- image processing in IDL 267
- image segmentation 310
- image sharpening 291
- image smoothing 285
- image statistics 319
- image thresholding 312
- image variance 326
- Image_Dimensions command 252
- ImageMagick
 - constructing convert command 414
 - convert command 411
 - obtaining 405
 - purpose of 406
 - role of PS_End with 414
 - role of PS_Start with 412
- ImageMagick properties
 - configuring in windows 440
- images
 - alternative display commands 241
 - bilinear interpolation 234
 - blended 260
 - controlling multiple layout 256
 - correct color display of 51
 - creating transparent PNGs 261
 - display order 226
 - displaying 24-bit 49
 - displaying in color 228, 247
 - displaying missing data in 225
 - displaying true-color 229
 - displaying with color palettes 248
 - displaying with indexed color 222
 - in FITS files 215
 - in flat binary format files 215
 - in resizeable graphics windows 265
 - incorrect color display of 24-bit 49
 - incorrect display colors 47
 - information about 213
 - locations in 252
 - maintaining aspect ratio of 243
 - missing data in 223
 - MODIS 279
 - nearest neighbor interpolation 234
 - positioning 232, 243
 - precise resizing 238
 - preparing for display 219
 - query functions 213
 - reading from file 209
 - resizing 232, 243
 - resizing in PostScript 240
 - scaling into bytes 220
 - scaling with missing data 223
 - scientific data format 214
 - substitute TV commands 51
 - traditional display commands 218
 - true-color 229
 - upside-down 226
 - using Value_Locate in 252
 - zooming into 252
- images plots
 - erasing before display 246
- ImageSelect command 213
- Image_Statistics command 319
- ImDisp command 51
- indexed color
 - setting 52
- indexed color model
 - diagram of 39
 - setting 52

IndGen command 76
inverse hyperbolic sine function 277
irregular data with contour plots 147

J

Journal command 15
journal of IDL commands 15
JPEG file
 creating 407

K

keyword inheritance 369
keyword parameters 11
keywords
 typing names of 127

L

Label_Date function 83
Label_Region command 316
Laplacian image sharpening 292
Layout keyword for graphics 446
LeeFilt command 305
less than operator 221
line plot
 histogram plot 110
line plots
 adding additional axis to 105
 adding data to 73, 89
 adding symbols to 106
 adding text to 106
 adding titles to 77
 annotating 76
 auto-scaling of axes 99
 calculated range of axes 107
 character size of 78, 79
 color model independent 116
 color of 73
 colored backgrounds 119
 colored lines 118
 colored symbols 118
 colors of 67
 coordinate systems 108
 defining own axis labels 79
 device independent 116
 drawing lines on 106
 filling colors on 106
 formatting axes with functions 81
 formatting axis labels 80
 in resizeable window 116, 120

irregular spacing of labels 83
labeling axes of 77
line styles of 83
logarithmic axes 103
missing data in 113
multiple on page 93
overplotting 89
overplotting on 73
positioning 91
reverse axis 102
scaling parameters 89
setting aspect ratio of 118
setting axis range 99
style of axes 102
suppressing labels of 81
symbols on 83
turning axis on/off 102
using symbols with 88
zooming into 101

lines

 drawing on plots 107
linestyles 84
little endian 216
LoadCT command 53
logarithmic axes 103
LogScl command 277
low-pass filter 285

M

Make_Transparent_Image command 264
map projections
 filled contour plot on 146
MaxWindowSize command 27
Median command 305
memory management
 using Temporary function for 274
memory management practices 274
missing data
 in contour plots 152
 in images 223
 in line plots 113
ModifyCT command 63
MODIS images 279
Morph_Close command 314
Morph_Distance command 314
Morph_Gradient command 314
Morph_HitOrMiss command 315
morphological operators 313
morphological properties 319
Morph_Open command 314

Morph_Thin command 315
Morph_Tophat command 315
MultiMargin keyword 256
multiple image plots 254
multiple plots
 creating with Layout keyword 446
 laying out 446
 setting outside margins 96
multiple plots on page 93, 446

N

NaN keyword 111
NaN values 114
NASA IDL Astronomy Library 215
netCDF file 214
NLevels keyword in contour plot 129
noise reduction
 in frequency domain 307
 in images 304
normalized coordinate system 108

O

object graphics system 2
OPlot command 73, 89
output
 presentation quality 405
 outside margins 96

P

parameters
 keyword 11
 positional 10
path
 adding coyote directory to 6
 printing list of directories on 6
PComp command 328
perimeter length of ROI 320
PickColorName command 42
piece-wise image scaling 278
pixel coordinate system 108
PlotS command 106
PNG file
 creating 407
PolyFill command 106, 401
PolyShade command 402
Position keyword 91
positioning graphics output 91
PostScript device 334
 closing 336

configuring 337
configuring interactively 365
creating "window" on 338
encapsulated output 350
encapsulated preview 351
isolatin encoding 352
keywords for configuring 338
language level 352
setting 335
window location on page 345
window orientation 345
PostScript graphics device 334
PostScript output
 CMYK color 345
 color 340
 color backgrounds 344
 color images in 375
 color on grayscale printers 345
 color reversal 342
 creating 333
 font substitution table 355
 fonts used in 352
 grayscale 340
 Greek characters in 359
 images sized in 371
 Landscape upside-down 347
 metric page size 349
 multiple pages of 336
 on non-PostScript printers 384
 positioning graphics in 382
 potential color problems 343
 printing 382
 protecting commands in 378
 resizing images in 240
 rotating fonts in 357
 setting aspect ratio 349
 special characters in 363
 surface plots in 197
 true-color image in 377
 TrueType fonts in 356
 viewing 384
 window aspect ratio 348
power spectrum
 bug in Center keyword 299
 centering 299
power spectrum of image 298
presentation quality output 405
Principal Component Analysis 326
principal components 326
PrintPath command 6

probability distribution function 271

procedures

 calling 12

PSConfig command 365, 412

PS_End command 197

PS_Start command 197, 412

PSWindow command 348

PSym keyword 84

Q

QHull command 155

Query_GIF command 213

Query_JPEG command 213

Query_PNG command 213

Query_TIFF command 213

R

RandomU command 149

Read_Binary command 216

Read_GIF command 212

Read_Image command 209, 212

Read_JPEG command 212

Read_PNG command 212

Read_TIFF command 212

Rebin command 233

Reform command 327

regions of interest 310

resizeable graphics window

 contour plot in 167

 line plots in 120

 surface plot in 201

resizeable graphics windows

 configuring properties of 435

 Coyote Graphics in 441

 displaying images in 265

 ImageMagick properties in 440

 PostScript properties in 438

reverse indices 283, 318

Roberts command 303

S

salt and pepper noise in images 304

saving IDL commands 15

ScaleModis command 279

Scale_Vector command 110, 111, 146, 252

scatter plot 194

scientific data file formats 214

SetDecomposedState command 52, 138,

186

Set_Plot command 335, 392

Set_Shading command 190

shaded surface plots 184, 187

Shade_Surf command 188

Shade_Volume command 402

Sharpen command 293

Simplex Greek font table 360

Smooth command 286

Sobel command 303

speckle noise in images 305

Subversion

 obtaining 5

Subversion repository

 Coyote Library programs in 5

Surface command 171

surface plots

 adding color to 180

 adding skirt to 178

 adding texture maps to 205

 adding title to 173

 basic 171

 color model independent 198

 customizing 177

 device independent 198

 elevation shading 186

 in PostScript 197

 in resizable graphics window 201

 lego style 188

 not true 3D 174

 rotating 176, 203

 rotating font for 197

 scatter plot 194

 setting color model 185

 shaded 184, 187

 shading parameters 190

 shading with data 192

 true 3D 203

 true 3D rotation 175

symbols

 creating 86

 SymCat command 88

 table of 86

SymCat command 88, 196

T

Temporary function 274

text

 adding to plots 107

 adding to plots interactively 109

texture maps 205
 3D scatter plot 194
 tick formatting functions 81
 TIFF file
 creating 407
 TimeStamp command 16
 transparency in Z-buffer 401
 transparent PNG files 258
 transparent PNG images
 creating 261
 Transpose command 327
 Triangulate command 149
 Trigrid command 149
 true-color image 229
 TrueType fonts
 selecting 356
 TV command 218
 incorrect colors 47
 substitutes for 51
 TVLCT command 44, 53
 TVscl command 218

U

Unicode characters in PostScript 352
 Unicode special characters 364
 Unicode values represented in IDL 364
 unsharp masking 295
 UserSym command 86

V

Value_Locate command 252, 281
 version of IDL required 4

W

warping image in Z-buffer 398
 watershed command 315
 watershed operator 315
 Where command 159
 gotcha! 159
 Window command 17
 window leveling 269
 WindowImage command 270
 Write_JPEG command 407
 Write_PNG command 407
 Write_TIFF command 407

X

XColors command 56
 XLoadCT command 53

XPalette command 59
 XStretch command 312
 XStretch command. 278
 XYOutS command 106

Z

Z-graphics device
 24-bit color in 393
 character size in 396
 configuring 388
 depth buffer 390
 frame buffer 390
 purpose of 387
 reading depth buffer 395
 snapshot of 391
 transparency in 401
 warping images in 398
 Z-buffer 390



David Fanning has been working with IDL software for 25 years. He has taught thousands of scientists and engineers to write more powerful, elegant, and easy-to-maintain programs. His web page, **Coyote's Guide to IDL Programming**, is a world-wide resource of IDL information. He is the owner of an IDL training and consulting company.



Coyote is a poet and ne'er-do-well who has been associated with David Fanning for 20 years. He is responsible for the humor and perspective in David's courses, and for the remaining typos in the book. He is extremely serious about his job.

Using Familiar IDL Tools Creatively

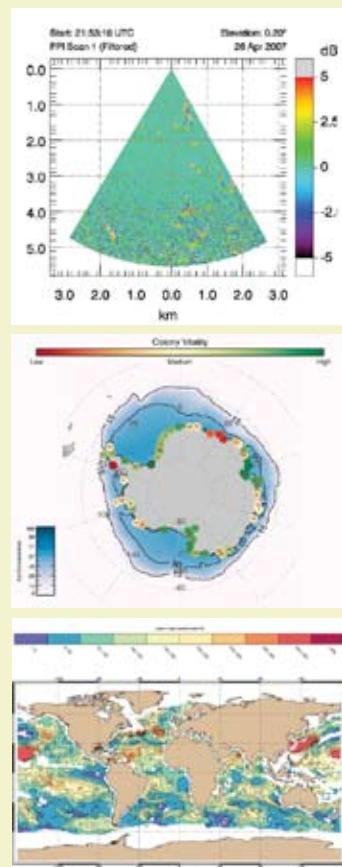
Marketing wisdom assures us the latest is the greatest. It's possible. There is something to be said, though, for the tried-and-true, the reliable, the traditional, the familiar tool that fits the hand and does its job well. IDL's traditional graphics commands are old-school. But that doesn't mean they aren't used every day by IDL programmers the world over.

This book explains how traditional graphics commands work, what their limitations are, and how to refurbish them with programs from the powerful Coyote Library to create IDL graphics programs that work in all versions of IDL in a simple, modern, and (dare I say it) non-traditional way.

You will learn how to:

- Turn your 24-bit graphics card into money well spent.
- Display graphic output in resizable graphics windows.
- Create high quality output for presentations and web pages automatically.
- Take full advantage of the **Coyote Graphics System**.

This informal, example-filled field guide is the first in a series that will introduce you to everything you need to know about IDL programming. It is written as if you were sitting in front of the instructor, taking a course in IDL graphics programming. I expect you to type the commands and do your homework. But, don't worry, Coyote is doing the grading.



ISBN-13: 978-0-9662383-5-8



90000

9 780966 238358