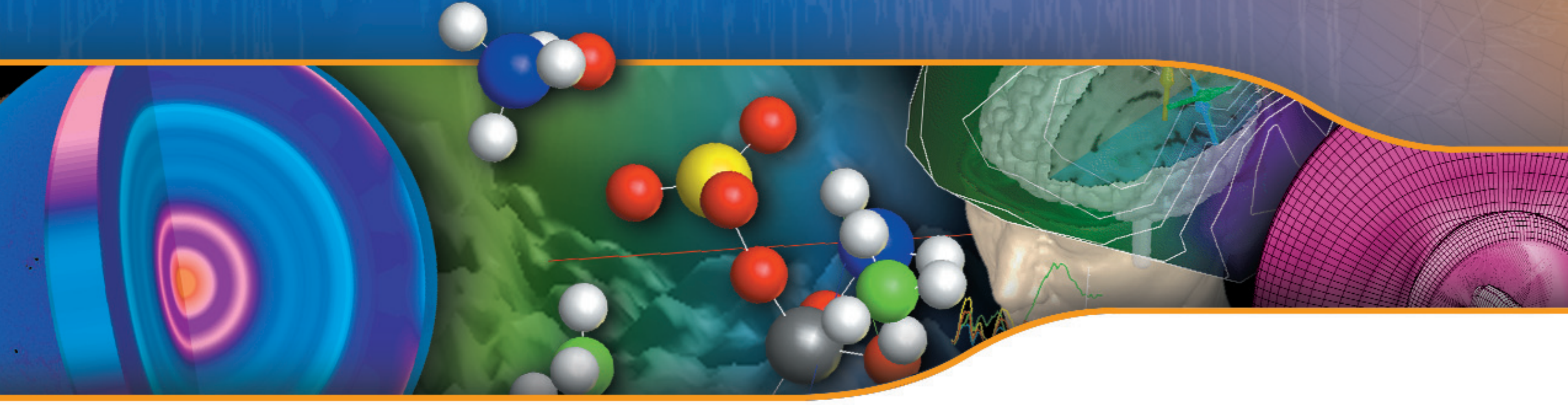
# IDL

# QuickStart



# IDL Quick Start

IDL 6.3 Version

# Table of Contents

# Chapter 1:  Introduction to IDL

## What is IDL?

IDL, the *Interactive Data Language*, is an interpreted computer language and interactive software environment that is ideal for data analysis, visualization, and cross-platform application development.  IDL is specifically designed for the visualization and analysis of large, multi-dimensional technical datasets.  IDL is the language of choice for technical professionals, offering simple syntax, array-oriented architecture, and rich library of analysis and visualization routines.  IDL combines all of the tools individuals need for any type of project -- from "quick-look", interactive analysis and display to large-scale commercial programming projects.

IDL has a rich library of built-in mathematical, statistical, signal & image processing, and analytical routines that provide proven algorithms that developers and scientists can rely on.  The language is specifically designed for visualizing large and complex datasets, from simple 2D plots to powerful OpenGL-accelerated 3D graphics.  In addition, IDL reads and writes virtually any data format, type and size, reducing the time users spend dealing with file I/O.  Thousands of technical professionals use IDL every day to rapidly develop algorithms, graphical user interfaces, and powerful visualizations with the ability to quickly crunch through large numerical problems. All of this is available in an easy-to-use, high-level, fully extensible programming environment.

## Introduction and Resources

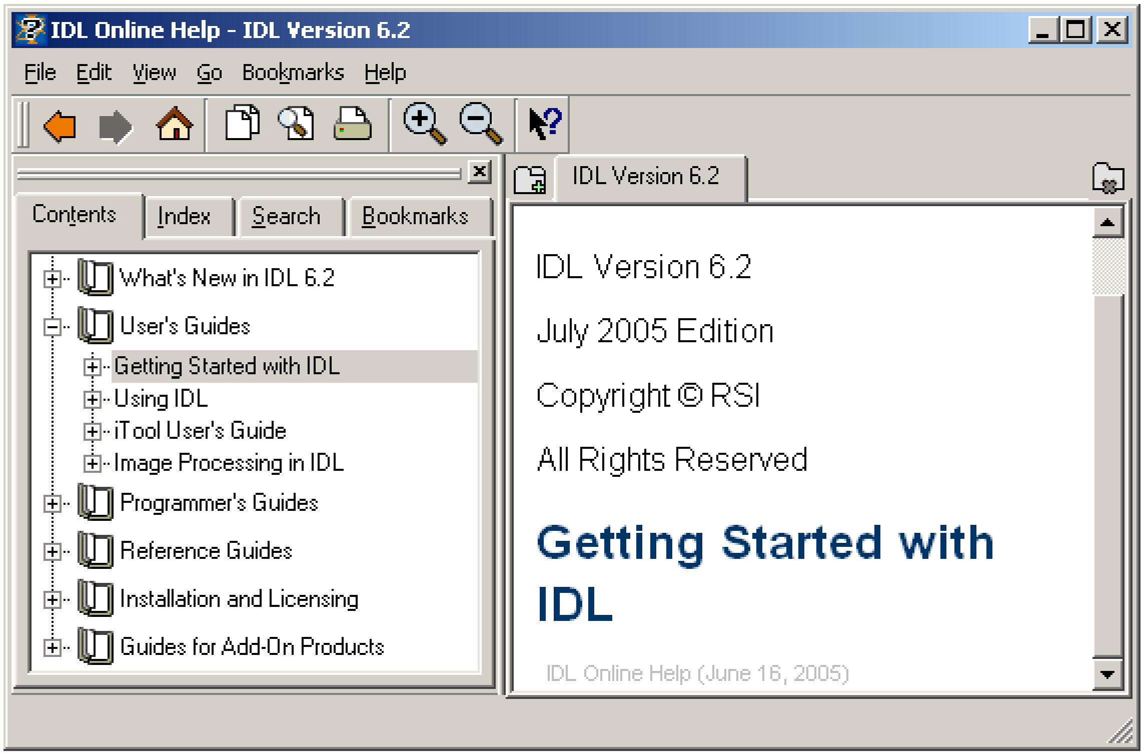
This is the manual for the *IDL Quick Start* tutorial and training course.  Users can work through the exercises in this manual either independently on their own, or interactively with the aid of an instructor from ITT as a 1-day intensive training course.  This manual covers a specific set of topics that represent a small subset of the overall functionality that IDL provides.  Our goal is to help you learn how to use IDL and we want you to be successful in your work.

ITT is committed to providing the highest quality level of customer support and assistance to our customers, and we provide a variety of support services to help users be as productive as possible.  In addition to the *IDL Quick Start* tutorial, the ITT Global Services Group offers a more extensive training program that consists of a series of courses taught with hands-on instruction by a team of skilled professionals.  ITT provides training classes across the country on a regular basis at a number of locations, and can also do custom on-site instruction when needed. Furthermore, the ITT Global Services Group also provides consulting, and we specialize in building complete solutions for unique data analysis and visualization applications.  The ITT team of responsive, creative consultants is ready to listen and deliver what the users want -- on time and within budget.  They can help define requirements and lead the way throughout the development cycle -- from prototyping to final installation -- or jump in mid-stream and provide expert assistance. Furthermore, the ITT Support Services department is available to answer questions between 8:00 AM and 5:00 PM, Mountain Standard Time. Users receive prompt, authoritative, and convenient replies to questions from qualified specialists who are trained in the sciences and engineering.

There are a number of resources that are available to individuals working with the IDL software package, and several of these resources can be found on the ITT website (www.ittvis.com). Some of these resources include :

| | |
|---|---|
| • Online Store : | http://www.ittvis.com/store/ |
| • Product Downloads : | http://www.ittvis.com/download/ |
| • Web Seminars : | http://www.ittvis.com/webinar/ |
| • Product Applications : | http://www.ittvis.com/appprofile/ |
| • Technical Support : | http://www.ittvis.com/services/ |
| • Training Programs : | http://www.ittvis.com/training/ |
| • Global Services : | http://www.ittvis.com/gsg/ |
| • User Forum : | http://www.ittvis.com/forum/ |
| • User-Contributed Library : | http://www.ittvis.com/codebank/ |

Finally, the IDL software package also includes a complete set of documentation manuals in either PDF format or the built-in Online Help system, which can be accessed by selecting "*Help > Contents…*" from the IDL development environment window. Another very good resource for individuals just getting started with the IDL software package is the *Getting Started with IDL* documentation manual [Fig. 1-1].



IDL Version 6.2

July 2005 Edition

Copyright © RSI

All Rights Reserved

Getting Started with IDL

IDL Online Help (June 16, 2005)

***Figure 1-1: The IDL Online Help system***

## The IDL Development Environment

IDL has a convenient interface called the IDL Development Environment (IDLDE) that includes built-in data input, visualization, analysis, and program editing & debugging tools.  To start the IDLDE perform one of the following :

- **Windows:** select "*Start > All Programs > RSI IDL #.# > IDL*"
- **UNIX et al.:** execute "*idlde*" at the shell prompt
- **Mac OS X:** double-click "*/Applications/rsi/idl_#.#/idlde*"

**Note:** If executing "*idlde*" doesn't work on UNIX and Linux, the IDL setup file (that defines environment variables specific to IDL) needs to be sourced.  This setup file is found in the "*bin/*" subdirectory of the IDL installation.  For example, with an installation of IDL in the default "*/usr/local/rsi/*" location the appropriate commands would be :

```
        Bourne or Korn shell:  $ . /usr/local/rsi/idl/bin/idl_setup.ksh
        C or TC shell:         % source /usr/local/rsi/idl/bin/idl_setup
```

It is a good idea to setup user accounts to source these files upon login in the resource configuration file in their home directory (e.g. .cshrc, .login, .kshrc, etc.).

After launching IDL the main IDLDE window appears [Fig. 1-2].  The eight sections of this window are described below.  On Windows, most sections of the IDLDE can be undocked (click and drag) and resized by moving the separator.  In addition, each of these individual sections, except the Document Panel and Main Menu Bar, can be turned on or off under the *Window* menu.

**The Main Menu Bar**

The Main Menu items, located at the top of the IDLDE, are used to control the configuration of IDL, develop IDL programs, and layout projects.  The main menu is used primarily by the software programmer to design applications in IDL.

**The Toolbar**

Toolbar buttons provide a shortcut to execute the most common tasks found in the main menu.

**The Project Window**

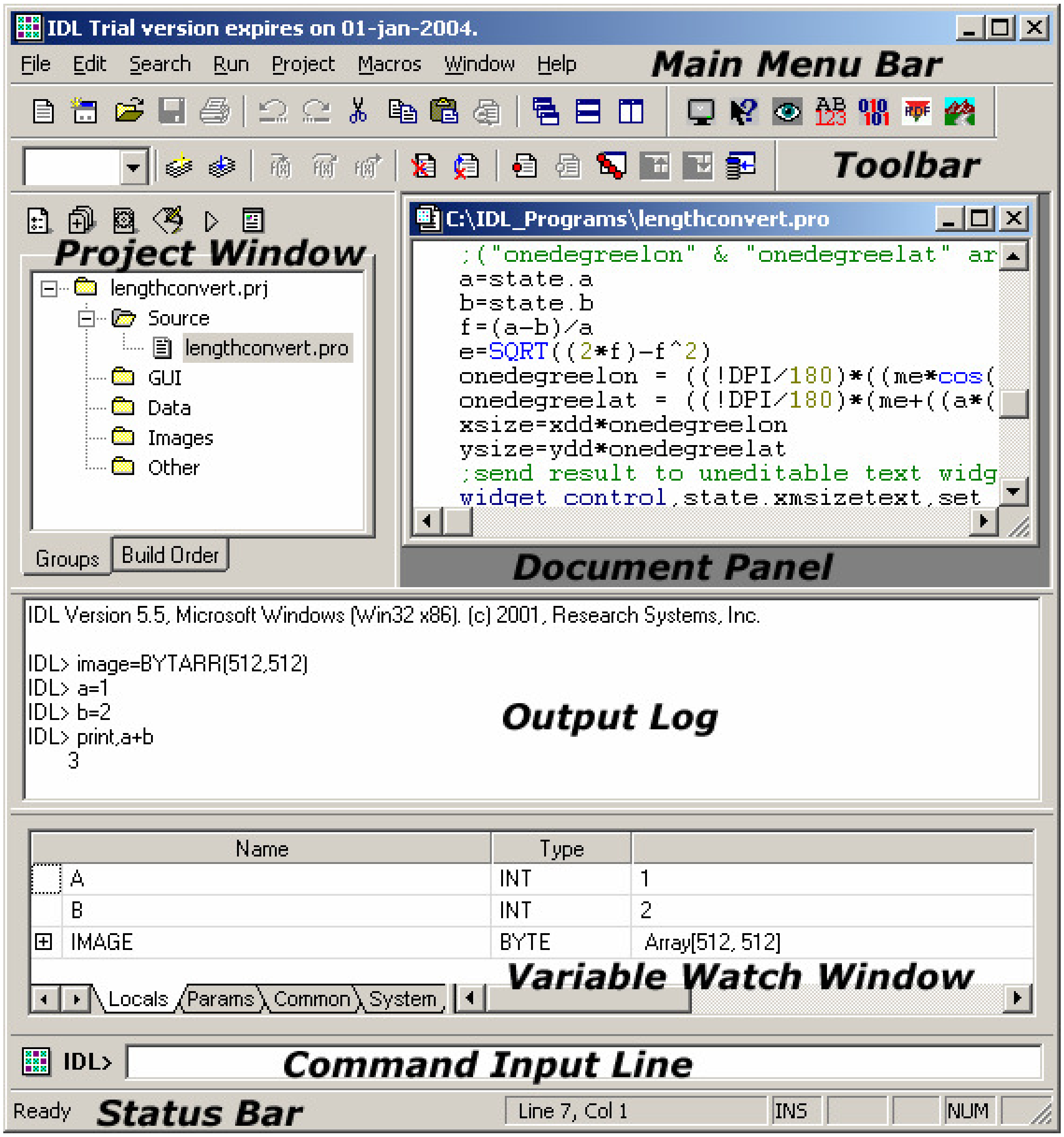
The Project Window allows users to manage, compile, run, and create distributions of all the files needed to develop an IDL application. All of the application files can be organized for ease of access, and to be easier to export to other developers, colleagues, or users.

**The Document Panel**

The Document Panel is where the IDL ASCII text editor and GUI builder windows are displayed, which allow the user to edit the IDL source code of their program modules and design GUIs (Windows only).

**The Output Log**

The standard output from the IDL interpreter is displayed in the Output Log window.



```
;("onedegreelon" & "onedegreelat" ar
a=state.a
b=state.b
f=(a-b)/a
e=SQRT((2*f)-f^2)
onedegreelon = ((!DPI/180)*((me*cos(
onedegreelat = ((!DPI/180)*(me+((a*(
xsize=xdd*onedegreelon
ysize=ydd*onedegreelat
;send result to uneditable text widg
widget_control,state.xmsizetext,set
```

```
IDL Version 5.5, Microsoft Windows (Win32 x86). (c) 2001, Research Systems, Inc.

IDL> image=BYTARR(512,512)
IDL> a=1
IDL> b=2
IDL> print,a+b
     3
```

| Name | Type | |
|---|---|---|
| A | INT | 1 |
| B | INT | 2 |
| ⊞ IMAGE | BYTE | Array[512, 512] |

***Figure 1-2: The IDL Development Environment***

**The Variable Watch Window**

Displays variables that are "in scope". By selecting the + sign next to array variable names the values are viewable. The tabs also allow viewing of parameters, common blocks, and system variables.

**The Command Input Line**

The Command Input Line is an IDL *prompt* where the user can enter IDL commands, much like a DOS or UNIX shell prompt. IDL statements can be typed directly into the command input line and executed by hitting Enter on the keyboard. To use the Command Input Line first make sure that input focus is at the IDL command prompt by clicking the mouse in the box right of the IDL> prompt. A blinking cursor appears if the input focus is at the command line.

**The Status Bar**

When the mouse pointer is positioned over a Toolbar button or an item is selected from the Main Menu in the IDLDE, the Status Bar displays a brief description.

## Installing the Tutorial Files

In addition to this manual, a package of files is provided that includes custom IDL programs and a number of data files that are used throughout the Quick Start tutorial. These files are located within the folder named "*IDL_QS_Files*". In order to complete the steps performed during the Quick Start exercises, this folder must be copied to the main IDL installation folder on the computer's local harddrive and the user must be given open write permissions to this location.

At this time, please perform the appropriate steps for your operating system to copy the entire "*IDL_QS_Files*" folder to the appropriate location on the computer. The default installation location for IDL that this folder should be copied to is :

- **Windows:** *C:\RSI\IDL##\*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/*
- **Mac OS X:** */Applications/rsi/idl_#.#/*

Once this is accomplished, a couple of settings in the Preferences must be modified so that IDL knows how to locate these files. IDL uses the concept of a "path" that stores the directories IDL will search for libraries, include files, custom user-written programs, and executive commands (much like UNIX and DOS). When a command is executed within IDL that is not part of IDL's built-in library, IDL will search the folders designated by this "path" (in order of declaration) for either a *.pro (IDL ASCII source code) or *.sav (IDL runtime binary) file with the same filename prefix as the routine being executed. Please use the following steps to make the necessary modifications to the Preferences for IDL :

1. select "*File > Preferences…"* from the IDLDE main menu
2. move to the "Startup" tab
3. change the "Working Directory:" setting by pressing the "*Browse…"* button
4. navigate to and select the "*IDL_QS_Files*" folder on the harddrive (you may need to double-click on the folder in order to successfully select)
5. move to the "Path" tab
6. press the "*Insert"* button
7. once again, navigate to and select the "*IDL_QS_Files*" folder on the harddrive
8. check the box immediately to the left of the folder listing, which tells IDL to search subdirectories recursively
9. press "*Apply"* in the *Preferences* dialog, followed by "*OK"*

## The Power of IDL

IDL is basically a rich library of data analysis and visualization routines. The software provides a wide variety of data input/output, processing, and graphics generation functionality. IDL can be useful in just about any situation when data needs to be analyzed.

Individuals who are new to IDL or have not been exposed to the software before may be interested in seeing some demonstrations of how the software can be utilized. One of the best ways to get a feel for the capabilities of IDL is to run the built-in demo system, which can be accessed by pressing the button on the IDLDE toolbar or executing the command *DEMO* at the IDL> command prompt :

```
IDL> demo
```

**Note:** The IDL source code for these demonstration programs can be found in the "*examples/demo/demosrc/*" subfolder of the IDL installation.

The demonstration programs are launched by a category on the left-hand side, then double-clicking on the link in the lower-right hand corner.

## A Brief Tour of the IDL Language

Although IDL has a number of interactive tools for data input, analysis, and visualization, it is (in essence) a programming language. The primary mechanism for using IDL is by executing **statements**, either at the command line or within programs, which control the actions of IDL. IDL statements are case insensitive (and most are also space insensitive to an extent). For instance, each of the following are acceptable IDL statements (and perform the same exact operation) :

```
1. IDL> PRINT,2*4
          8
2. IDL> print, 2 * 4
          8
3. IDL> Print, 2*4
          8
```

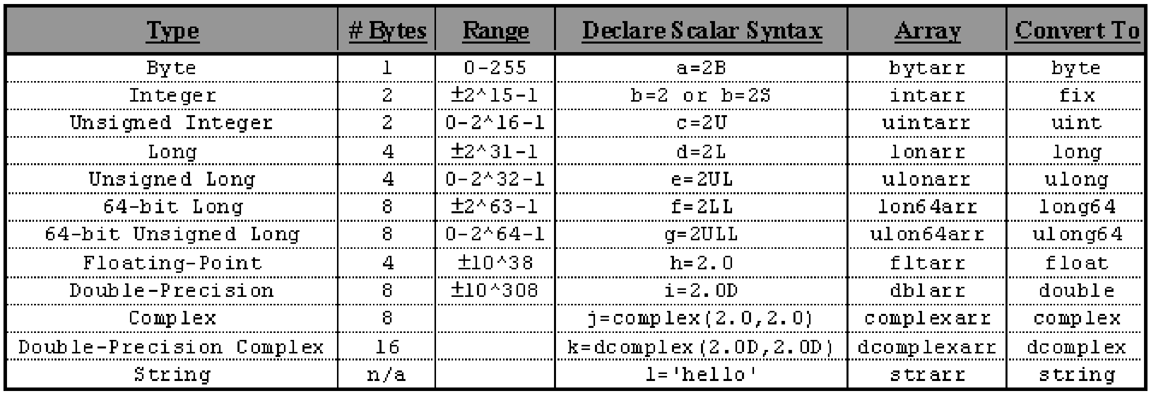
The *PRINT* routine prints the value of its argument into the IDL output log. Notice that the argument is evaluated before it is printed, resulting in the number (8) being output to the log window.

**Note:** IDL saves previously entered statements in a buffer, and these statements can be recalled to the command line with the UP-DOWN arrow keys on most keyboards while cursor focus is at the command line. The number of lines saved in the recall buffer can be changed in the Preferences for IDL.

In an IDL session, data (i.e. numbers and strings) is stored in what's known as **variables**. There are 3 basic types of IDL variables :

- **Scalar** (a single value)
- **Array** (from 1 to 8 dimensions – a **vector** is an array with only 1 dimension)
- **Structure** (an aggregate of various data types and other variables)

Any given variable in IDL also has a specific **data type**. There are 12 basic atomic data types in IDL (seven different types of integers, two floating-point types, two complex types, and a string) [Fig. 1-3]. The data type assigned to a variable is determined either by the syntax used when creating the variable, or as a result of some operation that changes the type of the variable (i.e. the IDL language is **dynamically typed**).



| Type | # Bytes | Range | Declare Scalar Syntax | Array | Convert To |
|---|---|---|---|---|---|
| Byte | 1 | 0-255 | a=2B | bytarr | byte |
| Integer | 2 | ±2^15-1 | b=2 or b=2S | intarr | fix |
| Unsigned Integer | 2 | 0-2^16-1 | c=2U | uintarr | uint |
| Long | 4 | ±2^31-1 | d=2L | lonarr | long |
| Unsigned Long | 4 | 0-2^32-1 | e=2UL | ulonarr | ulong |
| 64-bit Long | 8 | ±2^63-1 | f=2LL | lon64arr | long64 |
| 64-bit Unsigned Long | 8 | 0-2^64-1 | g=2ULL | ulon64arr | ulong64 |
| Floating-Point | 4 | ±10^38 | h=2.0 | fltarr | float |
| Double-Precision | 8 | ±10^308 | i=2.0D | dblarr | double |
| Complex | 8 |  | j=complex(2.0,2.0) | complexarr | complex |
| Double-Precision Complex | 16 |  | k=dcomplex(2.0D,2.0D) | dcomplexarr | dcomplex |
| String | n/a |  | l='hello' | strarr | string |

***Figure 1-3: IDL data types***

Variables do not have to be declared in IDL, and the data type of a variable can be determined by its usage. If a new variable is set equal to the sum of two integers, then the new variable will also be an integer. For example, start by declaring a scalar variable named "*a*" and set it equal to a value of (2) :

```
4. IDL> a = 2
```

By default, if a variable is set to a whole number it is assigned the 16-bit signed integer data type, as illustrated in the *Declare Scalar Syntax* column of Fig. 1-3. Next, declare a second scalar variable called "*b*" and set it equal to a floating-point value of (5) :

```
5. IDL> b = 5.0
```

IDL provides the ability to perform an arithmetic operation on these two variables and store the result in a new variable called "*c*" without having to first declare "*c*" :

```
6. IDL> c = a + b
```

The *HELP* routine is used to get information about the IDL session. In this case, use the *HELP* routine to obtain information on the three variables declared thus far :

```
7. IDL> help, a, b, c
   A               INT       =        2
   B               FLOAT     =       5.00000
   C               FLOAT     =       7.00000
```

Notice that when variables of different data types are combined in a single expression, the result has the data type that yields the highest precision.

So far all of the work performed has been with scalar variables, but the real power of IDL is in the fact that it's an **array-oriented** language. For example, declare a variable called "*array*" as an integer matrix with two dimensions of size 5 columns and 5 rows. Use the *INDGEN* function to set the value for each element of the array to its one-dimensional subscript (notice that IDL is a **row-major** language) :

```
8.  IDL> array = indgen (5, 5)
9.  IDL> help, array
    ARRAY           INT       = Array[5, 5]
10. IDL> print, array
           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
```

Since IDL is an array-oriented language, any operation that is applied to an array variable will automatically affect every element of the array without having to utilize FOR loops. For example, every element within the variable “*array*” can be multiplied by the scalar value (2) that is currently stored in the variable “*a*” with one simple statement :

```
11. IDL> array = array * a
12. IDL> print, array
           0       2       4       6       8
          10      12      14      16      18
          20      22      24      26      28
          30      32      34      36      38
          40      42      44      46      48
```

The ability to perform operations on only specific elements of an array is another power of the IDL language and is called **subscripting**. The square bracket characters “[“ and “]” are used to perform subscripting in IDL. Since IDL is row-major, the appropriate way to subscript a 2-dimensional array is with the syntax [*column#*, *row#*]. It is also important to keep in mind that indexes for subscripting start at 0 instead of 1. For example, to print only the value found in the element in the 2nd column and 4th row of the variable “*array*” execute the statement :

```
13. IDL> print, array [1,3]
          32
```

So far we have executed two different routines from the IDL language library, *PRINT* and *HELP*. There are 3 basic types of routines that can be used by executing statements within the IDL language :

- **Procedures**
- **Functions**
- **Executive Commands**

A **procedure** is a routine that simply performs a well-defined task. In contrast, a **function** is a routine that performs a well-defined task and also returns a value to the specified variable once it is finished executing. **Executive commands** are used to control the execution of IDL programs. The statements executed to run the 3 different types of IDL routines differ in their calling sequence syntax :

*Procedure:* *IDL>* `PROCEDURE, argument`
*Function:* *IDL>* `result = FUNCTION (argument)`
*Executive:* *IDL>* `.EXECUTIVE_COMMAND –flags argument`

For example, execute the following procedure, function, and executive command statements at the IDL> command prompt (Note: "0" is the number zero) :

14. *Procedure:* IDL> `CALENDAR, 1976`
15. *Function:* IDL> `time = SYSTIME (0)`
16. *Executive:* IDL> `.COMPILE arrow`

In the example statements above, the *CALENDAR* procedure displays a simple calendar for the year 1976 in an IDL graphics window, the *SYSTIME* function returns the current time as a date/time string into the variable "*time*", and the *.COMPILE* executive command compiles the *ARROW* routine from the IDL distribution library and opens its source code file into the IDLDE document panel. There are hundreds of routines built into IDL, and most fall within one of these three categories.

In order to view the value returned by the *SYSTIME* function into the variable "*time*" the *HELP* procedure can be utilized :

17. IDL> `help, time`

The IDL output log should report that "*time*" is a variable of type string that is equal to the current date/time in the format "*DOW MON DD HH:MM:SS YEAR*".

There are also 2 different types of **parameters** that any given IDL routine can accept :

- **Arguments**
- **Keywords**

In the examples above, only arguments were specified. **Arguments** are used to *pass information* (e.g. data) to and/or from the IDL routine. Arguments are *positional* in that the order in which they are passed dictates what the IDL routine does with the information in the variable/value specified. Some or all of the arguments for any given routine may be optional.

In contrast, **keywords** are always optional and can be specified in any order. Keyword parameters are used to *control the behavior* of the IDL routine being executed, or to specify a *named variable* into which a result will be placed. For instance, to control the behavior of the *HELP* procedure so it returns information about the amount of dynamic memory (in bytes) currently in use by the IDL session, set the *MEMORY* keyword equal to one :

18. IDL> `help, memory=1`

The *MEMORY* keyword is a binary behavioral parameter that is either "on" or "off", and this is controlled by setting the keyword to a value of 1 or 0, respectively. A shortcut to *setting* a keyword "on" is available by preceding the variable with the forward slash "/" character. In addition, keywords can be abbreviated to the smallest string that uniquely identifies them for the routine in question.  For example, the following is a shortcut to perform the same exact task as above :

```
19. IDL> help, /mem
```

Some keywords are used to return results from a routine.  For example, the *OUTPUT* keyword to the *HELP* procedure can be used to place a string array containing the formatted output of the *HELP* command into a named variable called "*text*" :

```
20. IDL> help, /mem, output=text
21. IDL> help, text
    TEXT            STRING    = Array[1]
22. IDL> print, text
```

Finally, it is worth mentioning that you can view the IDL software documentation and automatically jump to the index location of a specific keyword by executing the question-mark character "?" followed by the keyword :

```
23. IDL> ?help
```

This will launch the IDL Online Help system and display the reference guide entry for the *HELP* procedure.

# Chapter 2:  Two & Three Dimensional Plots

## The IDL Intelligent Tools (iTools)

The IDL Intelligent Tools (**iTools**) are a set of interactive utilities that combine data analysis and visualization with the ability to produce presentation quality graphics. The iTools allow users to continue to benefit from the control of a programming language, while enjoying the convenience of a point-and-click environment.  There are 6 primary iTool utilities built into the IDL software package.  Each of these six tools is designed around a specific data or visualization type :

- Two and three dimensional plots (line, scatter, polar, and histogram style)
- Surface representations
- Contours
- Image displays
- Mapping
- Volume visualizations

The iTools system is built upon an **object-oriented** component framework architecture that is actually comprised of only a single tool, which adapts to handle the data that the user passes to it.  The pre-built *iPlot*, *iSurface*, *iContour*, *iMap*, *iImage*, and *iVolume* procedures are simply shortcut configurations that facilitate ad hoc data analysis and visualization.  Each pre-built tool encapsulates the functionality (data operations, display manipulations, visualization types, etc.) required to handle its specific data type.  However, users are not constrained to work with a single data or visualization type within any given tool.  Instead, using the iTools system a user can combine multiple dataset visualization types into a single tool creating a hybrid that can provide complex, composite visualizations.

## Starting the iPlot Intelligent Tool and Loading Data

There are 3 ways in which a user can launch an iTool :

- Execute the statement for the appropriate procedure at the IDL> command prompt
- Select "*File > New > Visualization > iTool Name*" from the IDLDE main menu
- Windows users can launch an iTool directly by selecting "*Start > All Programs > RSI IDL #.# > iTools > iTool Name*", and Macintosh users can launch an iTool directly by double-clicking "*/Applications/rsi/idl_#.#/iTool Name*".

In this case, launch the iPlot tool by executing the procedure *iPlot* at the IDL> command prompt :

1. IDL> `iPlot`

A splash screen will appear while the iTools system is launching. The first time the iTools system is launched in any given session of IDL will always be the slowest since the iTools need to be compiled from their ASCII text source code files found in the "*lib/itools/*" subfolder of the IDL distribution (i.e. the user has access to the source code for the entire iTools system). Once the iTools system is up and running a separate window will come up entitled "*IDL iPlot*". Feel free to minimize the IDLDE window as the next steps of this exercise will involve the *iPlot* utility.

In the following exercise, the time-series data from the example data file "*time_series.txt*" will be input into the *iPlot* utility. This example data file is located in the "*data*" subfolder of the Quick Start directory :

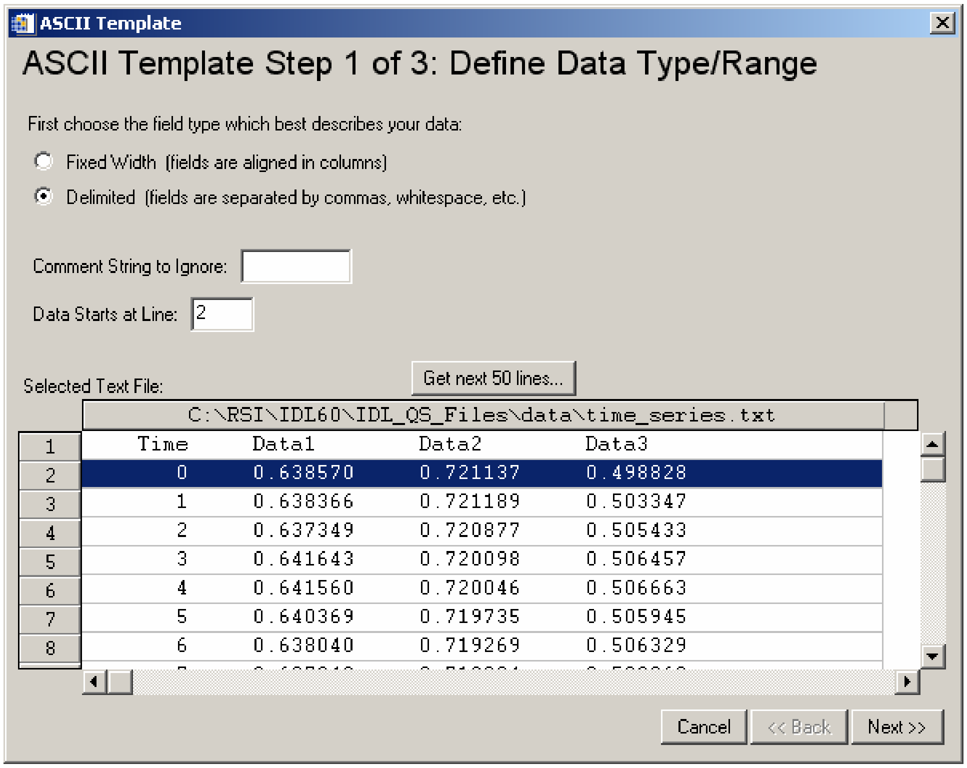
- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\time_series.txt*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/time_series.txt*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/time_series.txt*

The file "*time_series.txt*" contains standard ASCII text consisting of 4 columns of data separated by whitespace. The first column contains the time-stamp data (independent variable), and the rest of the columns contain various data measurements made at the associated time (dependent variables). Here are the first 4 lines of this ASCII text file :

```
Time      Data1         Data2         Data3
   0      0.638570      0.721137      0.498828
   1      0.638366      0.721189      0.503347
   2      0.637349      0.720877      0.505433
```

There are a number of different ways to load data into an iTool, and probably the easiest of these is to use the data input utilities built into the iTools system.

2. Select "*File > Open…*" from the *IDL iPlot* window.
3. Navigate to the location of the "*time_series.txt*" file, select it, and hit "*Open*".
4. This will launch the *ASCII Template* wizard [Fig. 2-1] that will guide the user through the steps of defining the layout of the data within the file.
5. Change the "Data Starts at Line:" field to *2* so the first line of column headers is skipped during the read process [Fig. 2-1].
6. Hit the "*Next >>*" button to proceed to step 2.
7. In Step 2 of 3 make sure all of the parameters match the default settings illustrated in Fig. 2-2.
8. Hit the "*Next >>*" button to proceed to step 3.
9. In Step 3 of 3 change the name of the fields to match the column headers above. This can be accomplished by clicking on the columns in the "Sample Record:" area and typing the appropriate text into the "Name:" field [Fig. 2-3].
10. Hit "*Finish*" to complete the *ASCII Template* wizard.



ASCII Template

ASCII Template Step 1 of 3: Define Data Type/Range

First choose the field type which best describes your data:

○ Fixed Width (fields are aligned in columns)

● Delimited (fields are separated by commas, whitespace, etc.)

Comment String to Ignore:

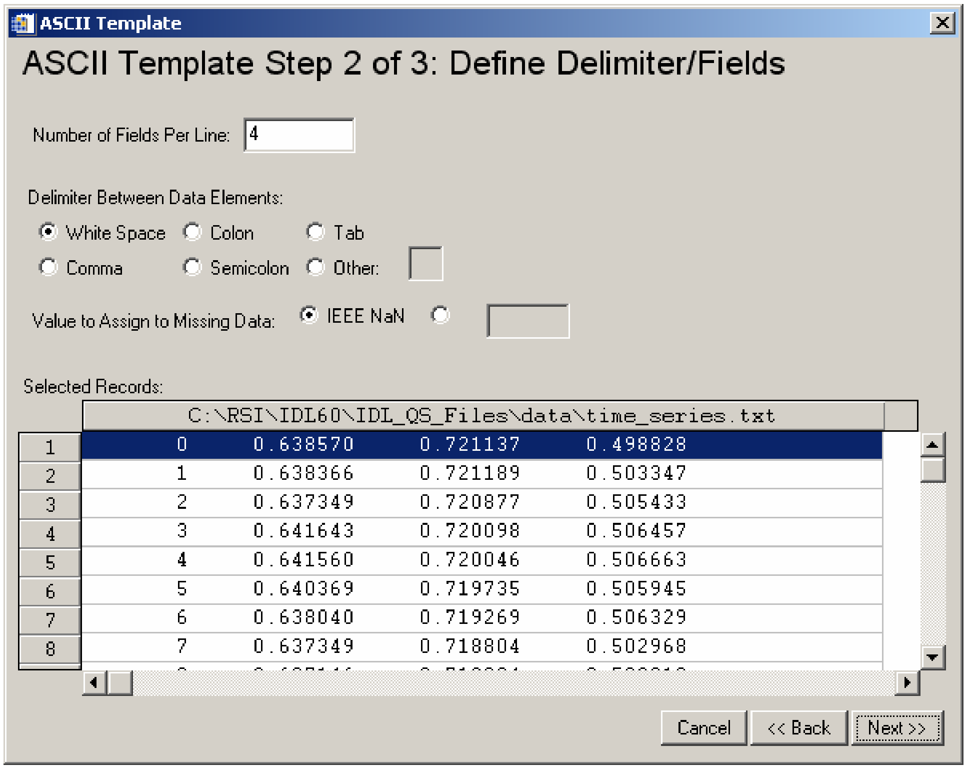
Data Starts at Line: 2

Selected Text File:

C:\RSI\IDL60\IDL_QS_Files\data\time_series.txt

| | | | | |
|---|---|---|---|---|
| 1 | Time | Data1 | Data2 | Data3 |
| 2 | 0 | 0.638570 | 0.721137 | 0.498828 |
| 3 | 1 | 0.638366 | 0.721189 | 0.503347 |
| 4 | 2 | 0.637349 | 0.720877 | 0.505433 |
| 5 | 3 | 0.641643 | 0.720098 | 0.506457 |
| 6 | 4 | 0.641560 | 0.720046 | 0.506663 |
| 7 | 5 | 0.640369 | 0.719735 | 0.505945 |
| 8 | 6 | 0.638040 | 0.719269 | 0.506329 |

***Figure 2-1: Step 1 of 3 for the ASCII Template wizard***



ASCII Template

ASCII Template Step 2 of 3: Define Delimiter/Fields

Number of Fields Per Line: 4

Delimiter Between Data Elements:

● White Space ○ Colon ○ Tab

○ Comma ○ Semicolon ○ Other:

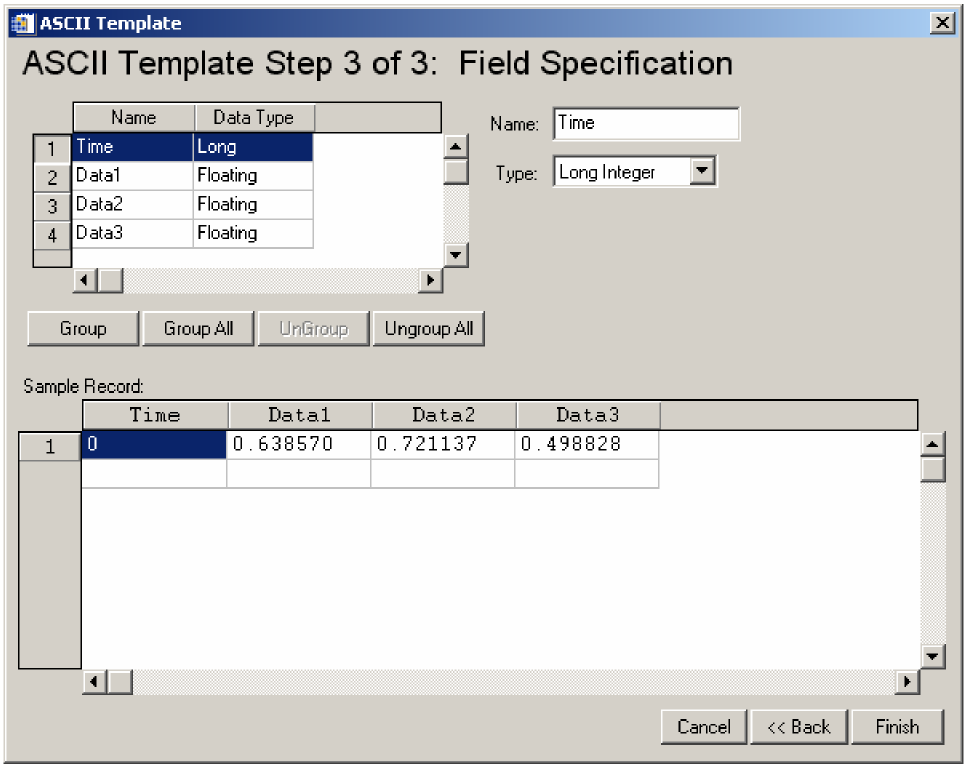
Value to Assign to Missing Data: ● IEEE NaN ○

Selected Records:

C:\RSI\IDL60\IDL_QS_Files\data\time_series.txt

| | | | | |
|---|---|---|---|---|
| 1 | 0 | 0.638570 | 0.721137 | 0.498828 |
| 2 | 1 | 0.638366 | 0.721189 | 0.503347 |
| 3 | 2 | 0.637349 | 0.720877 | 0.505433 |
| 4 | 3 | 0.641643 | 0.720098 | 0.506457 |
| 5 | 4 | 0.641560 | 0.720046 | 0.506663 |
| 6 | 5 | 0.640369 | 0.719735 | 0.505945 |
| 7 | 6 | 0.638040 | 0.719269 | 0.506329 |
| 8 | 7 | 0.637349 | 0.718804 | 0.502968 |

***Figure 2-2: Step 2 of 3 for the ASCII Template wizard***



ASCII Template

ASCII Template Step 3 of 3: Field Specification

|  | Name | Data Type |
|---|---|---|
| 1 | Time | Long |
| 2 | Data1 | Floating |
| 3 | Data2 | Floating |
| 4 | Data3 | Floating |

Name: Time

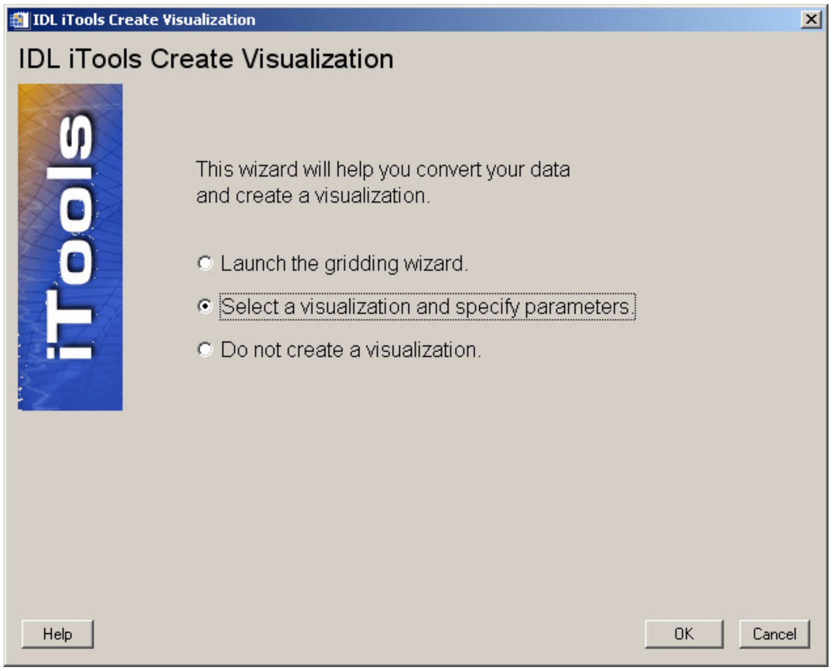
Type: Long Integer

Sample Record:

|  | Time | Data1 | Data2 | Data3 |
|---|---|---|---|---|
| 1 | 0 | 0.638570 | 0.721137 | 0.498828 |

**Figure 2-3: Step 3 of 3 for the ASCII Template wizard**

Once the user is finished running the *ASCII Template* wizard the *IDL iTools Create Visualization* dialog will come up [Fig. 2-4].



IDL iTools Create Visualization

IDL iTools Create Visualization

This wizard will help you convert your data and create a visualization.

○ Launch the gridding wizard.

● Select a visualization and specify parameters.

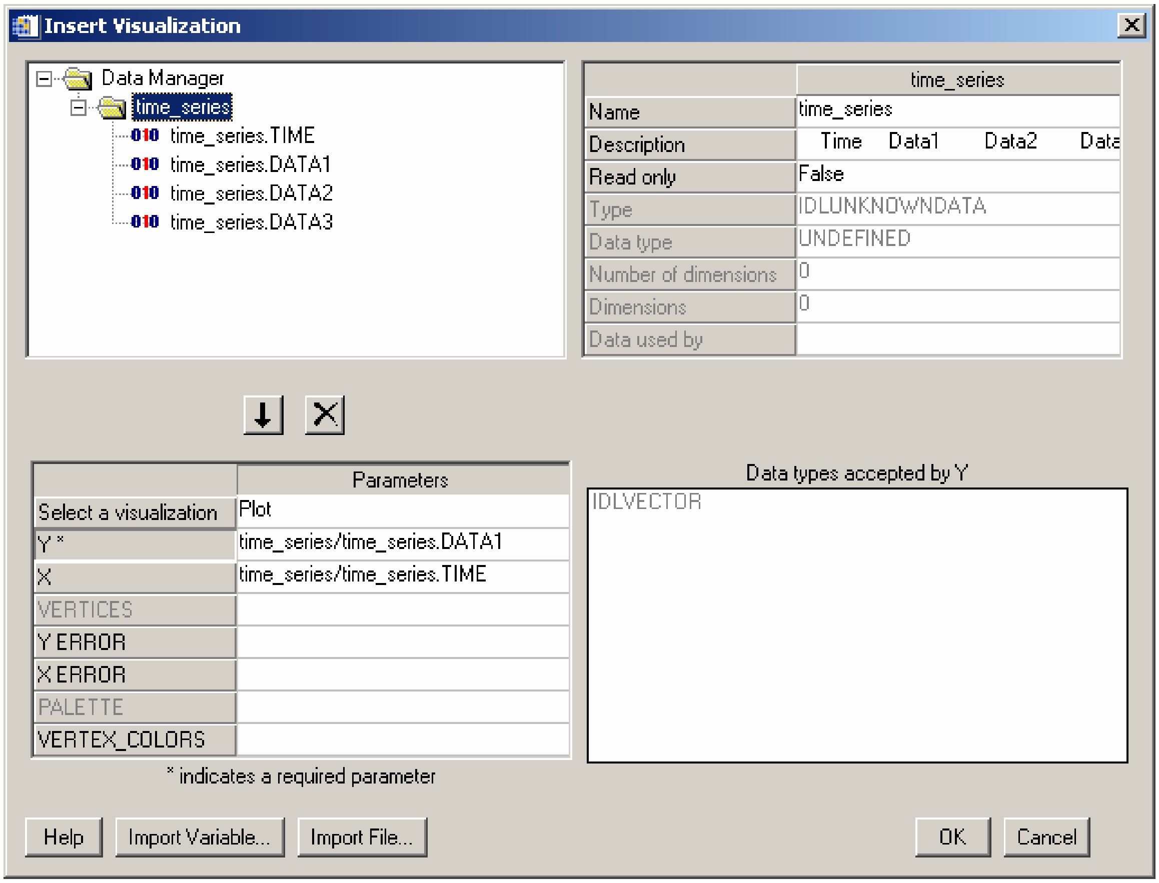
○ Do not create a visualization.

**Figure 2-4: The IDL iTools Create Visualization dialog**

11. In order to create a line plot visualization out of the time-series data, press "*Select a visualization and specify parameters*" and hit "*OK*".

This will bring up the iTools *Insert Visualization* dialog. This tool is used to select the data that is used to create the line plot visualization. In this case, there is one independent variable (X) contained in the field "time_series.TIME", and 3 dependent variables (Y) that can be plotted as a function of this time.

12. Click on the "time_series" item within the Data Manager panel and expand.
13. Select the "X" row in the lower left table and then double-click the "time_series.TIME" field from the upper left window to insert the time-stamp data as the independent variable [Fig. 2-5].
14. Select the "Y" row in the lower left table and then double-click the "time_series.DATA1" field from the upper left window to insert the first data measurement as the dependent variable [Fig. 2-5].
15. Once these selections have been made, press "*OK*".



Data Manager
- time_series
  - time_series.TIME
  - time_series.DATA1
  - time_series.DATA2
  - time_series.DATA3

| | time_series |
|---|---|
| Name | time_series |
| Description | Time Data1 Data2 Data |
| Read only | False |
| Type | IDLUNKNOWNDATA |
| Data type | UNDEFINED |
| Number of dimensions | 0 |
| Dimensions | 0 |
| Data used by | |

| | Parameters |
|---|---|
| Select a visualization | Plot |
| Y * | time_series/time_series.DATA1 |
| X | time_series/time_series.TIME |
| VERTICES | |
| Y ERROR | |
| X ERROR | |
| PALETTE | |
| VERTEX_COLORS | |

* indicates a required parameter

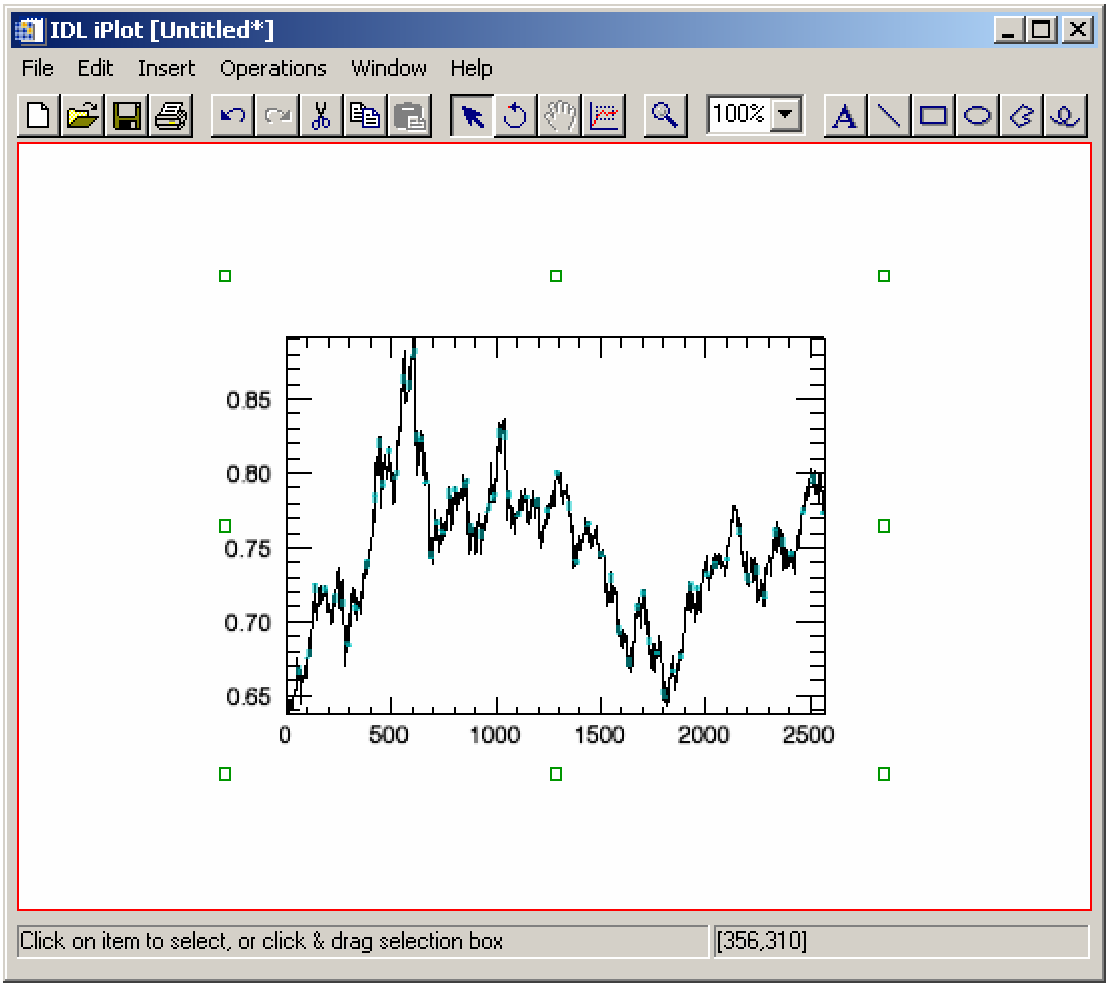
Data types accepted by Y

IDLVECTOR

***Figure 2-5: The iTools Insert Visualization dialog***

A simple line plot of the "*Time*" data (X axis) versus the "*Data1*" data (Y axis) will appear within the *IDL iPlot* window [Fig. 2-6]. Notice that both the X and Y axes have been automatically scaled to display the full data range of the respective variables. Currently the size of the *IDL iPlot* window should be the default when it was first launched, which makes it difficult to visualize the features within the line plot graphic. Consequently, use the following steps to maximize the visual area for the graphics window :

16. From the *IDL iPlot* window menu select "*Window > Zoom on Resize*".
17. Hit the maximize button □ in the upper-right hand corner of the window.



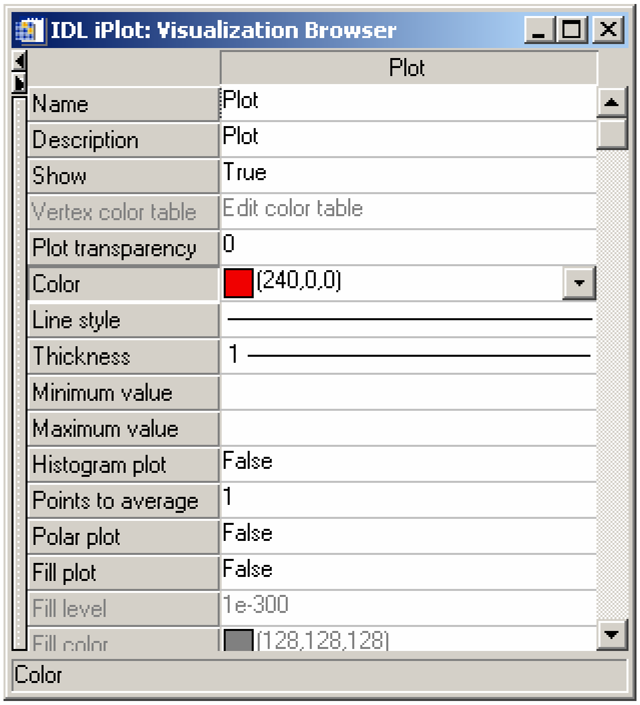
Click on item to select, or click & drag selection box

[356,310]

***Figure 2-6: The IDL iPlot window displaying graphic of time-series data***

Currently the line plot should be displayed with the default black color. Furthermore, the *iPlot* tool should be in selection mode with the line plot automatically selected (graphic will be surrounded by 8 small squares and highlighted with cyan color markers). If the line plot is not selected simply click on the line in any location so it appears as displayed in Fig. 2-6. In order to differentiate the line plot from the axes (and the other data variables which will soon be plotted) change the color to red by following these steps :

18. Right-click on the lineplot in order to see the standard iTools context menu.
19. Select "*Properties…*".
20. Within the *IDL iPlot: Visualization Browser* window, click on the box to the right of the "Color" property and select bright red [Fig. 2-7].
21. Once the desired change has been made, close the *IDL iPlot: Visualization Browser* window.



|  | Plot |
|---|---|
| Name | Plot |
| Description | Plot |
| Show | True |
| Vertex color table | Edit color table |
| Plot transparency | 0 |
| Color | 🟥 (240,0,0) |
| Line style | — |
| Thickness | 1 — |
| Minimum value |  |
| Maximum value |  |
| Histogram plot | False |
| Points to average | 1 |
| Polar plot | False |
| Fill plot | False |
| Fill level | 1e-300 |
| Fill color | ■ (128,128,128) |

Color

***Figure 2-7: The IDL iPlot: Visualization browser window (property sheet)***

Now that the line plot for the first dataset has been changed in color to red, the two remaining dependent data variables can be imported an overplotted within the existing visualization :

22. From the *IDL iPlot* window menu system, select "*Insert > Visualization…*".
23. Select the "Y" row in the lower left table and this time double-click the "time_series.DATA2" field from the upper left window to insert the second data measurement as the dependent variable.
24. Once this has been accomplished, hit "*OK*".

This will automatically insert a new lineplot within the existing data space and axes. Once again the line is plotted with the default black color, so use the following steps to change the color of this line to blue :

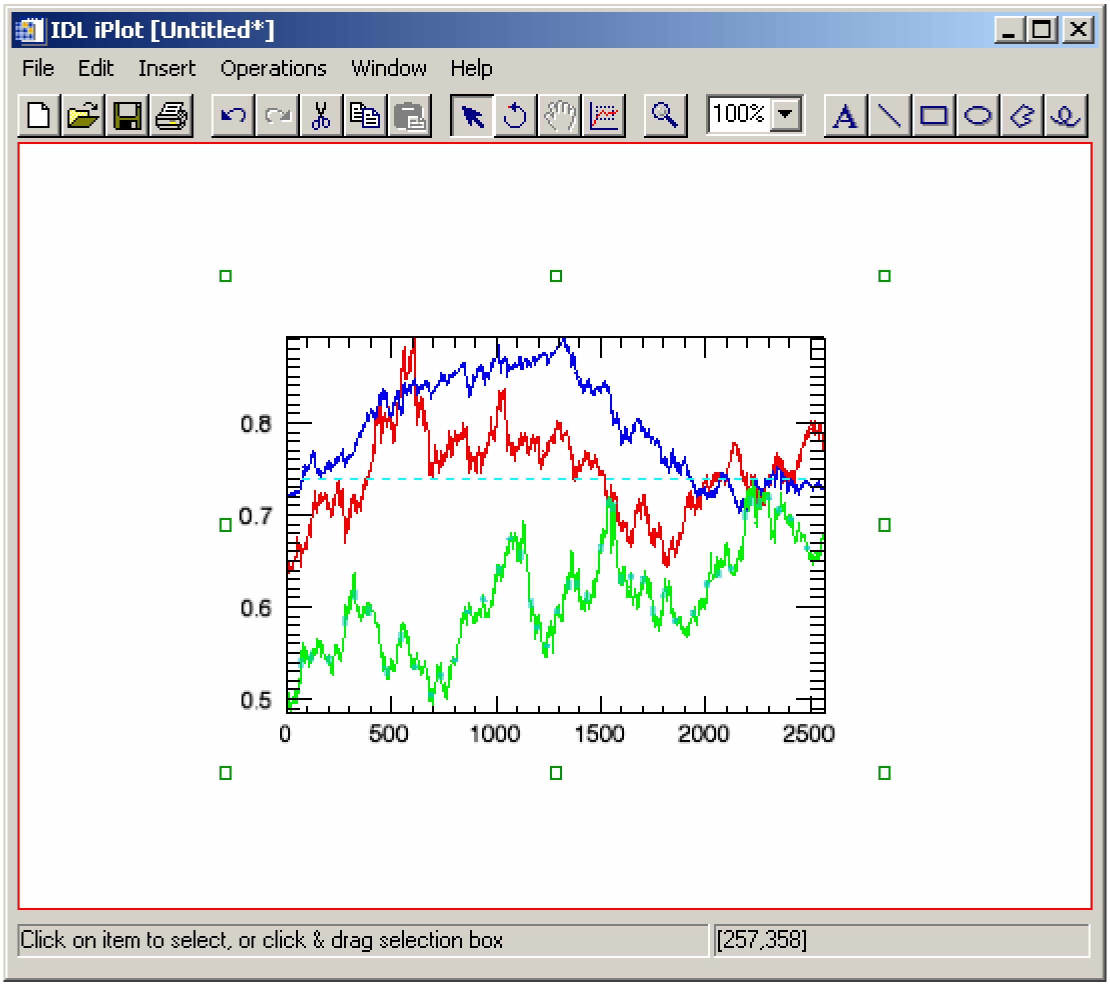
25. Select the new black line plot if it is not already (it should be selected by default).
26. Right-click, select "Properties…", and change the "Color" property to blue.
27. Close the *IDL iPlot: Visualization Browser* window.

Next, plot the third and final data variable and change its line color to green :

28. From the *IDL iPlot* window menu system, select "*Insert > Visualization…*".
29. This time select the "Y" row in the lower left table and then double-click the "time_series.DATA3" field from the upper left window to insert the second data measurement as the dependent variable.
30. Once this has been accomplished, hit "*OK*".
31. Select the new black line plot.

32. Right-click, select "Properties…", and change the "Color" property to green.
33. Close the *IDL iPlot: Visualization Browser* window.

Notice that when this final data variable was plotted, the scaling of the Y axis automatically changed in order to display the full data range of all 3 time-series measurements. In addition, dashed lines that are cyan in color will be displayed when a specific line plot is selected, which mark the minimum and maximum Y data range for that dataset. The line plot visualization should now look like Fig. 2-8.



Click on item to select, or click & drag selection box

[257,358]

***Figure 2-8: The line plot visualization of all 3 time-series datasets***

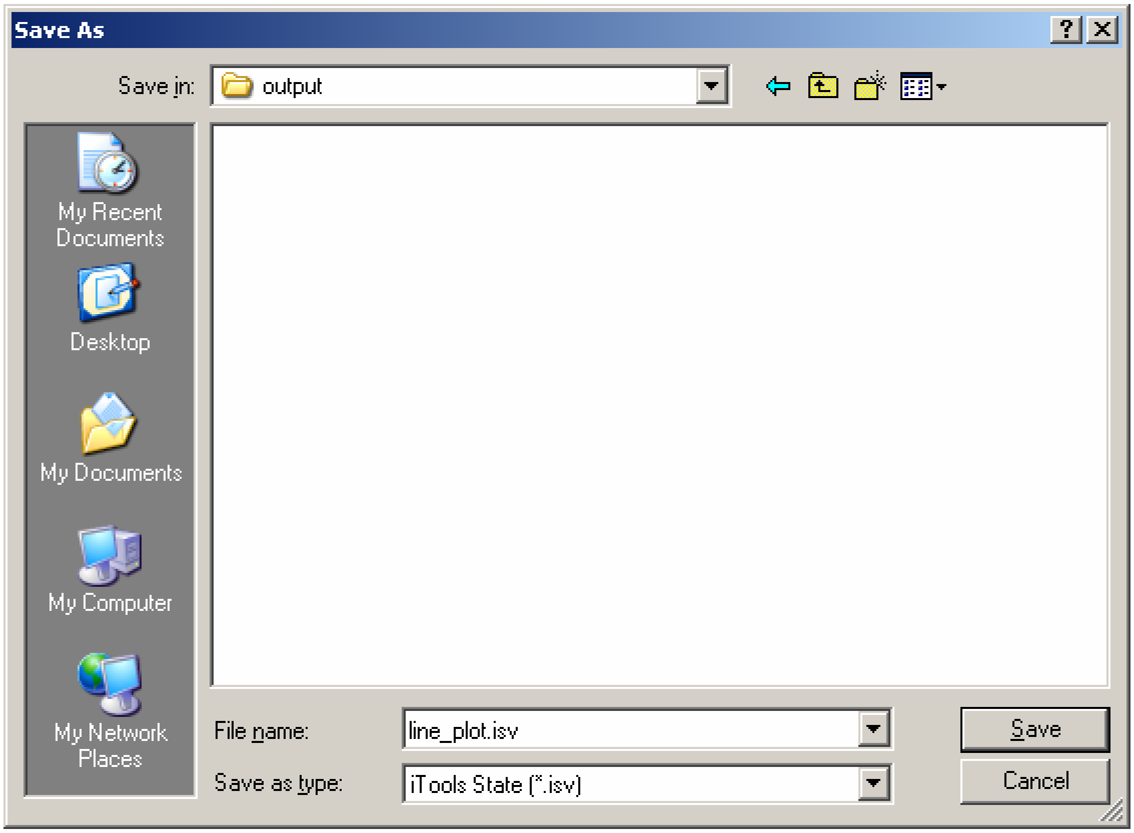
Now that all 3 time-series datasets have been loaded and plotted, it is a good idea to save the current **state** of the *iPlot* visualization utility. When an iTool is saved, the state of the utility in its entirety is output to a file on disk, and this state includes the layout of the current graphical visualizations, their properties, and the data itself. Use the following steps to save the current iTool state :

34. From the *IDL iPlot* window, select "*File > Save As…*".
35. Save the iTools state to a new file named "*line_plots.isv*" located in the "*output*" subfolder of the Quick Start directory [Fig. 2-9] :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\output\line_plots.isv*

- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/output/line_plots.isv*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/output/line_plots.isv*

36. Press the "*Save*" button.



File name: line_plot.isv

Save as type: iTools State (*.isv)

***Figure 2-9: Saving the iTool state to a file on disk***

## Selection of Objects, Undo/Redo, and Manipulation

The iTools graphical system is based on an object-oriented architecture where each element (graphic, display window, axes, annotations, etc.) have individual **objects** associated with them. These objects each have their own individual **properties**, such as the color of the line plots that were modified in the previous exercise. The power of an object-oriented architecture is that separate objects can be built into a **hierarchy** that provides the ability to create complex graphical visualizations.

There are three ways to select separate or multiple objects within the iTools system :

- Using the **Select/Translate** arrow manipulator
- Using the *Visualization Browser* window
- Selecting "*Edit > Select All*" from the iTool menu system

Experiment with selecting separate graphical objects or groups of objects using the Select/Translate arrow :

1. If necessary, click the Select/Translate arrow button ↖ on the toolbar to enter the object selection and translation manipulator mode.
2. Position the mouse cursor pointer over one of the line plots and click with the left mouse button. The line plot object will be highlighted with cyan markers, and a selection box will appear around the data space.
3. To select additional line plots, hold down the *Shift* key on the keyboard while clicking them.
4. The arrow button manipulator also provides translation functionality. While at least one of the line plots is selected, click and hold down the left mouse button while moving the mouse to re-position the data space within the view window. The mouse pointer changes to the translation pointer ✥.
5. In addition to being able to select and move the line plots, the axes are also their own individual objects. Click on one of the axes to select it, then hold down the left mouse button to move it within the view window. Each axis can be moved in a plane perpendicular to its direction.
6. Also experiment with the ability to click-and-drag an object selection box. This can be accomplished by clicking in the white area outside of the data space, holding down the mouse button, and dragging a rectangular box. Once the mouse button is released, any objects that fall within the selected region in their entirety will be selected and highlighted.

If at any time an action is taken within an iTool that the user wishes to reverse, they can utilize the built-in Undo/Redo system. There are two ways to access the Undo/Redo mechanism within any given iTool :

- Using the Undo/Redo ↶↷ buttons on the toolbar
- From the *Edit* menu system

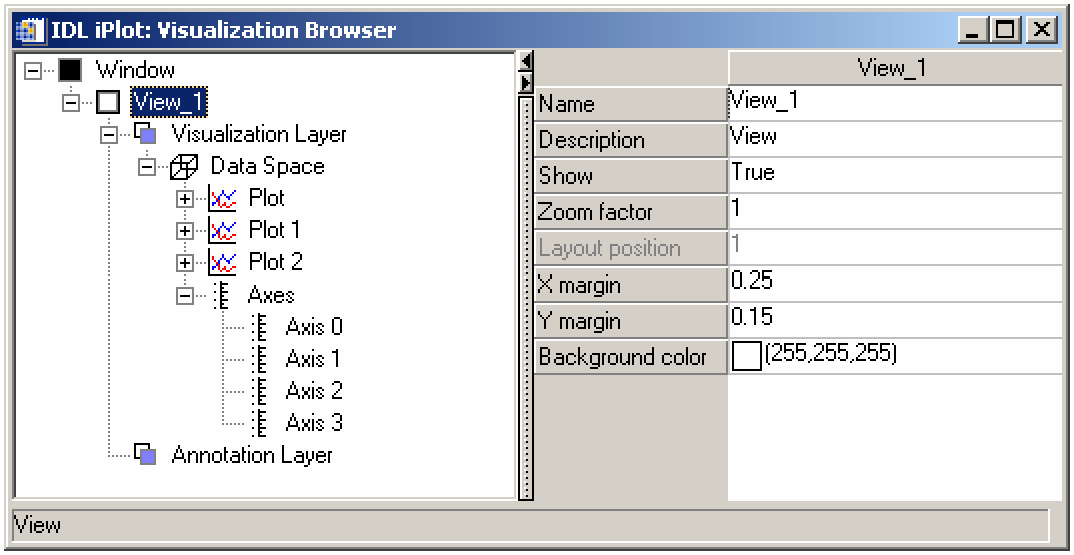
7. Experiment with the Undo/Redo system using both the toolbar buttons and the *Edit* menu system items in order to retrace and repeat the steps that have been performed within the current *IDL iPlot* utility.
8. Using the Select/Translate arrow or the Undo/Redo system, setup the current visualization in its previous appearance with the data space centered within the view window.
9. While in Select/Translate mode, click in an area outside of the data space so that no object appears selected.
10. Open the *Visualization Browser* window by selecting "*Window > Visualization Browser…*" from the menu system.

The *Visualization Browser* window displays the object-oriented hierarchy of the current graphical visualization for the iTool on the left-hand side, and the property sheet for the currently selected object on the right-hand side [Fig. 2-10]. Currently the main view object named "*View_1*" should be selected, which is the white graphics area within the *IDL iPlot* window.

11. Experiment with selecting various objects on the left-hand side of the *Visualization Browser* window.

Notice how the objects are once again highlighted within the *IDL iPlot* utility window when they are selected.  In addition, notice how the property sheet on the right-hand side of the *Visualization Browser* window updates to reflect the properties of the currently selected object.

Currently the 3 line plot graphical objects that are contained within the iTool are named "*Plot*", "*Plot 0*", and "*Plot 1*" by default [Fig. 2-10].  These line plot objects can be given more appropriate names that help associate them with the data they are displaying.



IDL iPlot: Visualization Browser

- Window
  - View_1
    - Visualization Layer
      - Data Space
        - Plot
        - Plot 1
        - Plot 2
        - Axes
          - Axis 0
          - Axis 1
          - Axis 2
          - Axis 3
    - Annotation Layer

| | View_1 |
|---|---|
| Name | View_1 |
| Description | View |
| Show | True |
| Zoom factor | 1 |
| Layout position | 1 |
| X margin | 0.25 |
| Y margin | 0.15 |
| Background color | □ (255,255,255) |

View

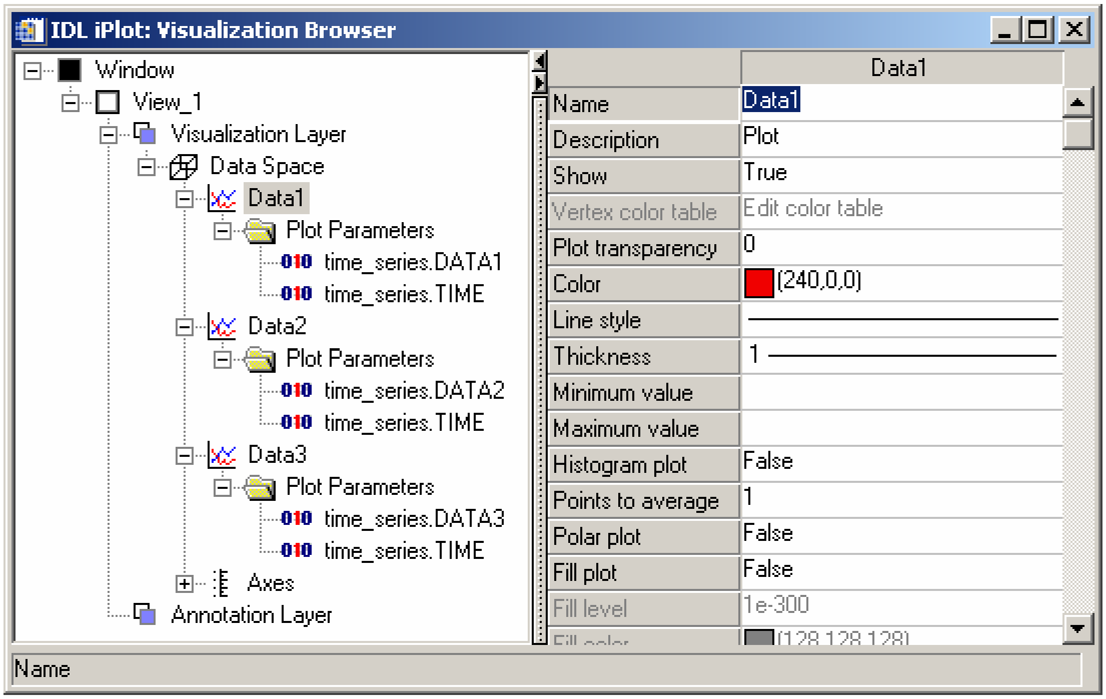
***Figure 2-10: The Visualization Browser window***

12. Click on the first line plot object named "Plot" within the *Visualization Browser* window.
13. On the right-hand side, change the "Name" property to the appropriate time-series field name from the original dataset "*Data1*".  It may help to expand the plot object tree to look at the parameters (data) that define the graphical object [Fig. 2-11].
14. Make sure to hit the *Enter* key on the keyboard after modifying the "Name" property so the change takes effect.
15. Repeat this process for the other 2 line plot objects, changing "*Plot 0*" to "*Data2*", and "*Plot 1*" to "*Data3*" [Fig. 2-11].
16. The final state of the *Visualization Browser* window should look like Fig. 2-11. Once this has been accomplished close the *Visualization Browser* window.

In addition to the translation manipulator mode accessed via the Select/Translate arrow button, there is also **Rotate**, **View Zoom**, **Scaling**, and **Data Range** manipulators that can be accessed using their respective buttons on the toolbar :

17. From the *IDL iPlot* window menu system, select "*Edit > Select All*".
18. Click the Rotate button to enter rotation manipulator mode.
19. Experiment with arbitrarily rotating the current graphic by clicking on it, holding down the mouse button, and moving the mouse.

20. Notice how the arrow keys on the keyboard can also be used to rotate the graphic (after it has been rotated by the mouse). Furthermore, notice how using the *Shift* and *Ctrl* keys in conjunction with the arrow keys will rotate the graphic by specific amounts.
21. Since the rotation of line plot graphics is usually not appropriate, use either the Ctrl-arrow button single degree rotation or the Undo button to restore the visualization to its original horizontal orientation.



| | Data1 |
|---|---|
| Name | Data1 |
| Description | Plot |
| Show | True |
| Vertex color table | Edit color table |
| Plot transparency | 0 |
| Color | 🟥 (240,0,0) |
| Line style | |
| Thickness | 1 |
| Minimum value | |
| Maximum value | |
| Histogram plot | False |
| Points to average | 1 |
| Polar plot | False |
| Fill plot | False |
| Fill level | 1e-300 |
| Fill color | (128,128,128) |

***Figure 2-11: Changing the Name properties of the 3 line plot objects***

The size of the graphics area can also be made smaller / larger in relation to the display canvas by using the View Zoom manipulator :

22. Click the Zoom button to enter View Zoom manipulator mode.
23. Position the mouse pointer over the viewplane. The pointer will change to a zoom pointer.
24. The view zoom manipulator works by clicking-and-dragging with the mouse cursor. Dragging the zoom pointer towards the top of the window (upwards direction) will enlarge the graphics. Dragging the zoom pointer towards the bottom of the window (downwards direction) will make the graphics area smaller.
25. Restore the graphics area (approximately) to its original size, and click on the Select/Translate arrow button in order to leave View Zoom mode.

In addition to using the View Zoom manipulator tool, the size of the graphics area can also be made smaller / larger using the constrained and unconstrained scaling available when using the Select/Translate arrow mode. Constrained scaling permits changing the size of only one dimension while preserving the size of the other dimensions.  In contrast, unconstrained rotation allows the user to scale all dimensions of an object in an arbitrary fashion.

26. While using the Select/Translate arrow mode, click on one of the line plots within the graphics area.
27. In addition to the selected line plot being highlighted with cyan color markers, the 8 green selection boxes will appear around the graphics area.
28. To perform constrained scaling, position the mouse cursor over one of the four selection boxes found midway along each side of the graphics area.  The mouse cursor will change to a double-headed ↔ arrow.
29. Click-and-drag with the mouse to scale the graphics in the desired direction.
30. To perform unconstrained scaling, position the mouse cursor over one of the four selection boxes found in the corners of the graphics area.  The mouse cursor will change to a four-headed ⤡ arrow.
31. Click-and-drag with the mouse to scale the graphics in an unconstrained fashion.
32. Restore the graphics area (approximately) to its original shape and size.

Finally, the Data Range tool allows the user to zoom into the data space itself, instead of simply resizing the graphics area within the viewplane.  This allows users to adjust the range displayed within the current graphics area, and zoom into the data space to get a better look at features at a smaller scale.

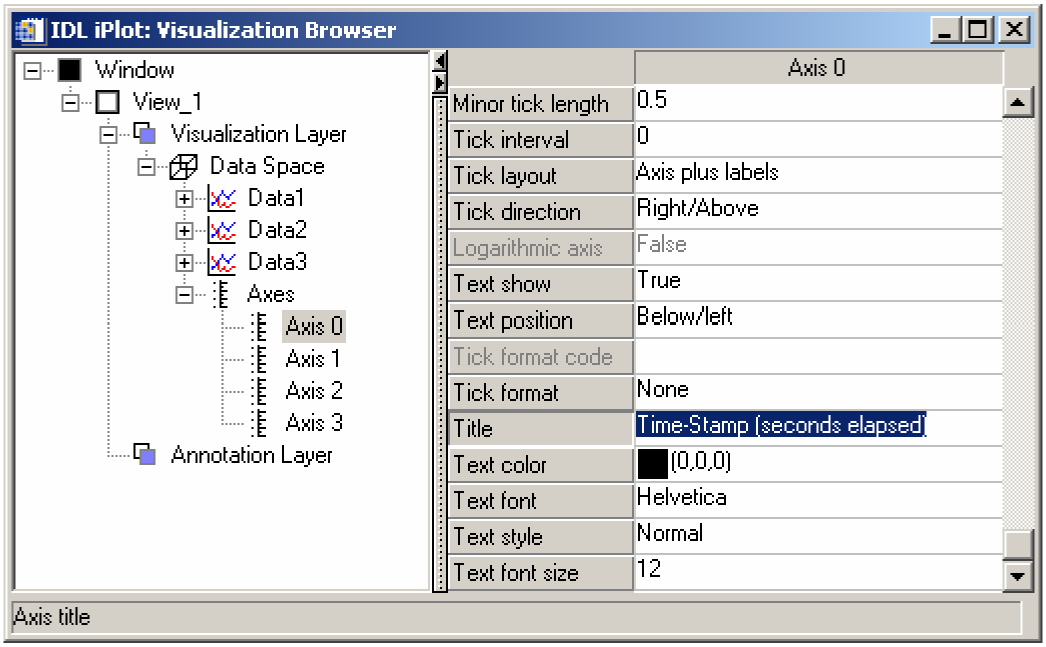
33. While one or all of the line plots are selected, click the Data Range button to enter data range manipulator mode.
34. Both the X and Y axes will be labeled with green translation arrows and zoom symbols ("+" and "-").  Clicking on these will change the range and location of the respective axis.
35. In addition, position the mouse cursor within the graphics area containing the line plots.  The mouse cursor will change to a cross-hair.
36. Click-and-drag to define a rectangular box within the graphics area, and once the box has been defined release the mouse button to zoom into the corresponding data range.
37. Select/Translate arrow button in order to leave Data Range mode.
38. In order to reset the data range to the original full extent of all 3 data variables, select "*Window > Reset Dataspace Ranges*" from the *IDL iPlot* window menu system.

## Labeling Axes, Annotation, and Inserting a Legend

Currently the line plots for the 3 time-series data variables are displayed with different colors, but there are no annotations present within the visualization window that describe the contents of the graphic.  In order to label the axes and change the direction of the tick marks, utilize the *Visualization Browser* window :

1. Select "*Window > Visualization Browser…*".
2. Expand the Axes group to see the 4 separate axis objects.
3. Select the first axis object named "*Axis 0*".  Notice that the bottom X-axis within the *iPlot* window is highlighted.

4. Scroll down the property sheet on the right-hand side and locate the "Title" property. Change the "Title" property for this axis to "*Time-Stamp (seconds elapsed)*" and hit *Enter* on the keyboard to complete the change [Fig. 2-12].



| | Axis 0 |
|---|---|
| Minor tick length | 0.5 |
| Tick interval | 0 |
| Tick layout | Axis plus labels |
| Tick direction | Right/Above |
| Logarithmic axis | False |
| Text show | True |
| Text position | Below/left |
| Tick format code | |
| Tick format | None |
| Title | Time-Stamp (seconds elapsed) |
| Text color | ■ (0,0,0) |
| Text font | Helvetica |
| Text style | Normal |
| Text font size | 12 |

***Figure 2-12: Modification of the Title property for the X axis***

5. Next, select the second axis object named "*Axis 2*". This selects the Y axis along the left-hand side of the graphics area.
6. Find the "Tick direction" property, click within the setting box, and change the droplist to "*Left/Below*".
7. Once again, scroll down and change the "Title" property to "*Data Values*".
8. Finally, select the last axis named "*Axis 3*" and change the "Tick direction" property to "*Right/Above*".
9. Once these tasks have been accomplished, close the *Visualization Browser* window.
10. An axis may still be selected, so click in the white viewplane outside of the graphics area to unselect all graphical objects.

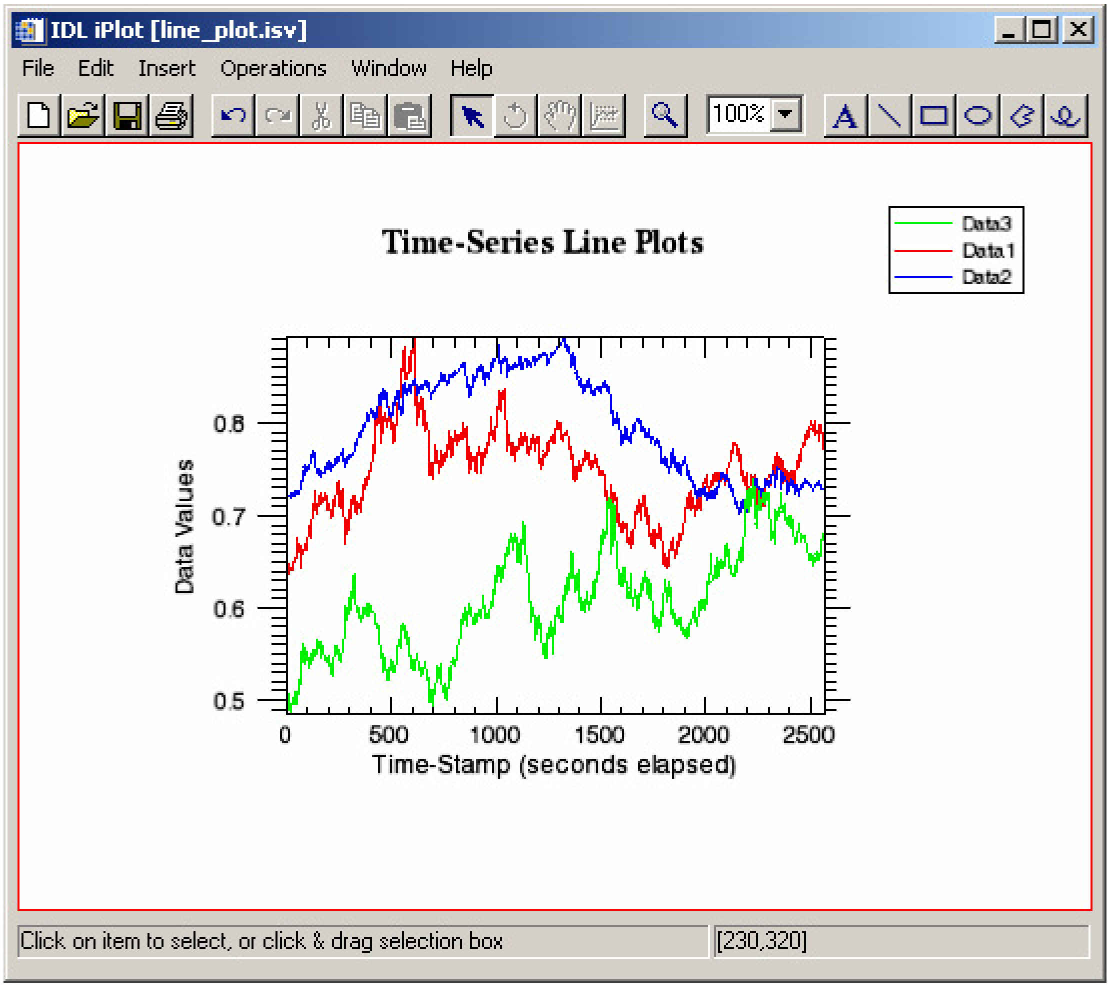
It may also be appropriate to provide a title for the overall visualization, and this can be inserted into the visualization using the Text annotation tool :

11. Click the Text A annotation button to enter text insertion mode.
12. Once the mouse cursor is moved over the viewplane, the pointer will change to a standard text selection cursor.
13. Click on a location outside and above the graphics area. A cursor is inserted into the annotation layer and text insertion mode is initiated.
14. Using the keyboard, enter the text "*Time-Series Line Plots*". Once finished typing this text, hit the *Enter* key to complete the text annotation.
15. While this text annotation object is highlighted, right-click on it and select "*Properties…*" from the context menu.

16. The font properties of this text annotation can be modified to make it appear like a legitimate graphics title. In the *Visualization Browser* window, change the "Text font" property to "*Times*".
17. Change the "Text style" property to "*Bold*".
18. Change the "Text font size" to "*16*", hit *Enter*, and close the *Visualization Browser* window.
19. Re-position the text annotation so it is centered above the graphics area.

Finally, a legend can be inserted into the *iPlot* visualization window in order to label the line plots with the names that were previously defined :

20. Click on one of the line plots, hold down the Shift key, and click on the remaining two so that all 3 line plots are selected and highlighted.
21. From the menu system, select "*Insert > New Legend*".
22. Position the legend in the desired location in the upper-right hand corner of the viewplane.
23. Click in a blank white area of the viewplane outside all graphical objects in order to deselect all objects.
24. At this point, it may be a good idea to re-save the current state of the *iPlot* utility. This can be accomplished by selecting "*File > Save*" from the menu system or hitting the Save 💾 button.
25. The current state of the *IDL iPlot* window should look similar to Fig. 2-13.



Click on item to select, or click & drag selection box

[230,320]

***Figure 2-13: The current state of the time-series line plot visualization***

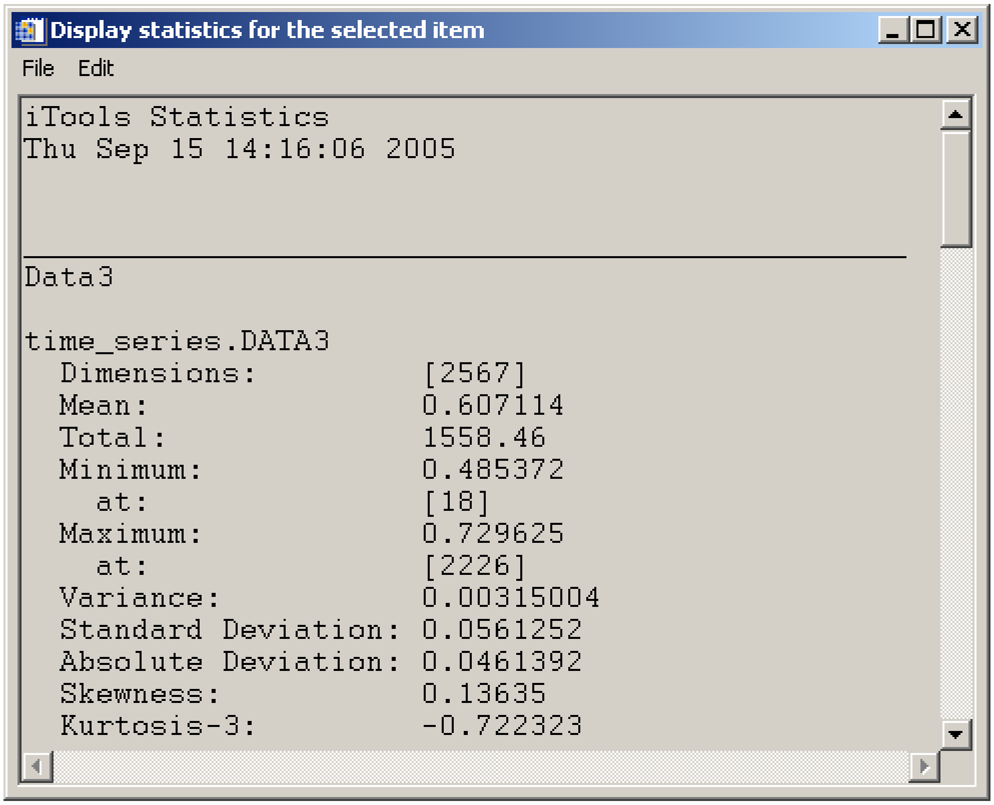
## Operations, Graphical Output, and Printing

So far the exercises in this chapter have demonstrated the data input, graphics creation, manipulation, undo/redo, and annotation capabilities of the iTools system. In addition to this standard functionality the iTools system also includes **operations**, which are data analysis and processing tools. Any given iTool will contain a suite of the most common operations for the data type and visualization style that the utility was designed for.

In the case of visualizing line plots of time-series data, one common operation that is performed in signal processing analysis is the application of a smoothing filter that will help eliminate some of the high-frequency noise contained within the data. In addition, the user may be interested in obtaining mathematical statistics on the time-series datasets :

1. Make sure the Select/Translate ↖ arrow button is selected.

2. Click on one of the line plots, hold down the Shift key, and click on the remaining two so that all 3 line plots are selected and highlighted.
3. From the menu system, select "*Operations > Filter > Smooth*". A separate dialog window entitled "Smooth" will appear.
4. This operation will apply a smoothing filter using a boxcar average over an area of a specified width. Change the value for the "Width" field to "*7*" and press the "*OK*" button in the Smooth window. Notice that the line plots are smoothed.
5. Click within the white viewplane outside the graphics area in order to deselect all objects.
6. Experiment with the Undo/Redo buttons the visualize the effect that the smoothing operation had on the line plot graphics.
7. Reselect the three line plots by clicking on each while holding down the Shift key. From the menu system, select "*Operations > Statistics…*". This will launch a separate window that displays the standard iTools statistics for each of the 3 line plot datasets [Fig. 2-14].
8. Once finished viewing the statistical information, close the iTools Statistics window.



```
iTools Statistics
Thu Sep 15 14:16:06 2005


Data3

time_series.DATA3
  Dimensions:         [2567]
  Mean:               0.607114
  Total:              1558.46
  Minimum:            0.485372
    at:               [18]
  Maximum:            0.729625
    at:               [2226]
  Variance:           0.00315004
  Standard Deviation: 0.0561252
  Absolute Deviation: 0.0461392
  Skewness:           0.13635
  Kurtosis-3:         -0.722323
```

***Figure 2-14: Statistics for the time-series datasets***

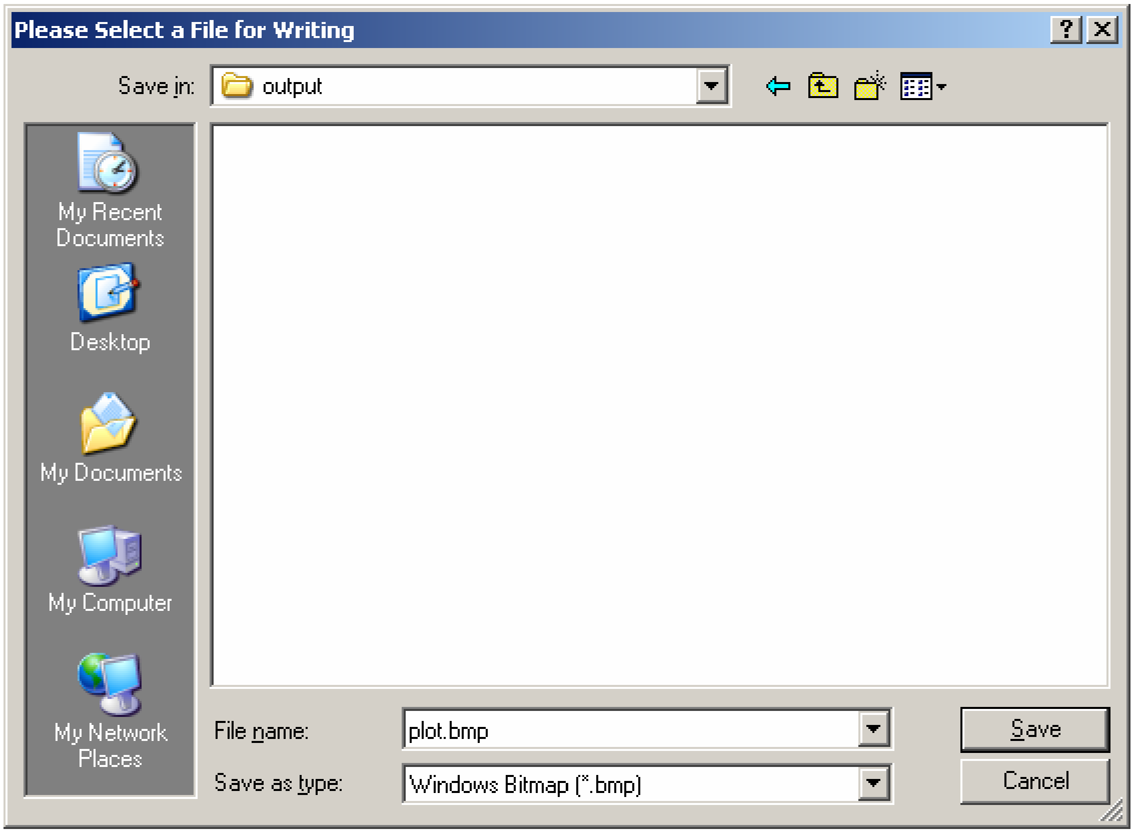
At this point, it may be necessary to obtain a softcopy or hardcopy version of the current *iPlot* visualization. There are a number of ways of exporting the current graphical visualization to the clipboard, image files on disk, or a printer :

9. Make sure the Select/Translate ↖ arrow button is selected. Click on an area outside the plot and drag a rectangle around the entire plot.

10. Select "*Edit > Copy*". This will copy the current viewplane window to the native operating system clipboard so that it can be pasted into other software applications.
11. If running on a Windows computer with Microsoft Office installed, open either Word or PowerPoint and paste the graphic into this separate application.

In addition to the standard clipboard operations, the iTools system also offers the ability to export the current visualization to a wide variety of generic image formats :

12. From the menu system, select "*File > Export…*". This will launch the *IDL Data Export Wizard*.
13. In Step 1 of 3, select the "*To a File*" option for the export destination and hit the "*Next >>*" button.
14. In Step 2 of 3, the user can select individual graphical objects to specify either the entire visualization or individual data items to export. In this case, in order to output the entire visualization select the "*Window*" object and hit "*Next >>*".
15. In step 3 of 3 the output "File Type:" should be set to *Windows Bitmap* by default. If this is not the case, select *Windows Bitmap* now.
16. Click the file selection button and specify an output filename of "*plot.bmp*" within the "*output*" subfolder of the Quick Start directory [Fig. 2-15].



Save in: output

File name: plot.bmp

Save as type: Windows Bitmap (*.bmp)

***Figure 2-15: Specifying an output filename for Windows Bitmap format***

17. Once this is accomplished, hit the "*Save*" button in order to return to the *IDL Data Export Wizard*.

18. The full path to the output file specified will be displayed within the "File Name:" field.  This output file path should be :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\output\plot.bmp*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/output/plot.bmp*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/output/plot.bmp*

19. Once this output filename is satisfactory, hit "*Finish*".

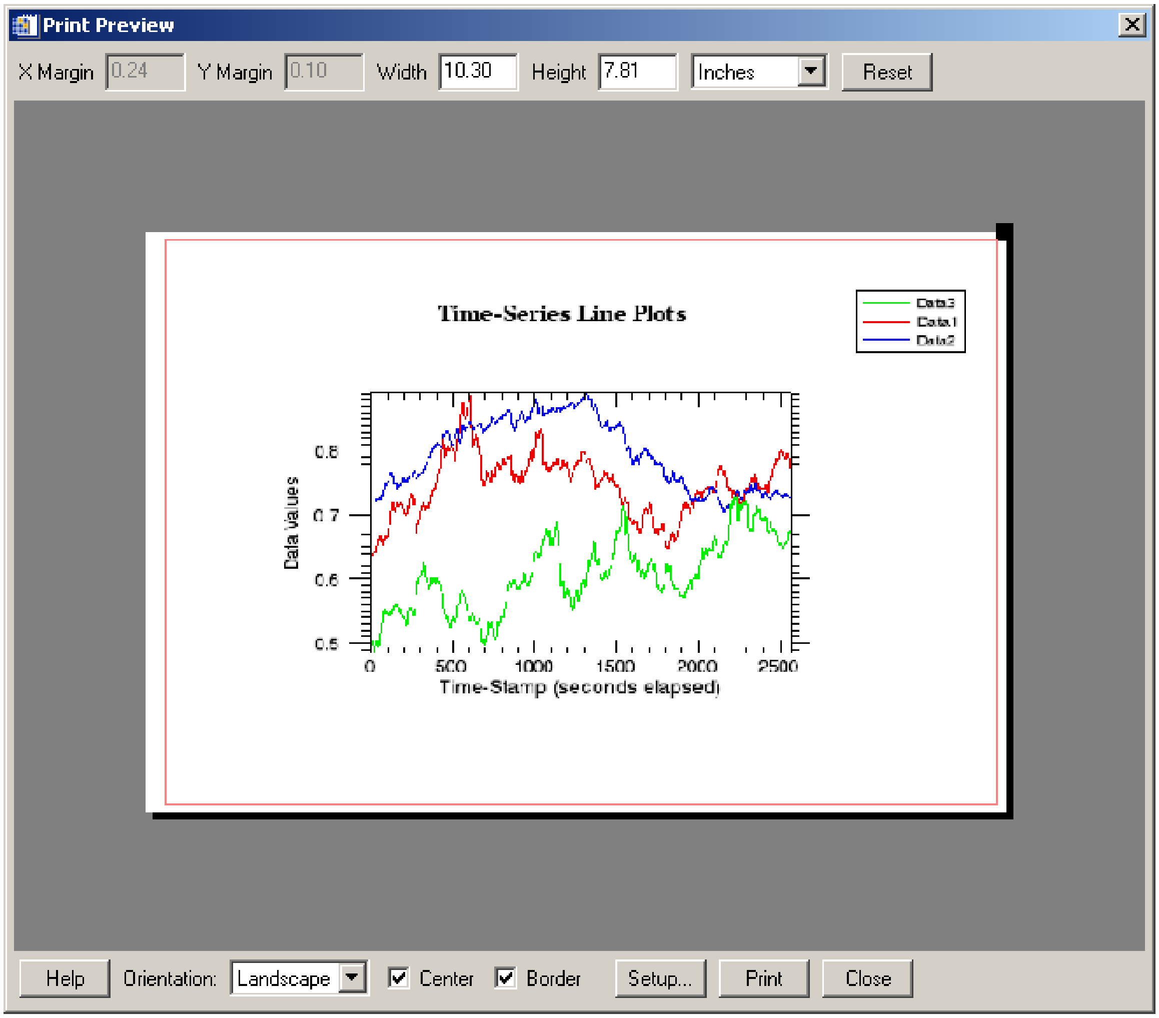
Looking within the "*output*" directory for the bitmap file that was just exported from the *iPlot* utility, the user should find a new file named "*plot.bmp*".  Looking at this file in any standard image visualization software that can read and display bitmap files will display the contents of the *iPlot* visualization as it currently appears.

The current *iPlot* visualization can also be sent directly to a printer for hardcopy output :

20. Select "*File > Print Preview…*" from the main menu system.

A separate *Print Preview* window will be displayed [Fig. 2-16].  This tool allows the user to control the layout of the iTools viewplane window in relation to a standard 8.5" x 11" piece of paper.  Within the *Print Preview* window pane are horizontal and vertical offset bars, along with a scaling box.

21. Change the orientation droplist to "*Landscape*".
22. Click on the scaling box in the upper-right corner and move up or down to make the size of the graphic larger or smaller, respectively.
23. Check the "*Center*" checkbox to automatically adjust the horizontal and vertical offsets so the graphic is centered within the page.  The user can also manually move the individual offset bars.
24. If the computer being used is configured with a printer connection, the "*Setup…*" button can be pressed to obtain the standard printer selection / configuration dialog.
25. In addition, if the computer being used is configured with a printer connection, the "*Print*" button can be pressed to direct output to the selected printer.



Print Preview

X Margin 0.24 Y Margin 0.10 Width 10.30 Height 7.81 Inches Reset

Orientation: Landscape ☑ Center ☑ Border

***Figure 2-16: The iTools Print Preview tool***

26. Once finished working with the *Print Preview* tool, press "*Close*".
27. At this point, the *IDL iPlot* window containing the time-series visualizations can be closed as it is no longer used from this point forward. This can be accomplished by selecting "*File > Exit*" from the menu system or hitting the Close ☒ button in the upper-right hand corner.
28. Restore the IDL Development Environment window so the IDL> command prompt can be accessed.

## Scatter, Histogram, Polar, and 3-Dimensional Plots

In addition to standard 2-Dimensional line plots, IDL also has the capability to display X-Y scatter plots, histograms, polar plots, and 3-D line plots. The iTools utilities can be launched with data automatically loaded using the **arguments** to the appropriate iTool procedure call. Furthermore, the properties of the graphical objects can be controlled by using the **keywords** to the iTool procedure. For example, create two 50 element vectors of uniformly-distributed, floating-point, pseudo-random numbers by using the *RANDOMU* function :

1. IDL> `x = RANDOMU (s, 50)`
2. IDL> `y = RANDOMU (s, 50)`

The *RANDOMU* function returns two variables "x" and "y" that contain data values ranging from 0 → 1.0 that are created by a random number generator. To visualize these two vectors in a **X-Y scatter plot**, simply load the variables as arguments to the *IPLOT* procedure and set the appropriate keywords in order to control the properties of the plot object. In this example, the *SCATTER* keyword is set in order to change the graphic style from the default line plot to scatter plot mode, the *SYM_INDEX* keyword is set to the symbol index for small rectangular boxes, and the *COLOR* keyword is set to a RGB triplet for the color red :

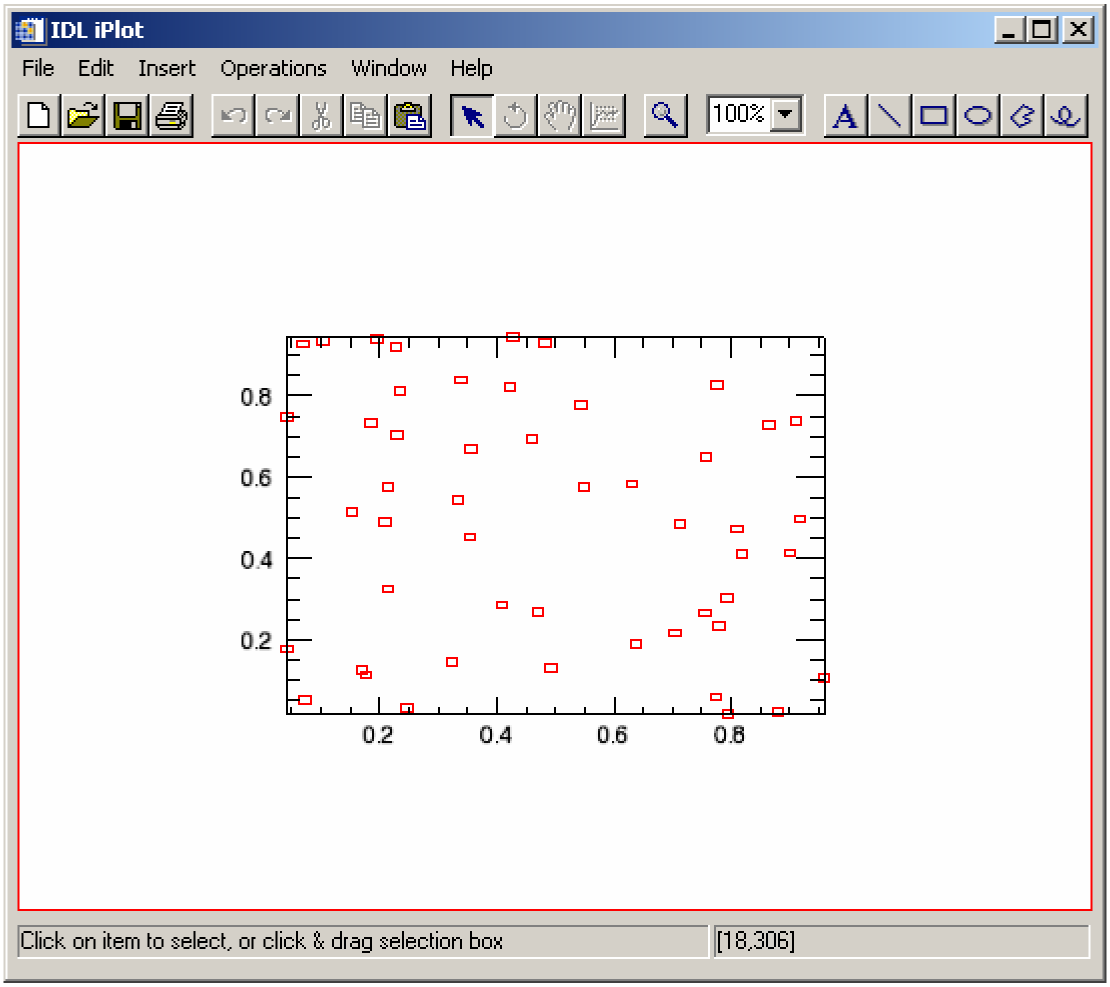
3. IDL> `iPlot, x, y, /SCATTER, SYM_INDEX=6, COLOR=[255,0,0]`

This will launch a new *IDL iPlot* window containing a X-Y scatter plot of the random data values plotted as small red rectangles [Fig. 2-17].

4. It may help to click in the white viewplane outside of the graphics area to deselect the scatter plot object in order to remove the cyan selection markers.

**Note:** The appearance of the X-Y scatter plot that results may differ slightly from what appears in Fig. 2-17 since the random numbers that are generated by the *RANDOMU* function will always be different.

5. Once finished viewing the X-Y scatter plot, close the *IDL iPlot* window.



Click on item to select, or click & drag selection box

[18,306]

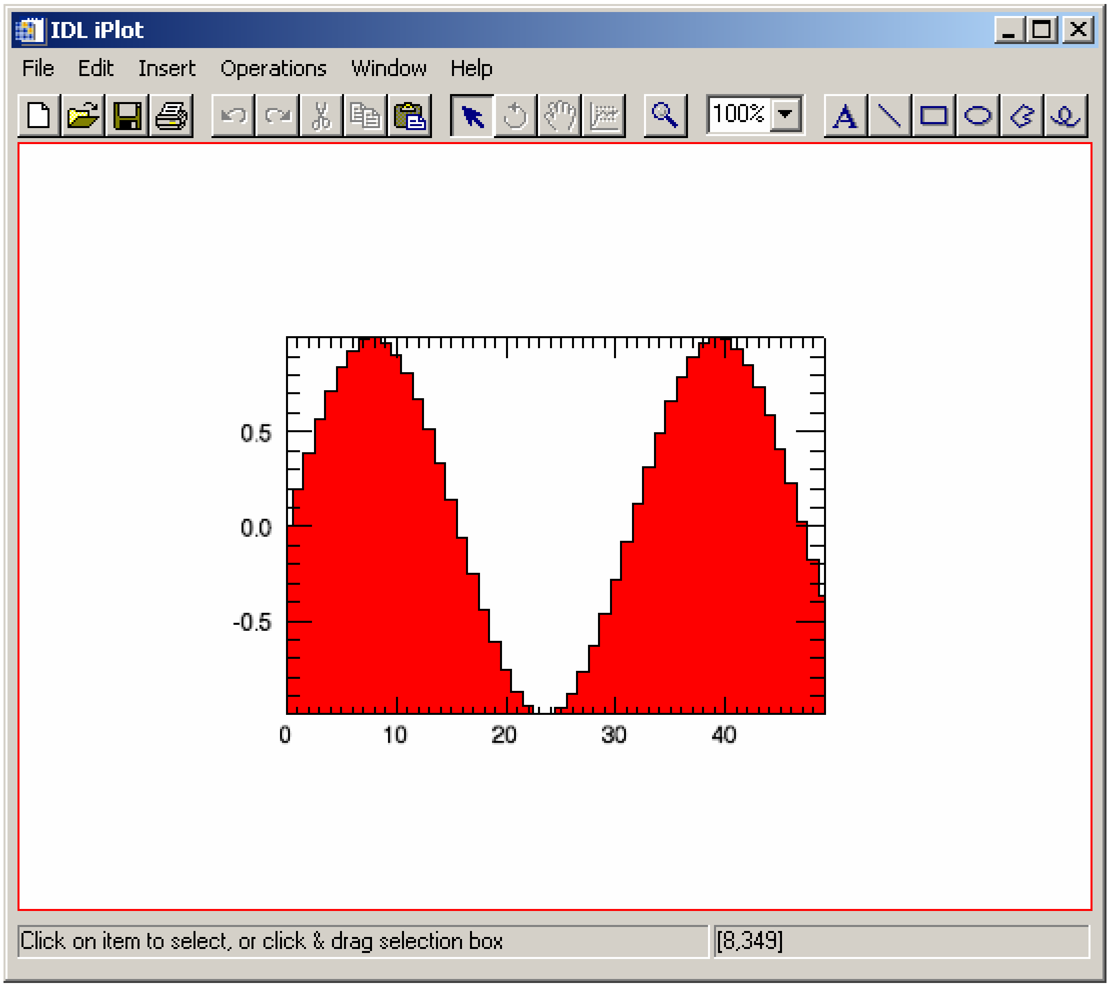
***Figure 2-17: An X-Y scatter plot of the randomly generated numbers***

A **histogram** can also be useful for visualizing 2-Dimensional signals or a density plot showing a distribution of data values. In a histogram plot, only horizontal and vertical lines are used to connect the plotted points. For example, plot a simple sine wave using a histogram style plot :

6. IDL> `sine = SIN (FINDGEN (50) / 5)`
7. IDL> `iPlot, sine, /HISTOGRAM, /FILL_BACK, FILL_COLOR=[255,0,0]`

The resulting *IDL iPlot* visualization window should look like Fig. 2-18.

8. Once finished viewing the histogram plot, close the *IDL iPlot* window.



Click on item to select, or click & drag selection box

[8,349]

***Figure 2-18: A histogram style plot of a sine wave***

So far all of the data that has been plotted is in a standard Cartesian (X-Y) coordinate system. In some cases, the data may use a **polar** coordinate system, which involves **radius** distances measured at various angles (**theta**) that are expressed in units of radians. For example, create a vector named "theta" that stores angles which encompass a full 360 degrees in 15° increments, and create a vector named "radius" that contains distances which range from 0 → 24 :

```
9.  IDL> theta = INDGEN (25) * 15 * !DTOR
10. IDL> radius = FINDGEN (25)
11. IDL> iPlot, radius, theta, /POLAR, XRANGE=[-24,24], $
         YRANGE=[-24,24]
```

The resulting *IDL iPlot* visualization window should look like Fig. 2-19.

12. Once finished viewing the polar plot, close the *IDL iPlot* window.

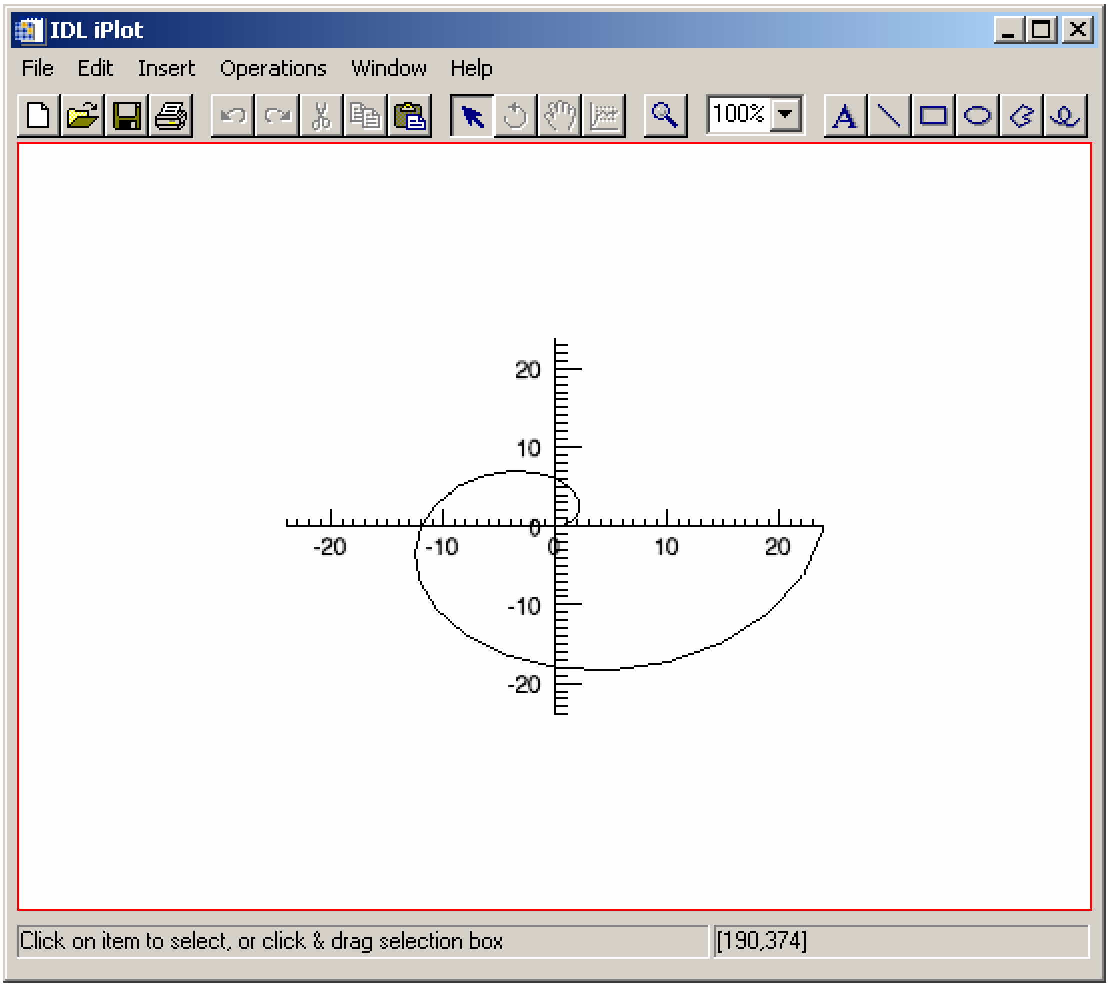


***Figure 2-19: A polar line plot of radius values that increase incrementally***

Finally, the *iPlot* utility is not limited to two dimensions, and quite often it is necessary to visualize datasets that have 3 variables (X, Y, Z) in three dimensions. For example, consider the flight test data that is stored in the file "*flight_test.txt*". This example data file is located in the "*data*" subfolder of the Quick Start directory :

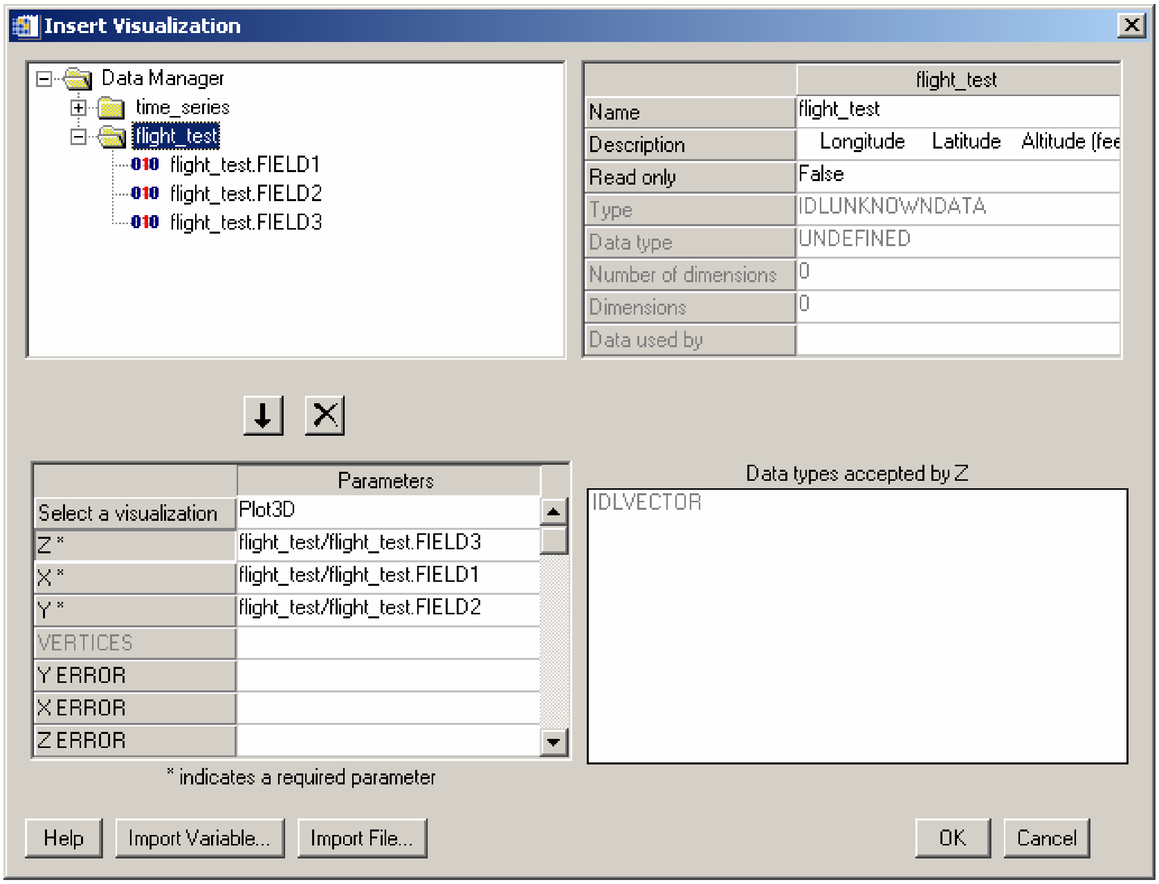
- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\flight_test.txt*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/flight_test.txt*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/flight_test.txt*

The file "*flight_test.txt*" contains standard ASCII text consisting of 3 columns of data separated by whitespace. The first column contains longitude location in units of decimal degrees, the second column contains latitude position, and the third column contains the altitude of the aircraft. Here are the first 2 lines of this ASCII text file :

```
    Longitude      Latitude     Altitude (feet)
     -97.3943       35.4237          798
```

This flight test data will be loaded into a new *iPlot* utility in the same manner as the time-series data that was used in the previous exercise. Start by launching a new blank *IDL iPlot* window :

13. IDL> `iPlot`
14. Select "*File > Open…*" from the menu system.
15. Select the "*flight_test.txt*" file and hit "*Open*".
16. In Step 1 of 3 for the *ASCII Template* wizard, make sure to change the "Data Starts at Line:" field to "2".
17. In the remaining steps of the *ASCII Template* wizard, all settings can be left as their defaults. Simply hit "*Next >>*", followed by "*Finish*".
18. In the *IDL iTools Create Visualization* dialog, press "*Select a visualization and specify parameters*" and hit "*OK*". This will display the *Insert Visualization* dialog.
19. In the *Select a Visualization* row in the lower-left hand corner, click on "*Plot*" and change the graphic type droplist to "*Plot3D*" [Fig. 2-20].
20. Define the X parameter for the Plot3D visualization by clicking on "X" in the table in the lower-left corner, select the "*flight_test.FIELD1*" in the upper-left hand corner, then press the ↓ down arrow button. Do the same with "*flight_test.FIELD2*" as the Y parameter, and "*flight_test.FIELD3*" as the Z parameter [Fig. 2-20].
21. Once this is accomplished, hit "*OK*" to load the 3-Dimensional line plot.



Insert Visualization

- Data Manager
  - time_series
  - flight_test
    - flight_test.FIELD1
    - flight_test.FIELD2
    - flight_test.FIELD3

| | flight_test |
|---|---|
| Name | flight_test |
| Description | Longitude Latitude Altitude (fee |
| Read only | False |
| Type | IDLUNKNOWNDATA |
| Data type | UNDEFINED |
| Number of dimensions | 0 |
| Dimensions | 0 |
| Data used by | |

| | Parameters |
|---|---|
| Select a visualization | Plot3D |
| Z * | flight_test/flight_test.FIELD3 |
| X * | flight_test/flight_test.FIELD1 |
| Y * | flight_test/flight_test.FIELD2 |
| VERTICES | |
| Y ERROR | |
| X ERROR | |
| Z ERROR | |

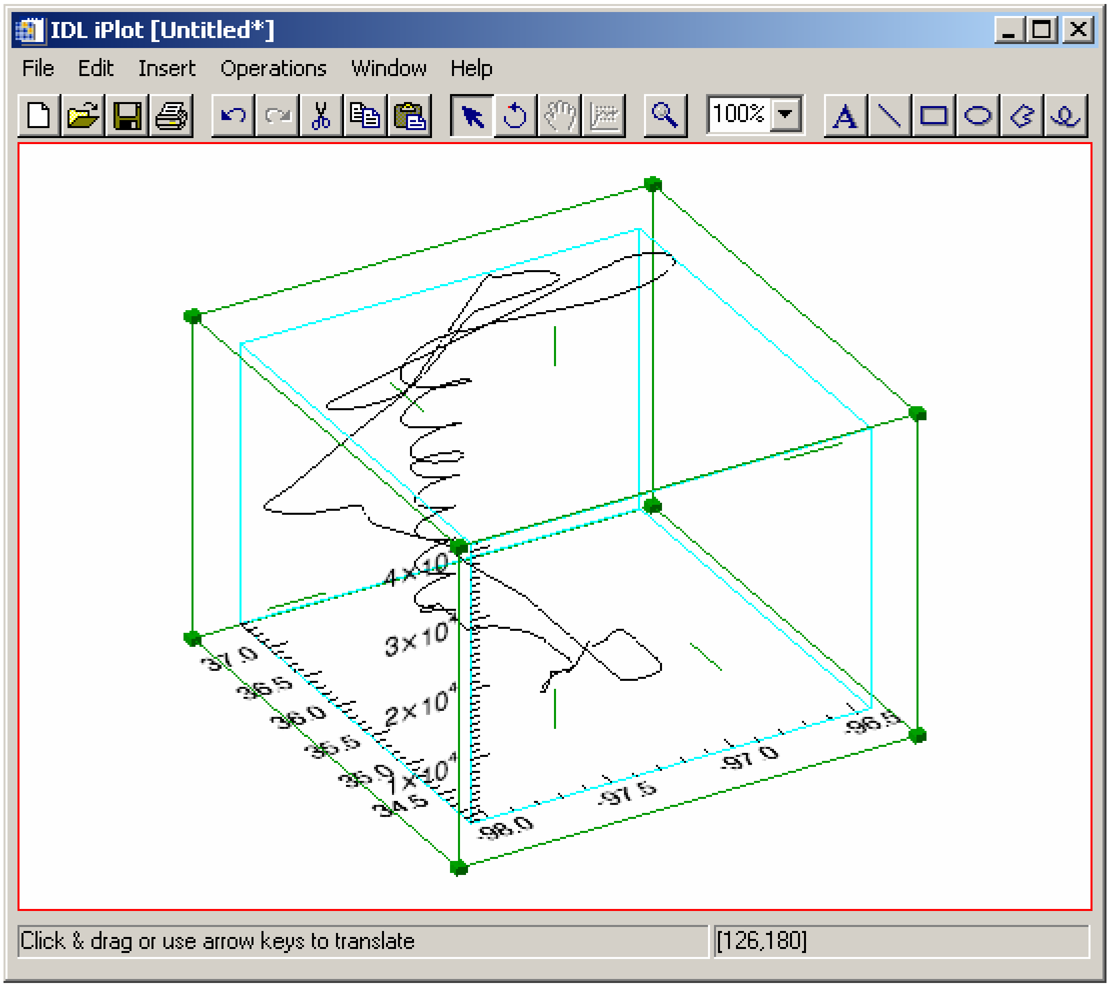
* indicates a required parameter

Data types accepted by Z

IDLVECTOR

***Figure 2-20: Specifying the parameters for the 3-D line plot***

The resulting *IDL iPlot* window should look like Fig. 2-21.



Click & drag or use arrow keys to translate

[126,180]

***Figure 2-21: The 3-Dimensional line plot of flight test data***

All of the same translation, rotate, zoom, and scaling manipulators are available within the iTools system when a 3-Dimensional graphic is displayed, although they act in three dimensions.

22. Once finished viewing the 3-Dimensional line plot, close the *IDL iPlot* window.
23. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```

# Chapter 3:  Contours and Surfaces

## Contouring a 2-Dimensional Dataset

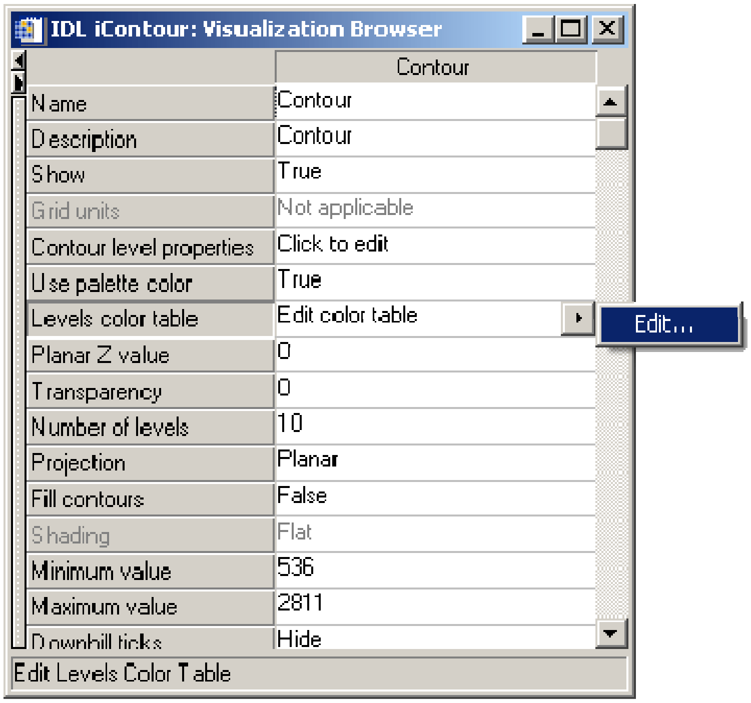
IDL has the ability to create **contour** plots of data stored in a rectangular 2-D array or a set of irregular unstructured points (X, Y, Z).  One of the most appropriate datasets that can be visualized with contours is a digital elevation model, or "DEM", which is a gridded 2-D matrix of elevation values that define the topography for a region.  In the following exercise, DEM data for the Grand Canyon in Arizona will be displayed within the *iContour* utility.  This example data is stored in a file named "*Grand_Canyon_DEM.tif*" that is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:**
  *C:\RSI\IDL##\IDL_QS_Files\data\Grand_Canyon_DEM.tif*
- **UNIX et al.:**
  */usr/local/rsi/idl_#.#/IDL_QS_Files/data/Grand_Canyon_DEM.tif*
- **Mac OS X:**
  */Applications/rsi/idl_#.#/IDL_QS_Files/data/Grand_Canyon_DEM.tif*

Although the file "*Grand_Canyon_DEM.tif*" is in TIFF image format, the pixels of this dataset contain actual elevation data values in units of meters.  Use the following steps to load this dataset into the *iContour* utility :

1. IDL> `iContour`
2. Select "*File > Open…*" from the *IDL iContour* window.
3. Select the "*Grand_Canyon_DEM.tif*" file and hit "*Open*".
4. A contour graphical object with the default properties will be automatically inserted into the *IDL iContour* window.
5. It may help to maximize the iTool window in order to see the graphic, so select "*Window > Zoom on Resize*" from the menu system and maximize the *IDL iContour* window.
6. Select the contour plot object, right-click, and select "*Properties…*".
7. Within the *Visualization Browser* window, click on the box to the right of the "Levels color table" property and select "*Edit…*" [Fig. 3-1].
8. A separate dialog entitled *Palette Editor* will be displayed.  In the "*Load Predefined…*" droplist, select the "*RAINBOW*" color table.
9. Press "*OK*" to dismiss the *Palette Editor* window.
10. At this point the *IDL iContour* window will be brought to the foreground. Relocate the *Visualization Browser* window, which may be behind the *IDL iContour* window.  Change the "Number of levels" property to a value of "*10*", hit *Enter* on the keyboard to complete the change, and close the *Visualization Browser* window [Fig. 3-1].

The DEM data for the Grand Canyon should now be displayed with a sequence of color contours that resembles a topographic map.  The elevation at which the contours are drawn are automatically computed by dividing the data value range (minimum to maximum) into 10 evenly sized levels.  The exact elevation for any given contour, and all other relevant properties, can also be modified by the user if necessary.



| | Contour |
|---|---|
| Name | Contour |
| Description | Contour |
| Show | True |
| Grid units | Not applicable |
| Contour level properties | Click to edit |
| Use palette color | True |
| Levels color table | Edit color table |
| Planar Z value | 0 |
| Transparency | 0 |
| Number of levels | 10 |
| Projection | Planar |
| Fill contours | False |
| Shading | Flat |
| Minimum value | 536 |
| Maximum value | 2811 |
| Downhill ticks | Hide |

Edit Levels Color Table

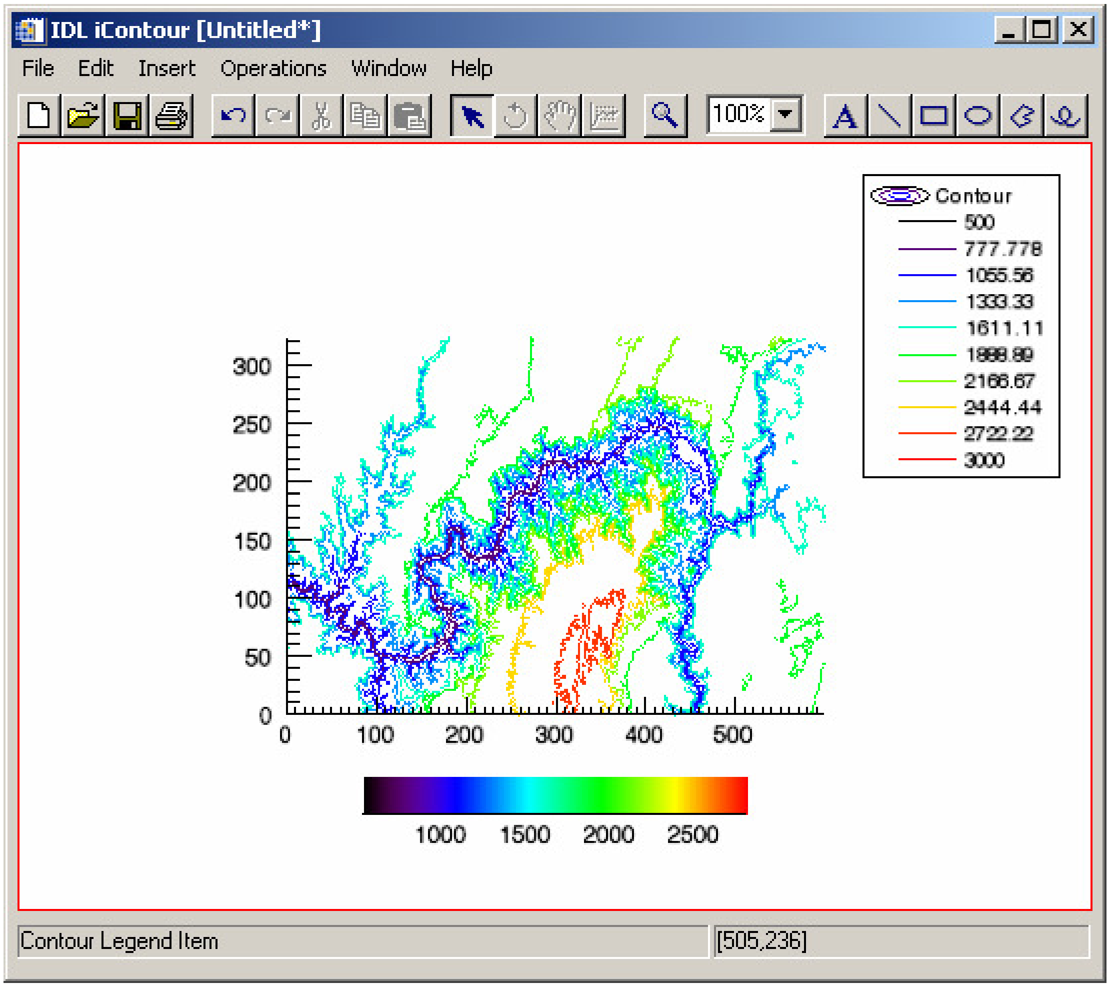
***Figure 3-1: Modifying the properties of the contour plot***

The iTools system also have built-in tools for inserting a colorbar and legend for a contour plot graphic :

11. Make sure the contour plot is selected.
12. Select "*Insert > Colorbar*".  A colorbar showing the color table and range of elevations is automatically inserted below the graphics area.
13. Reselect the contour plot.
14. Select "*Insert > New Legend*".  Re-position the legend that is inserted in the upper-right hand corner.

The final *IDL iContour* window should appear like Fig. 3-2.

15. Once finished viewing the contour plot visualization, close the *IDL iContour* window.



Contour Legend Item [505,236]

***Figure 3-2: Contour plot visualization of DEM data for the Grand Canyon***

## Creating a Surface from a 2-Dimensional Dataset

A **surface** representation provides a way to visualize a gridded 2-Dimensional array of data values as a 3-Dimensional object. The height of the surface at any given X-Y location is defined by the data value (Z). In some cases, the 2-Dimensional data may be in the form of 3 unstructured (X, Y, Z) vectors instead of a gridded matrix. For example, elevation data is quite often measured and stored in irregular X-Y-Z format. In order to visualize this type of data as a surface graphical object, the data must first be **gridded** using an interpolation algorithm that converts the scattered data values into a 2-Dimensional array. Fortunately, the iTools system has a built-in *Gridding Wizard* to help walk the user through the process of gridding irregular X-Y-Z data.

In the following exercise, the X-Y-Z data from the example data file "*elevation.txt*" will be input into the *iSurface* utility. This example data file is located in the "*data*" subfolder of the Quick Start directory :

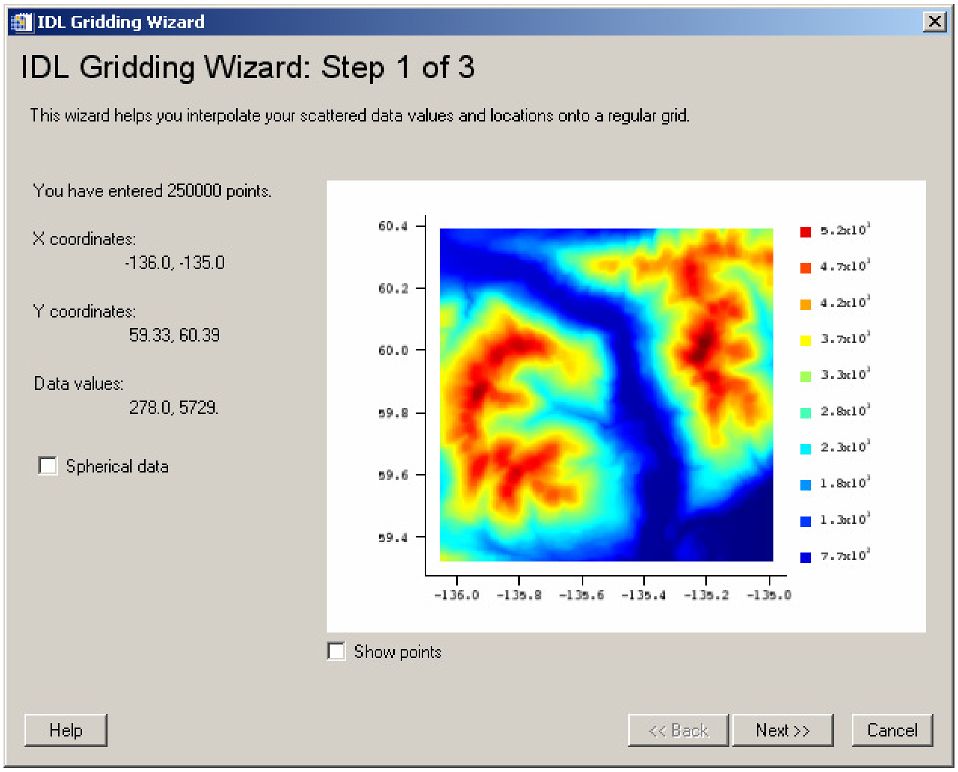
- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\elevation.txt*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/elevation.txt*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/elevation.txt*

The file "*elevation.txt*" contains standard ASCII text consisting of 3 columns of data separated by commas. The first column contains the longitude position, the second column contains latitude, and the third stores the elevation data values. Here are the first 2 lines of this ASCII text file :

```
Longitude,Latitude,Elevation
-136.049,59.3303,3537
```

Use the following steps in order to load the data from the file "*elevation.txt*" into the *iSurface* utility :

24. IDL> `iSurface`
25. Select "*File > Open…*" from the menu system.
26. Select the "*elevation.txt*" file and hit "*Open*".
27. In Step 1 of 3 for the *ASCII Template* wizard, make sure to change the "Data Starts at Line:" field to "*2*".
28. In the remaining steps of the *ASCII Template* wizard, all settings can be left as their defaults. Simply hit "*Next >>*", followed by "*Finish*". There may be a short delay while IDL ingests the data from this ASCII text file.
29. In the *IDL iTools Create Visualization* dialog, select "*Launch the gridding wizard*" and press "*OK*".
30. The *IDL Gridding Wizard* window will appear. In Step 1 of 3, the wizard describes the data that was input and displays a preview of the resulting 2-Dimensional grid. Confirm that this dialog appears as it does in Fig. 3-3 and hit "*Next >>*".
31. In Step 2 of 3, change the "Dimension:" parameter for both the "X coordinates:" and "Y coordinates:" from the default setting of "*25*" to "*500*" [Fig. 3-4] and hit "*Next >>*".
32. In Step 3 of 3, change the "Please choose a gridding method:" droplist setting to "*Linear*" [Fig. 3-5] and hit "*Finish*".



IDL Gridding Wizard

IDL Gridding Wizard: Step 1 of 3

This wizard helps you interpolate your scattered data values and locations onto a regular grid.

You have entered 250000 points.

X coordinates:
-136.0, -135.0

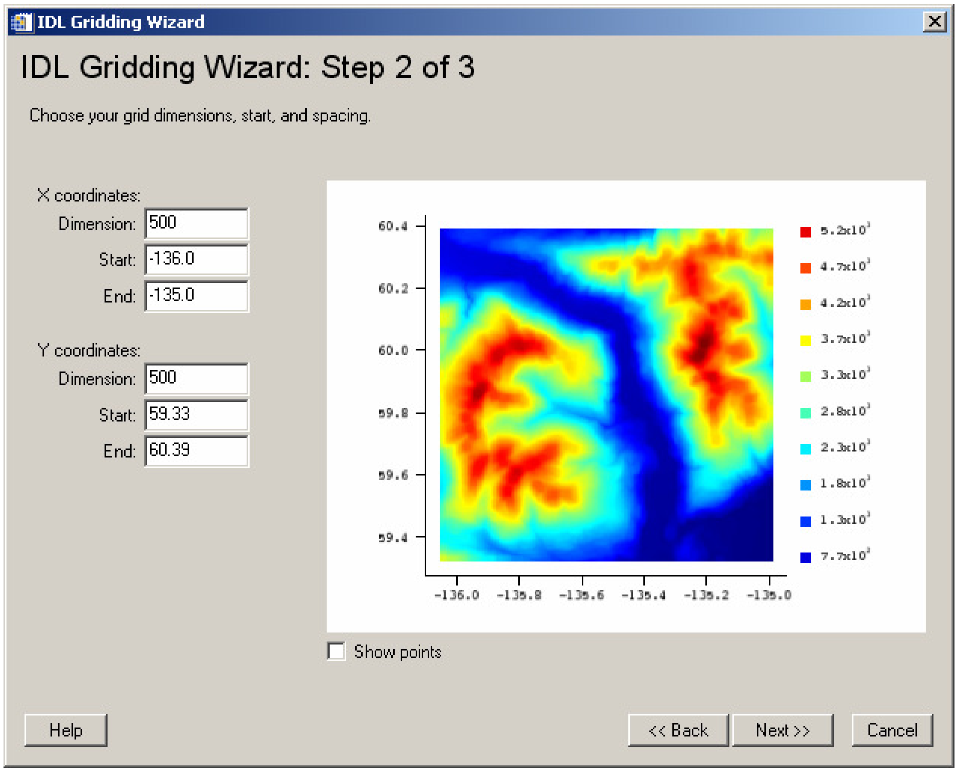
Y coordinates:
59.33, 60.39

Data values:
278.0, 5729.

□ Spherical data

□ Show points

***Figure 3-3: Step 1 of 3 for the IDL Gridding Wizard***



IDL Gridding Wizard

IDL Gridding Wizard: Step 2 of 3

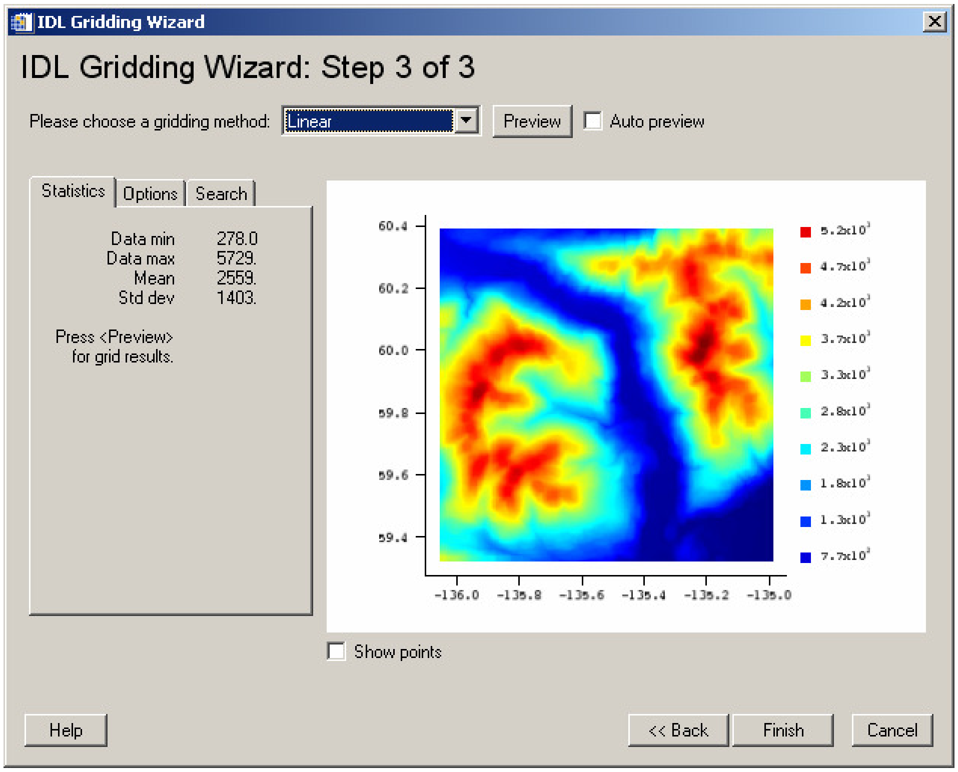
Choose your grid dimensions, start, and spacing.

X coordinates:
Dimension: 500
Start: -136.0
End: -135.0

Y coordinates:
Dimension: 500
Start: 59.33
End: 60.39

□ Show points

***Figure 3-4: Step 2 of 3 for the IDL Gridding Wizard***



IDL Gridding Wizard: Step 3 of 3

Please choose a gridding method: Linear

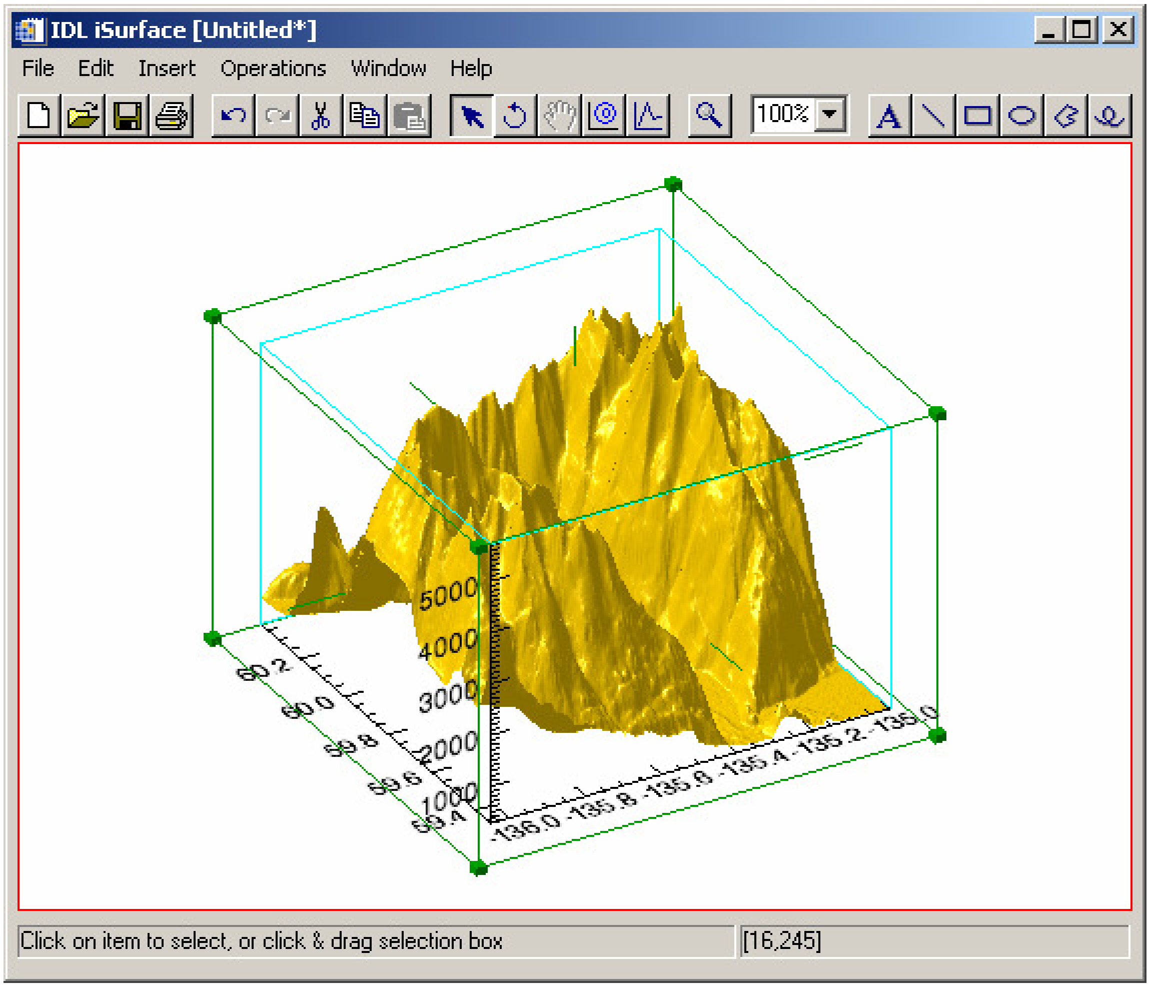
| | |
|---|---|
| Data min | 278.0 |
| Data max | 5729. |
| Mean | 2559. |
| Std dev | 1403. |

Press <Preview> for grid results.

***Figure 3-5: Step 3 of 3 for the IDL Gridding Wizard***

After the *IDL Gridding Wizard* is completed, the selected gridding and interpolation algorithm is used to convert the unstructured X-Y-Z data into a 2-Dimensional grid. Once this processing is finished, a surface object for the elevation data will be displayed within the *IDL iSurface* window [Fig. 3-6]. At this point, it may be beneficial to maximize the *iSurface* utility :

33. From the *IDL iSurface* window menu system, select "*Window > Zoom on Resize*".
34. Maximize the *IDL iSurface* window.



Click on item to select, or click & drag selection box

[16,245]

*__Figure 3-6: Surface visualization of the X-Y-Z format elevation data__*

## Manipulating 3-Dimensional Graphical Objects

The object selection, translation, view zoom, undo/redo, and annotation tools all behave in the same exact manner as they do in 2-Dimensional graphics mode. However, when an iTool is displaying a 3-Dimensional graphical object, the scaling and rotate manipulators behave in a slightly different fashion.

There are 3 types of scaling available for 3-D objects :

- Constrained Scaling
  - Multiple-Axis Scaling (⤢ mouse pointer)
  - Single-Axis Scaling (↔ mouse pointer)
- Unconstrained Scaling (⤧ mouse pointer)

The **multiple-axis scaling** mouse pointer for 3-D objects is obtained when the mouse pointer is positioned over a corner of a 3-D object's data space. Dragging the constrained scaling pointer scales the object a fixed distance along all axes in the direction of the drag. In contrast, the **single-axis scaling** is accessed when the mouse pointer is positioned over an axis "*whisker*", which are small linear hash marks on the edges of the data space in the direction of the three orthogonal axes. Dragging along an axis *whisker* scales the object only in the direction of the axis (and the arrows of the mouse pointer). Finally, unconstrained scaling of 3-D objects can be performed when a single side of the data space bounding box is dragged, scaling the object along the dimensions of the selected side.

1. Experiment with the 3 different styles of scaling with the 3-D surface object within the *IDL iSurface* window.
2. Use the single-axis constrained scaling along the Z-dimension whisker to adjust the vertical exaggeration of the surface object. It may be appropriate to reduce the vertical exaggeration in order to make a more realistic visualization of the terrain.
3. If at any time an undesirable scaling operation is performed, remember that the Undo tool can be used to reverse the operation.

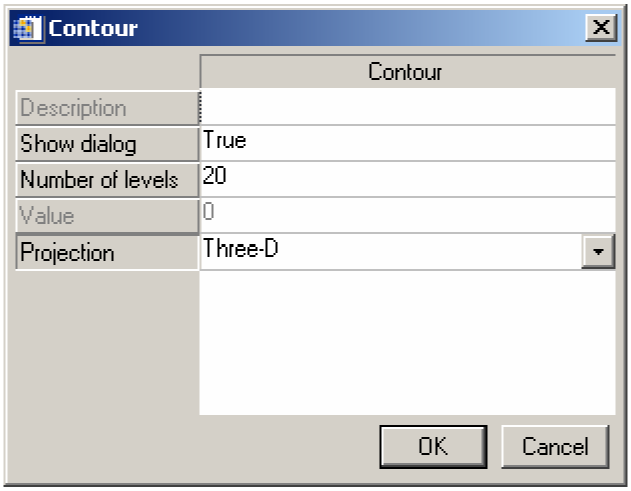
The rotation of 3-D objects in the iTools system can be done in a freehand (unconstrained) fashion or constrained along each of the three orthogonal axes. When a 3-D object is selected within rotation mode, a **rotation sphere** consisting of circular X, Y, and Z dimension axes is displayed around the object. To rotate an object in a constrained fashion, position the mouse cursor over one of the three axis circles in the rotation sphere until the cursor changes to a constrained rotation ↻ pointer and drag in the desired direction. To rotate an object in an freehand fashion simply position the mouse pointer anywhere on the object until the cursor changes to an unconstrained rotation ↻ pointer and drag in any arbitrary direction.

4. Experiment with both constrained and freehand rotation of the 3-D surface object.

## Adding Contours and a Texture Map

One of the benefits of the object-oriented design of the iTools system is the ability to make composite visualizations that include more than one graphic. The data that is currently being used to define the surface graphical object is digital elevation data, and as the previous exercise demonstrated it is quite appropriate to visualize DEM data using a contour plot. Fortunately, the iTools has a built-in operation that allows the user to automatically add contours to the current visualization utility.

1. Using the Select/Translate ↖ arrow, select the surface object within the *IDL iSurface* window.
2. While the surface object is highlighted, select "*Operations > Contour*" from the menu system. The *Contour* dialog will appear [Fig. 3-7].
3. Within the *Contour* dialog, change the "Number of levels" parameter to "*20*" [Fig. 3-7].
4. Click in the box to the right of "Projection" parameter and change its setting to "*Three-D*" [Fig. 3-7].
5. Once these changes have been made, press the "*OK*" button. This will insert the contour graphical object within the same *iSurface* utility as the existing surface object.



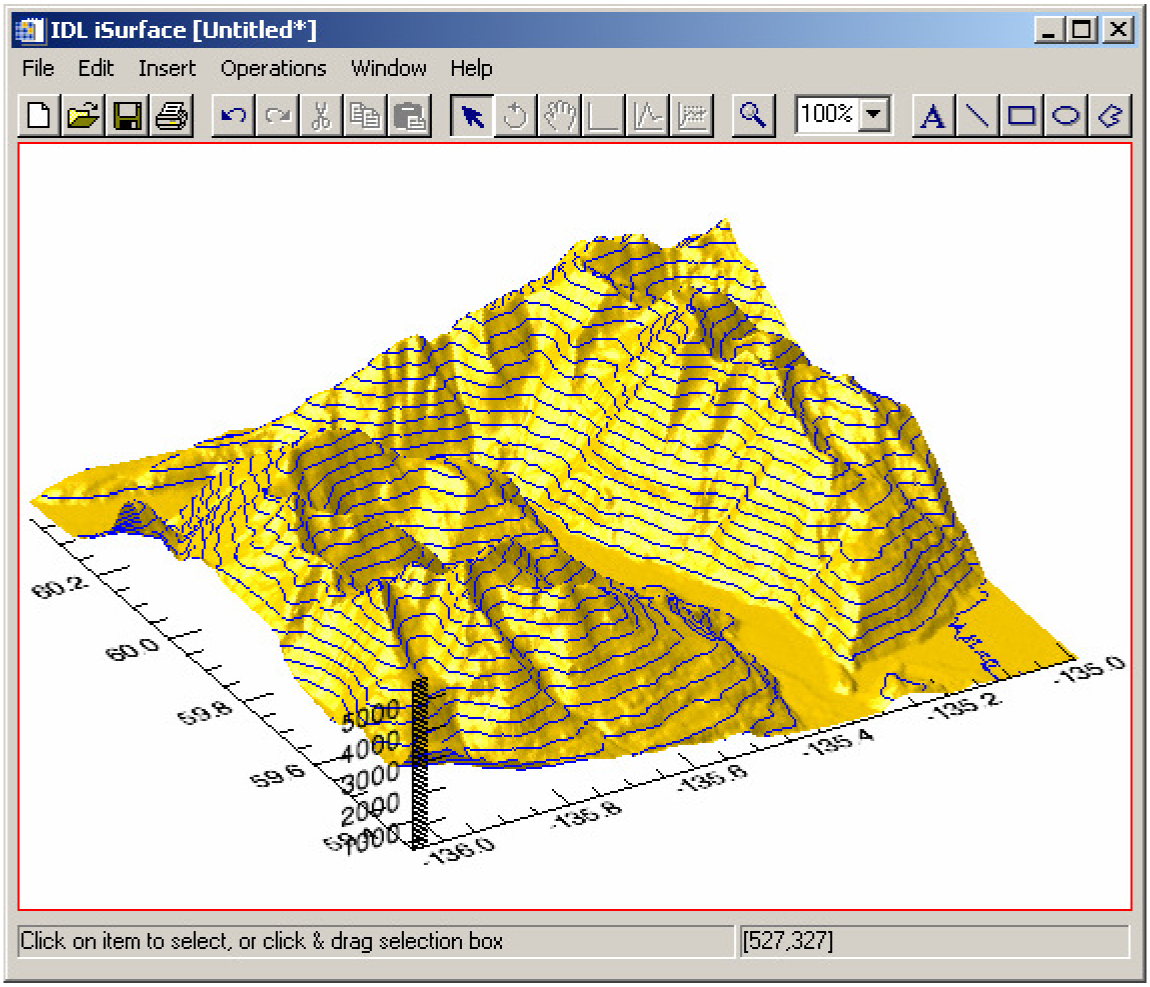
| | Contour |
|---|---|
| Description | |
| Show dialog | True |
| Number of levels | 20 |
| Value | 0 |
| Projection | Three-D |

***Figure 3-7: Options when inserting a contour visualization***

In order to get a better view of these contours, it may be beneficial to change their color so it contrasts with the surface.

6. Select "*Window > Visualization Browser…*" from the menu system.
7. Make sure the Contour object is selected. Within the property sheet along the right-hand side, click on the box to the right of the "Contour level properties" item and select "*Edit…*".
8. A separate window entitled "*Contour Levels*" will appear. The first column of this dialog can be used to adjust the properties of all of the contours. Change the "Color" property for "All Levels" to bright blue and hit "*OK*" to dismiss this dialog.
9. Close the *Visualization Browser* window.

The resulting visualization should look similar to Fig. 3-8.



Click on item to select, or click & drag selection box

[527,327]

***Figure 3-8: Surface with contours draped on top***

In addition to the ability to drape contours on top of a surface, IDL also has the capability to drape images on top of 3-Dimensional graphical objects. This visualization technique involves the use of what's known as a **texture map**.

The example "*data*" subfolder of the Quick Start directory contains a PNG image file of a satellite image that covers the same exact area as the elevation data :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\satellite.png*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/satellite.png*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/satellite.png*

This image can be draped on top of the surface as a texture map using the following steps :

10. First, remove the contours so they will not hinder the display of the texture map image. This can be accomplished by opening the *Visualization Browser*, right-click on the Contour object, and select "*Delete*".
11. Close the *Visualization Browser* window.
12. Within the *IDL iSurface* window select the surface object, right-click, and select "*Parameters…*".

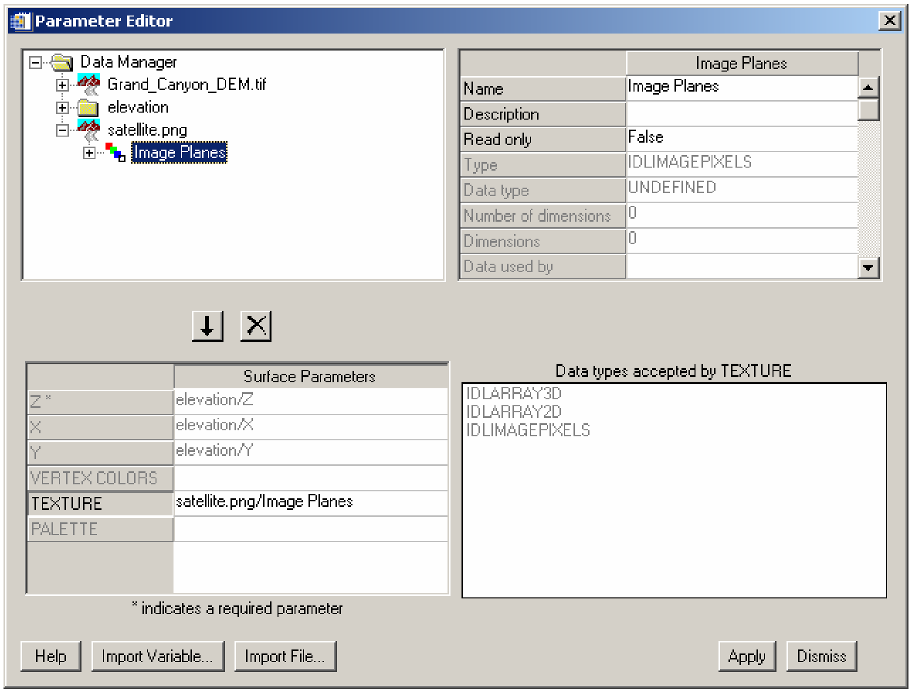
The *Parameter Editor* dialog will appear. In the lower-left hand corner the parameters that define the current surface object are listed. Currently, only the "Z", "X", and "Y" parameters has been defined using the elevation data. Notice that there is also a parameter called "TEXTURE", and it is this parameter that can be used to specify a texture map image to drape on top of the surface. However, the "*satellite.png*" file must first be opened and input into the iTools system.

13. From the *Parameter Editor* window, press the "*Import File…*" button. This will launch the *File Import* dialog.
14. Within the *File Import* dialog, press the file selection 📂 button, navigate to the appropriate location, select the file "*satellite.png*" and hit "*Open*".
15. Once this file is selected, the *File Import* wizard should recognize the format as being "*Portable Network Graphics*".
16. Hit "*OK*" to dismiss the *File Import* dialog.
17. Back within the *Parameter Editor*, expand the new "*satellite.png*" image item so the "Image Planes" contained within are visible. This image is in 24-bit RGB true color mode, so it has 3 image planes.
18. Select the "Texture" row located in the lower-left table, and then double-click the "Image Planes" from the Data Manager in the upper-left corner of the window in order to load this image as the surface's texture map [Fig. 3-9].
19. Once this is accomplished, press "*Apply*" followed by "*Dismiss*" to complete the changes.

The resulting visualization should look similar to Fig. 3-10.

20. Once finished viewing the surface visualization, close the *IDL iSurface* utility.
21. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```



- Data Manager
  - Grand_Canyon_DEM.tif
  - elevation
  - satellite.png
    - Image Planes

| | Image Planes |
|---|---|
| Name | Image Planes |
| Description | |
| Read only | False |
| Type | IDLIMAGEPIXELS |
| Data type | UNDEFINED |
| Number of dimensions | 0 |
| Dimensions | 0 |
| Data used by | |

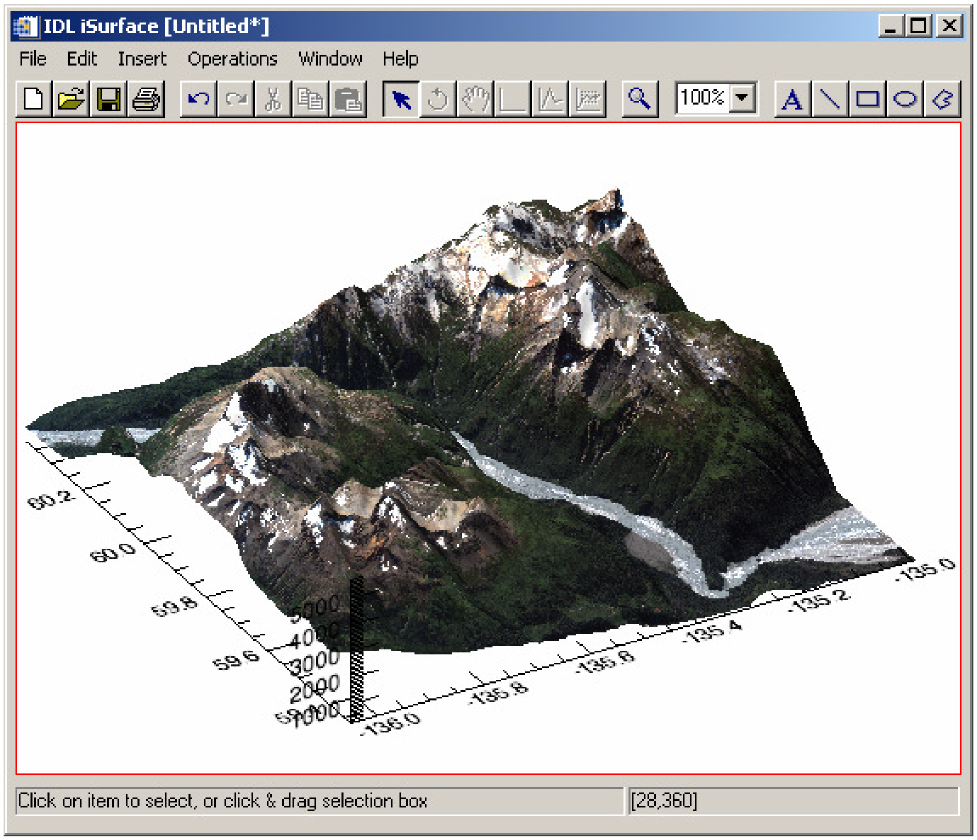
| | Surface Parameters |
|---|---|
| Z * | elevation/Z |
| X | elevation/X |
| Y | elevation/Y |
| VERTEX COLORS | |
| TEXTURE | satellite.png/Image Planes |
| PALETTE | |

* indicates a required parameter

Data types accepted by TEXTURE

IDLARRAY3D
IDLARRAY2D
IDLIMAGEPIXELS

***Figure 3-9: Using the satellite image to define a texture map***



Click on item to select, or click & drag selection box

[28,360]

***Figure 3-10: Surface with texture map draped on top***

# Chapter 4:  Working with Images

## Image Display, Enhancement, and Regions of Interest

The IDL *iImage* tool can be used to display gridded 2-Dimensional arrays of data in image form.  The *iImage* utility has built-in support for the input of images in BMP, DICOM, GIF, JPEG, JPEG2000, PICT, PNG, and TIFF formats.  Once an image is loaded into the *iImage* tool there are several options for visualizing, manipulating, and processing the imagery.  Furthermore, the library of IDL routines contains a rich suite of image processing algorithms.

In the following exercise, the image data from the example data file "*MRI.dcm*" will be input into the *iImage* utility.  This example data file is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\MRI.dcm*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/MRI.dcm*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/MRI.dcm*

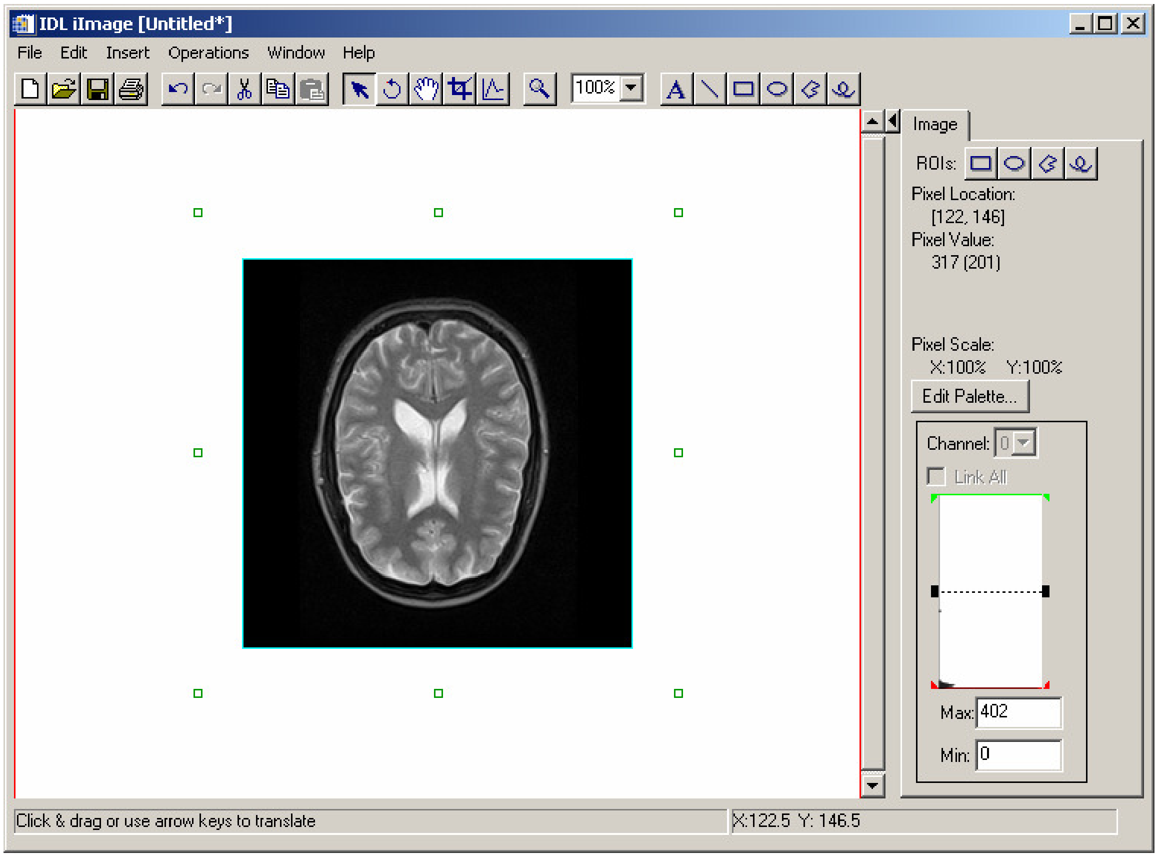
The file "*MRI.dcm*" is in DICOM format, which is the industry standard for storing imagery and diagnostic patient information in the radiology community.  This file contains a grayscale (black-and-white) image of a MRI exam involving a single slice through a human brain.  The MRI image can be loaded into the *iImage* utility using the following steps :

16. From the IDL Development Environment window, select "*File > New > Visualization > iImage*".  A separate *IDL iImage* window will appear.
17. Click on the open file 📂 button on the toolbar of the *IDL iImage* window.
18. Select the "*MRI.dcm*" file and hit "*Open*".  An image graphical object will be automatically inserted into the utility.

Along the right-hand side of the *iImage* utility is a panel that is not present on the other iTools utilities.  This panel contains cursor query, histogram manipulation, and ROIs (Regions Of Interest) tools specific to the image graphical object.

19. While the image object is selected and the iTool is in Select/Translate arrow mode, move the mouse cursor over the image and watch the "Pixel Location:" and "Pixel Value:" fields update.

The origin location for the image (0,0) is considered to be the pixel in the lower-left hand corner.  Within the "Pixel Value:" field there is actually 2 numbers displayed: the first being the actual image data pixel value, and the second within parentheses is the scaled display value [Fig. 4-1].  Since the computer monitor can only display 256 (or sometimes less) levels of discrete brightness for each color channel, all image data is scaled to fit within the range of 0 → 255 automatically by the *iImage* utility if it does not already.  The "Min:" and "Max:" fields within the *Image* panel show that the actual data range for this image is 0 → 402, so these image pixel data values must be scaled in a linear fashion to fit within the corresponding display range of 0 → 255.



Pixel Location:
[122, 146]
Pixel Value:
317 (201)
Pixel Scale:
X:100% Y:100%
Max: 402
Min: 0
Click & drag or use arrow keys to translate
X:122.5 Y: 146.5

***Figure 4-1: The iImage utility displaying a MRI image of a human brain***

The *Image* panel also displays a small **histogram** (density) plot of the pixel data values displayed on its side [Fig. 4-1]. The histogram plot ranges from the minimum pixel data value on the bottom to the maximum pixel value on the top. Since the current image is dominated by very dark pixels (i.e. the black area around the border), the scaling of this histogram is skewed and difficult to visualize in such a small display window. Fortunately, the iTools system has a built-in operation for creating a separate *iPlot* utility that displays the density distribution for data objects using a histogram plot.

20. Make sure the image object is selected.
21. From the *IDL iImage* window menu system, select "*Operations* > *Histogram*".

A separate *IDL iPlot* window appears, and the histogram density plot for the image is displayed in a more robust fashion. Notice that most of the pixels in the image (more than 40,000) have a very low (dark) data value.

22. Once finished viewing the image histogram, close the separate *IDL iPlot* window.

The image histogram window within the *Image* panel has 3 horizontal bars that the user can interactively click-and-drag in order to modify the input data range that is used to compute the output scaled image [Fig. 4-1]. The red bar is used to designate the minimum threshold value, the green bar corresponds to the maximum threshold value, and the black bar in the middle is used to move the current range up and down within the image histogram. In other words, all input image data pixel values that are equal to or less than the current location of the red bar are saturated to an output display value of 0 (black), all input image data pixel values that are equal to or greater than the current location of the green bar are saturated to an output display value of 255 (white), and all of the pixel values in between are scaled in a linear fashion from 1 → 254.

23. Click on the green maximum image stretch bar and move it down within the histogram plot window. Notice how the "Max:" field within the Image panel updates with the selected pixel data value, and the image display is automatically updated based on the stretch modification.
24. Click on the red minimum image stretch bar and move it up within the histogram plot window.
25. Finally, click on the black range location bar and move it up and down to re-position the current image stretch.
26. Once finished experimenting with the histogram manipulation tool, reset the stretch to its original state by moving the green and red bars to the top and bottom of the window, respectively.

The Image panel within the iImage utility also has built-in **regions of interest** (ROI) tools that can be used to specify certain areas within an image for processing and analysis.

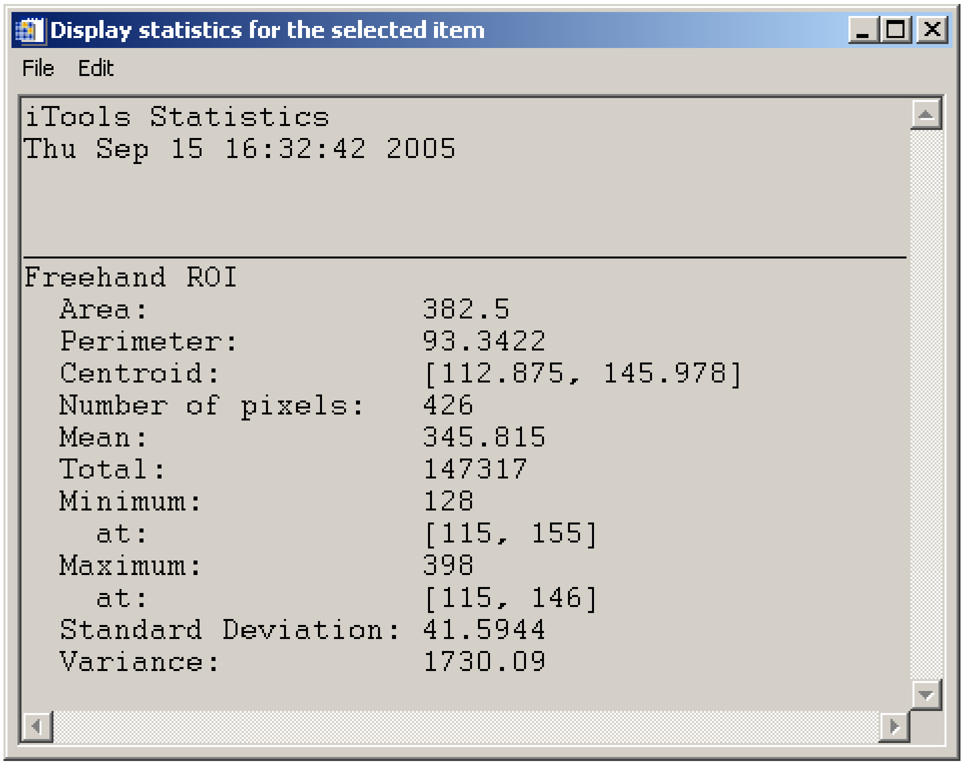
27. While the image object is selected, click on the freehand ROI definition button to enter into ROI definition mode.
28. Once the mouse cursor is positioned over the image object it will change to the ROI definition pencil pointer. Click-and-drag with the mouse to draw a ROI on top of a feature within the MRI image. When the mouse button is released the ROI definition will complete to a solid polygon, and the new ROI will be highlighted with small green boxes.

Once the ROI definition is complete, the object can be resized and moved by clicking on or within the green highlight boxes. Now that a region of interest has been defined, the user can obtain useful information about the image pixels contained within the region such as statistics :

29. While the ROI object is highlighted, select "*Operations > Statistics…*" from the menu system.

A separate dialog entitled "*Display statistics for the selected item*" will appear, which lists useful statistical information on the image data contained within the region of interest [Fig. 4-2].

**Note:** The exact statistical information that is reported will vary from what appears in Fig. 4-2 since the ROI is drawn in an arbitrary fashion.



```
iTools Statistics
Thu Sep 15 16:32:42 2005

Freehand ROI
  Area:                 382.5
  Perimeter:            93.3422
  Centroid:             [112.875, 145.978]
  Number of pixels:     426
  Mean:                 345.815
  Total:                147317
  Minimum:              128
    at:                 [115, 155]
  Maximum:              398
    at:                 [115, 146]
  Standard Deviation:   41.5944
  Variance:             1730.09
```

***Figure 4-2: Statistical information on the image data within the defined ROI***

30. Once finished viewing the statistical information, close the "*Display statistics for the selected item*" dialog.
31. While the ROI object is selected and highlighted, right-click on it and select "*Delete*" in order to remove it from the current visualization.

## Color Palettes, Filtering, and Morphology

Currently the MRI image is displayed with a simple grayscale color palette that was read-in from the DICOM file on disk. This palette displays the image in a very simple black-and-white fashion. In addition to the standard grayscale color palette, IDL also has a number of pre-built color tables that can be applied to singe channel images in order to give them a colored appearance.

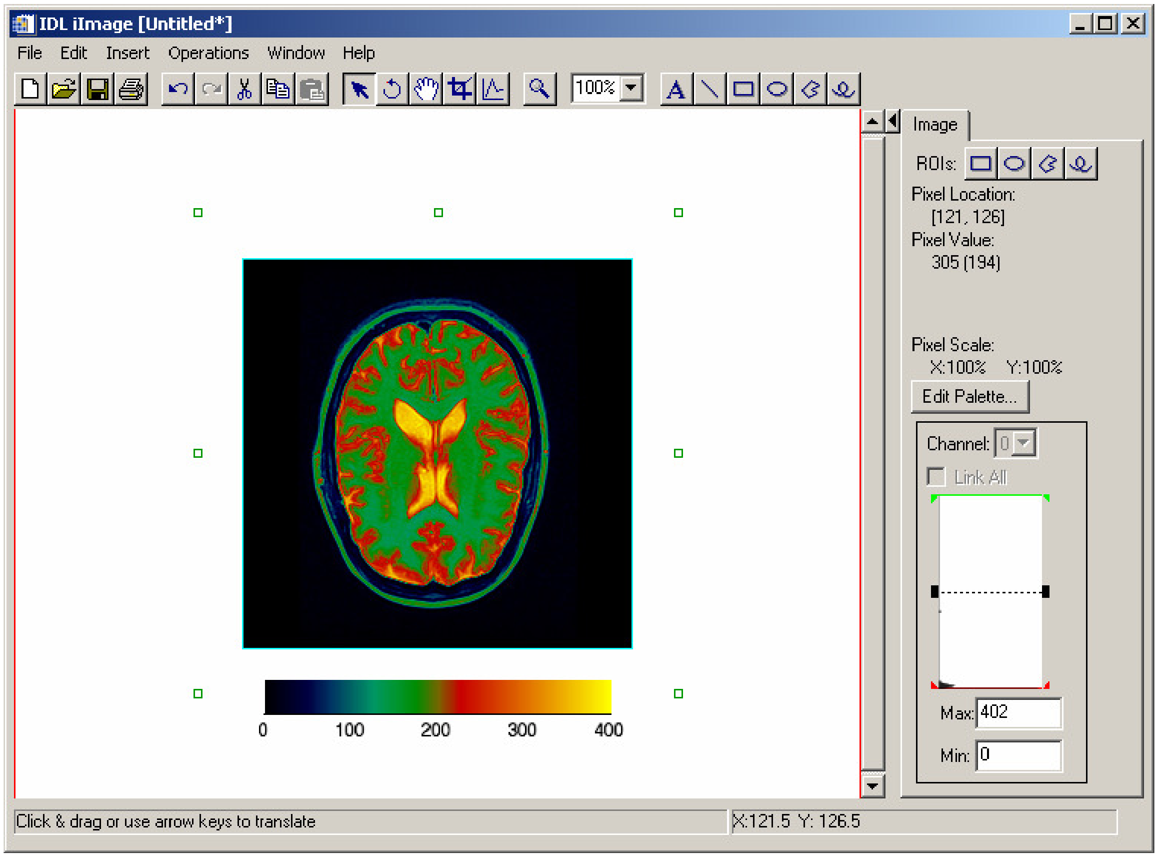
1. Select the image, then press the "*Edit Palette…*" button on the *Image* panel.
2. A separate *Palette Editor* dialog will appear. Within this window press the "*Load Predefined…*" droplist and select the "*BLUE/GREEN/RED/YELLOW*" color palette.
3. Press "*OK*" to dismiss the *Palette Editor* dialog.

Instead of displaying the image with the input Black → Gray → White color palette, the *IDL iImage* utility is now applying the selected color table and displaying the image brightness values through the color range of Black → Blue → Green → Red → Yellow. This allows the user to visualize the features within the image in color and can help highlight aspects of the image that were not readily visible in standard black-and-white mode.

4. Experiment with moving the green, red, and black stretch manipulation bars on the image histogram once again to see the effect the new color palette has on the image display.
5. Once finished experimenting with the histogram manipulation tool, reset the stretch to its original state by moving the green and red bars to the top and bottom of the window, respectively.

At this point it may be useful to insert a colorbar into the current visualization so the user can see how the input image pixel data range maps to the colors that are displayed.

6. Make sure the image object is selected, and from the menu system select "*Insert > Colorbar*" [Fig. 4-3].



Pixel Location: [121, 126]
Pixel Value: 305 (194)
Pixel Scale: X:100% Y:100%
Channel: 0
Max: 402
Min: 0
Click & drag or use arrow keys to translate
X:121.5 Y: 126.5

***Figure 4-3: The MRI image displayed with a color palette***

In addition to the extraction of statistics, there are a number of other standard image processing tools found within the *Operations* menu. The operations available for an image object include the ability to apply a filter, highlight shapes using morphological operators, and rotate or flip the image.

7. Make sure the image object is highlighted and select "*Operations > Filter > Smooth*" from the menu system.

8. Within the *Smooth* dialog, leave all parameters set to their default values and simply press the "*OK*" button.

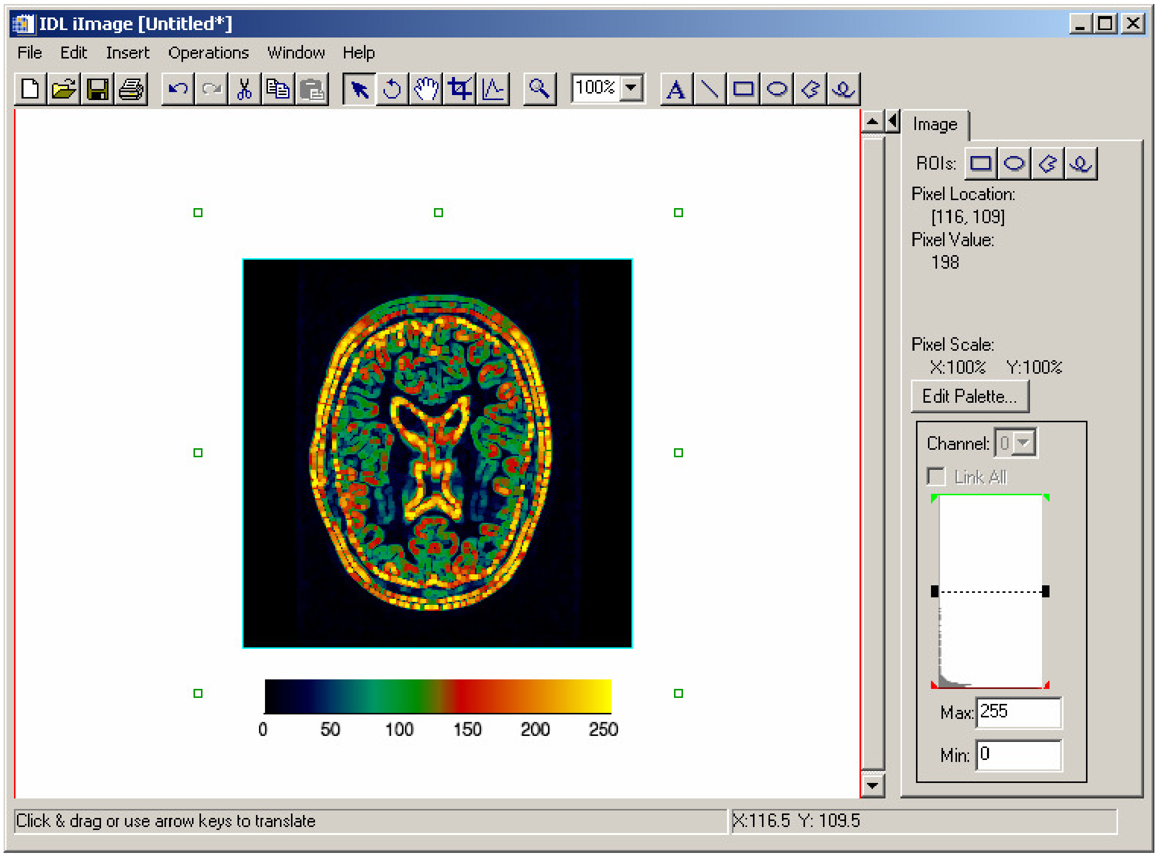
This operation will apply a smoothing image filter, which helps to remove unwanted noise using a weighted average. Notice how the image histogram and colorbar automatically update based on the image that is returned from this operation.

9. Select "*Operations* > *Filter* > *Sobel Filter*" from the menu system.

The Sobel filter operation uses the Sobel edge enhancement algorithm to detect and highlight edges within the image.

10. Finally, select "*Operations* > *Morph* > *Morph Open*" from the menu system.
11. Within the *Morph Open* dialog, leave all parameters set to their default values and simply press the "*OK*" button.

The opening morphological operator removes noise from an image while maintaining the overall sizes of objects in the foreground. The resulting visualization should look similar to Fig. 4-4.



Pixel Location:
[116, 109]
Pixel Value:
198
Pixel Scale:
X:100% Y:100%
Channel: 0
Max: 255
Min: 0

Click & drag or use arrow keys to translate
X:116.5 Y: 109.5

***Figure 4-4: The result of applying Smooth, Sobel and Morph Open operations***

12. Once finished viewing the MRI image visualization, close the *IDL iImage* window.

## Line Profiles, Contouring, and Advanced Processing

In the following exercise, the image data from the example data file "*nebula.jpg*" will be input into the *iImage* utility. This example data file is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\nebula.jpg*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/nebula.jpg*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/nebula.jpg*

The file is in JPEG format and contains an image of the famous Ring Nebula taken by the Hubble Space Telescope. In contrast to the single channel MRI image used in the last exercise, this dataset is a 3 channel (Red, Green, Blue) 24-bit true color image. In this case, the three color channels are displayed together in order to visualize the color image. Use the following steps to load the "*Nebula.jpg*" image into a new *iImage* utility :

1. IDL> `iImage`
2. Click on the open file 📂 button on the toolbar of the *IDL iImage* window.
3. Select the "*Nebula.jpg*" file and press "*Open*".

The color image be automatically displayed within the *IDL iImage* window.

4. Move the mouse cursor over the image and notice how the "Pixel Value:" field in the *Image* panel is now reporting 3 pixel values: one each for the R (red), G (green), and B (blue) image channels that are combined together to make the color display.

In this case, the image data values already fall within the display range of 0 → 255, so no scaling is needed.

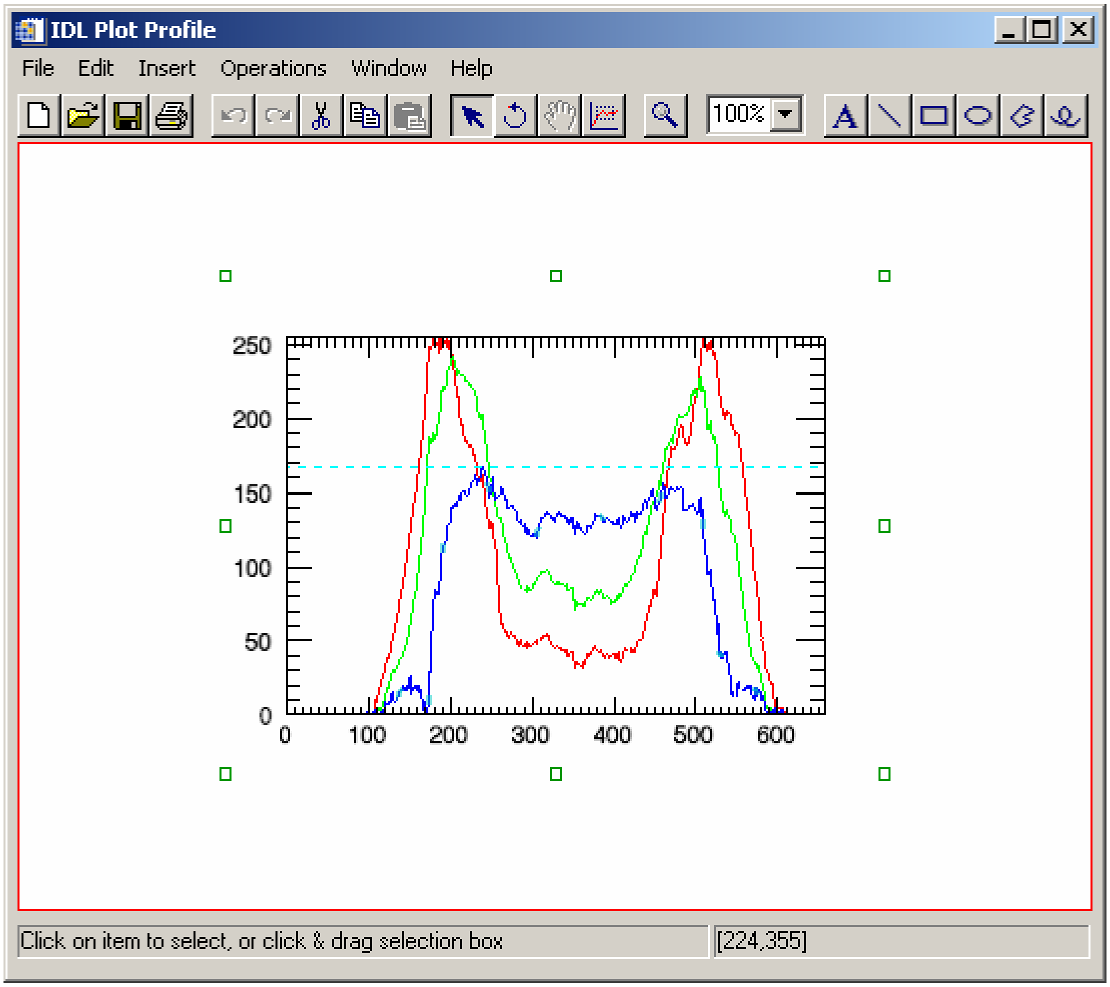
The *iImage* utility has a built-in line profile tool that will plot image pixel values along a user-defined transect. This tool will automatically launch a secondary *iPlot* utility containing line plots for the pixel values along the selected line for each of the three red, green, and blue image channels.

5. While the image object is selected, click on the line profile 📈 button on the toolbar.

Once the mouse cursor is positioned over the image object it will change to the line profile pointer. In order to define a line profile, the user must left-click at the desired starting location for the transect, hold down the mouse button, drag to the desired ending location for the transect, and release the mouse button in order to complete the profile. Once this is accomplished a separate *iPlot* utility will be launched containing the 3 line profiles (one for each color channel).

6. Using the mouse, click-and-drag from the upper-right hand corner of the image to the lower-left hand corner.

The resulting *IDL Line Profile* window should look similar to Fig. 4-5.



Click on item to select, or click & drag selection box

[224,355]

***Figure 4-5: Line profiles extracted from the color JPEG image***

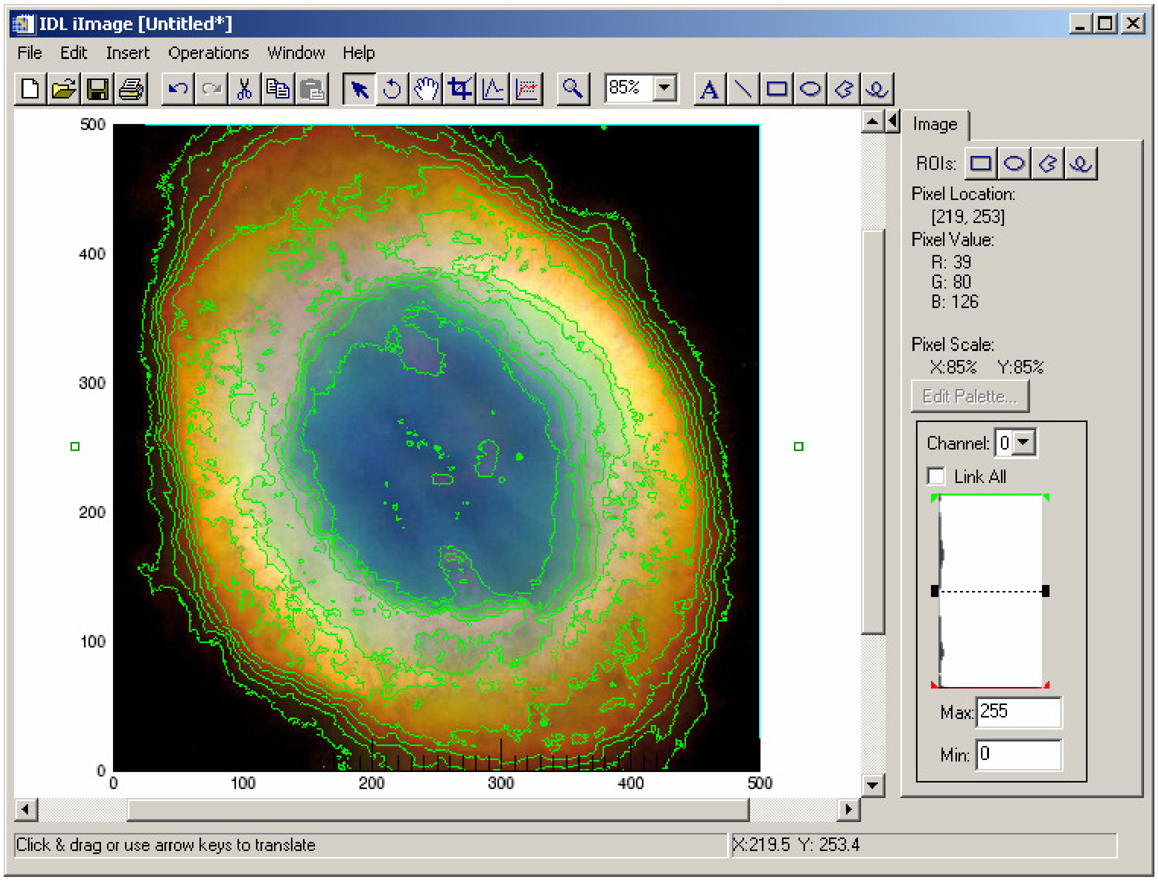
7. Once finished viewing the line profiles, close the *IDL Line Profile* window.
8. To remove the line from the current visualization, click on it and press the *Delete* key on the keyboard.

Since image data is actually a gridded 2-Dimensional array of pixel values, it can also be visualized using some of the other graphical objects available in the iTools system.  For example, it is quite easy to insert a contour plot into the existing utility in order to make a composite visualization.

9. Make sure the image object is selected.
10. From the menu system, select "*Operations* > *Contour*".
11. Within the Contour dialog box, change the "Number of levels" parameter to "*10*".
12. Press the "*OK*" button to dismiss the *Contour* dialog box.
13. Select "*Window* > *Visualization Browser…*" to view the property sheet for the new contour object.
14. Within the *Visualization Browser* window, make sure the contour object is selected, click on the box to the right of the "Contour level properties" item, and select "*Edit…*".

15. Within the first "All Levels" column, change the "Color" property to bright green and hit "*OK*".
16. Close the *Visualization Browser* window.

The resulting visualization should look similar to Fig. 4-6.



Pixel Location:
[219, 253]
Pixel Value:
R: 39
G: 80
B: 126

Pixel Scale:
X:85% Y:85%

Channel: 0
□ Link All

Max: 255
Min: 0

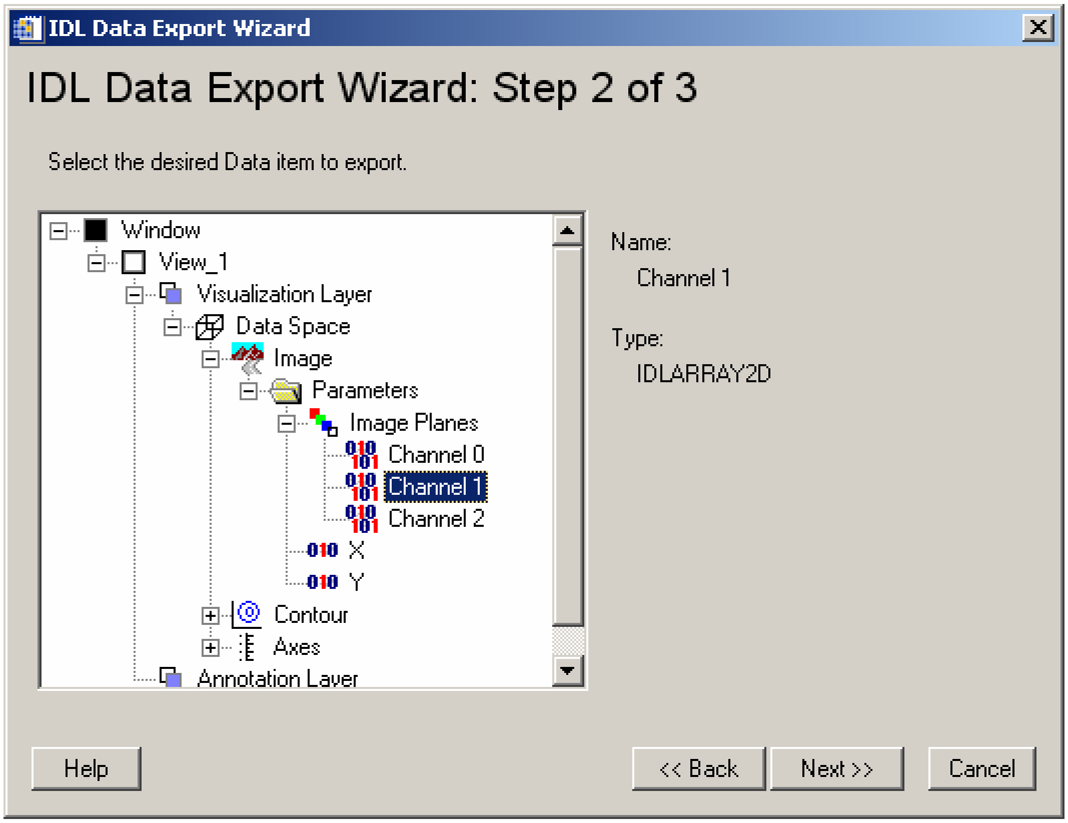
Click & drag or use arrow keys to translate
X:219.5 Y: 253.4

***Figure 4-6: Contour map overlaid on top of the image display***

In addition to the analysis capabilities exposed within the iTools system under the *Operations* menu, the IDL library includes hundreds of routines that provide advanced processing capabilities. Some of these processing algorithms are very specialized, and it is not appropriate to place an item within the *Operations* menu in the iTools system to run all of these more advanced tools. Fortunately, the iTools system was designed in a manner that allows the user to easily pass data back-and-forth between an iTool utility and the IDL> command prompt.

In the following exercise, the green channel for the current image will be exported from the iImage utility to the IDL> command prompt level within the IDL Development Environment for specialized processing. First, the green channel image will be smoothed using the SMOOTH function in an effort to minimize noise. Then, the *WATERSHED* function from the IDL library will be used to segment the image into watershed regions and their boundaries. The watershed algorithm considers the grayscale image as a surface, where each local minimum can be thought of as the point to which water falling on the surrounding region drains. The boundaries of the watersheds lie on the tops of the ridges. This operator labels each watershed region with a unique index pixel value, and sets the boundaries to zero so they are readily visible.

17. From the *IDL iImage* window menu system, select "*File > Export…*".
18. In Step 1 of 3 for the *IDL Data Export Wizard*, select "To an IDL Variable" and hit "*Next >>*".
19. In Step 2 of 3, expand the object hierarchy tree until the green image plane "*Channel 1*" can be selected [Fig. 4-7].



IDL Data Export Wizard

IDL Data Export Wizard: Step 2 of 3

Select the desired Data item to export.

Name:
Channel 1

Type:
IDLARRAY2D

***Figure 4-7: Selecting the green channel image for export to IDL***

20. Press "*Next >>*", and within Step 3 of 3 leave the "IDL Variable Name:" field set to the default "*Channel_1*" and hit "*Finish*".
21. Bring-up the IDL Development Environment window so the IDL> command prompt can be accessed. Execute the following statements at the IDL> command prompt.
22. IDL> `HELP, Channel_1`

This will report to the output log that a 2-Dimensional byte array with the same spatial size as the Ring Nebula image loaded into the *iImage* utility now exists within IDL's main memory space :

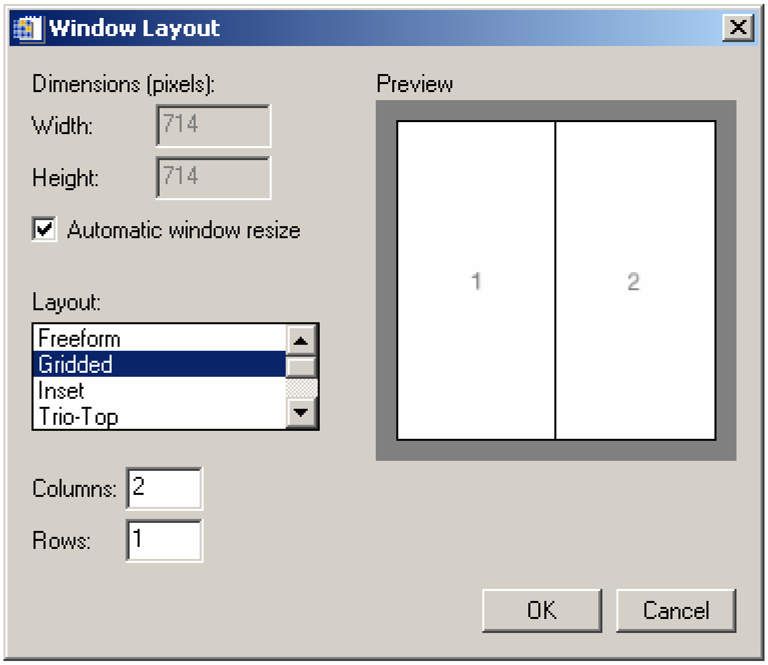
```
CHANNEL_1       BYTE      = Array[500, 500]
```

This is a duplicate copy of the same data that is stored in the *iImage* utility, so it can be acted upon in a manner independent of the image that is currently being displayed. Now execute the advanced watershed analysis :

23. IDL> `smoothed = SMOOTH (Channel_1, 9, /EDGE_TRUNCATE)`
24. IDL> `segmented = WATERSHED (smoothed, CONNECTIVITY=8)`

Once this is accomplished, a new variable named "*segmented*" exists at the main IDL level which contains the resulting watershed image. This new image can be loaded into the existing *iImage* utility and compared to the original image using the following steps :

25. From the *IDL iImage* menu system, select "*Window > Layout…*". This will bring-up a separate dialog entitled "*Window Layout*" [Fig. 4-8].
26. Within the *Window Layout* dialog, change the "Columns:" field to "*2*" and hit *Enter* on the keyboard. The *Preview* pane will update to show the new viewplane which will be inserted into the existing *iImage* utility [Fig. 4-8].



Dimensions (pixels):
Width: 714
Height: 714
☑ Automatic window resize
Layout:
Freeform
Gridded
Inset
Trio-Top
Columns: 2
Rows: 1

***Figure 4-8: The Window Layout tool within the iTools system***

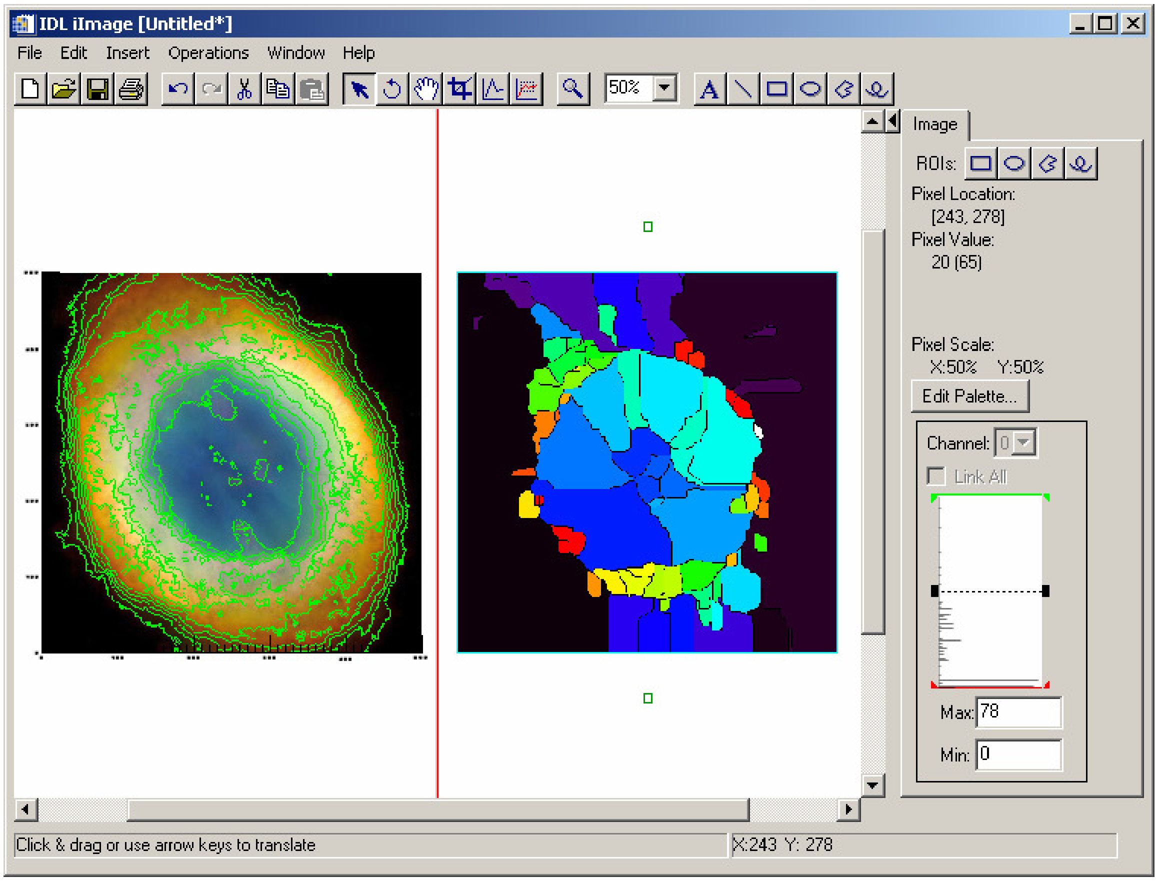
27. Press "*OK*" to dismiss the *Window Layout* dialog.
28. The user will be returned to the *iImage* utility, where the original viewplane will be selected and highlighted with a red border. Click within the new viewplane on the right-hand side in order to designate it as the target for future visualization insertion. Once this is accomplished, the viewplane on the right-hand side of the utility will be outlined in red.
29. Select "*Insert > Visualization…*" from the menu system.
30. Within the *Insert Visualization* dialog, press the "*Import Variable…*" button.
31. Within the *IDL Variable Browser* window, select the "*SEGMENTED*" variable and hit "*OK*".
32. Back within the *Insert Visualization* dialog, click on the new "*SEGMENTED*" data object and use it to define "IMAGEPIXELS" parameter of a new *Image* object visualization.

33. Press "*OK*" to dismiss the *Insert Visualization* dialog and insert the image for the watershed analysis into the second viewplane.

The watershed analysis has segmented the image into discrete regions, which are currently being displayed with the default grayscale color palette. It is more effective to visualize this watershed image using one of IDL's built-in color tables.

34. Select the new watershed segmentation image and press the "*Edit Palette…*" button.
35. Within the Palette Editor dialog click on the "*Load Predefined…*" droplist and select the "*Rainbow + white*" color table.
36. Press "*OK*" to dismiss the *Palette Editor* dialog.

The resulting visualization should look similar to Fig. 4-9.



Pixel Location: [243, 278]
Pixel Value: 20 (65)
Pixel Scale: X:50% Y:50%
Max: 78
Min: 0

Click & drag or use arrow keys to translate
X:243 Y: 278

***Figure 4-9: Display of the watershed analysis segmentation image***

37. Once finished viewing the two images, close the *IDL iImage* utility.
38. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```

# Chapter 5:  Volume Rendering

## Displaying 3-Dimensional Volumetric Data

A volumetric dataset consists of a 3-Dimensional array of numbers that represent a certain measurement made at each element location within the 3-D space.  Each data element location within the 3-D array is called a **voxel**, which is analogous to a three-dimensional pixel.  One manner in which volumetric datasets can be created is by stacking numerous 2-D image slices together into a three-dimensional array.

In the following exercise, volumetric data of numerous image slices from the Visible Human project stacked together will be visualized using the *iVolume* utility.  This volume consists of seventy 128 x 128 image slices acquired through the torso of a human body.  This example data is stored in a file named "*torso.dat*" that is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\torso.dat*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/torso.dat*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/torso.dat*

The "*torso.dat*" file does not have any particular file format, and the data is stored within this file in a **flat binary** fashion.  Consequently, the user will be required to provide IDL with the information it needs in order to successfully read the data from the file on disk.  Use the following steps to load this dataset into the *iVolume* utility :

1. IDL> iVolume
2. Select "*File > Open…*" from the *IDL iVolume* window.
3. Within the *Open* dialog, change the "Files of type:" droplist to "*All files (*)*".
4. Select the "*torso.dat*" file and hit "*Open*".

The dialog for the *Binary Template* wizard will appear.  This wizard helps walk the user through the steps of providing IDL with information on the structure of the binary file so the data can be read into the *iVolume* utility.  The user must provide the following basic information in order for the data to be successfully input :

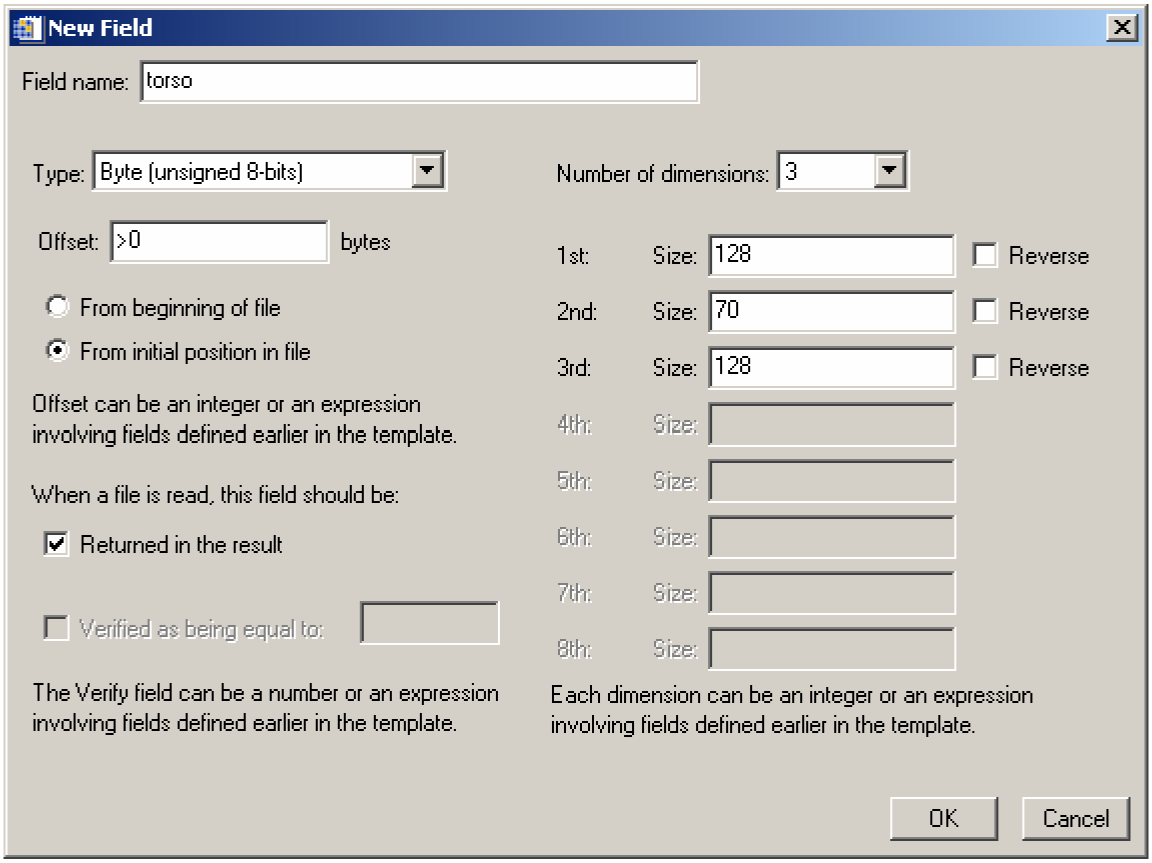
- Number of dimensions
- Size of each dimension
- Data type
- Offset (size of header)
- Byte order (Little Endian or Big Endian)

Without this information, there is no way for the software to know how to read-in the binary data from the file on disk.

5. Start by pressing the "*New Field…*" button within the *Binary Template* wizard.

A field is basically a data segment that will be read-in from the file on the harddrive. In this case, the "*torso.dat*" file only contains the stacked image data, so the entire volume will be input from the file as one field.

6. Within the *New Field* dialog, set the "Field name:" parameter to the string "*torso*" [Fig. 5-1].
7. The data type for this volumetric dataset is unsigned 8-bits (Byte), so the "Type:" droplist can be left as its default setting [Fig. 5-1].
8. The selected data file does not contain a header, so the "Offset:" parameter can be left as its default "*>0*" setting [Fig. 5-1].
9. The volumetric dataset has three dimensions, so set the "Number of dimensions:" droplist to "*3*" [Fig. 5-1]. Once this is accomplished, the "Size:" text boxes for the 1st, 2nd, and 3rd dimensions will become active.
10. The seventy images are stored within this volume using an interleave known as *band-interleaved by line* (BIL). Consequently, the dimensions of the 3-D array are 128 x 70 x 128 [Fig. 5-1].
11. Once these settings have been completed, press "*OK*" to dismiss the *New Field* dialog.


New Field

Field name: torso

Type: Byte (unsigned 8-bits)

Offset: >0 bytes

○ From beginning of file

● From initial position in file

Offset can be an integer or an expression involving fields defined earlier in the template.

When a file is read, this field should be:

☑ Returned in the result

□ Verified as being equal to:

The Verify field can be a number or an expression involving fields defined earlier in the template.

Number of dimensions: 3

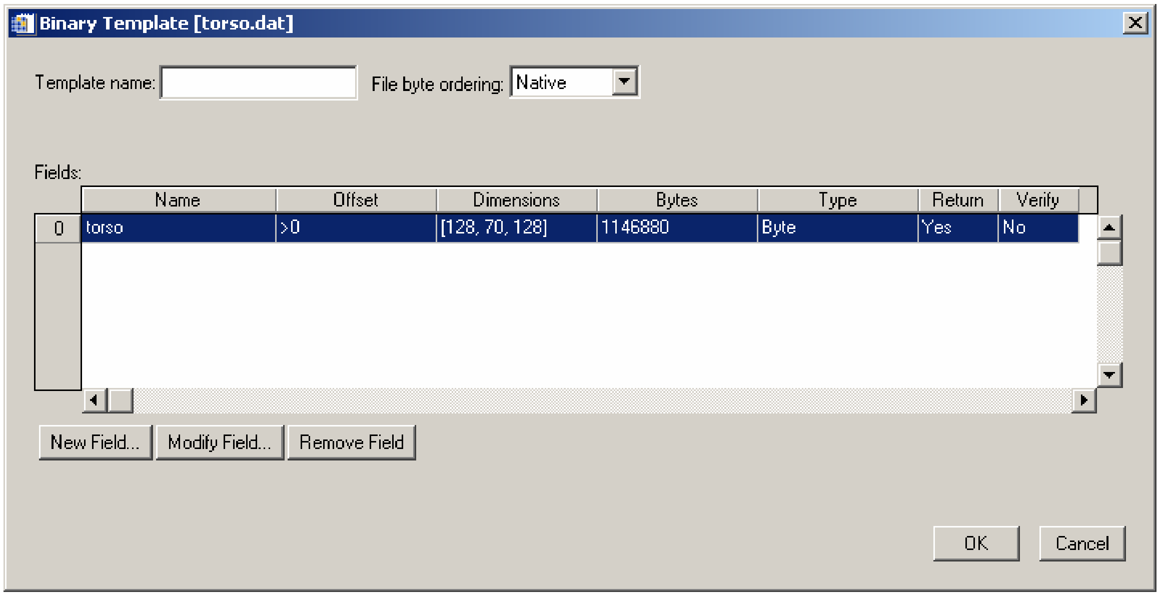
| | | | |
|---|---|---|---|
| 1st: | Size: | 128 | □ Reverse |
| 2nd: | Size: | 70 | □ Reverse |
| 3rd: | Size: | 128 | □ Reverse |
| 4th: | Size: | | |
| 5th: | Size: | | |
| 6th: | Size: | | |
| 7th: | Size: | | |
| 8th: | Size: | | |

Each dimension can be an integer or an expression involving fields defined earlier in the template.

***Figure 5-1: The New Field dialog within the Binary Template wizard***

The user will be returned to the *Binary Template* wizard, which will show the parameters for the "*torso*" field definition that was just created [Fig. 5-2].

12. Since the data type for the selected file is only single bytes, the user does not need to be concerned with the byte order, and the "File's byte ordering:" droplist can be left as its default setting of "*Native*" [Fig. 5-2].
13. Press "*OK*" to complete the *Binary Template* wizard.



Binary Template [torso.dat]

Template name:

File byte ordering: Native

Fields:

|  | Name | Offset | Dimensions | Bytes | Type | Return | Verify |
|---|---|---|---|---|---|---|---|
| 0 | torso | >0 | [128, 70, 128] | 1146880 | Byte | Yes | No |

***Figure 5-2: The completed Binary Template wizard***

Once the *Binary Template* wizard is finished a 3-Dimensional volume object will be automatically inserted into the *IDL iVolume* utility. By default, the data space and axes will be shown, but the volume itself will not be rendered.

There are a number of ways to visualize volumetric data within the *iVolume* utility. Of course, the individual **image planes** that were stacked up to create the volume can be extracted and viewed in each of the three (X-Y, Y-Z, and X-Z) orthogonal planes. A technique known as **isosurfacing** can be used to contour the data, which creates a surface that shows the voxels in the 3-D data space that all have a specified data value (the isovalue). In addition, an **interval volume** can be extracted which is a tetrahedral mesh that spans the 3-D data space between two isosurfaces created at different data values. Finally, **true volume rendering** can be performed, which maps the voxel data elements to colors and opacity values through a set of lookup tables, then projects the 3-D volume to a two-dimensional graphical visualization. By default, the volume object within the iVolume utility is setup to perform volume rendering.

14. While the data space is selected, click on the "*Render*" button within the *Volume* panel on the right-hand side of the *iVolume* utility.

A volume rendering of the 3-Dimensional array is created using the default color palette (grayscale) and opacity settings. The reason the *iVolume* utility forces the user to press the "*Render*" button in order to visualize the object is because the volume rendering process of very large datasets can be very computationally intensive. In most cases, it is not feasible to constantly render the volume object when any of the standard iTools manipulators (Translate, Rotate, Zoom, etc.) are used to modify the visualization, especially when performing a click-and-drag operation.

15. Using the Select/Translate arrow ↖ button, click on the data space and move it slightly in one direction within the viewplane. The volume rendering of the object disappears.

Since the current volumetric dataset is relatively small, the Auto-Render option can be turned on so the iVolume utility will automatically update the visualization after every action.

16. Click on the small white checkbox to the left of the Auto-Render option to enable the auto-rendering capability.

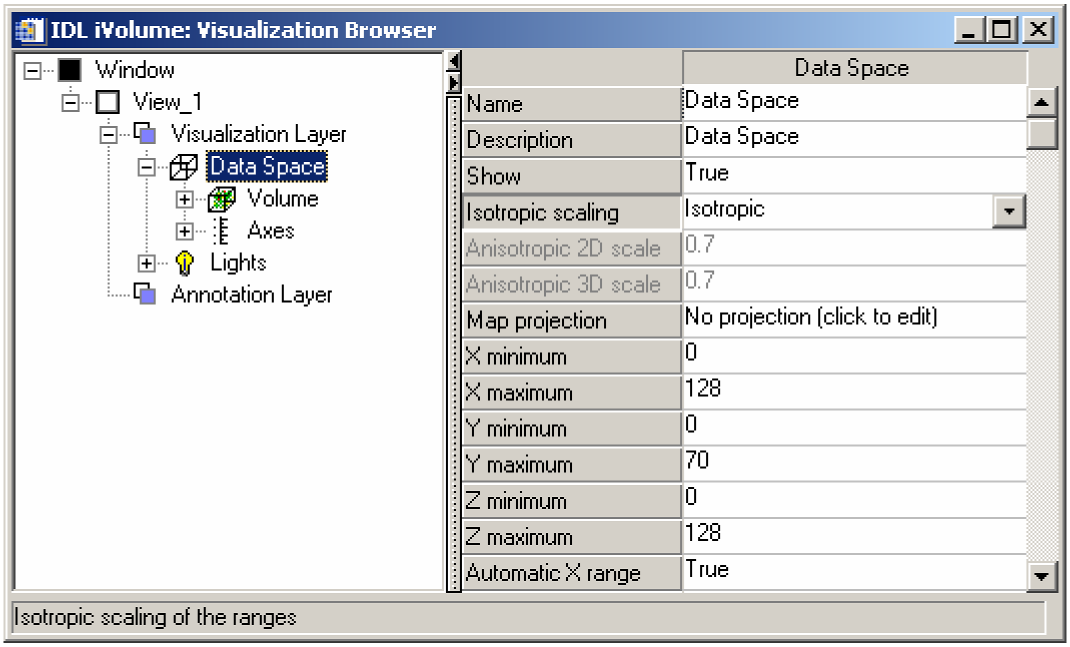
Now the volume will be constantly rendered while object manipulations are being performed.

## Data Space Scaling

Currently the data space is displayed with its default geometry, which scales the volume object in each of the three dimensions so it fits within a perfect cube. However, the size of this volume is actually 128 x 70 x 128, so it is not equal in all three dimensions. Consequently, the volume object is being exaggerated in the Y dimension (70) to fit the same 3-Dimensional space as the other X and Z dimensions (128). In order to display the volume with its true geometry, the scaling of the data space must be set to **isotropic**.

1. From the IDL iVolume menu system, select "Window > Visualization Browser…".
2. On the left-hand side of the Visualization Browser window, select the "Data Space" object [Fig. 5-3].
3. In the Data Space property sheet, click on the box to the right of the "Isotropic scaling" property and set the droplist to "Isotropic" [Fig. 5-3]. The volume object within the iVolume utility will automatically update and display the data space in an isotropic fashion.
4. Close the Visualization Browser window.

The data space is now displayed in an isotropic fashion and the length of each of the three X-Y-Z axes is proportionate to their size.



| | Data Space |
|---|---|
| Name | Data Space |
| Description | Data Space |
| Show | True |
| Isotropic scaling | Isotropic |
| Anisotropic 2D scale | 0.7 |
| Anisotropic 3D scale | 0.7 |
| Map projection | No projection (click to edit) |
| X minimum | 0 |
| X maximum | 128 |
| Y minimum | 0 |
| Y maximum | 70 |
| Z minimum | 0 |
| Z maximum | 128 |
| Automatic X range | True |

Isotropic scaling of the ranges

***Figure 5-3: Changing the scaling of the data space object to isotropic***

## Volume Object Properties

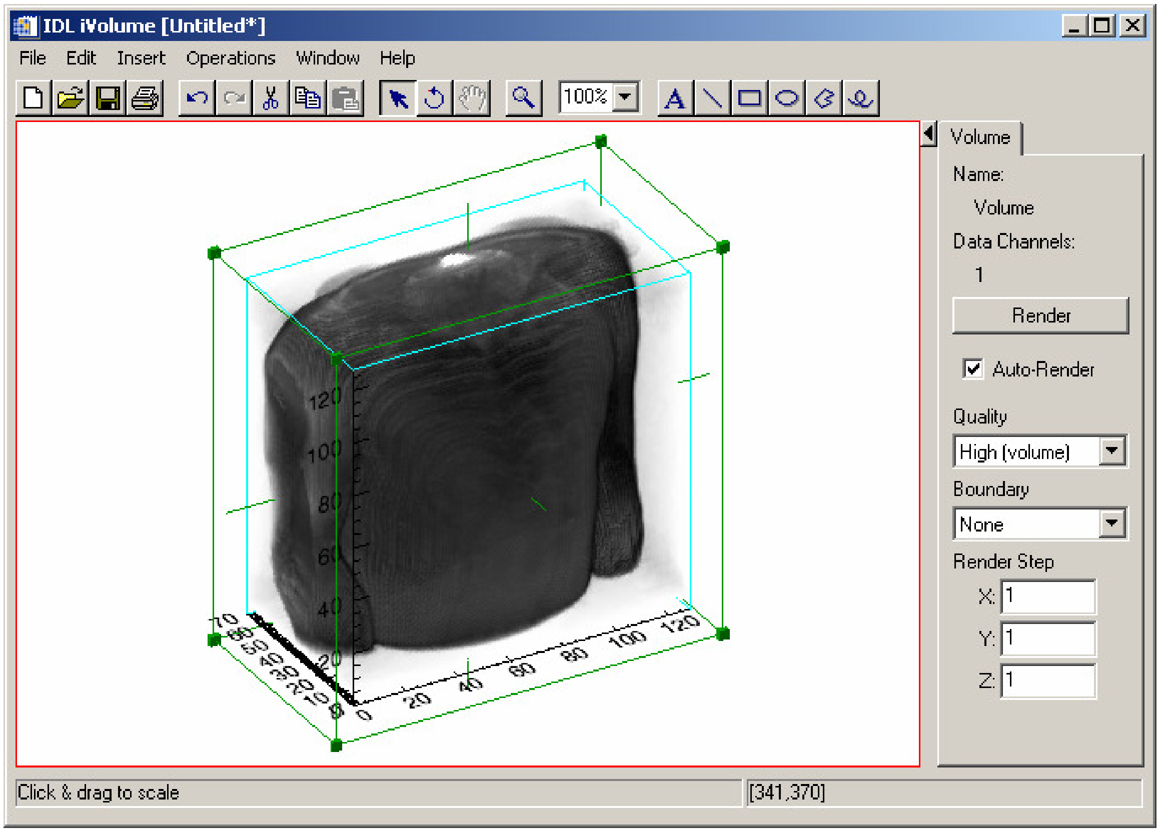
Volume objects have numerous properties that control the manner in which they are rendered. These properties define the rendering attributes for the visualization. The first properties of the volume rendering that can be easily modified by the user are present in the **Quality** and **Boundary** droplists on the *Volume* panel along the right-hand side of the *iVolume* utility. By default, the *iVolume* utility displays the volume with a low rendering quality and solid walls boundary.

There are two rendering qualities available for the volume object visualization :

- **Low (textures)**: Rendering is performed with a stack of 2-D texture-mapped semi-transparent polygons. The polygons are oriented so that the flat sides face the viewer as directly as possible. On most computers, low quality mode renders the volume faster but with less accuracy.
- **High (volume)**: Rendering is performed with a ray-casting volume renderer. This quality is more CPU-intensive and will usually take much longer to complete than low quality mode, but creates a much more accurate visualization.

The Boundary option displays an internal translucent solid cube within the data space, which is useful when the volume is not automatically rendered so that the user can locate and select the volume when no graphic is visible. Since the Auto-Render option is currently enabled, there is no need to display the boundary.

1. Change the Quality droplist to "*High (volume)*" and the Boundary droplist to "*None*". Notice the effect these changes have on the volume rendering visualization [Fig. 5-4].



Name:
Volume

Data Channels:
1

☑ Auto-Render

Quality: High (volume)

Boundary: None

Render Step
X: 1
Y: 1
Z: 1

Click & drag to scale

[341,370]

***Figure 5-4: Volume rendering with high quality and no internal extents***

Once the rendering quality is set to high, the Render Step "X:", "Y:", and "Z:" fields will become sensitive, which allow the user to specify the stepping factor (in screen dimensions) through the voxel matrix [Fig. 5-4]. By default, these fields are set to "*1*" so that each and every voxel is considered when the volume is rendered. Changing this setting to "*2*" would render only half as many voxels in the specified screen dimension.

It is much quicker and easier to modify the other properties of the volume visualization when the Auto-Render functionality is turned off. Once the desired changes have been made, the "*Render*" button can be pressed in order to display the visualization of the volume dataset.

2. Un-check the Auto-Render checkbox.

The **opacity** lookup table controls the transparency of any given voxel, while the current **color palette** defines what color is applied based on the voxel's data value. Manipulation of the color palette and opacity table is critical to controlling and improving the appearance of a volume rendering.

3. While the volume object is selected, right-click on it and select "*Properties…*".
4. Within the *Visualization Browser* window, click on the "Edit color/opacity table" field to the right of the "Color & opacity table 0" property and select "*Edit…*". This will bring-up the *Palette Editor* dialog.

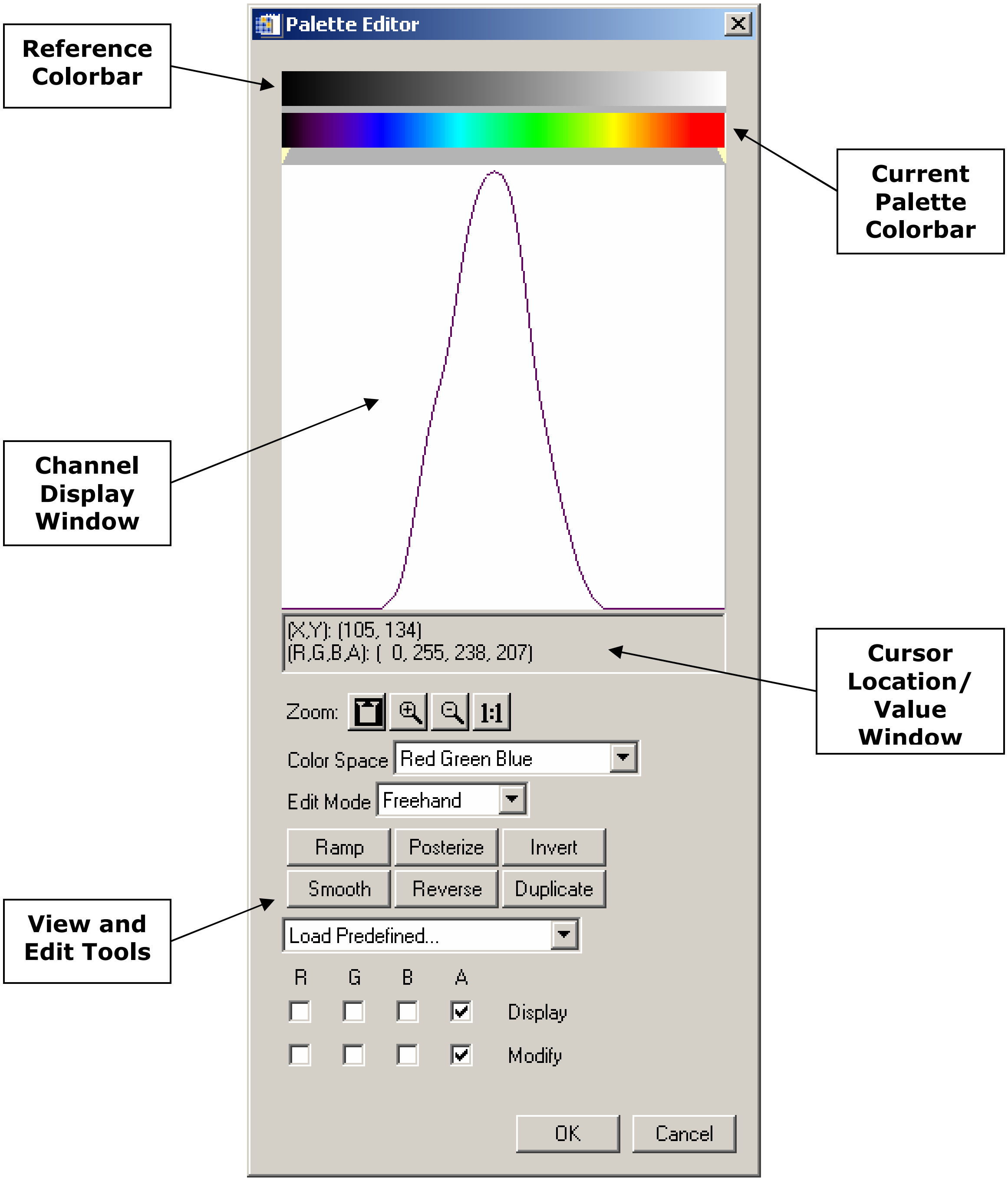
5. Within the "*Load Predefined…*" droplist, scroll down and select the "*Rainbow + white*" color table.

Next, the opacity table can be modified in order to control the transparency effect that is applied during the volume rendering. The *Palette Editor* dialog contains a number of items [Fig. 5-5] :

- **Reference colorbar** along the top illustrating data value range increasing from left-to-right using a grayscale ramp.
- **Current palette colorbar** that displays the currently loaded color table being applied to the visualization.
- **Channel display window**, which contains line plots for the individual red, green, and blue color palette channel vectors, along with a purple line for the Alpha channel (opacity lookup table).
- **Cursor location/value window**.
- **View and edit tools**, including zoom options, color space, editing operations, and channel selection.

The user can use the mouse cursor within the channel display window to modify the selected channels by interactively drawing the desired lookup table curve. The 3 color channel vector line plots are displayed with increasing brightness from bottom-to-top, and the combination of these 3 vectors creates the "*Rainbow + white*" color table that was just loaded. Since the R, G, B color channels have already been set to the desired color palette in the previous step, there is no need to modify or visualize these channels.

6. At the bottom of the *Palette Editor* window, un-check each of the R, G, B color channels in both the "Display" and "Modify" rows [Fig. 5-5].



(X,Y): (105, 134)
(R,G,B,A): ( 0, 255, 238, 207)

Color Space: Red Green Blue
Edit Mode: Freehand

| R | G | B | A | |
|---|---|---|---|---|
| ☐ | ☐ | ☐ | ☑ | Display |
| ☐ | ☐ | ☐ | ☑ | Modify |

***Figure 5-5: The Palette Editor dialog with a gaussian distribution curve for the alpha (opacity) channel***

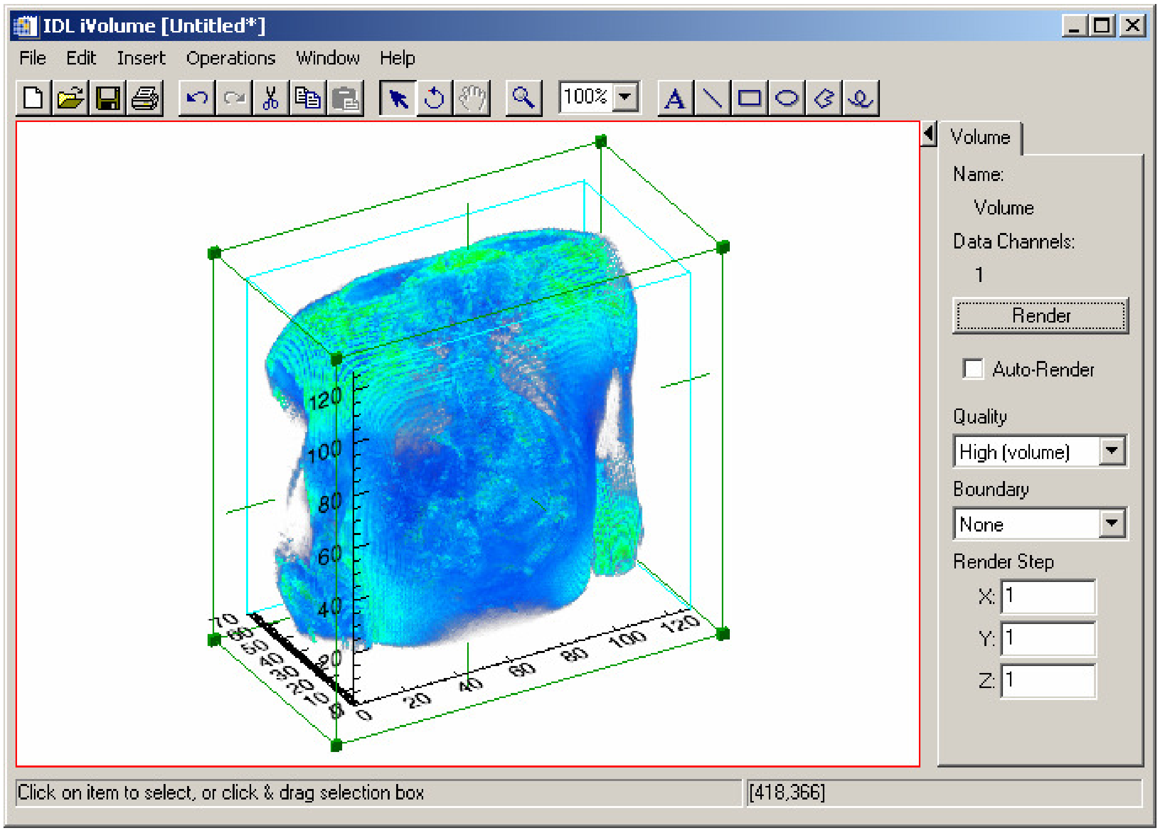
Now that all of the boxes under the R, G, B color channels are un-checked, only the **Alpha** channel (A) is displayed with the purple line plot. This is a plot of the lookup table for the current opacity effect, ranging from 0% opaque (100% transparent) at the bottom of the channel display window to 100% opaque (0% transparent) at the top. By default, the current opacity lookup table is a straight ramp from minimum data value (left-hand side) to maximum data value (right-hand side). This straight line has a slope of 1, meaning that the volume is rendered in a progressively more opaque fashion as the voxel data values get larger. Since the "Modify" field under the A channel is still checked, the user can click with the mouse cursor within the channel display window and draw a new opacity lookup table line in an arbitrary fashion.

7. Using the mouse cursor, click-and-drag within the channel display window and draw a standard gaussian shaped distribution curve [Fig. 5-5].
8. Click on the "*Smooth*" button a few times to give the line a smoother appearance if necessary.
9. Once the desired curve has been drawn, click "*OK*" to dismiss the *Palette Editor* dialog.

Creating a gaussian shaped curve for the opacity lookup table has the effect of highlighting (i.e. making more opaque) those voxels within the volume that have a medium data value around the middle of the overall data range.

10. Press the "*Render*" button in order to display the volume visualization.

The resulting visualization should look similar to Fig. 5-6.



Name:
Volume
Data Channels:
1

☐ Auto-Render

Quality
High (volume)

Boundary
None

Render Step
X: 1
Y: 1
Z: 1

Click on item to select, or click & drag selection box

[418,366]

***Figure 5-6: The volume rendering using a gaussian curve for the opacity***

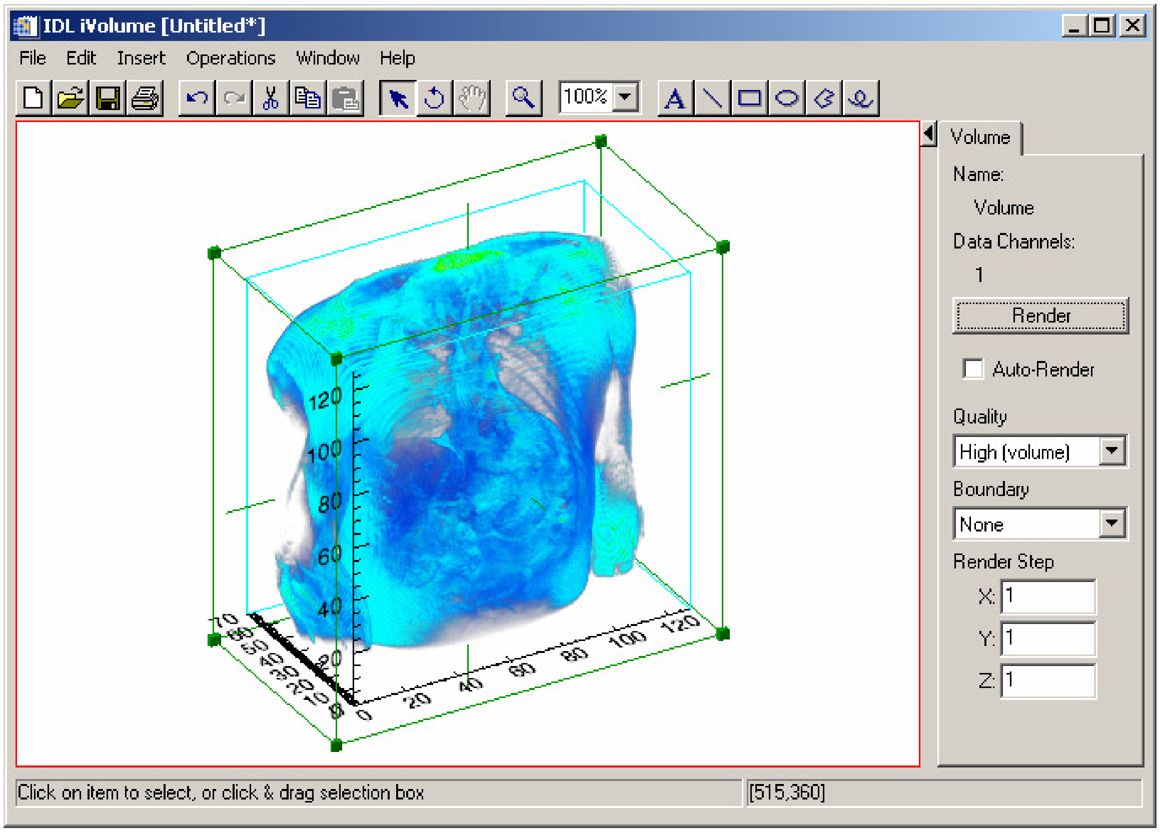
There are a number of other properties for the volume object that can be modified in order to control the appearance and quality of the visualization. By default, when rendering a volume object to the screen the voxel values that are selected to compute the visualization properties are selected using nearest neighbor selection process. If higher quality rendering is desired, the "**Interpolation**" property can be changed so this task if performed using a trilinear interpolation.

11. If the *Visualization Browser* window is not still visible, select "*Window > Visualization Browser…*". Within the *Visualization Browser* window, scroll down and change the "Interpolation" property for the Volume object to "*Trilinear*".
12. Press the "*Render*" button in order to display the volume visualization.

The rendering of the volume object should visually improve. When a volume is rendered, gradients within the volume are used to approximate a surface normal for each voxel, and the lighting sources in the current visualization are then applied to illuminate the object. **Gradient shading** can be enabled by modifying the "Use lighting" property. Furthermore, both sides of the voxels can be lighted by changing the "Voxel gradient" property.

13. Change the "Use lighting" property to "*True*" and the "Voxel gradient" property to "*Two-sided*".
14. Press the "*Render*" button in order to display the volume visualization.

The resulting visualization should look similar to Fig. 5-7.



Name:
Volume
Data Channels:
1

☐ Auto-Render

Quality
High (volume)

Boundary
None

Render Step
X: 1
Y: 1
Z: 1

Click on item to select, or click & drag selection box

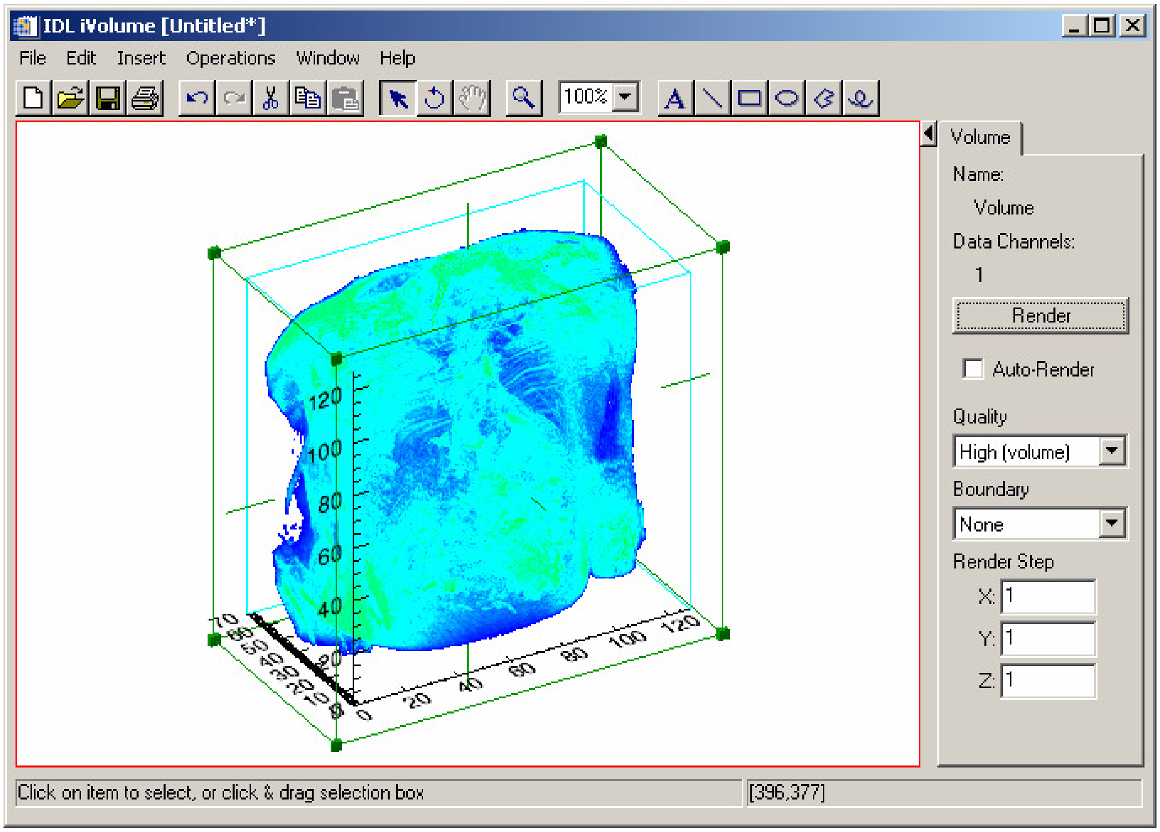
[515,360]

***Figure 5-7: The rendering of the volume object with modified properties***

By default, the volume object is rendered using the color palette and opacity table in a composite function technique known as **Alpha blending**. In Alpha blending, each voxel occludes other voxels behind it according to the opacity of the voxel in front, thereby allowing the viewer to see features within the 3-D volume. However, the *iVolume* utility also supports a number of methods for blending the projected voxels together to form an image. One common technique for projecting a volume into an image is the **Maximum Intensity Projection** (MIP).

15. Within the *Visualization Browser* window, change the volume object's "Composite function" property to "*Maximum intensity projection*".
16. Press the "*Render*" button in order to display the volume visualization.

The volume object rendering will change to display a MIP of the volume based on the current orientation of the 3-D data space. The resulting visualization should look similar to Fig. 5-8.



Name:
Volume
Data Channels:
1

☐ Auto-Render

Quality
High (volume)

Boundary
None

Render Step
X: 1
Y: 1
Z: 1

Click on item to select, or click & drag selection box [396,377]

***Figure 5-8: Maximum Intensity Projection (MIP) of the volume object***

17. Once finished viewing the MIP visualization, revert the setting for the "Composite function" property back to the default "*Alpha blending*" and close the *Visualization Browser* window.

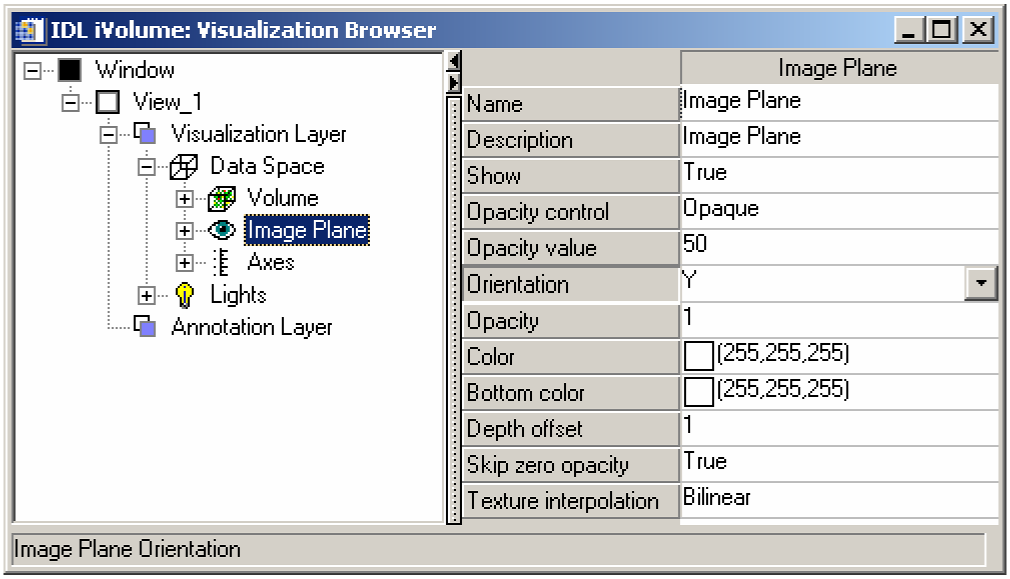
## Image Planes, Subvolumes, and Isosurfaces

In addition to the true volume rendering capabilities, the *iVolume* utility also offers a wide variety of other tools that allow the user to visualize volumetric datasets using additional graphical objects. For example, the user has the ability to extract and analyze individual image slices from any of the 3 orthogonal planes.

1. While the volume object is selected, use "*Operations > Volume > Image Plane*" to insert a new image object into the existing data space.

A cyan color rectangular box outlining the default image plane oriented in the Y-Z direction and located in the exact center of the volume is inserted. The user can click on this cyan box to move the location of the image plane within the current orientation, and the center voxel location for the image plane is displayed in the lower-right hand corner of the *IDL iVolume* window. For more advanced modifications to the image plane, the *Visualization Browser* must be utilized.

2. Select "*Window > Visualization Browser…*" from the menu system.
3. Click on the new "*Image Plane*" object so its property sheet is displayed [Fig. 5-9].
4. Change the "Opacity control" property to "*Opaque*" so the image is readily visible [Fig. 5-9].
5. Change the "Orientation" property to "*Y*" so the image plane is oriented in the X-Z direction [Fig. 5-9].



IDL iVolume: Visualization Browser

- Window
  - View_1
    - Visualization Layer
      - Data Space
        - Volume
        - Image Plane
        - Axes
      - Lights
    - Annotation Layer

| | Image Plane |
|---|---|
| Name | Image Plane |
| Description | Image Plane |
| Show | True |
| Opacity control | Opaque |
| Opacity value | 50 |
| Orientation | Y |
| Opacity | 1 |
| Color | □ (255,255,255) |
| Bottom color | □ (255,255,255) |
| Depth offset | 1 |
| Skip zero opacity | True |
| Texture interpolation | Bilinear |

Image Plane Orientation

***Figure 5-9: Modifying the image plane properties***

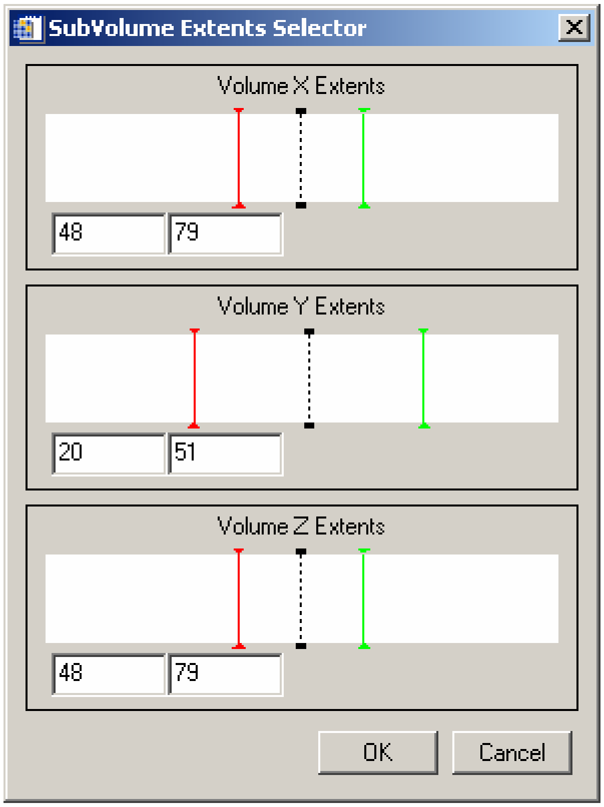
The *iVolume* utility also has the ability to display a **subvolume** within the current data space. This allows the user to specify a subset of the 3-Dimensional array to use when rendering the volume object.

6. Within the *Visualization Browser* window, select the "*Volume*" object.
7. Click on the "Edit Subvolume extents" item next to the "Subvolume" property and select "*Edit…*".
8. Within the *SubVolume Extents Selector* dialog box, use the keyboard to enter a specific voxel range for a subvolume to display [Fig. 5-10]. This will extract a 32 x 32 x 32 voxel subvolume located in the center of the volumetric dataset.

- Volume X Extents :  48 → 79
- Volume Y Extents :  20 → 51
- Volume Z Extents :  48 → 79

9. Click "*OK*" to dismiss the *SubVolume Extents Selector* dialog and close the *Visualization Browser* window.
10. Press the "*Render*" button in order to display the volume visualization.

The subvolume extent will be rendered instead of the full 3-Dimensional dataset.



SubVolume Extents Selector

Volume X Extents: 48 79

Volume Y Extents: 20 51

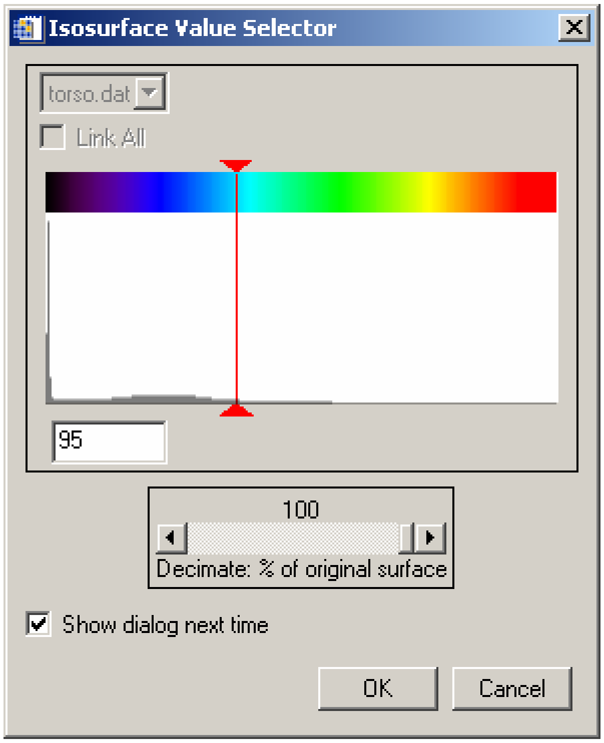
Volume Z Extents: 48 79

***Figure 5-10: Specifying a subvolume to render***

Finally, the volumetric dataset can be contoured in order to highlight all of the voxels that have the same specific data value.  These voxels are highlighted with a planar object called an **isosurface** that is automatically loaded into the existing data space.

11. While the subvolume object is selected, use the "*Operations > Volume > Isosurface*" menu item to launch the *Isosurface Value Selector* tool [Fig. 5-11].
12. Use the mouse to drag the red data value selector line until the number "*95*" appears in the text box.
13. Press the "*OK*" button to dismiss the *Isosurface Value Selector* tool.

The isosurface is computed for the volume and a new graphical object is inserted into the *iVolume* utility.  The resulting visualization should look similar to Fig. 5-12.



torso.dat

□ Link All

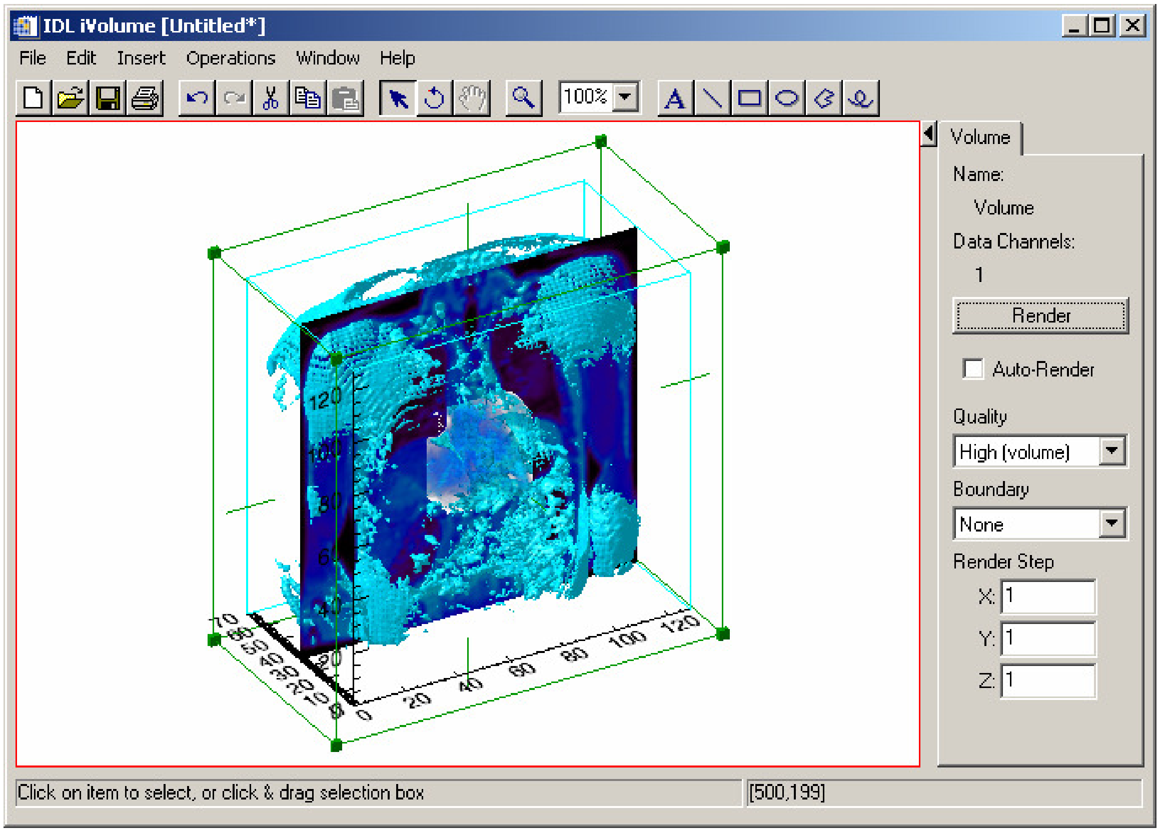
95

100

Decimate: % of original surface

☑ Show dialog next time

*Figure 5-11: Selecting an isosurface data value*



Name:
Volume

Data Channels:
1

□ Auto-Render

Quality: High (volume)

Boundary: None

Render Step
X: 1
Y: 1
Z: 1

Click on item to select, or click & drag selection box [500,199]

*Figure 5-12: Composite visualization of the volumetric dataset*

14. Once finished viewing the volume visualization, close the *IDL iVolume* window.
15. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```

# Chapter 6:  Advanced Signal Processing

## What is Signal Processing?

A **signal** is basically the record of a process that occurs in relation to an independent variable.  This independent variable can be any of a number of things, but in most cases it is **time**, in which case the signal is actually called a "**time-series**". Consequently, this exercise will only work with time-series signals.  In addition, although IDL's digital signal processing tools can work in more than one dimension, this exercise will only work with one-dimensional signals (i.e. vectors).

A **digital signal** is essentially a sequence of real values observed at discrete points in time that is stored as numbers on your computer.  The term "digital" actually describes two different properties of the signal :

- The values of the signal are only measured at discrete points in time as a result of **sampling**.  In general most signals have a constant **sampling interval**.
- The signal can take only discrete values as defined by the dynamic range of the instrument and the precision at which the data is stored on the computer.

Quite often it is difficult or even impossible to make sense of the information contained in a digital signal by looking at it in its raw form.  In addition, any signal obtained from an instrument measuring a physical process will almost always contain noise.  **Signal processing** is a technique that involves using computer algorithms to analyze and transform the signal in an effort to create natural, meaningful, and alternate representations of the useful information contained in the signal while suppressing the effects of noise.  In most cases signal processing is a multi-step process that involves both numerical and graphical methods.

## Curve Fitting

The problem of curve fitting can be summarized as follows :

*Given a tabulated set of data values {$x_i$, $y_i$} and the general form of a mathematical model (i.e. a function f(x) with unspecified parameters), determine the parameters of the model that minimize an error criterion.*

In the following exercise, the signal data from the example data file "*curve.csv*" will be input into the current IDL session using the Import ASCII macro.  This example data file is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\curve.csv*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/curve.csv*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/curve.csv*

The file "*curve.csv*" is standard ASCII text containing 2 columns of data that represent 2 variables ("*x*" and "*y*"). This ASCII text file uses comma-separated values as the file format, which can be easily viewed in spreadsheet form within Microsoft Excel.

1. From the main IDL development environment window, select "*Macros > Import ASCII*".
2. Within the file selection dialog choose the "*curve.csv*" file and hit "*Open*".

This will launch the standard ASCII Template wizard which was used in previous exercises to load data into the iTools system.

3. Within Step 1 of 3, make sure to change the "Data Starts at Line:" field to *2* and hit "*Next >>*".
4. In Step 2 of 3 all settings can be left as their defaults, so simply hit "*Next >>*".
5. In Step 3 of 3, change the "Name:" of *FIELD1* and *FIELD2* to *X* and *Y*, respectively.
6. Once this is accomplished, press "*Finish*".

Once the ASCII Template wizard has finished running the user will be returned to the main IDL development environment window where a new structure variable named "*curve_ascii*" now exists. This variable stores the data that was input from the file, and the *HELP* procedure can be used to acquire information on this variable :

```
7. IDL> HELP, curve_ascii, /ST
** Structure <14a2b70>, 2 tags, length=400, data length=400, refs=1:
   X               FLOAT     Array[50]
   Y               FLOAT     Array[50]
```

The *POLY_FIT* function within IDL performs a least-squares polynomial fit with optional error estimates and returns a vector of coefficients. The *POLY_FIT* function uses matrix inversion. A newer version of this routine, *SVDFIT*, uses Singular Value Decomposition (SVD), which is more flexible but slower. Another version of this routine, *POLYFITW*, performs weighted least-squares fitting. In addition, the *CURVEFIT* function uses a gradient-expansion algorithm to compute a non-linear least squares fit to a user supplied function with an arbitrary number of parameters. The user supplied function may be any non-linear function where the partial derivatives are known or can be approximated.

Using the tabulated polynomial data now stored in the "*curve_ascii*" variable, use the *POLY_FIT* fitting algorithm to fit a 3rd degree polynomial to the points :

```
8.  IDL> coeff = POLY_FIT (curve_ascii.x, curve_ascii.y, 3, yFit)
9.  IDL> iPlot, curve_ascii.x, curve_ascii.y, LINESTYLE=6, SYM_INDEX=2
10. IDL> iPlot, curve_ascii.x, yFit, /OVERPLOT
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-1.

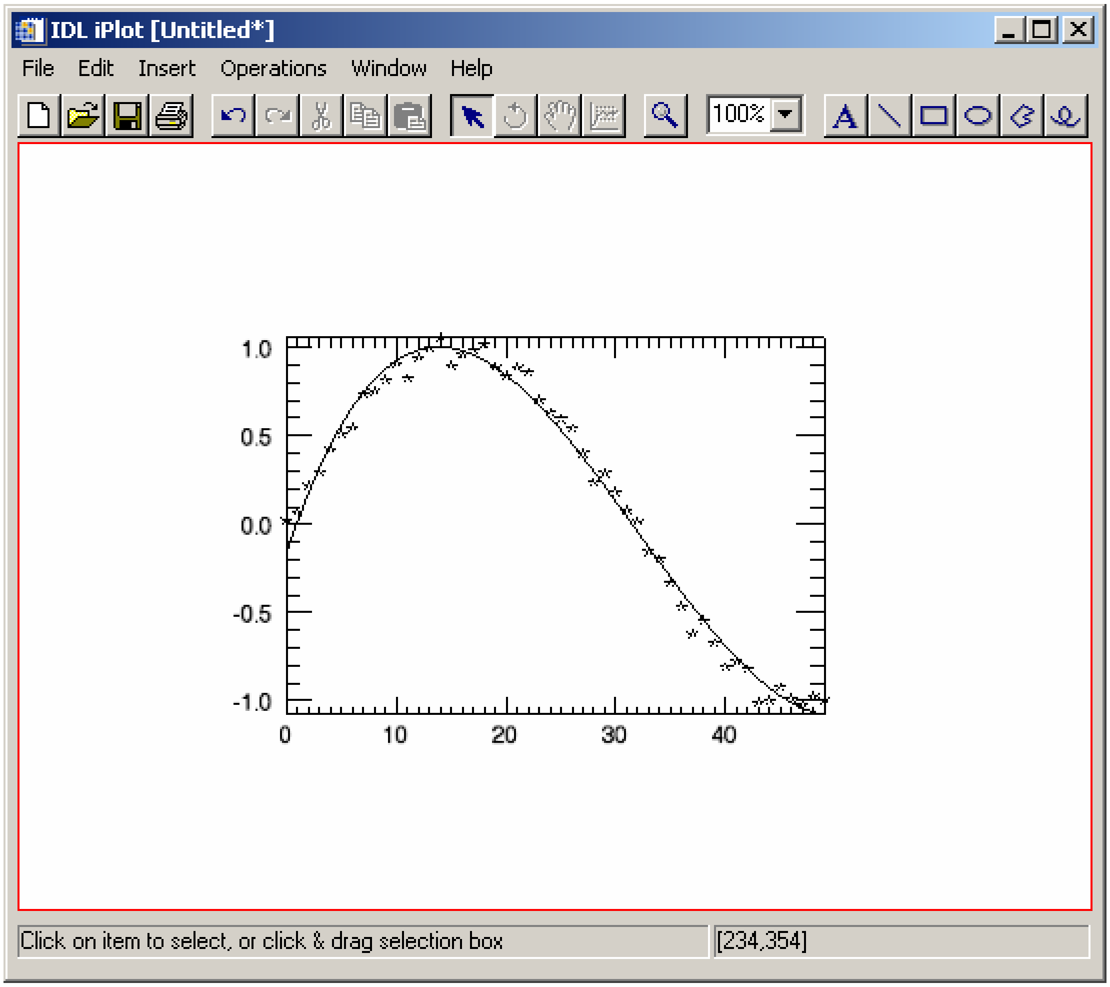


***Figure 6-1: Simulated instrument response data (* symbols) and the fitted function (solid line)***

Notice that POLY_FIT returns a vector containing the 4 estimated coefficients for the 3rd degree polynomial :

```
11. IDL> PRINT, coeff
         -0.154124
          0.183942
       -0.00846933
       8.83802e-005
```

12. Once finished viewing the fitted curve, close the *IDL iPlot* window.

## Simple Noise Removal

A general discussion of time-series analysis assumes that a time-series is comprised of four components :

- A trend or long-term movement (a constant or uniformly varying background level)
- A cyclical fluctuation about the trend (a superposition of sinusoidal variations, each with a definite amplitude, frequency, and phase)
- A pronounced seasonal effect (sinusoidal variation with a large frequency value)
- A residual, irregular, or random effect (additive noise)

It is the identification and extraction of these components that can be of interest when performing analysis of signals. In addition, adjacent observations are likely to be correlated, especially as the time intervals between them become shorter. The analysis of the correlation between signals can also be of interest and will be covered later within this exercise.

Start by creating a test signal that can be used for analysis. An IDL batch file called "*test_signal.pro*" has been supplied within the "*lib*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\lib\test_signal.pro*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/lib/test_signal.pro*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/lib/test_signal.pro*

The file "*test_signal.pro*" contains a package of multiple statements needed to create a test signal. To execute this batch file simply use the special batch execution character "@" followed by the filename :

1. IDL> `@test_signal`

**Note**: If executing this statement results in the error "`% Error opening file.`", then the "IDL_QS_Files/lib/" subfolder that contains this batch file is not included in IDL's path. Please go back to chapter 1 of this Quick Start tutorial and perform the steps delineated in the section entitled "Installing the Tutorial Files" in order to rectify this problem.

Once this batch file has executed there will be 2 new variables called "*x*" (the independent variable time) and "*s*" (the actual signal data) within the IDL session :

2. IDL> `HELP, x, s`

```
X                FLOAT     = Array[100]
S                FLOAT     = Array[100]
```

Both of these variables are vectors of 32-bit floating-point precision data that are 100 elements in length. To visualize this time-series signal simply make a call to the *iPlot* procedure and maximize the display window :

3. IDL> `iPlot, x, s, VIEW_GRID=[2,2], VIEW_TITLE='Original Signal'`
4. Select "*Window > Zoom on Resize*" from the IDL iPlot window menu system.
5. Press the □ button in the upper-right hand corner to maximize the window.

The resulting *IDL iPlot* window will contain 4 view sub-windows. The line plot for the independent variable "*x*" as the X component and the signal variable "*s*" as the Y component will appear in the upper-left hand corner view window. The resulting visualization should look similar to Fig. 6-2.

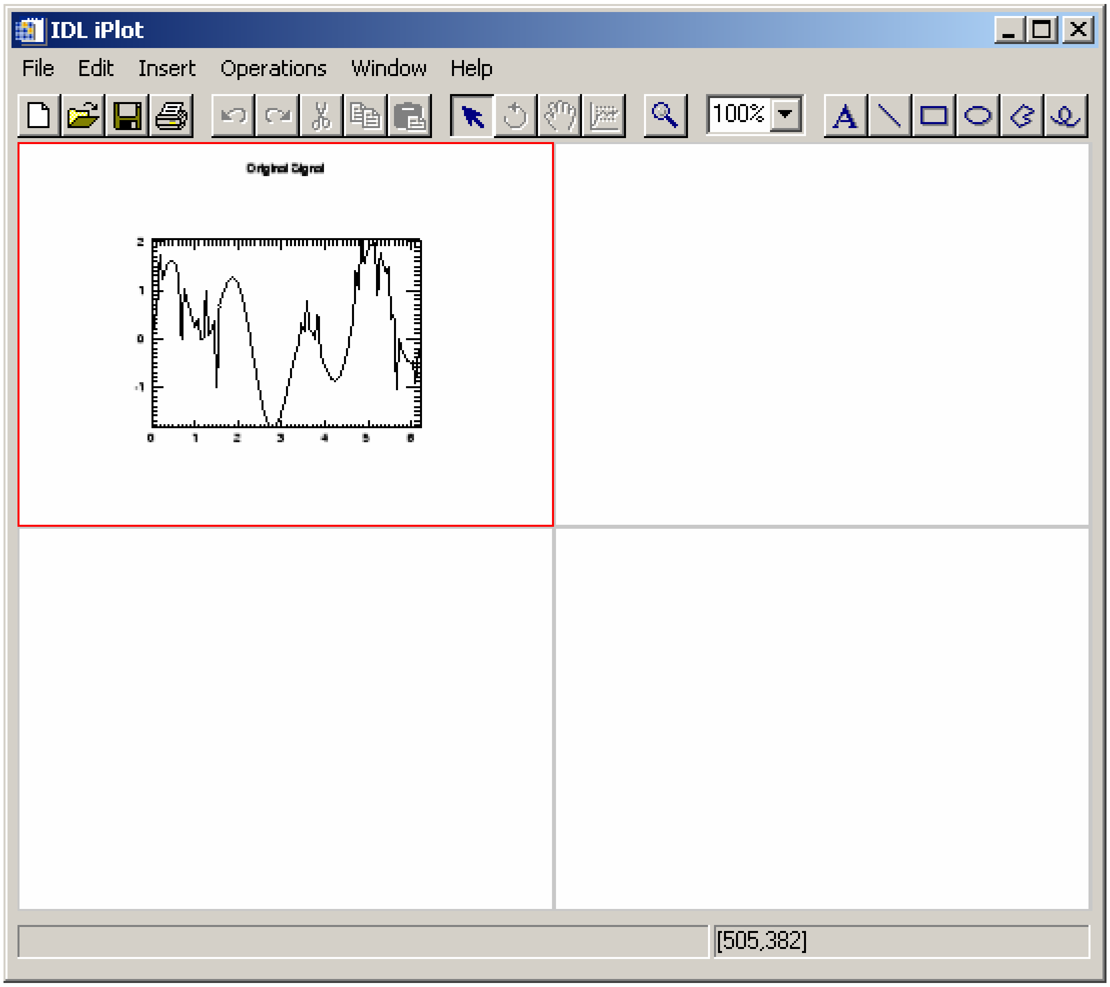


*Figure 6-2: Simple line plot of test signal data*

This signal has very sharp components that are analogous to noise. One way to attenuate certain types of noise in a signal is by using the *SMOOTH* function within IDL. The *SMOOTH* function filters an input array using a running-mean top hat kernel of a user-specified width (boxcar average). The *SMOOTH* function moves the kernel over the input array, element-by-element. At each step, the mean of the elements of the input array that fall under the window of the kernel is calculated, and this value is placed at the location of the centroid of the kernel at its current array element location. For example, consider a simple vector with five elements :

| | | | | |
|---|---|---|---|---|
| 3.00 | 1.00 | 7.00 | 2.00 | 4.00 |

When applying a *SMOOTH* function the output will be the average across the user-specified kernel size. For instance, when using a kernel size of 3 the element in the exact center of this vector would be replaced with the value (1.00 + 7.00 + 2.00) / 3 = 3.33. This is the output that results from smoothing the above vector using a kernel with a size of 3 :

| | | | | |
|---|---|---|---|---|
| 3.00 | 3.66 | 3.33 | 4.33 | 4.00 |

**Note:** By default, the endpoints outside the kernel window are replicated and placed in the output array unchanged. To override this effect use the EDGE_TRUNCATE keyword to the SMOOTH function.

Apply the *SMOOTH* function to the test signal and visualize the results :

```
6. IDL> smoothed = SMOOTH (s, 3)
7. IDL> iPlot, x, smoothed, /VIEW_NEXT, VIEW_TITLE='Smoothed'
```

The smoothed signal will be displayed in the upper-right hand corner view window. Smoothing a signal is this manner tends to distort the phase of the signal. Another interesting way to visualize the effect is by viewing the residual signal that is left after you remove the mean signal generated by the *SMOOTH* function from the original signal :

```
8. IDL> residual = s - smoothed
9. IDL> iPlot, x, residual, /VIEW_NEXT, VIEW_TITLE='Residual Signal'
```

The residual signal will contain information about the noise that the smoothing process removed.

Another way to remove certain types of unwanted noise from a signal is by using the *MEDIAN* function. The filtering process used by *MEDIAN* is similar to *SMOOTH*, but the application of the kernel is quite different. In *MEDIAN*, the median value (instead of the mean value) of the elements under the kernel is placed in the output array at the current centroid location. For example, if a *MEDIAN* function is applied to the same example vector :

| | | | | |
|---|---|---|---|---|
| 3.00 | 1.00 | 7.00 | 2.00 | 4.00 |

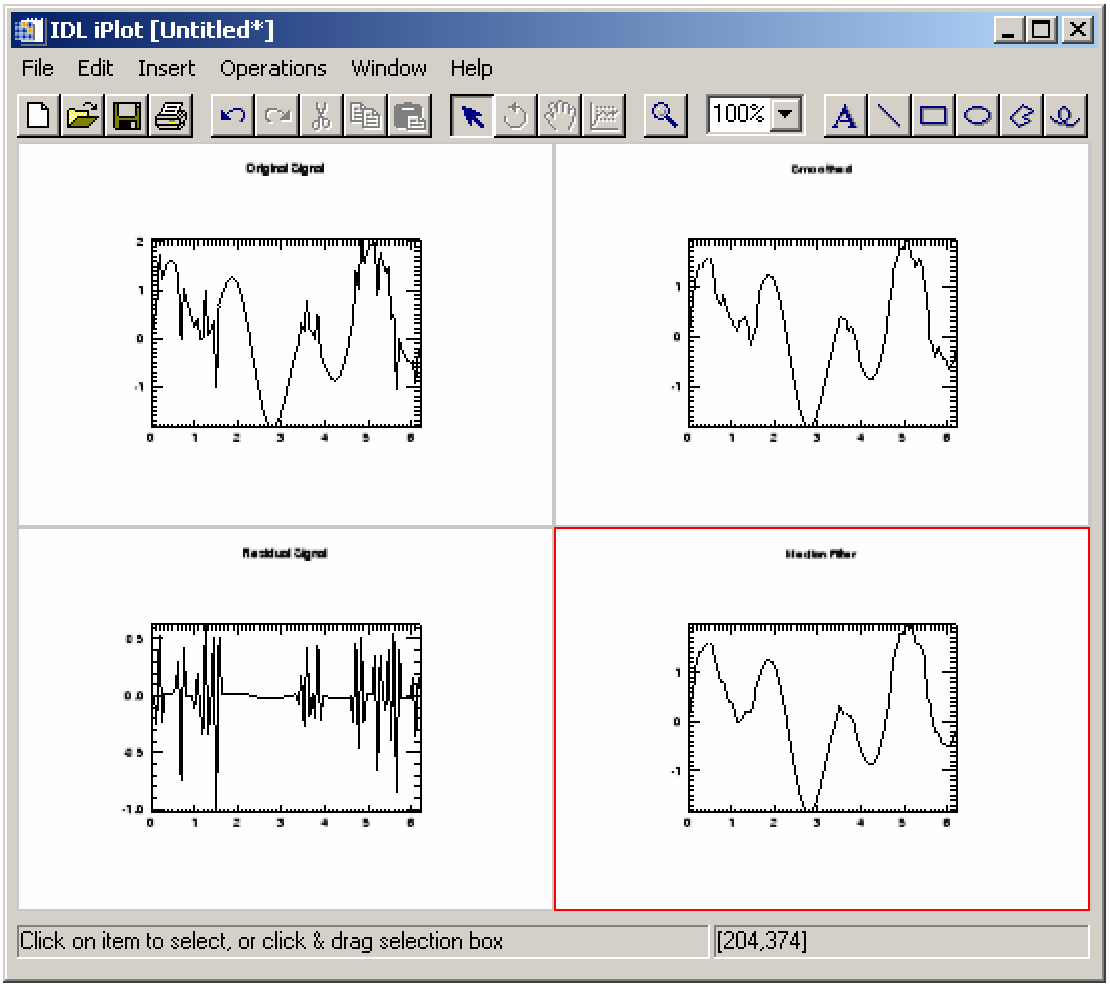
With a kernel of size 3 the *MEDIAN* function will return :

| | | | | |
|---|---|---|---|---|
| 3.00 | 3.00 | 2.00 | 4.00 | 4.00 |

Apply the *MEDIAN* function to the test signal and visualize the results :

```
10.IDL> median_filt = MEDIAN (s, 3)
11.IDL> iPlot, x, median_filt, /VIEW_NEXT, VIEW_TITLE='Median Filter'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-3.



Click on item to select, or click & drag selection box

***Figure 6-3: Result of applying SMOOTH and MEDIAN functions to test signal***

12. Once finished viewing the modified signal line plots, close the *IDL iPlot* window.

In addition, the *CONVOL* function in IDL also provides the ability to convolve a signal with a user-defined kernel array. Convolution is a general process that can be used for various types of smoothing, shifting, differentiation, edge detection, etc..

Digital filters can also be used to remove unwanted frequency components (e.g. noise) from a sampled signal. Two broad classes of filters are Finite Impulse Response (a.k.a. Moving Average) filters and Infinite Impulse Response (a.k.a. Autoregressive Moving Average) filters.

Digital filters that have an impulse response that reaches zero in a finite number of steps are called Finite Impulse Response (FIR) filters. An FIR filter is implemented by convolving its impulse response with the data sequence it is filtering. IDL's *DIGITAL_FILTER* function computes the impulse response of an FIR filter based on Kaiser's window, which in turn is based on the Bessel function. *DIGITAL_FILTER* constructs lowpass, highpass, bandpass, or bandstop filters.

Obtain the coefficients of a non-recursive lowpass filter for the equally spaced data points and convolve it with the original signal :

```
13. IDL> coeffs = DIGITAL_FILTER (0.0, 0.25, 50, 12)
14. IDL> digital_filt = CONVOL (s, coeffs, /EDGE_WRAP)
15. IDL> iPlot, x, s, VIEW_GR=[3,1], VIEW_TI='Original Signal'
16. IDL> iPlot, x, digital_filt, /VIEW_NE, VIEW_TI='Digital Filter'
```

Another filtering technique, known as the Savitzky-Golay smoothing filter (a.k.a. least-squares or "*DISPO*"), also provides a digital smoothing that can be used to process a noisy signal. IDL's *SAVGOL* function returns the coefficients of a Savitzky-Golay smoothing filter, which can then be applied using the *CONVOL* function. The filter is defined as a weighted moving average with weighting given as a polynomial of a certain degree. The returned coefficients, when applied to a signal, perform a polynomial least-squares fit within the filter window.

Use the *SAVGOL* function to obtain the coefficients of a Savitzky-Golay smoothing filter and visualize the results of both the *DIGITAL_FILTER* and *SAVGOL* functions :

```
17. IDL> coeffs = SAVGOL (5, 5, 0, 2)
18. IDL> savgol = CONVOL (s, coeffs, /EDGE_WRAP)
19. IDL> iPlot, x, savgol, /VIEW_NE, VIEW_TI='Savitzky-Golay'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-4.

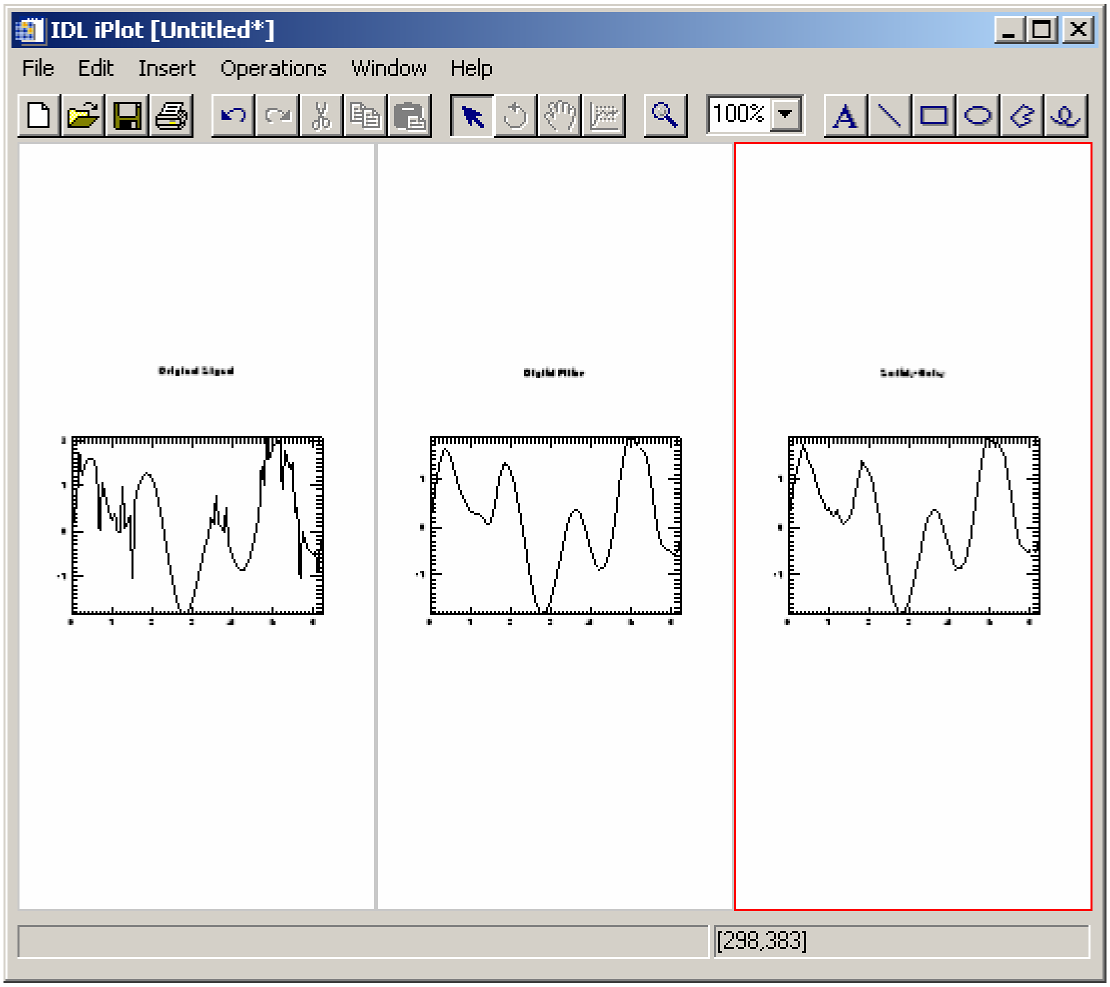


***Figure 6-4: Result of applying DIGITAL_FILTER and SAVGOL functions to test signal***

20. Once finished viewing the modified signal line plots, close the *IDL iPlot* window.

## Correlation Analysis

The first step in the analysis of a time-series is the transformation to a stationary series. A stationary series exhibits statistical properties that are unchanged as the period of observation is moved forward or backward in time. Specifically, the mean and variance of a stationary series remain fixed in time. The sample autocorrelation function is commonly used to determine the stationarity of a time-series. The autocorrelation of a time-series measures the dependence between observations as a function of their time differences or **lag**. In other words, the autocorrelation of a function shows how a function is related to itself as a function of a lag value. A plot of the sample autocorrelation coefficients versus corresponding lags can be very helpful in determining the stationarity of a time-series.

Compute the autocorrelation of the synthesized signal data versus a set of user-defined lag values and display the results :

```
1. IDL> lag = INDGEN (n/2) * 2
2. IDL> acorr = A_CORRELATE (s, lag)
3. IDL> iPlot, lag, acorr, VIEW_TI='Auto-Correlation'
4. IDL> iPlot, [0,n], [0,0], LINESTYLE=1, YRANGE=[-1,1], /OVER
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-5.

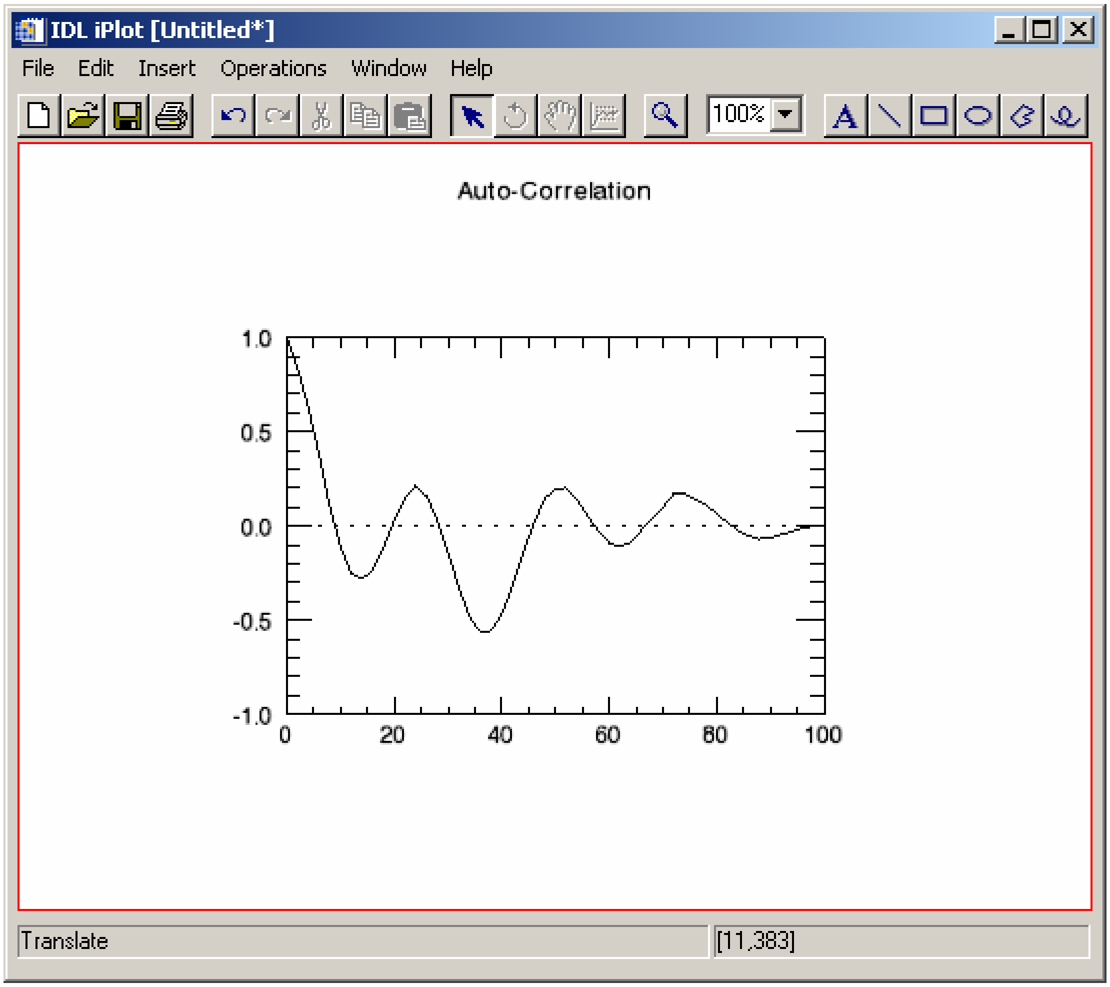


***Figure 6-5: Autocorrelation function for test signal***

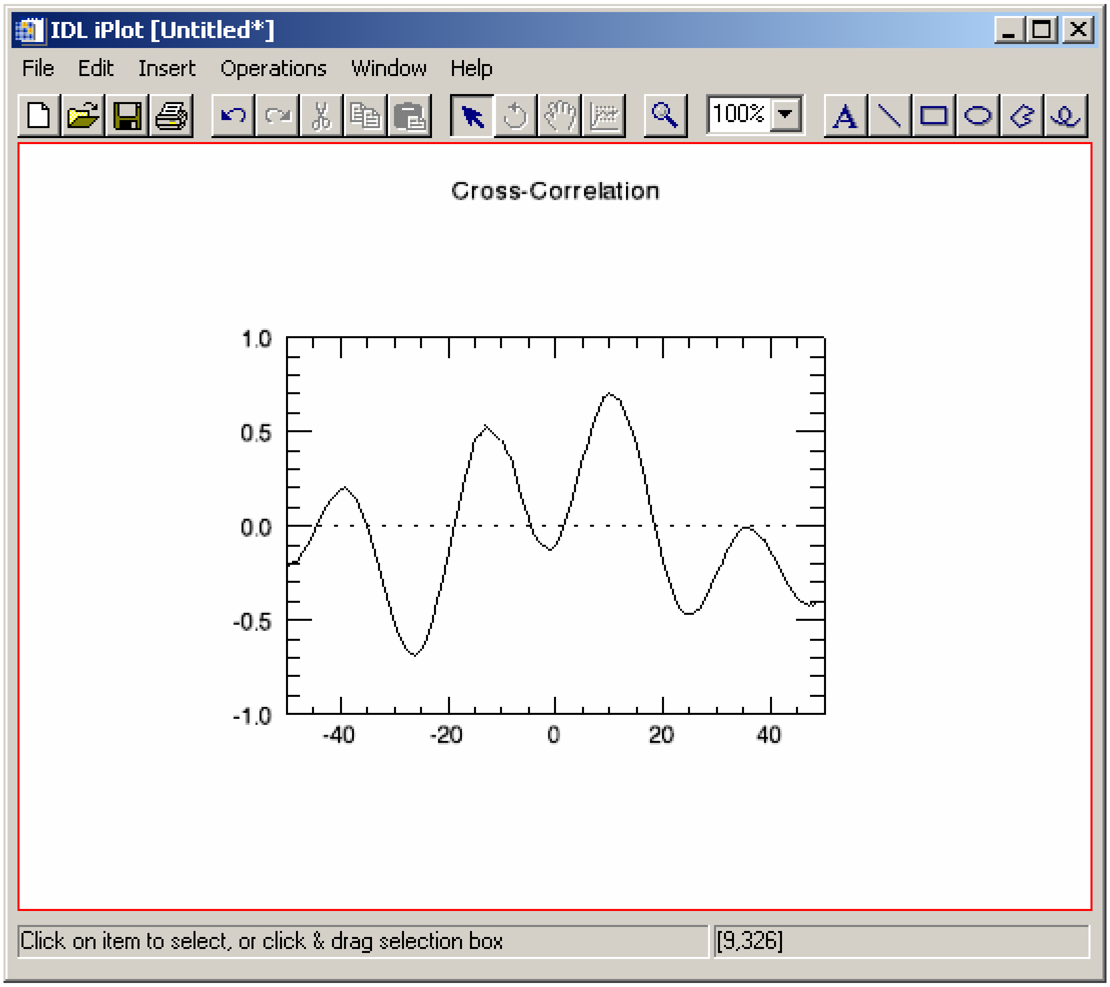
The autocorrelation measures the persistence of a wave within the whole duration of a time-series. When the autocorrelation goes to zero, the process is becoming random, implying there are no regularly occurring structures. An autocorrelation of 1 means the signal is perfectly correlated with itself. An autocorrelation of –1 means the signal is perfectly anticorrelated with itself.

5. Once finished viewing the autocorrelation results, close the *IDL iPlot* window.

The *C_CORRELATE* function in IDL can be used to compute the cross-correlation of two sample populations as a function of the lag. To demonstrate the use of this function, construct another signal that is simply the reverse of the original :

```
6.  IDL> lag = INDGEN (n) - n/2
7.  IDL> t = REVERSE (s)
8.  IDL> ccorr = C_CORRELATE (s, t, lag)
9.  IDL> iPlot, lag, ccorr, VIEW_TI='Cross-Correlation'
10. IDL> iPlot, [-n/2,n/2], [0,0], LINESTYLE=1, YRANGE=[-1,1], /OVER
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-6.



Click on item to select, or click & drag selection box

[9,326]

***Figure 6-6: Cross-correlation between two series at a set of user-defined lag values***

11. Once finished viewing the cross-correlation results, close the *IDL iPlot* window.

## Signal Analysis Transforms

Most signals can be decomposed into a sum of discrete (usually sinusoidal) signal components. The result of such decomposition is a frequency spectrum that can uniquely identify the signal. IDL provides three main transforms to decompose a signal and prepare it for analysis :

- The Discrete Fourier Transform
- The Hilbert Transform
- The Wavelet Transform

The **Discrete Fourier Transform** (DFT) is the most widely used method for determining the frequency spectra of digital signals (and is the only transform that will be covered in this Quick Start). This is due in part to the development of an efficient computer algorithm for computing DFTs known as the **Fast Fourier Transform** (FFT). IDL implements the Fast Fourier Transform in its *FFT* function. The DFT is essentially a way of estimating a Fourier transform at a finite number of discrete points, which works well with a digital signal since it is also sampled at discrete intervals.

The Fourier transform is a mathematical method that converts an input signal from the physical (time or space) domain into the frequency (or Fourier) domain. Functions in physical space are plotted as functions of time or space, whereas the same function transformed to Fourier space is plotted versus its frequency components. What the Fourier transform will help illustrate is that real signals can often be comprised of multiple sinusoidal waves (with the appropriate amplitude and phase) that when added together will re-create the real signal. In other words, the FFT algorithm decomposes the input time-series into an alternative set of basis functions, in this case sines and cosines.

The result of a FFT analysis is complex-valued coefficients, and *spectra* are derived from these coefficients in order to visualize the transform. Spectra are essentially the amplitude of the coefficients (or some product or power of the amplitudes) and are usually plotted versus frequency. There are a number of different ways to calculate spectra, but the primary forms are :

- Real and imaginary parts of Fourier coefficients
- Magnitude spectrum
- Phase spectrum
- Power spectrum

The most direct way to visualize spectra of a Fourier transformation of a signal is to plot the **real and imaginary parts** of the spectrum as a function of frequency. For example, calculate and display a Fourier transform for a new signal that consists of a sine wave. For simplicity assume the vector "*x*" contains time-series values in units of seconds :

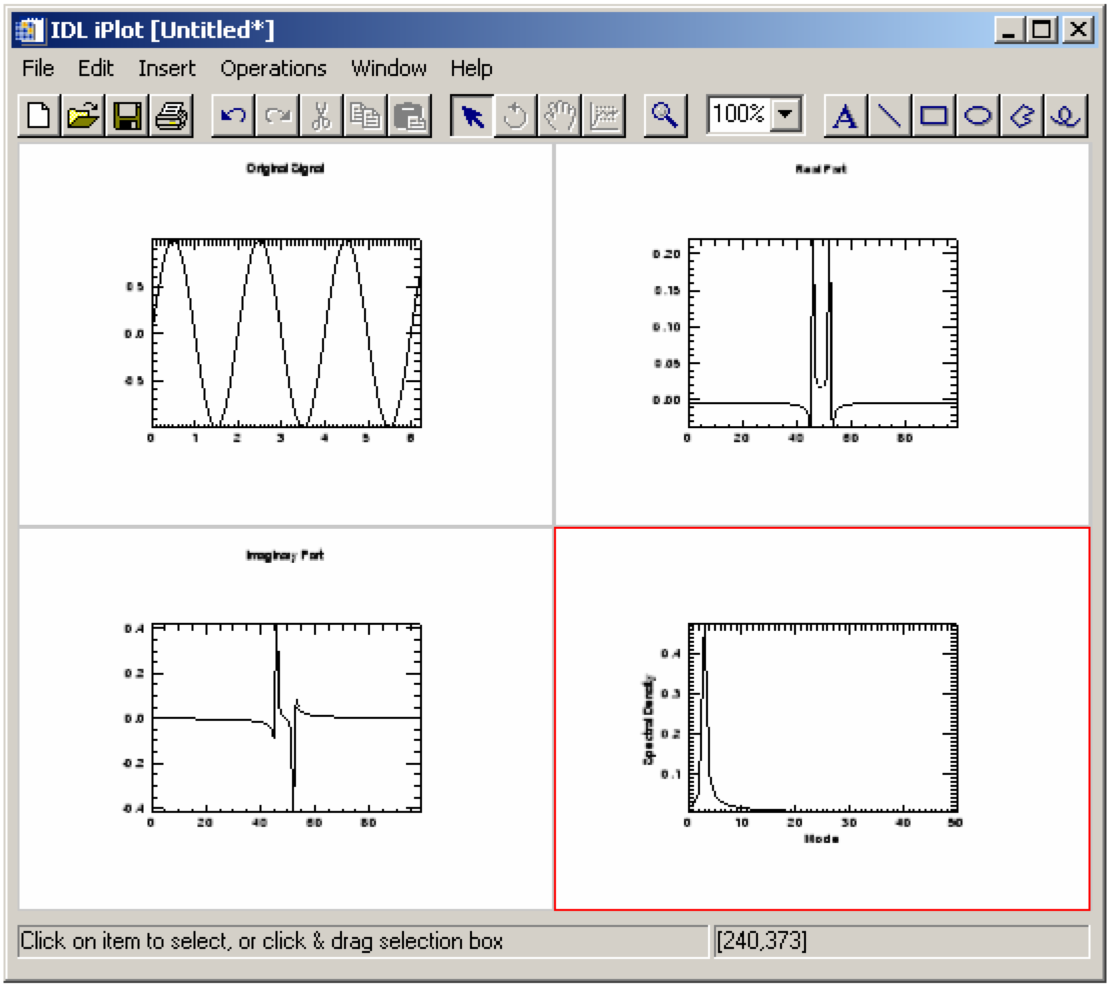
1. IDL> `sine = SIN (3.15 * x)`
2. IDL> `dft = FFT (sine)`
3. IDL> `real = FLOAT (SHIFT (dft, n/2-1) )`
4. IDL> `imaginary = IMAGINARY (SHIFT (dft, n/2-1) )`

```
5. IDL> iPlot, x, sine, VIEW_GR=[2,2], VIEW_TI='Original Signal'
6. IDL> iPlot, real, /VIEW_NE, VIEW_TI='Real Part'
7. IDL> iPlot, imaginary, /VIEW_NE, VIEW_TI='Imaginary Part'
```

Although the real and imaginary parts contain some useful information, it is more common to display the **magnitude spectrum** since it actually has physical significance. Since there is a one-to-one correspondence between a complex number and its magnitude (and phase for that matter), no information is lost in the transformation from a complex spectrum to its magnitude. In IDL, the magnitude can be easily computed using the absolute value (*ABS*) function :

```
8. IDL> magSpec = ABS (dft)
9. IDL> iPlot, magSpec, /VIEW_NE, XRANGE=[0,n/2], XTITLE='Mode', $
         YTITLE='Spectral Density'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-7.



Click on item to select, or click & drag selection box

[240,373]

**Figure 6-7: Real and imaginary parts of DFT plus the magnitude spectrum for the signal with sinusoidal component**

From this magnitude spectrum the mode at which the sinusoidal pattern occurs matches the peak of the spectral density (in this case Mode=3.15). Assuming the independent time variable "*x*" is in units of seconds, the plot of the original signal on top shows that this sinusoidal component (SIN(3.15*x)) has a wavelength (X-distance of one sine cycle) of approximately 2 seconds. Consequently, the frequency of this sinusoidal component is approximately (1 / 2 seconds) = 0.5 Hz.

10. Once finished viewing the magnitude spectrum, close the *IDL iPlot* window.

In addition to the sine component, add a cosine variation of a sinusoidal pattern to the signal with a wavelength of 0.5 seconds and a frequency of 2 Hz :

11. IDL> `cosine = COS (12.5 * x)`
12. IDL> `y = sine + cosine`
13. IDL> `iPlot, x, sine, COLOR=[255,0,0], NAME='SIN (3.15 * x)'`
14. IDL>
```
iPlot, x, cosine, COLOR=[0,0,255], NAME='COS (12.5 * x)', $
/OVER
```
15. IDL>
```
iPlot, x, y, COLOR=[0,128,0], NAME='Combined Signal', $
/OVER
```
16. Using the mouse cursor, click and drag a rectangular selection box that encompasses the entire plot dataspace so that everything is highlighted.
17. From the *iPlot* menu system select "*Insert* > *New Legend*" in order to insert a legend.

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-8.

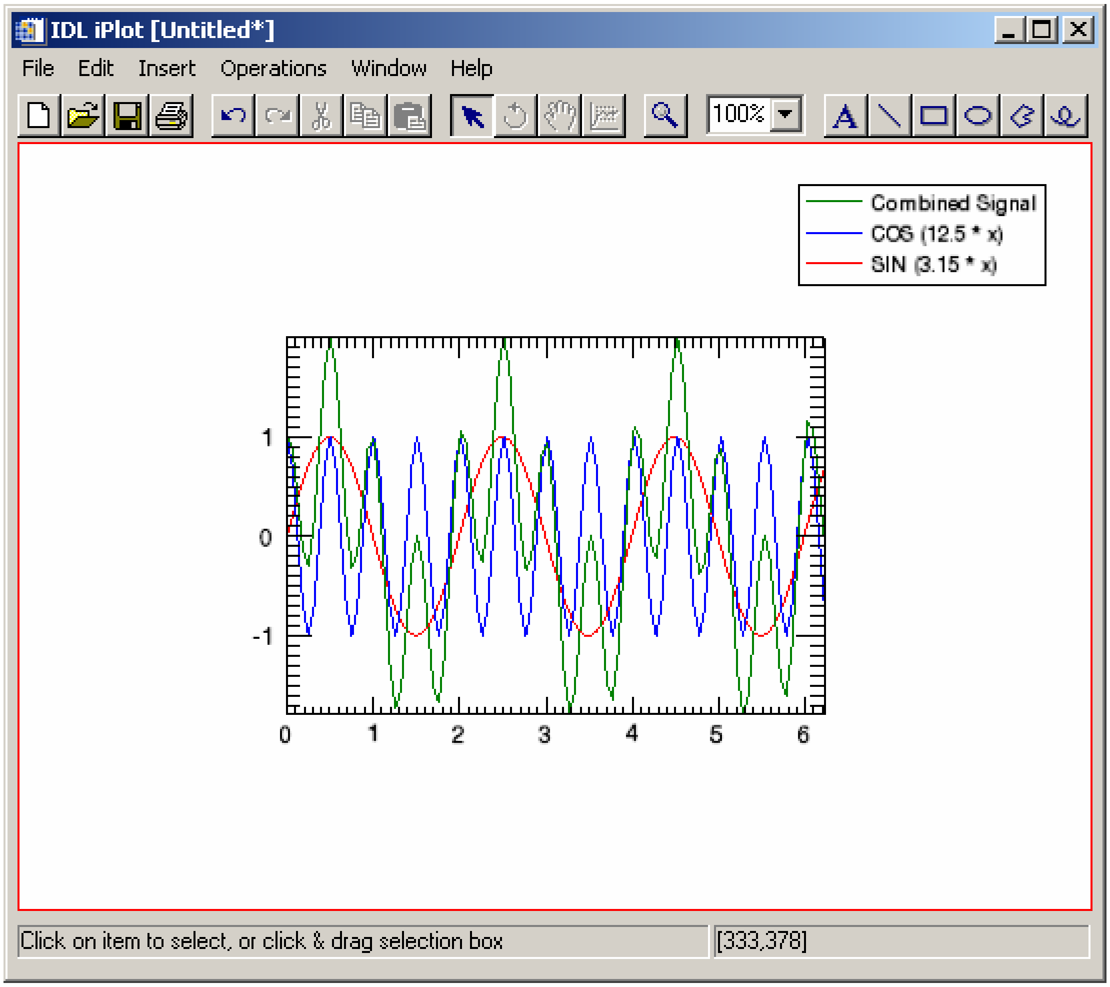


***Figure 6-8: Synthetic signal created from addition of two sinusoidal components***

18. Once finished viewing the combined signal, close the *IDL iPlot* window.

Signals are usually sampled at regularly spaced intervals in time. For example, the test signal dataset currently contained in the IDL variable "*y*" is a discrete representation of three superposed sine and cosine waves at n=100 points. In this case, the independent variable "*x*" that was used to create the test signal "*y*" is sampled at regularly spaced intervals of time, which is assumed to be in units of seconds. Consequently, the difference between any two neighboring elements in this vector "x" is known as the **sampling interval** (δ) :

```
19. IDL> samplingInterval = x[1] - x[0]
20. IDL> HELP, samplingInterval
       SAMPLINGINTERVAL   FLOAT     =     0.0628319
```

Since it is assumed that the time values are in units of seconds, it follows that the sampling frequency for this test signal is ~16 Hz :

```
21. IDL> samplingFreq = 1 / samplingInterval
```

```
22. IDL> HELP, samplingFreq
    SAMPLINGFREQ    FLOAT     =       15.9155
```

The **Nyquist frequency** (or “folding frequency”) is a special property of the sampling interval. The Nyquist frequency is basically the critical sampling interval at which measurements must be made in order to accurately portray the signal with only discrete values. For example, the critical sampling of a simple sine wave is two sample points per cycle. In other words, a digital signal has to have a sampling interval that is equal to or smaller than ½ the sinusoidal component’s wavelength. Consequently, the Nyquist Frequency is defined as (1/2*δ) :

```
23. IDL> Nyquist = 1 / (2 * samplingInterval)
24. IDL> HELP, Nyquist
    NYQUIST         FLOAT     =       7.95775
```

If a discretely sampled signal is not bandwidth-limited to frequencies less than the Nyquist frequency then **aliasing** occurs. When aliasing occurs the power spectrum will have a feature that has been falsely translated back across the Nyquist frequency. For example, construct another signal that adds a sinusoidal component with a frequency greater than the Nyquist frequency (7.96 Hz in this example). The new component will be (SIN(70*x)), which has a wavelength of approximately 0.089 seconds and a frequency of 11.14 Hz :

```
25. IDL> frq = FINDGEN (n) * samplingFreq / (n-1)
26. IDL> y = sine + cosine + SIN (70*x)
27. IDL> dft = FFT (y)
28. IDL> magSpec = ABS (dft)
29. IDL> iPlot, x, y, VIEW_GR=[2,1], VIEW_TI='Original Signal'
30. IDL> iPlot, frq, magSpec, /VIEW_NE, XRANGE=[0,Nyquist], $
         XTITLE='Frequency (Hz)', YTITLE='Spectral Density'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-9.

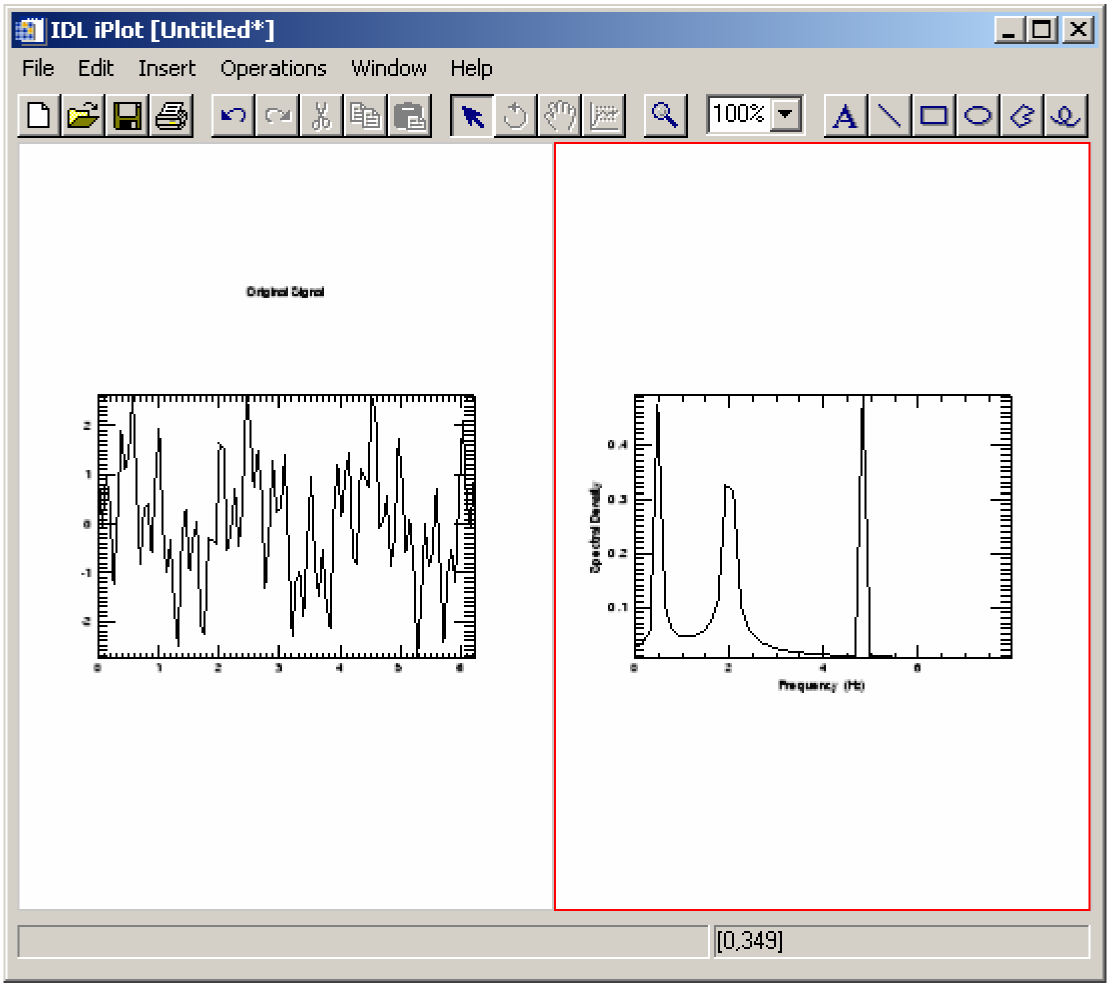


*Figure 6-9: Magnitude spectrum for signal with sinusoidal components*

The original "sine" and "cosine" components (at 0.5 Hz and 2 Hz respectively) are readily apparent in this magnitude spectrum, but there is also an aliased feature located at 4.77 Hz ( Nyquist - (11.14-Nyquist) ). Furthermore, the shape of the magnitude spikes vary from relatively narrow spikes to more diffuse features. For instance, the peak at 2 Hz is more spread-out, and this is due to an effect known as smearing or **leakage**. The leakage effect is a direct result of the definition of the DFT and is not due to any inaccuracy in the FFT.

**Note:** Leakage can be reduced by increasing the length of the time sequence, or by choosing a sample size that includes an integral number of cycles of the frequency component of interest.

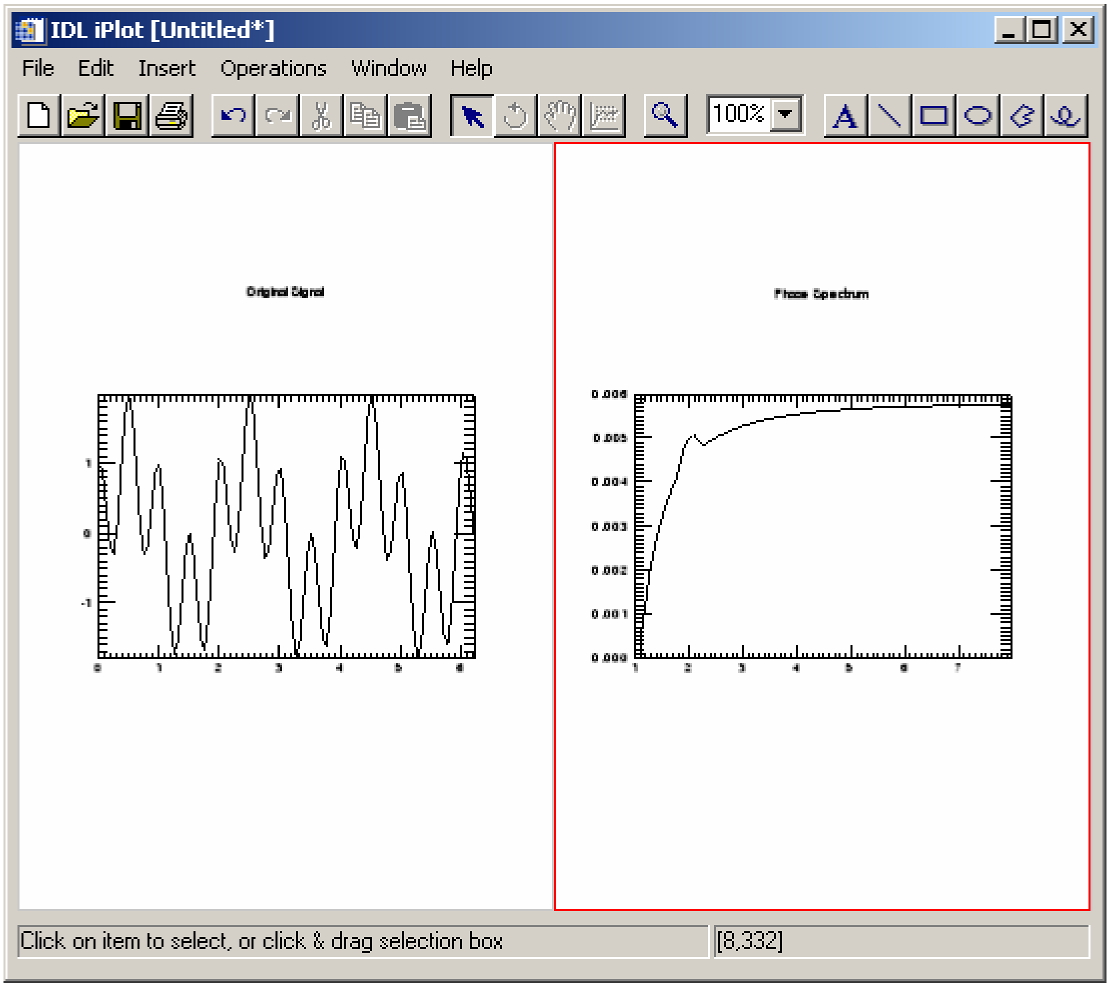
31. Once finished viewing the magnitude spectrum, close the *IDL iPlot* window.

The **phase spectrum** of a Fourier transform can also be easily computed in IDL using the arc-tangent (ATAN) function. By convention, the phase spectrum is usually plotted versus frequency on a logarithmic scale :

```
32. IDL> y = sine + cosine
33. IDL> dft = FFT (y)
34. IDL> phiSpec = ATAN (dft)
```

```
35. IDL> iPlot, x, y, VIEW_GR=[2,1], VIEW_TI='Original Signal'
36. IDL> iPlot, frq, phiSpec, /VIEW_NE, VIEW_TI='Phase Spectrum', $
         XRANGE=[1,Nyquist], YRANGE=[0,0.006]
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-10.



Click on item to select, or click & drag selection box

[8,332]

***Figure 6-10: Phase spectrum for signal with sinusoidal components***

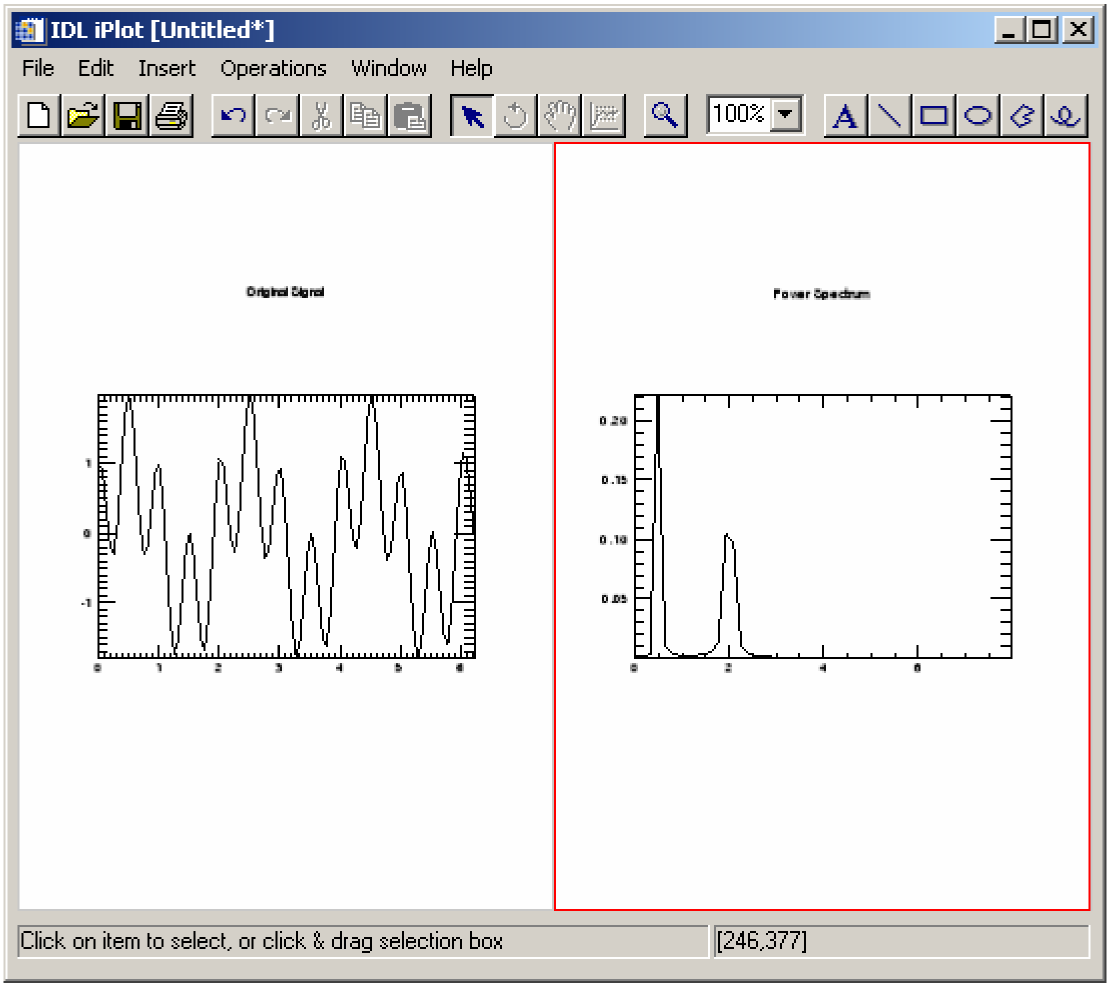
The same sinusoidal frequency component in the signal at 2 Hz shows up as a discontinuity in this phase plot.

37. Once finished viewing the phase spectrum, close the *IDL iPlot* window.

Finally, another way to visualize the same information is with the *power spectrum*, which is the square of the magnitude of the complex spectrum :

```
38. IDL> powSpec = (ABS (dft) ) ^ 2
39. IDL> iPlot, x, y, VIEW_GR=[2,1], VIEW_TI='Original Signal'
40. IDL> iPlot, frq, powSpec, /VIEW_NE, XRANGE=[0,Nyquist], $
         VIEW_TI='Power Spectrum'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-11.



Click on item to select, or click & drag selection box

[246,377]

***Figure 6-11: Power spectrum for signal with sinusoidal components***

41. Once finished viewing the power spectrum, close the *IDL iPlot* window.

## Windowing

The leakage effect mentioned previously is a direct consequence of the definition of the Discrete Fourier Transform, and of the fact that a finite time sample of a signal often does not include an integral number of some of the frequency components in the signal. The effect of this leakage can be reduced by increasing the length of the time sequence or by employing a windowing algorithm. IDL's HANNING function computes two windows that are widely used in signal processing :

- Hanning Window
- Hamming Window

For example, compare the power spectrum from the original signal to one that has the Hanning window applied :

```
1. IDL> hann = FFT (HANNING (n) * y )
2. IDL> hannPowSpec = (ABS (hann) ) ^ 2
3. IDL> iPlot, frq, powSpec, XRANGE=[0,3], XTITLE='Frequency (Hz)', $
        YTITLE='PSD', LINESTYLE=2
4. IDL> iPlot, frq, hannPowSpec, XRANGE=[0,3], /OVER
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6-12.

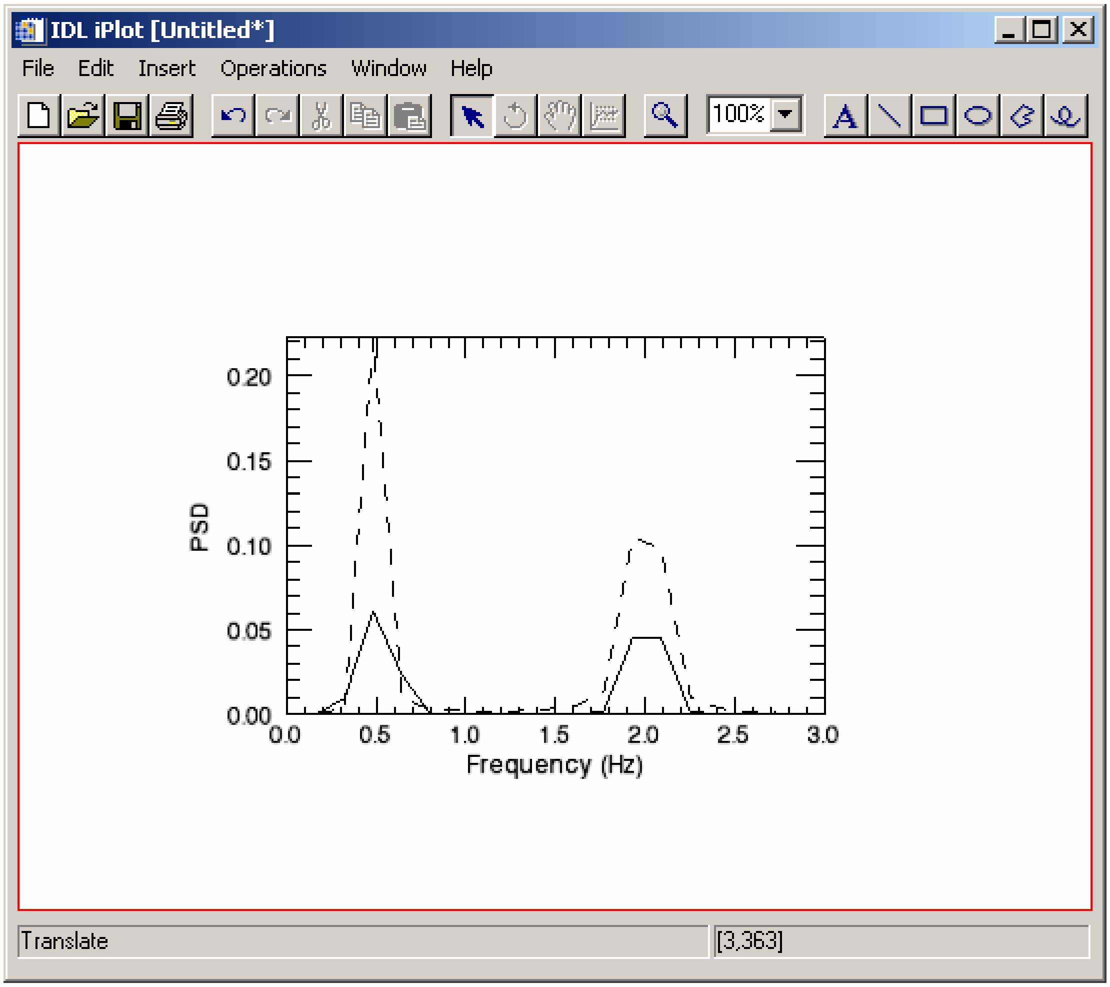


***Figure 6-12: Power spectrum of original (dashed) and Hanning windowed (solid) signal***

The power spectrum of the Hanning windowed signal shows some mitigation of the leakage effect.

5. Once finished viewing the power spectrums, close the *IDL iPlot* window.

## Wavelet Analysis

Wavelet analysis is becoming a popular technique for both signal and image analysis. By decomposing a signal using a particular wavelet function, one can construct a picture of the energy within the signal as a function of both spatial dimension (or time) and wavelet scale (or frequency). The wavelet transform is used in numerous fields such as geophysics (seismic events), medicine (EKG and medical imaging), astronomy (image processing), and computer science (object recognition and image compression).

The *WV_CWT* function within IDL can be utilized to calculate the continuous wavelet transform for a signal. Use the *WV_CWT* function to compute the transform for the signal currently stored in the variable "*y*" using a Gaussian wavelet function :

```
1. IDL> cwt = WV_CWT (y, 'Gaussian', 2)
```

Next, create the power spectrum for this continuous wavelet transform :

```
2. IDL> ps = ABS (cwt) ^ 2
```

An effective method for visualizing the continuous wavelet power spectrum is by using an image display with a color palette applied. The *LOADCT* routine within IDL can be used to automatically create the approriate color table vectors for a number of pre-defined color palettes. Use the *LOADCT* routine in conjunction with *TVLCT* to create the color table for a rainbow paletter ranging from blue to red :

```
3. IDL> ct = BYTARR (3, 256)
4. IDL> LOADCT, 34
5. IDL> TVLCT, r, g, b, /GET
6. IDL> ct[0,*] = r
7. IDL> ct[1,*] = g
8. IDL> ct[2,*] = b
```

Finally, display the continuous wavelet power spectrum within the iImage utility :

```
9. IDL> iImage, ps, RGB_TABLE=ct, VIEW_TI='Wavelet Power Spectrum'
```

The resulting *IDL iImage* visualization window should look similar to Fig. 6-13.

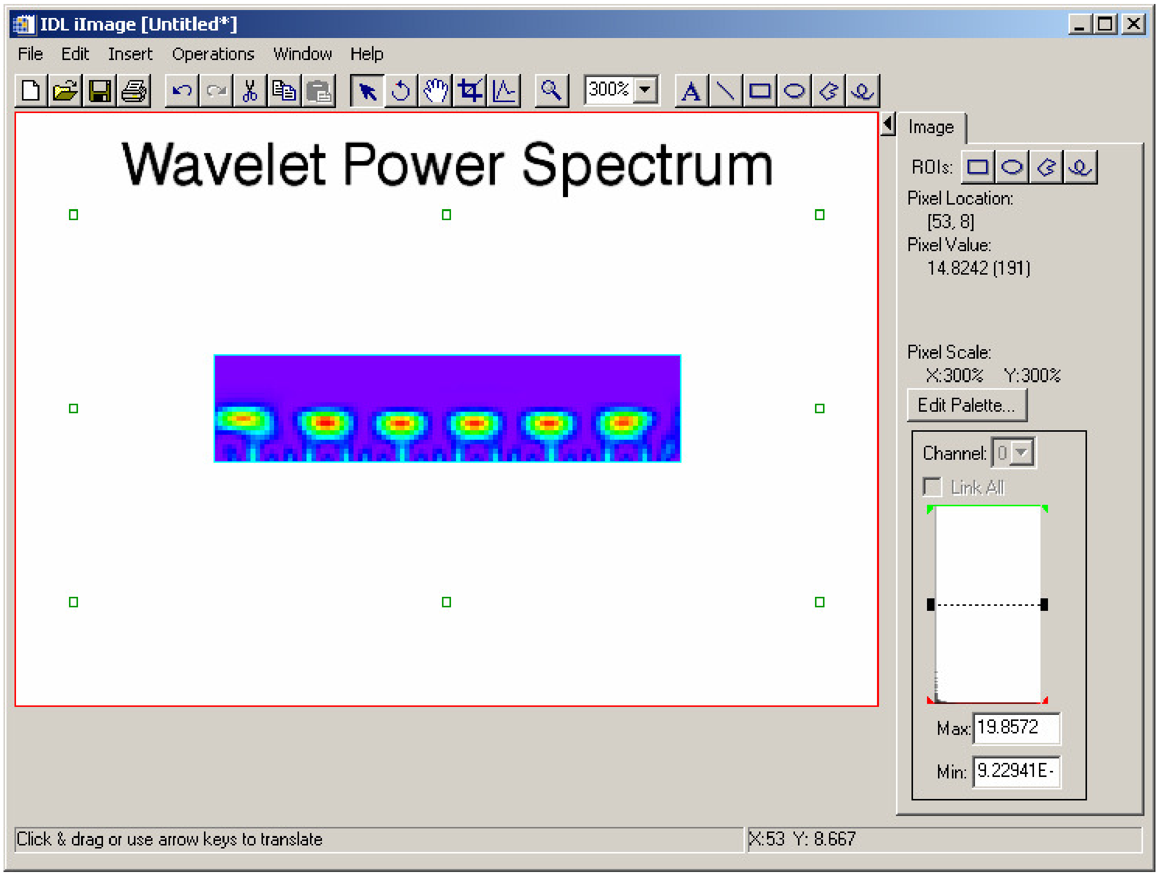


***Figure 6-13: Continuous wavelet power spectrum displayed as an image***

The display of the wavelet power spectrum clearly illustrates the sinusoidal components contained in the input signal.

10. Once finished viewing the continuous wavelet power spectrum, close the *IDL iImage* window.
11. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```

# Chapter 7: Advanced Image Processing

## Digital Images and Advanced iImage Operations

IDL provides a powerful environment for image processing and display. Digital images are easily represented as two-dimensional arrays in IDL and can be processed just like any other array. Within an image array the value of each pixel represents the intensity and/or color of that position in the scene. Images of this form are known as sampled or **raster** images, because they consist of a discrete grid of samples. IDL contains many procedures and functions specifically designed for image display and processing. In addition, the *iImage* tool allows the user great flexibility in manipulating and visualizing image data.

In the following exercise, the image from the example data file "*meteorite.bmp*" will be input into IDL. This example data file is located in the "*data*" subfolder of the Quick Start directory :

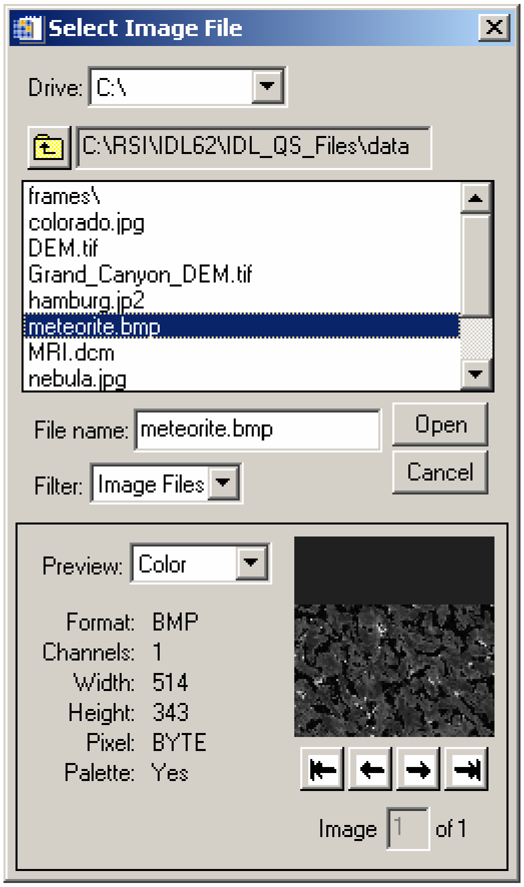
- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\meteorite.bmp*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/meteorite.bmp*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/meteorite.bmp*

The file is in Windows bitmap format and contains an image of a thin section taken through the Shergotty meteorite that is believed to represent a sample of the surface of Mars. Input the image data into the current IDL session by utilizing the *Import Image* macro built into the IDL Development Environment :

1. Start by selecting "*Macros > Import Image*" from the main IDL Development Environment window. A dialog entitled *Select Image File* will appear.
2. Navigate to the "*data*" subfolder of the Quick Start directory and select the "*meteorite.bmp*" file. Information on the image and a small preview will be displayed in the bottom of the *Select Image File* dialog [Fig. 7-1].
3. Press the "*Open*" button to read the image data into IDL and dismiss the *Import Image* wizard.

Once the *Import Image* macro is finished running the user will be returned to the main IDLDE window where a new variable named "*meteorite_image*" is now present within the current IDL session. The *HELP* procedure can be used to obtain information on this variable :

4. IDL>
```
HELP, meteorite_image
METEORITE_IMAGE STRUCT    = -> <Anonymous> Array[1]
```



Drive: C:\

C:\RSI\IDL62\IDL_QS_Files\data

File name: meteorite.bmp

Filter: Image Files

Preview: Color

Format: BMP
Channels: 1
Width: 514
Height: 343
Pixel: BYTE
Palette: Yes

Image 1 of 1

***Figure 7-1: The Import Image macro dialog***

The output from the *HELP* procedure shows that the "*meteorite_image*" variable is actually a structure containing multiple pieces of data and information read in from the BMP image file. To obtain information on the contents of this structure variable the *HELP* procedure must be executed with the *STRUCTURE* keyword set :

```
5. IDL> HELP, meteorite_image, /STRUCTURE
   ** Structure <14ca450>, 5 tags, length=177112, data
   length=177106, refs=1:
      IMAGE           BYTE       Array[514, 343]
      R               BYTE       Array[256]
      G               BYTE       Array[256]
      B               BYTE       Array[256]
      QUERY           STRUCT     -> <Anonymous> Array[1]
```

The actual image data from the BMP file is stored in the *IMAGE* tag of the structure variable, which contains a 2-dimensional array that has 514 columns and 343 rows with an 8-bit (BYTE) data type. The *R*, *G*, and *B* tags within the structure variable are provided to store the color table vectors that can be stored within 8-bit BMP files, which in this case are not necessary since the image is in simple grayscale mode. Furthermore, the *QUERY* tag stores yet another sub-structure that contains other useful information on the BMP image file.

In order to access the data that is stored within the fields of this "*meteorite_image*" structure, the period "." character must be used to reference the tags. For example, to view the information within the *QUERY* sub-structure field the following syntax must be utilized :

6. IDL> `HELP, meteorite_image.query, /STRUCTURE`

```
** Structure <14ca298>, 7 tags, length=40, data length=36,
refs=3:
   CHANNELS        LONG                 1
   DIMENSIONS      LONG      Array[2]
   HAS_PALETTE     INT              1
   NUM_IMAGES      LONG                 1
   IMAGE_INDEX     LONG                 0
   PIXEL_TYPE      INT              1
   TYPE            STRING    'BMP'
```

The *QUERY* field contains some useful information on the BMP image file. In order to access the actual image data stored within the "*meteorite_image*" structure in a manner that does not require a lot of typing, extract the *IMAGE* field of the structure and assign it to a new variable named "*image*" :

7. IDL> `image = meteorite_image.image`

Once this is accomplished a new variable is created at the main IDL level that is simply called "*image*" :

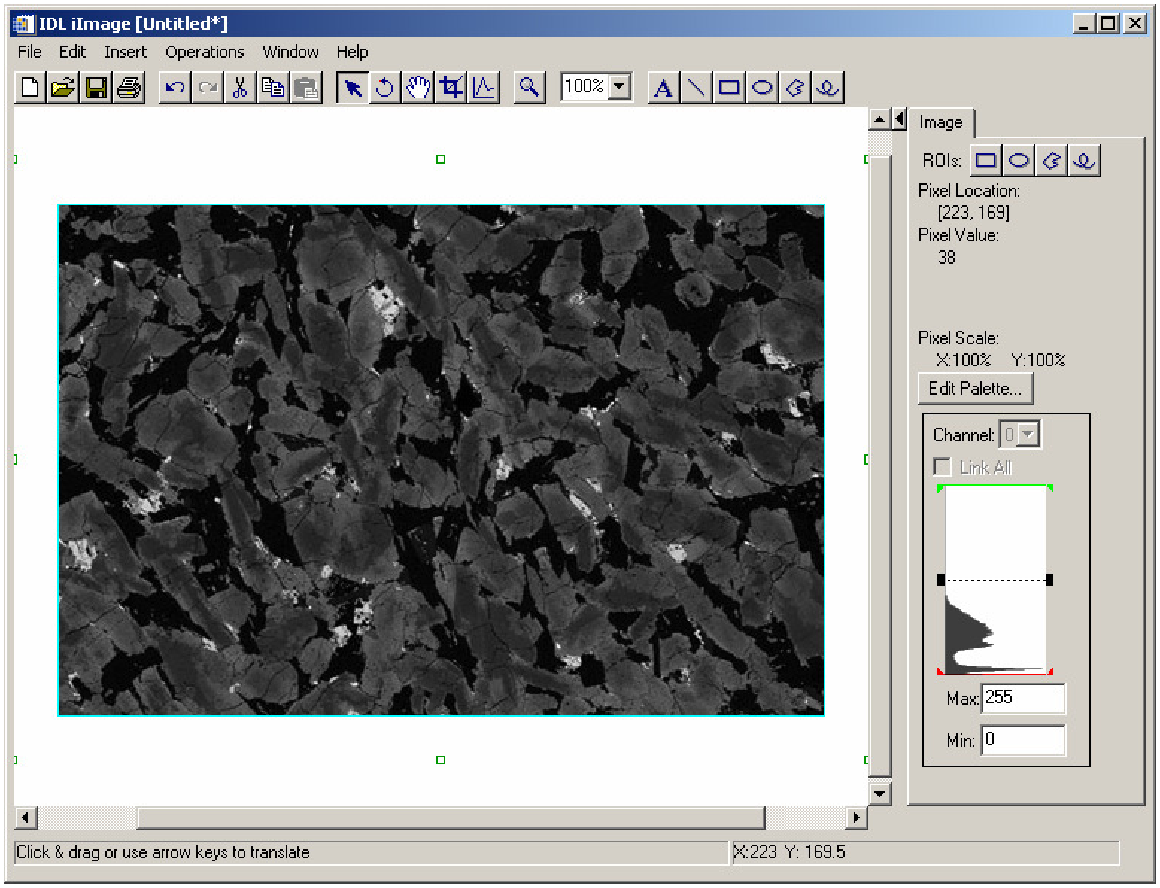
8. IDL> `HELP, image`

```
IMAGE           BYTE      = Array[514, 343]
```

Now that the image data has been extracted into a simple variable it can be easily visualized by loading it into the *iImage* utility :

9. IDL> `iImage, image`

The resulting *IDL iImage* visualization window should look similar to Fig. 7-2.



Pixel Location:
[223, 169]
Pixel Value:
38

Pixel Scale:
X:100% Y:100%

Channel: 0

Max: 255

Min: 0

Click & drag or use arrow keys to translate

X:223 Y: 169.5

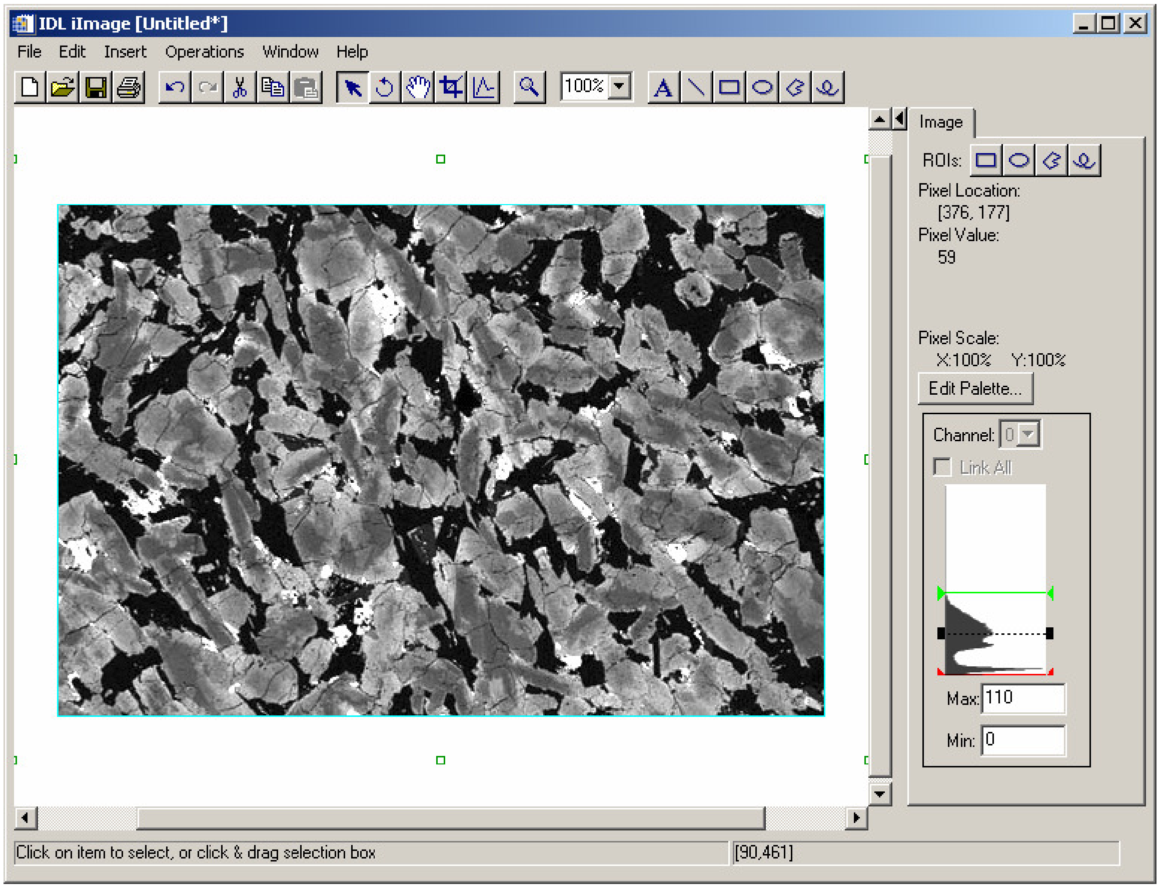
***Figure 7-2: Display of the Shergotty meteorite image within the iImage utility***

On the right-hand side of the *IDL iImage* window the "Min:" and "Max:" boxes show that the pixels in the image range throughout the full 8-bit range (0 → 255). However, the histogram plot window illustrates that most of the pixels within the image have brightness values in the lower half of the data range. The histogram plot is essentially portraying the overall dark gray to black appearance of the image.

A simple form of **image enhancement** can be obtained by moving the histogram threshold bars within the *iImage* utility. This will adjust the range of pixel data values that are mapped to the 256 levels of gray displayed on the screen. The **stretching** of the image in the defined range is performed in a linear fashion and this provides a form of **contrast** enhancement.

10. Within the "Max:" field box, type a pixel data value of 110 and press the *Enter* key on the keyboard. This will saturate all pixels in the image with a value of 110 or higher to white, while stretching the pixels with values 0 → 109 throughout the full range of the grayscale display.

The resulting *IDL iImage* visualization window should look similar to Fig. 7-3.



Pixel Location:
[376, 177]
Pixel Value:
59

Pixel Scale:
X:100% Y:100%

Channel: 0

Max: 110

Min: 0

Click on item to select, or click & drag selection box

[90,461]

***Figure 7-3: Contrast enhancement of the image via a linear stretch***

This manipulation of the histogram stretch bars only affects the display of the image and does not change the actual pixel values for the image dataset. Notice that the "Pixel Value:" field within the Image panel now displays a number in parentheses next to the actual pixel data value. This number in parentheses is the output grayscale intensity for the current pixel according to the stretch that is being applied.

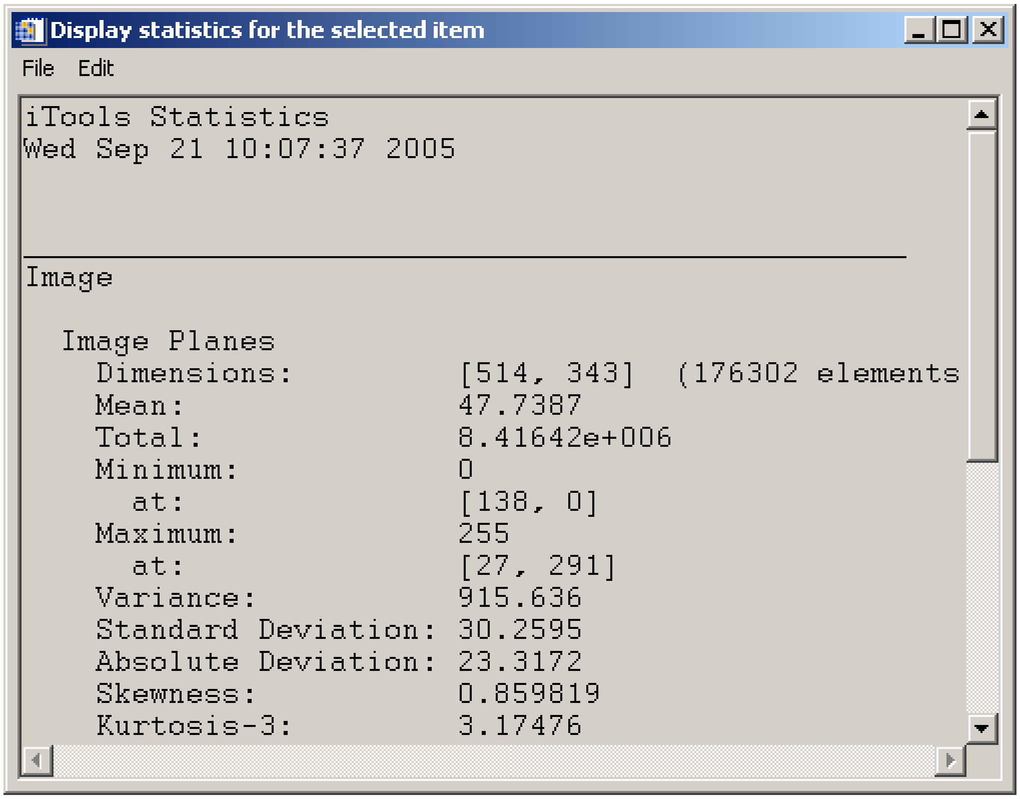
11. Change the "Max:" field back to "*255*" by either typing in the text box or clicking on the green stretch bar and dragging it back up to the top of the histogram display window.

There are a number of analysis tools found within the *Operations* menu of the *iImage* utility. The operations that are built into the iTools system represent some of the most common image processing tasks.

12. While the image object is selected within the *IDL iImage* window, select "*Operations > Statistics…*" from the menu system.

A separate dialog will appear that displays some statistical information on the current image dataset [Fig. 7-4]. Notice that the average (mean) pixel value for this image is 47.7387, which explains the relatively dark appearance of the original image.

13. Once finished viewing the image statistics, close the *Display statistics for the selected item* dialog.



```
iTools Statistics
Wed Sep 21 10:07:37 2005



Image

  Image Planes
    Dimensions:          [514, 343]  (176302 elements
    Mean:                47.7387
    Total:               8.41642e+006
    Minimum:             0
      at:                [138, 0]
    Maximum:             255
      at:                [27, 291]
    Variance:            915.636
    Standard Deviation:  30.2595
    Absolute Deviation:  23.3172
    Skewness:            0.859819
    Kurtosis-3:          3.17476
```

***Figure 7-4: Statistics for the meteorite thin section image***

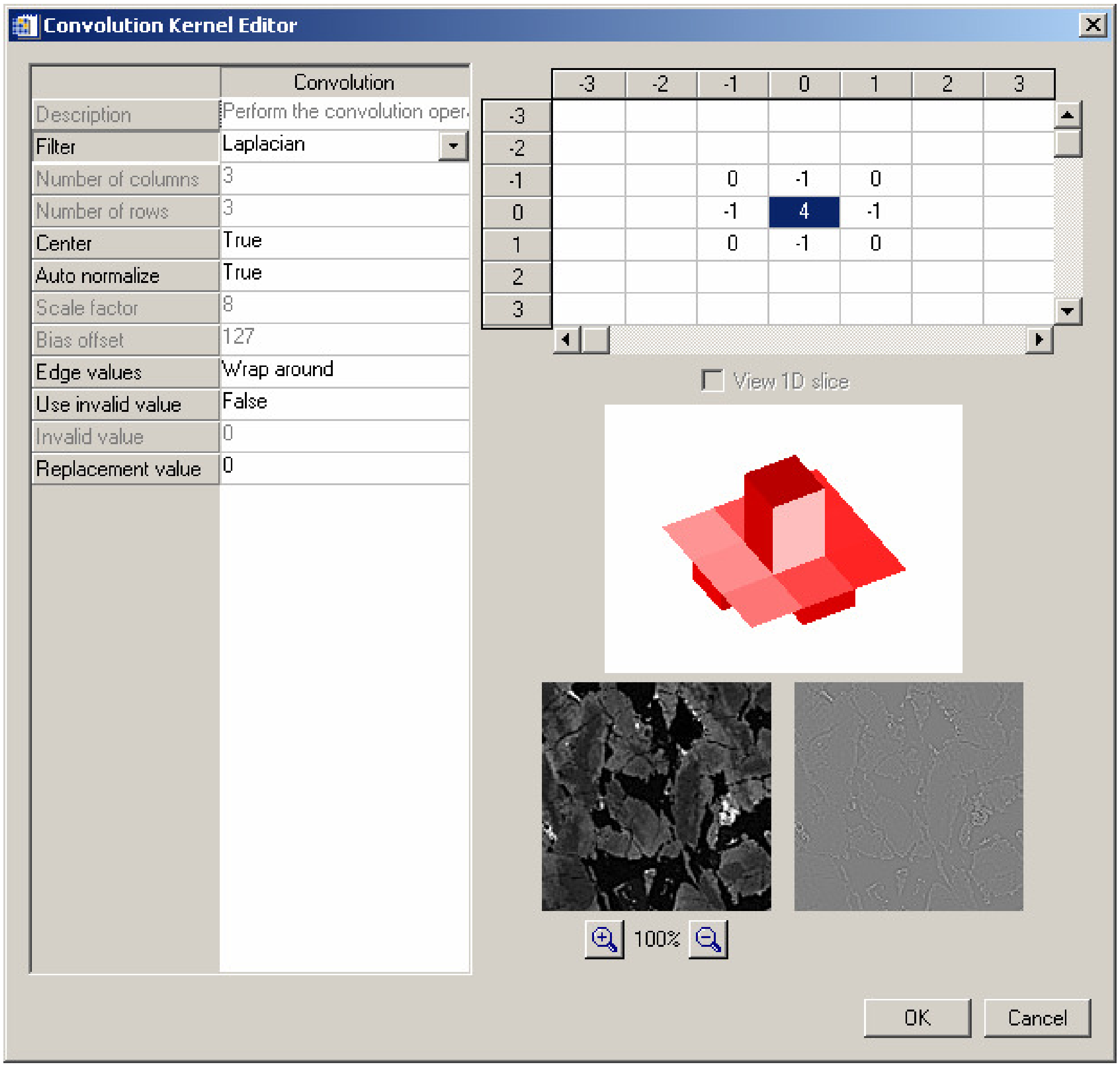
One of the operations built into the *iImage* utility is the **Unsharp Masking** technique, which applies a **sharpening** filter to the image. Digital Unsharp Masking is a digital image processing technique that increases the contrast where subtle details are set against a diffuse background. This operation suppresses features which are smooth (those with structures on large scales) in favor of sharp features (those with structure on small scale), resulting in a net enhancement of the contrast of fine structure in the image.

14. From the *IDL iImage* window menu system select "*Operations > Filter > Unsharp Mask*".
15. Within the *Unsharp Mask* dialog that pops up, leave all parameters set to their default values and simply press "*OK*".

Notice that the fine detail within the image is enhanced by applying the Unsharp Mask operation.

The *iImage* utility also has a built-in tool for convolving an image array with a kernel. **Convolution** is a simple matrix algebra operation that can be used for various types of smoothing, shifting, differentiation, edge detection, etc..

16. Select "*Operations > Filter > Convolution*" from the *iImage* menu system. A separate dialog entitled *Convolution Kernel Editor* will appear [Fig. 7-5].



Convolution Kernel Editor

| | Convolution |
|---|---|
| Description | Perform the convolution oper |
| Filter | Laplacian |
| Number of columns | 3 |
| Number of rows | 3 |
| Center | True |
| Auto normalize | True |
| Scale factor | 8 |
| Bias offset | 127 |
| Edge values | Wrap around |
| Use invalid value | False |
| Invalid value | 0 |
| Replacement value | 0 |

| | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|
| -3 | | | | | | | |
| -2 | | | | | | | |
| -1 | | | 0 | -1 | 0 | | |
| 0 | | | -1 | 4 | -1 | | |
| 1 | | | 0 | -1 | 0 | | |
| 2 | | | | | | | |
| 3 | | | | | | | |

***Figure 7-5: The iTools Convolution Kernel Editor dialog***

The *Convolution Kernel Editor* window allows the user to select from a list of pre-defined kernels, or define their own user-defined kernel.  The convolution of these different kernels will have a wide variety of effects on the resulting image display.

The **Laplacian** filter can be applied by convolving a Laplacian kernel with the image. A Laplacian filter is an edge enhancement filter that operates without regard to edge direction. Laplacian filtering emphasizes maximum values within the image by using a kernel with a high central value typically surrounded by negative weights in the up down and left-right directions and zero values at the kernel corners.  The Laplacian kernel convolution is a form of **high pass** filter, which removes the low frequency components of an image while retaining the high frequency (local variations). It can be used to **enhance edges** between different regions as well as to sharpen an image.

17. Using the droplist next to the "Filter" parameter in the upper left hand corner of the *Convolution Kernel Editor* dialog, select the "*Laplacian*" kernel.

18. Notice that a surface representation of the current kernel is displayed within this dialog. The user can click on this surface and rotate it in order to visualize the structure of the kernel.
19. Once the "*Laplacian*" kernel has been selected press "*OK*" to apply the convolution operation and dismiss the *Convolution Kernel Editor* dialog.
20. At this point, it is beneficial to change the range for the current stretch to the following values :
    - Max: 140
    - Min: 115

The application of the Laplacian filter will enhance the edges between different regions (in this case mineral grains) within the image. The resulting *IDL iImage* visualization window should look similar to Fig. 7-6.

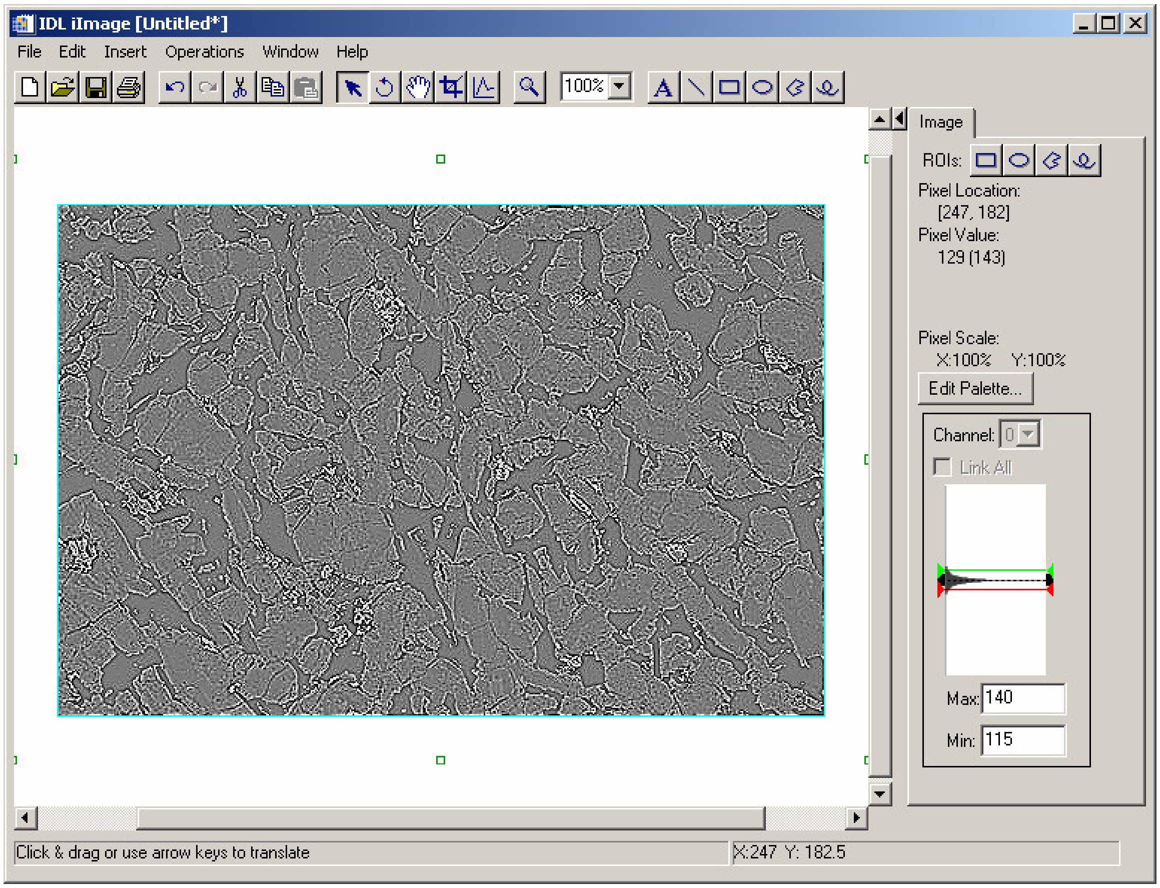


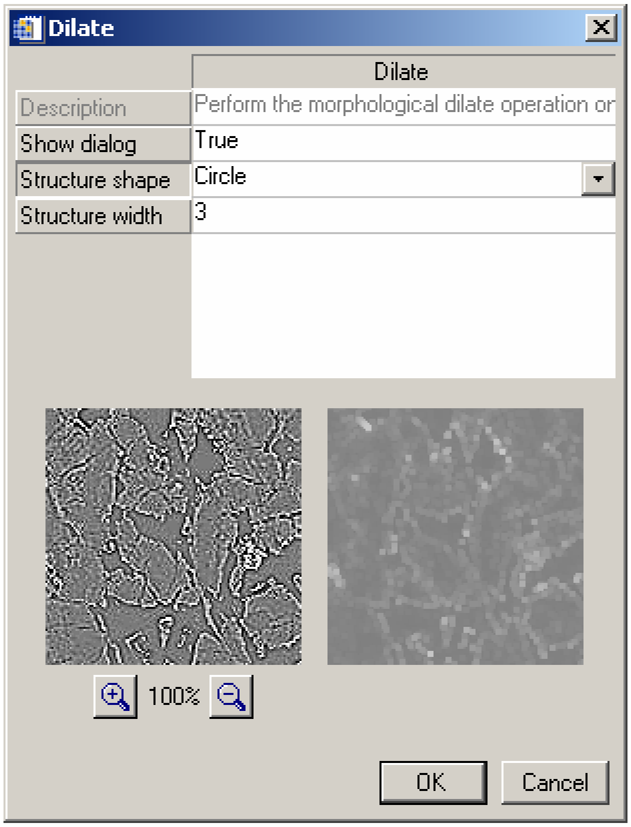
***Figure 7-6: Application of a Laplacian filter convolution***

IDL also has a number of morphological image operators built into its library of routines. Mathematical morphology is a method of processing digital images on the basis of shape. Some of these morphological algorithms have been added to the operations within the iTools system. For example, the dilate operator, which is commonly known as the "fill", "expand", or "grow" operator, can be used to further enhance the boundaries between mineral grains in the current image.

21. Select "*Operations > Morph > Dilate*" from the *iImage* menu system. A separate dialog entitled *Dilate* will appear [Fig. 7-7]. The parameters

associated with the dilate operation are displayed in this dialog along with a preview of the operation.

22. Click on the box to the right of the "Structure shape" field and change the setting to "Circle" [Fig. 7-7].
23. Press the "*OK*" button to apply the dilate operation and dismiss the *Dilate* dialog.

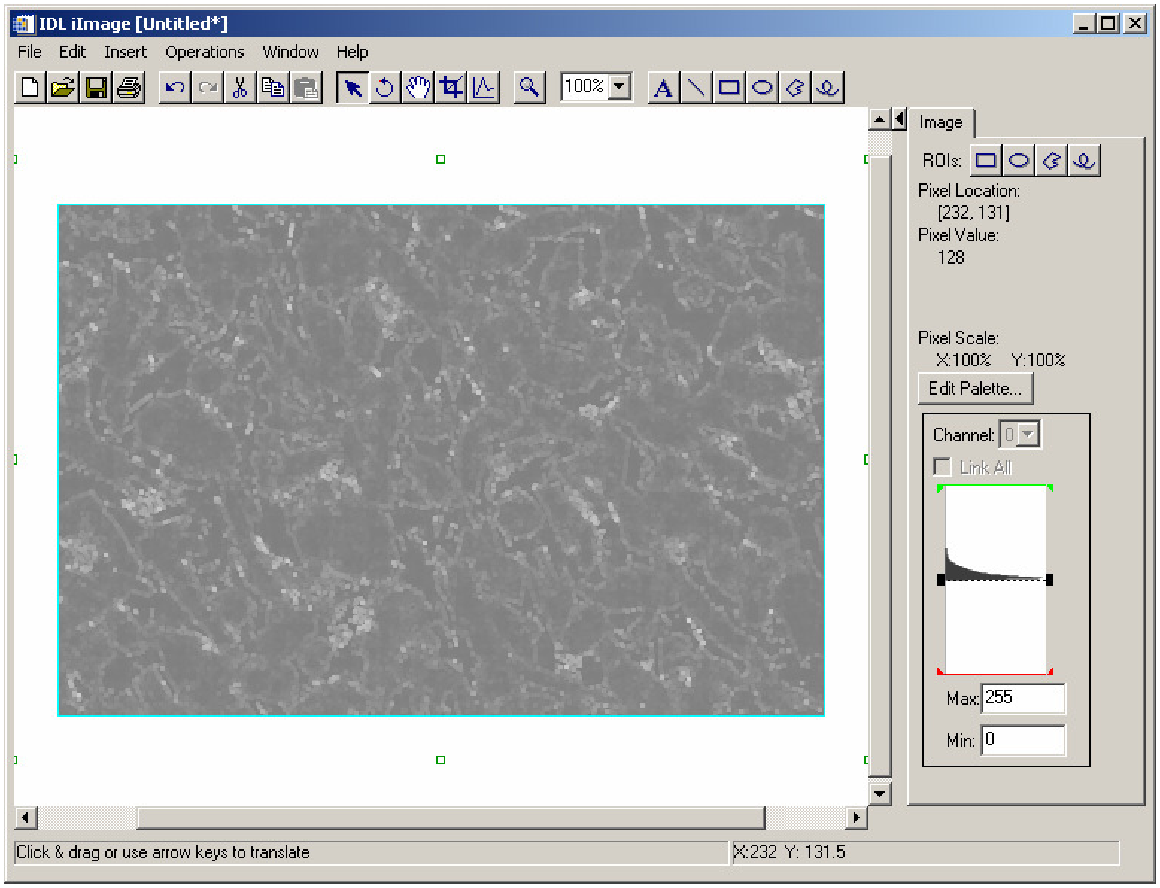


Dilate

| | Dilate |
|---|---|
| Description | Perform the morphological dilate operation or |
| Show dialog | True |
| Structure shape | Circle |
| Structure width | 3 |

***Figure 7-7: The iTools Dilate operation dialog***

The resulting *IDL iImage* visualization window should look similar to Fig. 7-8.



Pixel Location:
[232, 131]
Pixel Value:
128
Pixel Scale:
X:100% Y:100%
Channel: 0
Max: 255
Min: 0
Click & drag or use arrow keys to translate
X:232 Y: 131.5

***Figure 7-8: Application of the Dilate morphological operation***

24. Once finished viewing the processed image, close the *IDL iImage* window.

In addition to sharpening, high pass filtering, and edge enhancement techniques, IDL also offers operations to perform image **smoothing**, **low pass filtering**, and **noise removal**. Re-launch the *iImage* utility with the original image so these techniques can be investigated :

25. IDL> `iImage, image`

The median operation replaces each pixel with the median of the two-dimensional neighborhood of the specified width. In an ordered set of values, the median is a value with an equal number of values above and below it. Median filtering is effective in removing salt and pepper noise (isolated high or low values). The resulting image will have a less grainy appearance than the original.

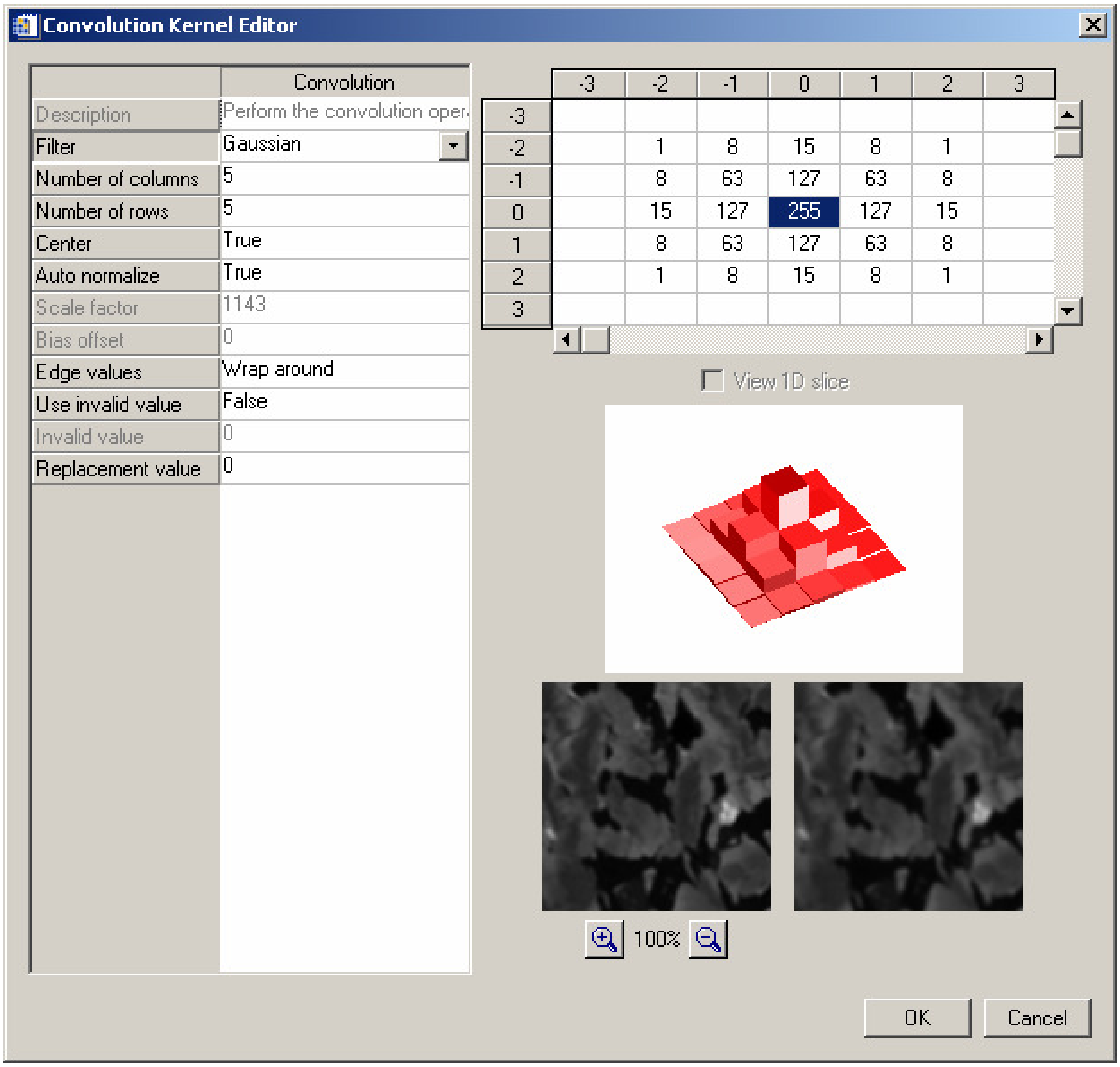
26. From the *iImage* menu system, select "*Operations > Filter > Median*".
27. Within the *Median* dialog window, leave all of the default settings and press "*OK*".

The smooth operation will compute the boxcar average of a specified width for the image. Smoothing is similar to the median filter except the pixels are replaced with the average (mean) value across the neighborhood. This tends to have the effect of blurring the edges within the image and making them more diffuse.

28. From the *iImage* menu system, select "*Operations > Filter > Smooth*".
29. Press the "*OK*" button to apply the smoothing operation and dismiss the *Smooth* dialog window.

Finally, the Convolution tool can be used once again to apply a low pass filter to the image. The Gaussian kernel provides a form of low pass filtering that preserves the low frequency components of an image.

30. Select "*Operations > Filter > Convolution*" from the *iImage* menu system.
31. Within the *Convolution Kernel Editor* dialog, change the "Filter" selection droplist to "*Gaussian*" [Fig. 7-9].
32. Edit the number of columns and rows fields so the kernel has a size of 5 x 5 [Fig. 7-9].
33. Press the "*OK*" button to apply the Gaussian filter and dismiss the dialog.



Convolution Kernel Editor

| | Convolution |
|---|---|
| Description | Perform the convolution oper. |
| Filter | Gaussian |
| Number of columns | 5 |
| Number of rows | 5 |
| Center | True |
| Auto normalize | True |
| Scale factor | 1143 |
| Bias offset | 0 |
| Edge values | Wrap around |
| Use invalid value | False |
| Invalid value | 0 |
| Replacement value | 0 |

| | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
|---|---|---|---|---|---|---|---|
| -3 | | | | | | | |
| -2 | | 1 | 8 | 15 | 8 | 1 | |
| -1 | | 8 | 63 | 127 | 63 | 8 | |
| 0 | | 15 | 127 | 255 | 127 | 15 | |
| 1 | | 8 | 63 | 127 | 63 | 8 | |
| 2 | | 1 | 8 | 15 | 8 | 1 | |
| 3 | | | | | | | |

***Figure 7-9: Application of the Gaussian filter convolution***

The resulting *IDL iImage* visualization window should look similar to Fig. 7-10.

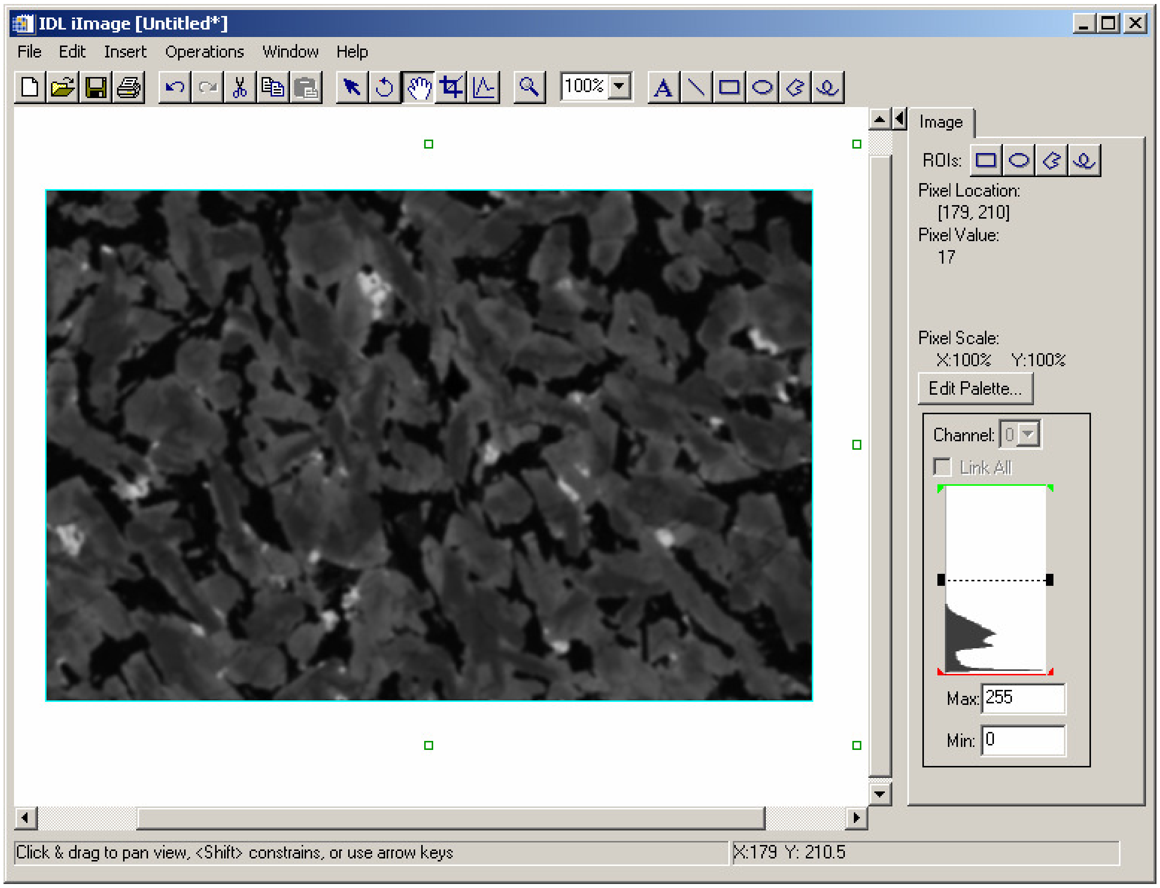


***Figure 7-10: Image display after noise removal, smoothing, and low pass filtering***

34. Once finished viewing the processed image, close the *IDL iImage* window.

## Thresholding, Clipping, and Histogram Equalization

Although the *iImage* utility has a lot of built-in analytical techniques, the bulk of IDL's image processing capabilities must be accessed using routines within the IDL language. IDL's image processing library contains a number of routines for contrast enhancement, filtering, feature extraction, image segmentation, geometry transformations, and regions of interest analysis. In addition, the IDL language has built-in operators that can be utilized to perform simple image processing techniques such as masking and stretching.

**Thresholding** (also known as masking) is used to isolate features within an image above, below, or equal to a specified pixel value. The value (known as the threshold level) determines how the masking occurs. In IDL, thresholding is performed using the relational operators. IDL's relational operators are illustrated in Table 7-1 :

| OPERATOR | DESCRIPTION |
|---|---|
| EQ | Equal to |
| NE | Not equal to |
| GE | Greater than or equal to |
| GT | Greater than |
| LE | Less than or equal to |
| LT | Less than |

***Table 7-1 : IDL's Relational Operators***

For example, in order to threshold the Shergotty meteorite image and identify the pixels that have a value greater than 70 (byte) simply execute the following statement :

1. IDL> `mask = image GT 70B`

This expression creates a new variable named "*mask*" that is a 2-dimensional array of the same size as the original image. This new "*mask*" variable contains a binary image where each pixel has a value of either one (original image pixel value was greater than 70) or zero (original image pixel value was equal to or less than 70). At this point, the user may wish to view this binary threshold image by loading it into the *iImage* utility :

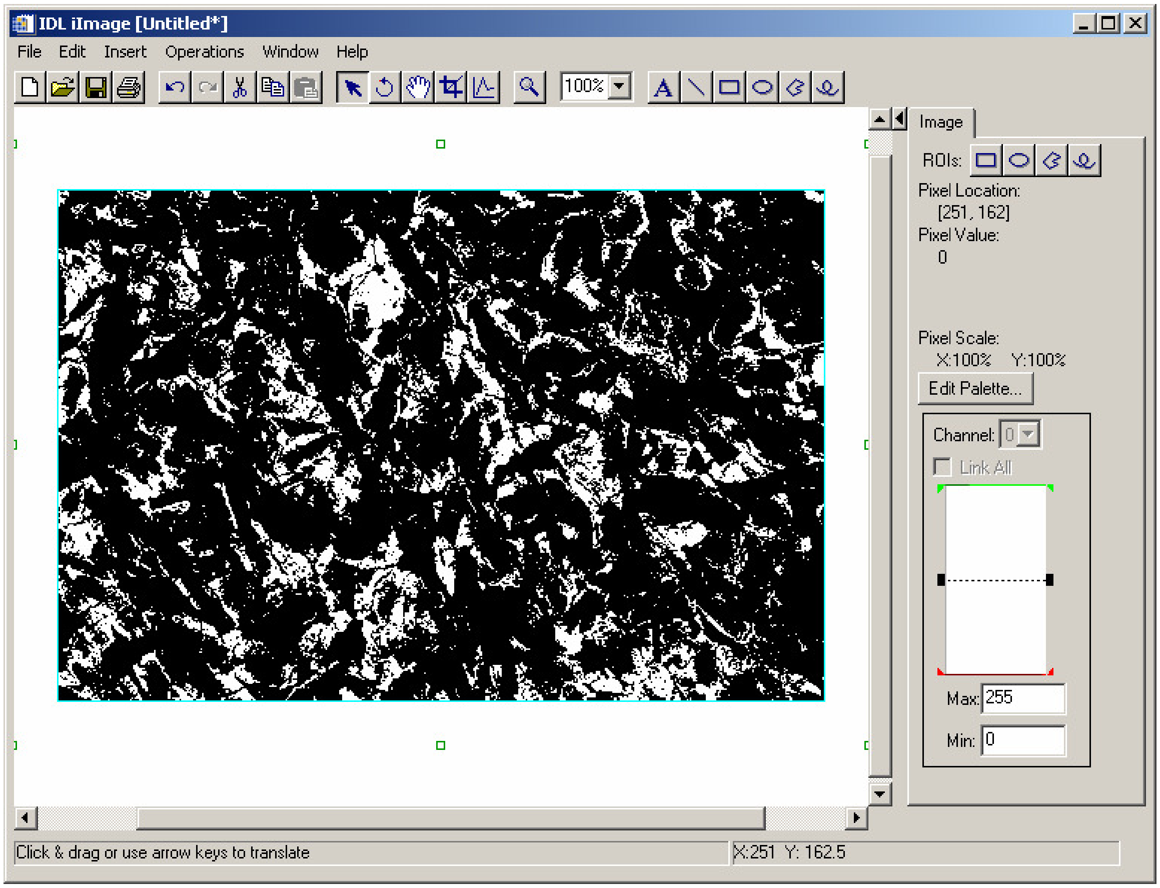
2. IDL> `iImage, mask`

The resulting image display within the *IDL iImage* window should appear completely black. This is due to the fact that all of the pixels within the "*mask*" binary image have a value of 0 or 1, which are very difficult to discern (and very dark) within a 0 → 255 grayscale display. Consequently, the *BYTSCL* function should be utilized when displaying binary images so the pixels with a value of 1 are actually mapped to 255 (white). The *BYTSCL* function scales all values of an array into a specified range (0 → 255 by default) :

3. Close the existing *IDL iImage* window.
4. Re-issue the *iImage* statement, but this time wrap the mask image variable with a dynamic call to the *BYTSCL* function :

   IDL> `iImage, BYTSCL (mask)`

The resulting *IDL iImage* visualization window should look similar to Fig. 7-11.



Pixel Location: [251, 162]
Pixel Value: 0
Pixel Scale: X:100% Y:100%
Channel: 0
Max: 255
Min: 0
Click & drag or use arrow keys to translate
X:251 Y: 162.5

***Figure 7-11: Threshold image showing pixels with values greater than 70 (white) and less than or equal to 70 (black)***

5. Once finished viewing the binary threshold image, close the *IDL iImage* window.

Binary threshold images can also be used to **mask-out** the pixels in an image that do not qualify based on the given expression. For example, in order to display only those pixels from the original image that have a value greater than 70 simply execute the following statements :

6. IDL> `masked = image * mask`
7. IDL> `iImage, masked`

The resulting *IDL iImage* visualization window should look similar to Fig. 7-12.

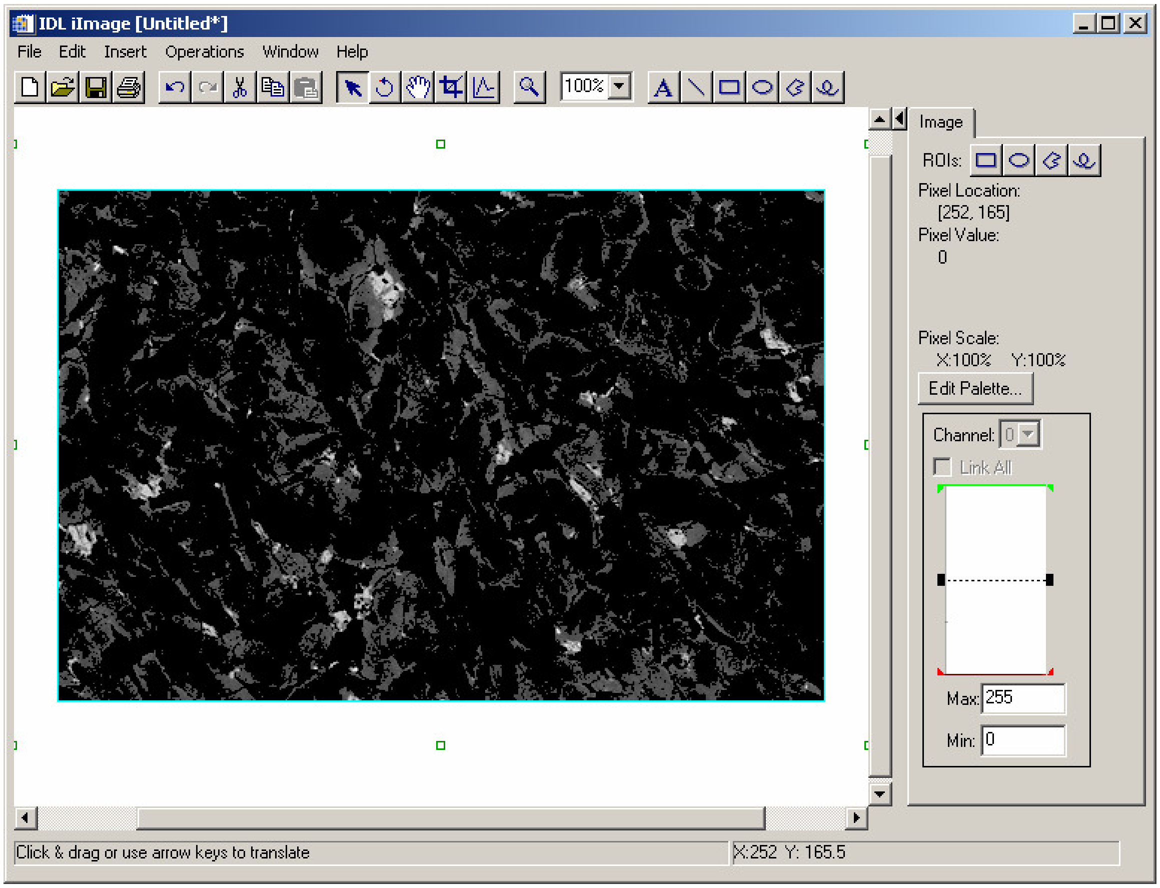


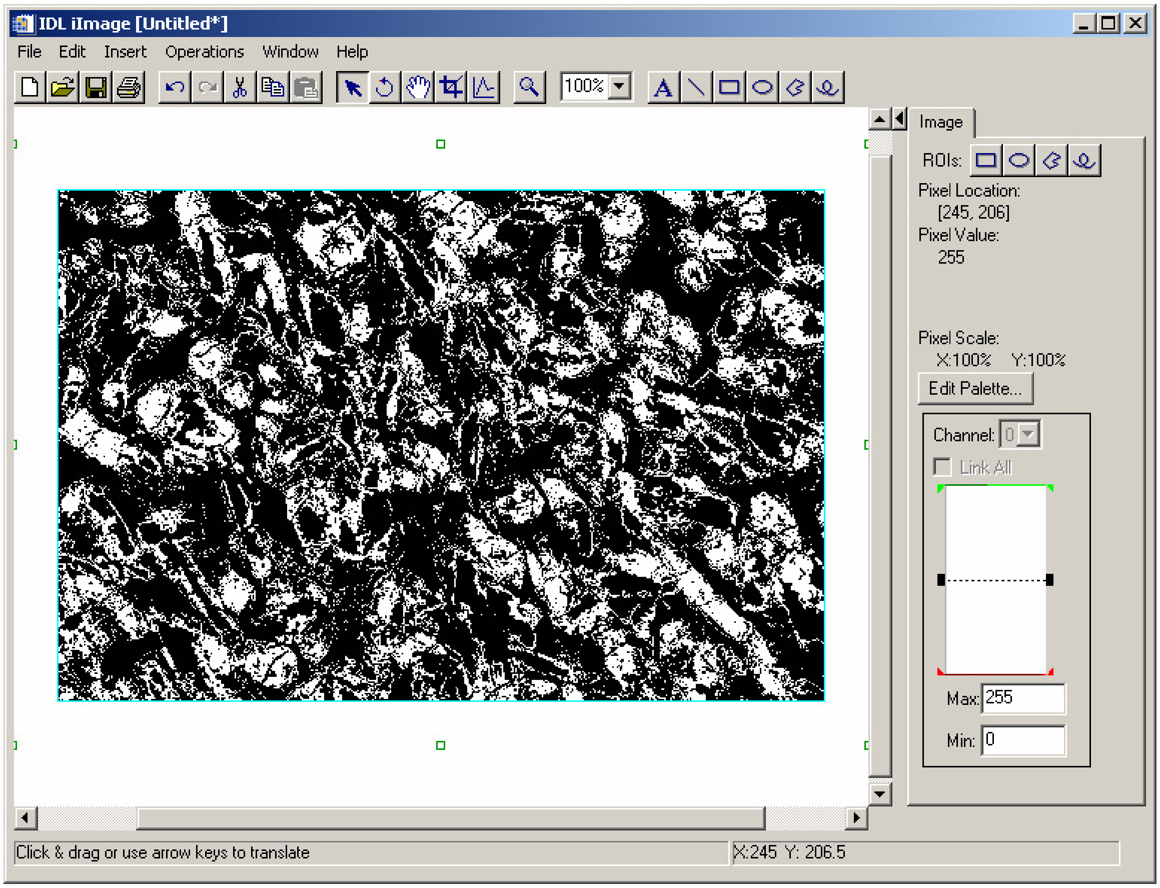
***Figure 7-12: Display of the original image with all pixels that have a value of 70 or less masked-out (i.e. displayed as black)***

8. Once finished viewing the masked image, close the *IDL iImage* window.

The user can also provide both upper and lower bounds when creating threshold images by using the Boolean operators built into IDL (*AND*, *NOT*, *OR*, and *XOR*). For example, create a threshold image that identifies those pixels which have a data value between 50 and 70 :

```
9.  IDL> mask = (image GE 50B) AND (image LE 70B)
10. IDL> iImage, BYTSCL (mask)
```

The resulting *IDL iImage* visualization window should look similar to Fig. 7-13.



Pixel Location: [245, 206]
Pixel Value: 255
Pixel Scale: X:100% Y:100%
Channel: 0
Max: 255
Min: 0
Click & drag or use arrow keys to translate
X:245 Y: 206.5

***Figure 7-13: Threshold image showing all pixels with data values between 50 and 70***

11. Once finished viewing the threshold image, close the *IDL iImage* window.

Clipping is similar to thresholding because pixels with data values above or below a specified level are all set to the same value. However, when clipping an image the pixels that do not satisfy the expression are set to the selected level and the resulting image is not binary in nature. Clipping can be used to enhance features within an image.

In IDL, clipping is performed with the minimum (<) and maximum (>) operators. In order to clip an image the user must design an expression that contains an image array, the appropriate operator, and the clipping level. For example, to clip the meteorite thin section image so that all pixels with a value greater than or equal to 50 are set to a value of 50 simply execute the following statements :

12. IDL> `clipped = image < 50B`
13. IDL> `iImage, clipped`

The resulting *IDL iImage* visualization window should look similar to Fig. 7-14.

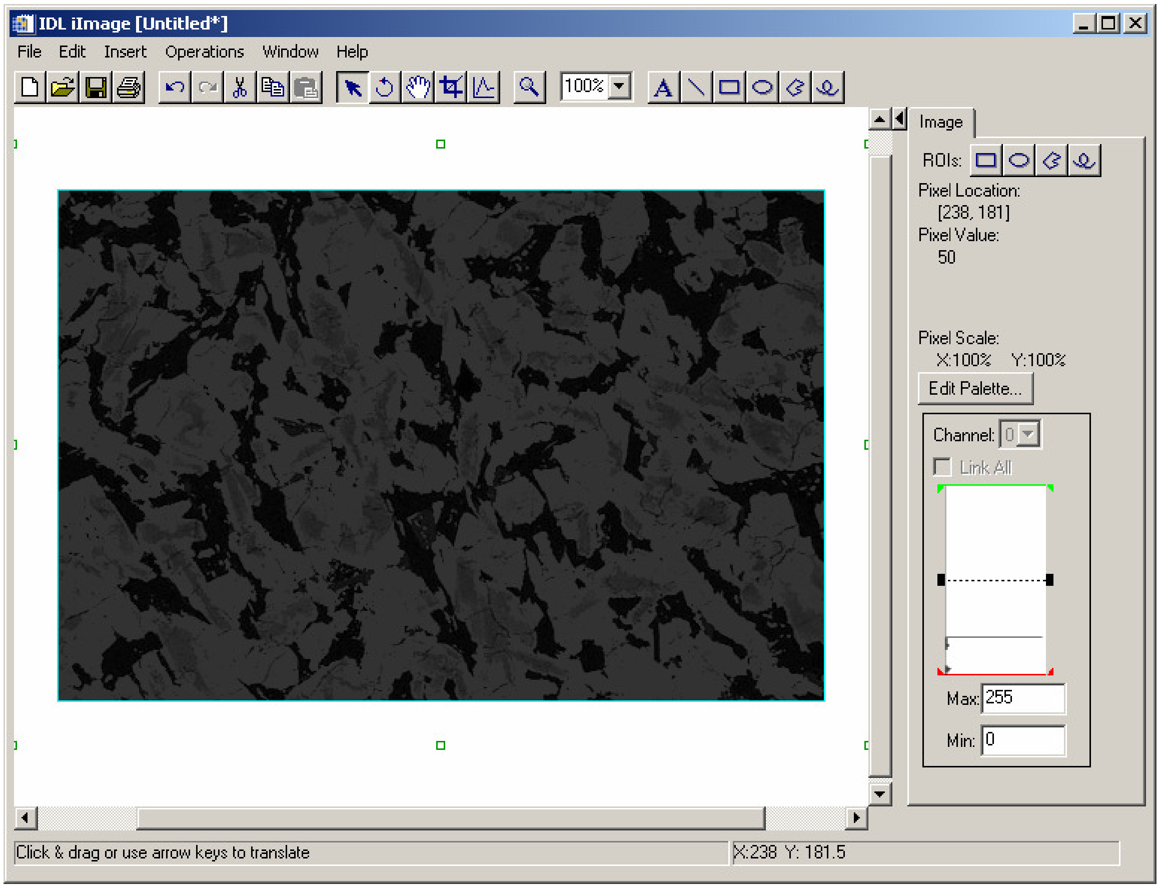


***Figure 7-14: Clipped image showing all pixels with data values 50 or higher set to a brightness level of 50***

14. Once finished viewing the clipped image, close the *IDL iImage* window.

When clipping is used in conjunction with byte-scaling it is equivalent to performing a stretch on an image. For example, in order to stretch the image between the range of 25 → 100 simply execute the following statements :

```
15. IDL> stretched = BYTSCL (image > 25B < 100B)
16. IDL> iImage, stretched
```

It is worth mentioning that the same stretching technique can be obtained by utilizing the *MIN* and *MAX* keywords to the *BYTSCL* function :

```
17. IDL> stretched = BYTSCL (image, MIN=25, MAX=100)
18. IDL> iImage, stretched
```

The resulting *IDL iImage* visualization window(s) should look similar to Fig. 7-15.

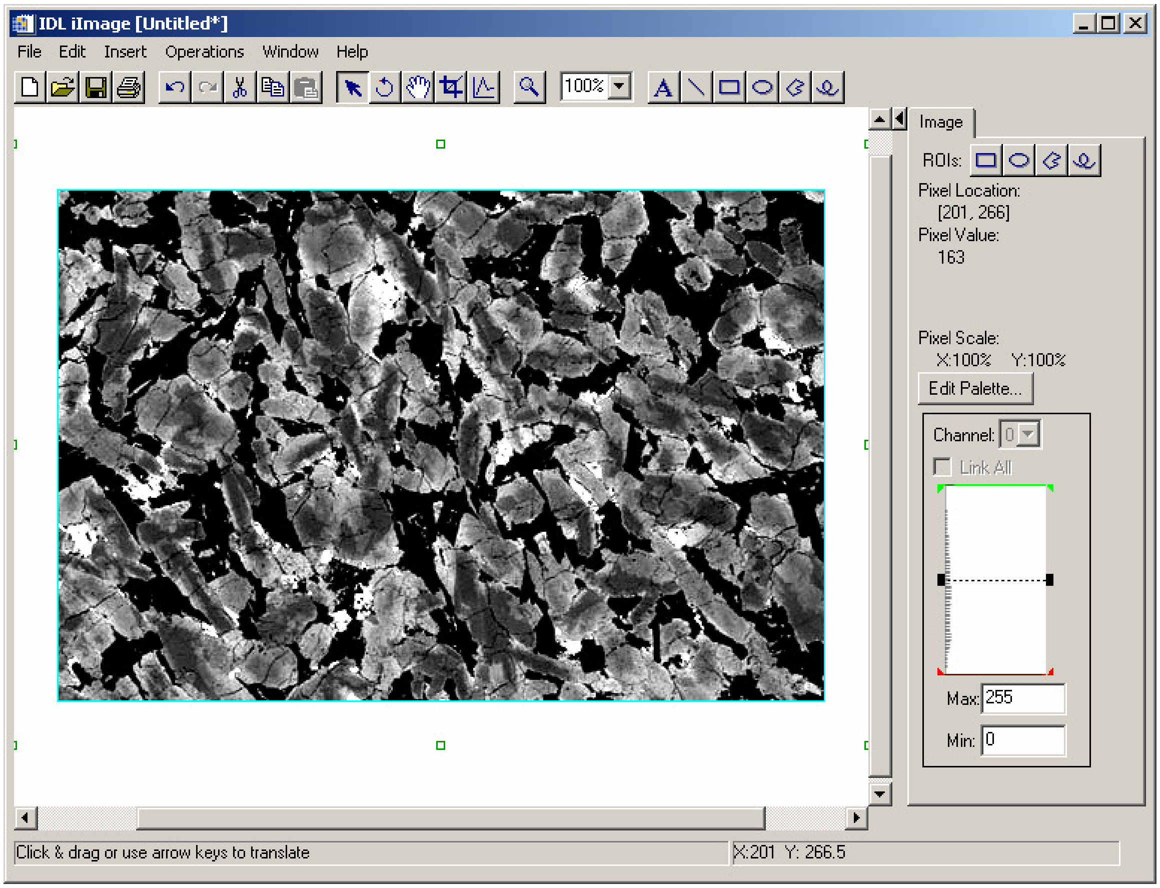


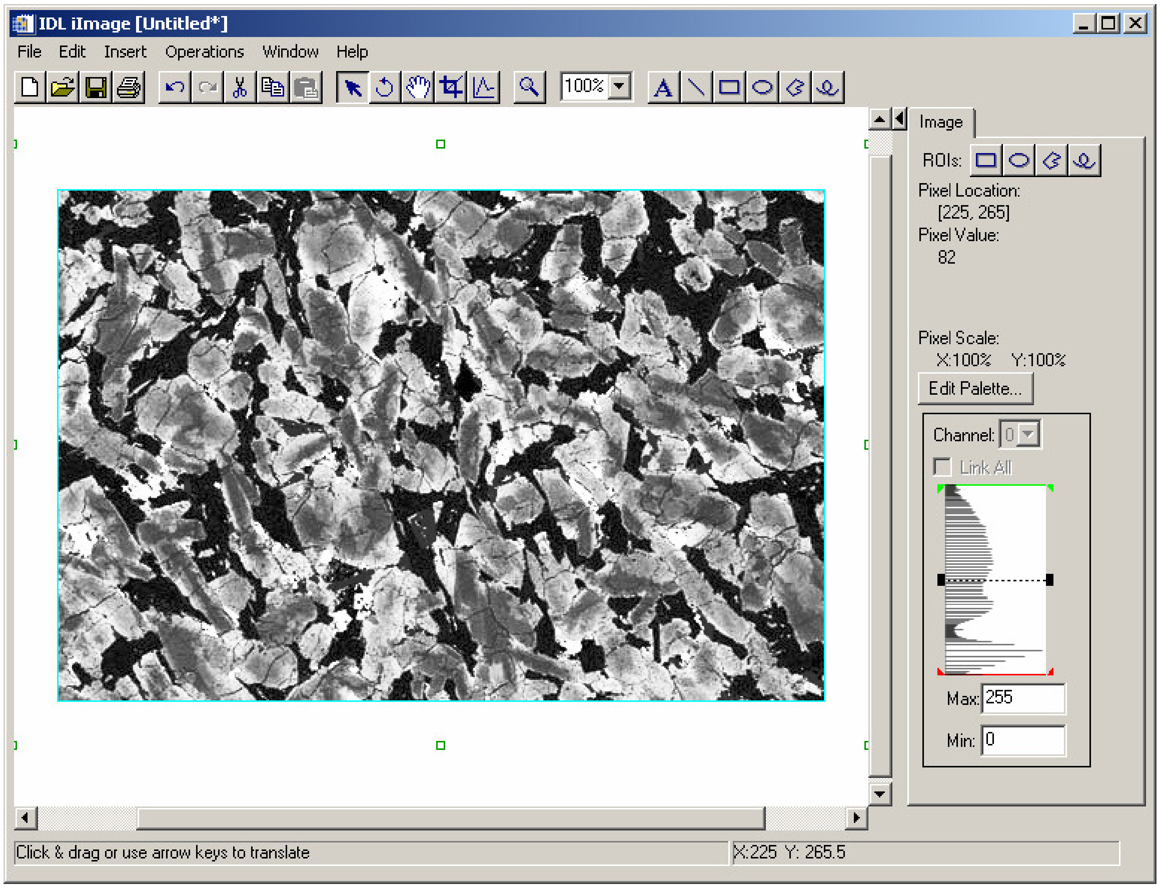
***Figure 7-15: Stretched image that highlights all pixels with data values between 25 and 100***

19. Once finished viewing the stretched image, close the *IDL iImage* window(s).

In addition to simple linear stretching techniques, IDL also has routines that allow the user to stretch the image using other histogram manipulations. For example, the *HIST_EQUAL* function can be used to apply a histogram equalization stretch to the image data. Histogram equalization employs a monotonic, non-linear mapping which re-assigns the intensity values of pixels in the input image such that the output image contains a uniform distribution of intensities (i.e. a flat histogram). Execute the following statements in order to derive and display the histogram-equalized version of the meteorite thin section image :

20. IDL> `equalized = HIST_EQUAL (image)`
21. IDL> `iImage, equalized`

Notice that the resulting image has improved contrast and the histogram has a very even distribution throughout the 0 → 255 range. The resulting *IDL iImage* visualization window should look similar to Fig. 7-16.



Pixel Location: [225, 265]
Pixel Value: 82
Pixel Scale: X:100% Y:100%
Channel: 0
Max: 255
Min: 0

Click & drag or use arrow keys to translate
X:225 Y: 265.5

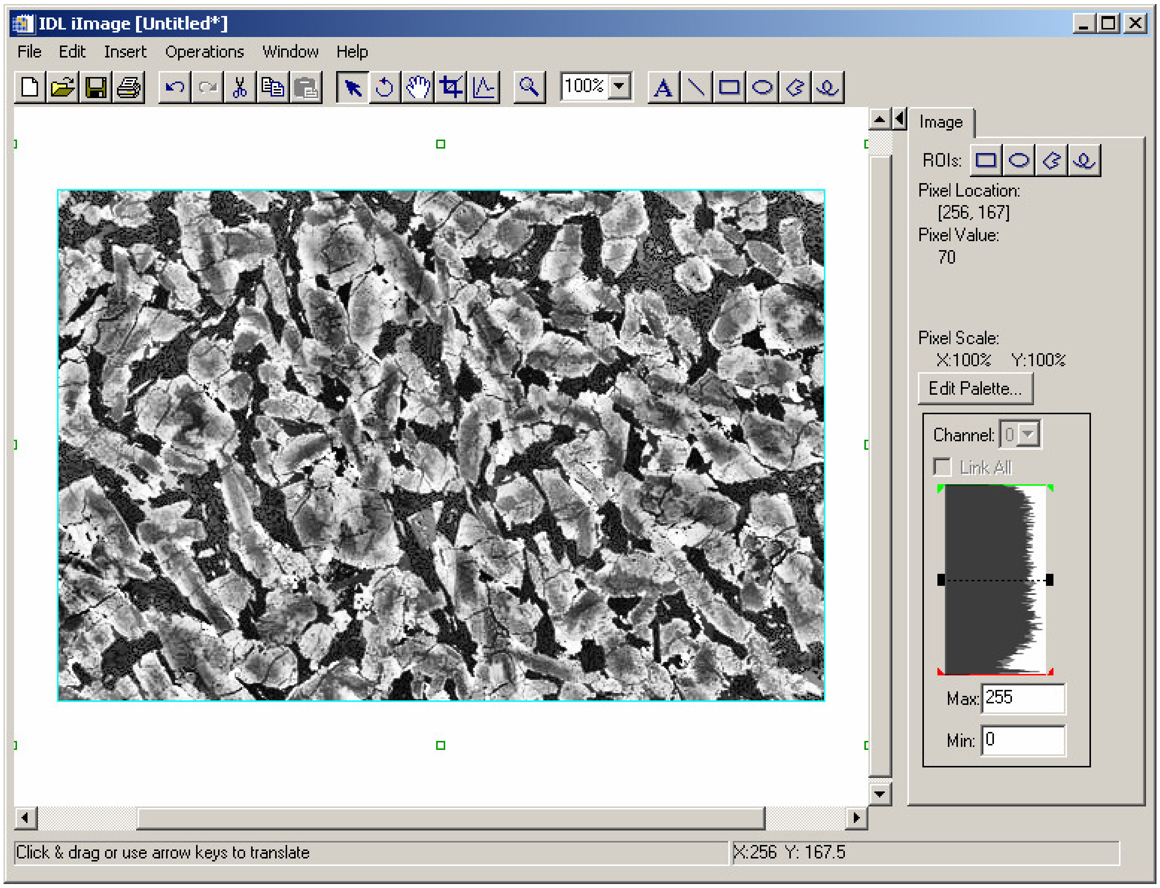
***Figure 7-16: Image display with a histogram equalization stretch***

22. Once finished viewing the histogram-equalized image, close the *IDL iImage* window.

In addition to the standard histogram equalization provided by the *HIST_EQUAL* function, IDL also provides the *ADAPT_HIST_EQUAL* function which performs adaptive histogram equalization (a form of automatic image contrast enhancement). Adaptive histogram equalization involves applying contrast enhancement based on the local region surrounding each pixel. Each pixel is mapped to an intensity proportional to its rank within the surrounding neighborhood. This method of automatic contrast enhancement has proven to be broadly applicable to a wide range of images and to have demonstrated effectiveness. Execute the following statements in order to apply the adaptive histogram equalization and display the resulting image :

```
23. IDL> adaptive = ADAPT_HIST_EQUAL (image)
24. IDL> iImage, adaptive
```

The resulting *IDL iImage* visualization window should look similar to Fig. 7-17.



Pixel Location:
[256, 167]
Pixel Value:
70

Pixel Scale:
X:100% Y:100%

Channel: 0

Max: 255

Min: 0

Click & drag or use arrow keys to translate

X:256 Y: 167.5

***Figure 7-17: Image display with an adaptive histogram equalization stretch***

25. Once finished viewing the adaptive histogram-equalized image, close the *IDL iImage* window.

## Morphological Operations and Image Segmentation

Morphological image processing operations reveal the underlying structures and shapes within binary and grayscale images. While individual morphological operations perform simple functions, they can be combined to extract specific information from an image. Morphological operations often precede more advanced pattern recognition and image analysis operations such as segmentation. Shape recognition routines commonly include image thresholding or stretching to separate foreground and background image features.

Morphological operations apply a **structuring element** or morphological mask to an image. A structuring element that is applied to an image must be 2 dimensional, having the same number of dimensions as the array to which it is applied. A morphological operation passes the structuring element, of an empirically determined size and shape, over an image. The operation compares the structuring element to the underlying image and generates an output pixel based upon the function of the morphological operation. The size and shape of the structuring element determines what is extracted or deleted from an image. In general, smaller structuring elements preserve finer details within an image than larger elements.

Start by thresholding the Shergotty meteorite image in order to identify the dark mineral grains with a pixel value less than or equal to 20 :

```
1. IDL> minerals = image LE 20B
```

Next, create a structuring element array with a square shape that will help extract objects with sharp rectangular edges :

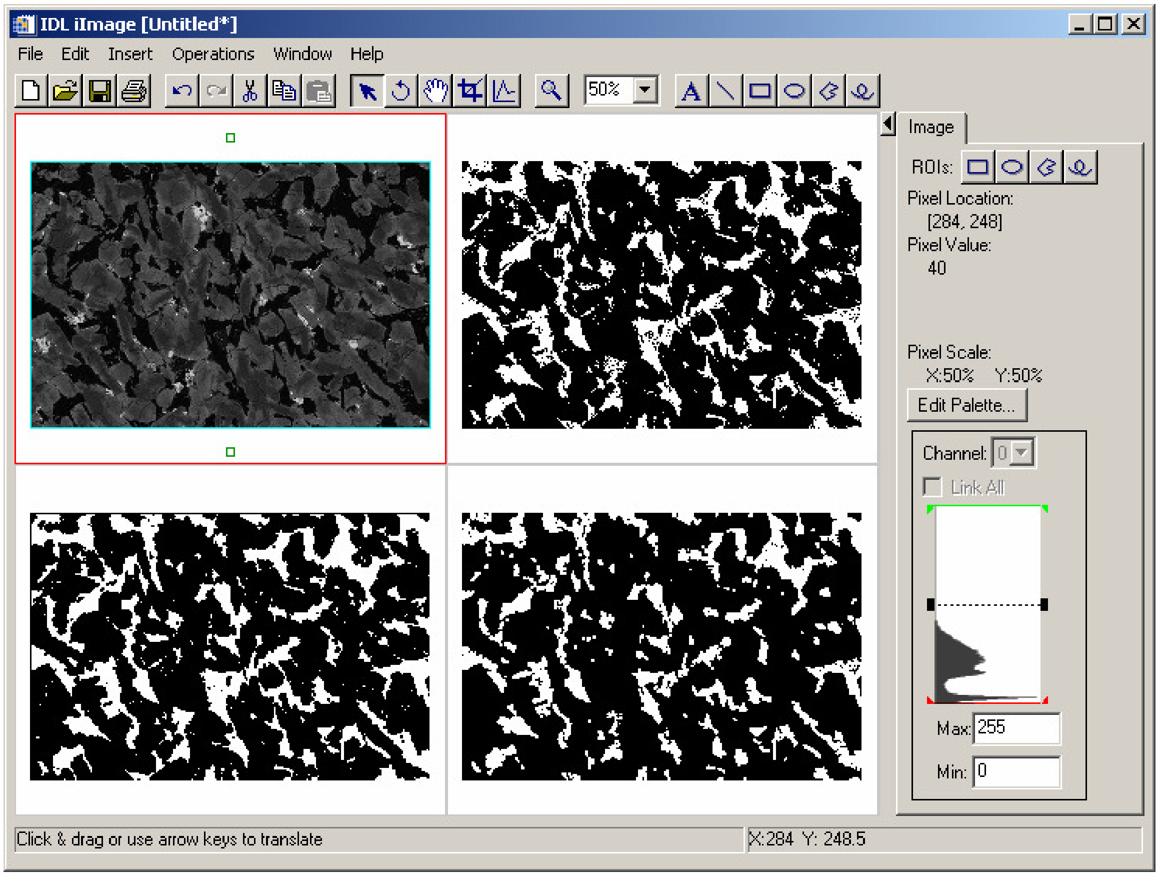
```
2. IDL> structElem = BYTARR (3,3) + 1B
3. IDL> PRINT, structElem
      1   1   1
      1   1   1
      1   1   1
```

The *MORPH_CLOSE* function can be used with this structuring element to apply the closing operator to the binary threshold image. The closing operator has the effect of clumping the threshold image, thereby filling in holes within and connecting gaps between neighboring regions. In addition, the *MORPH_OPEN* function can be subsequently used to apply the opening operator, which will have a sieving effect on the image that helps to remove small isolated regions. Apply these morphological operations and visualize the results in comparison to the original image :

```
4. IDL> clumped = MORPH_CLOSE (minerals, structElem)
5. IDL> sieved = MORPH_OPEN (minerals, structElem)
6. IDL> iImage, image, VIEW_GR=[2,2]
7. IDL> iImage, BYTSCL (minerals), /VIEW_NE
8. IDL> iImage, BYTSCL (clumped), /VIEW_NE
9. IDL> iImage, BYTSCL (sieved), /VIEW_NE
```

10. Click on each individual window pane and change the canvas zoom droplist to "*50%*".

The resulting *IDL iImage* visualization window should look similar to Fig. 7-18.



Pixel Location:
[284, 248]
Pixel Value:
40
Pixel Scale:
X:50% Y:50%
Max: 255
Min: 0

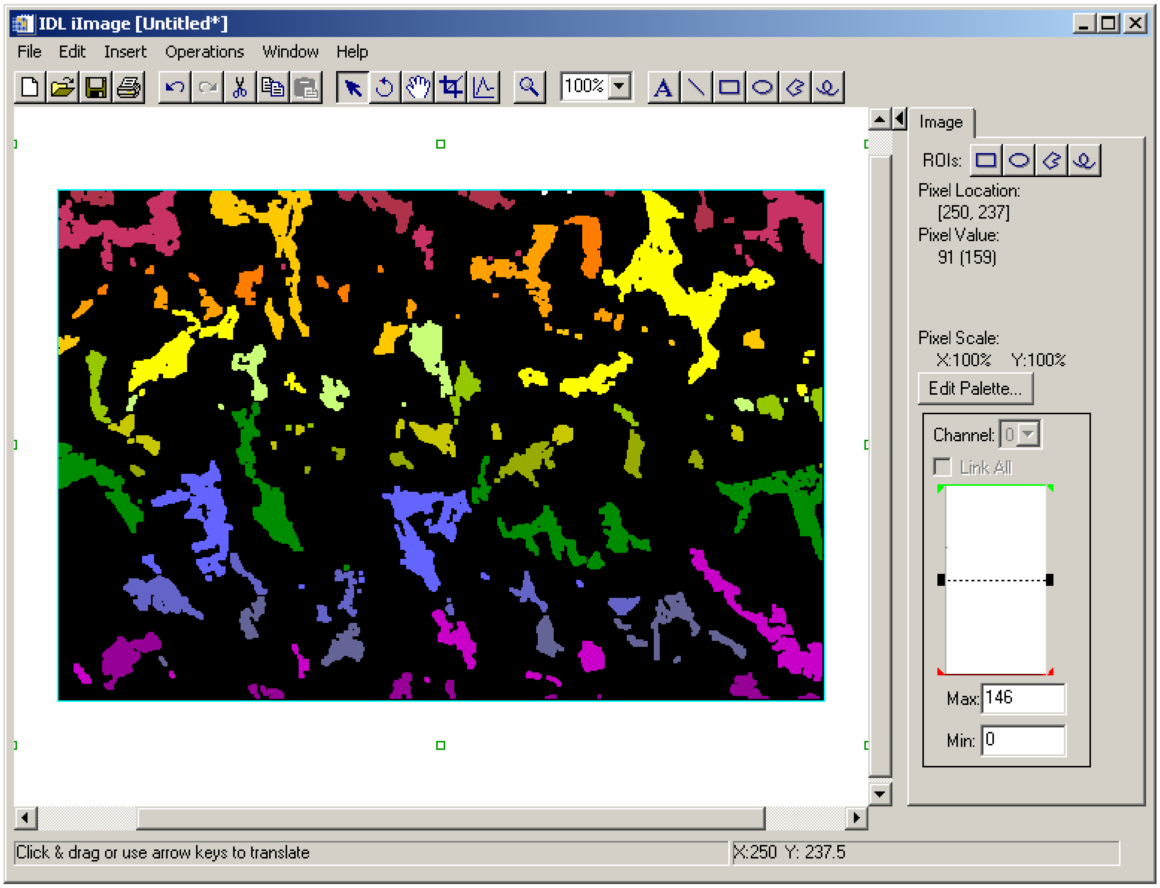
Click & drag or use arrow keys to translate
X:284 Y: 248.5

***Figure 7-18: Display of original image (upper left), binary threshold image of dark mineral grains (upper right), application of a clumping operation (lower left), followed by a sieving operation (lower right)***

Finally, the *LABEL_REGION* function can be used to perform image segmentation, which will consecutively label all of the regions, or *blobs*, of the clumped and sieved binary image with a unique region index. The resulting segmentation image can be displayed with a color table in order to visualize the separate distinct mineral grains within the meteorite image :

11. IDL> `segmented = LABEL_REGION (sieved)`
12. IDL> `iImage, segmented`
13. Within the Image panel on the right hand side of the *IDL iImage* window, press the "*Edit Palette…*" button.
14. Within the *Palette Editor* dialog, click on the "*Load Predefined…*" droplist and select "*Rainbow18*" from the dropdown menu.
15. Press the "*OK*" button to dismiss the *Palette Editor* dialog.

The resulting *IDL iImage* visualization window should look similar to Fig. 7-19.



Pixel Location:
[250, 237]
Pixel Value:
91 (159)
Pixel Scale:
X:100% Y:100%
Channel: 0
Max: 146
Min: 0
Click & drag or use arrow keys to translate
X:250 Y: 237.5

***Figure 7-19: Image segmentation of the separate dark mineral grains***

26. Once finished viewing the segmentation image, close the *IDL iImage* window.

## Processing Images in Alternate Domains

So far all of the processing and visualization of image data has been performed in the **spatial domain**. This means that the digital image is represented by pixel values that have a particular spatial location (i.e. column and row). However, a pixel's value and location can also be represented in other domains. Transforming an image into an alternate domain can provide a basis for performing image filters, noise removal, sharpening, or feature extraction. In addition, domain transformations also provide additional information about an image and can enable robust image compression techniques.

In the frequency or **Fourier domain**, the value and location are represented by sinusoidal relationships that depend upon the frequency of a pixel occurring within an image. In this domain, pixel location is represented by its X and Y frequencies and its value is represented by an amplitude. Images can be transformed into the frequency domain to determine which pixels contain the most important information and whether repeating patterns occur.

In addition to the Fourier domain, IDL also has the ability to transform images into the wavelet (time-frequency), Hough, and Radon domains.  In the **wavelet domain** the value and location of pixels are represented by sinusoidal relationships that only partially transform the image into the frequency domain.  The wavelet transformation process is the basis for many image compression algorithms.  The image information within the **Hough domain** shows the pixels of the original (spatial) image as sinusoidal curves.  If the points of the original image form a straight line, their related sinusoidal curves in the Hough domain will intersect. Masks can be easily applied to the image within the Hough domain to determine if and where straight lines occur.  The image information within the **Radon domain** shows a line through the original image as a point.  Specific features and geometries within the original image will produce peaks within the Radon domain and can be easily identified.

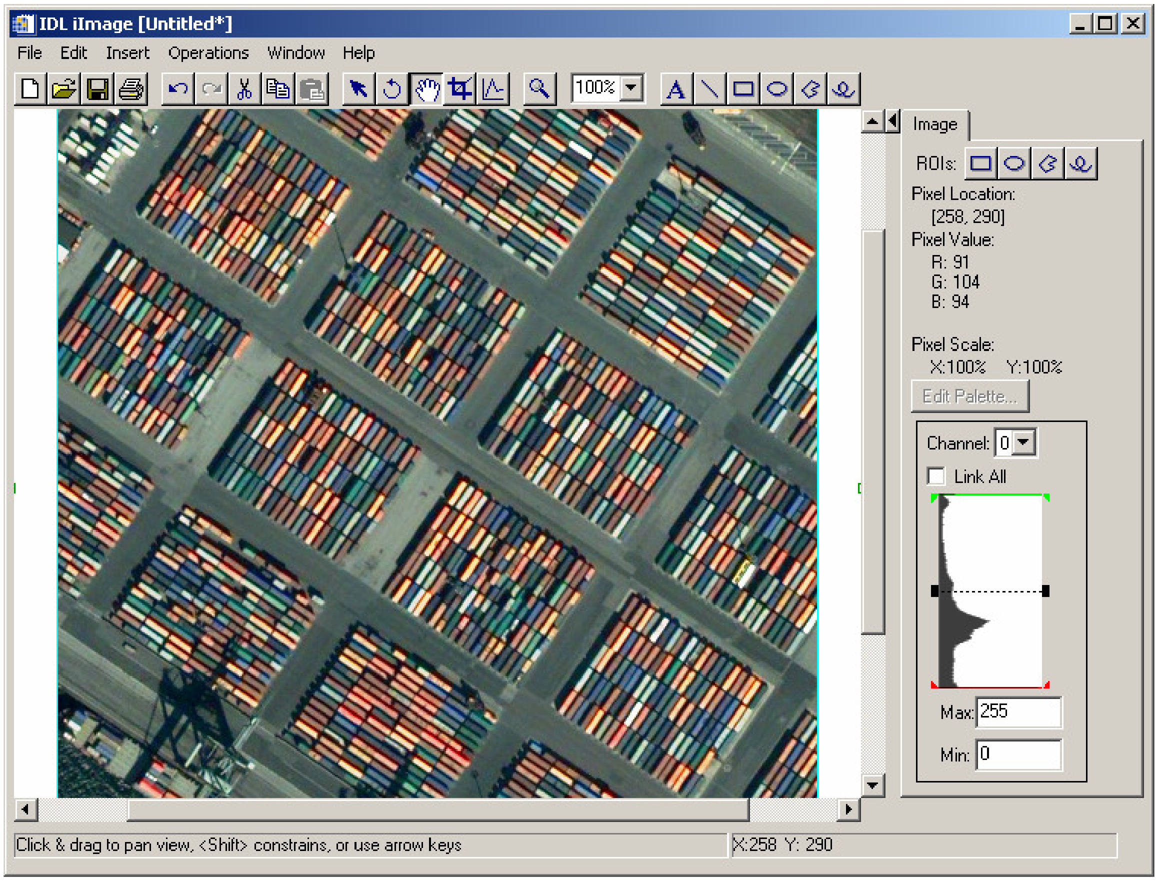
In IDL, the *FFT* routine can be utilized to perform a **Fast Fourier Transformation** and convert an image from the spatial domain into the frequency domain.  In the following exercise, the image data from the example data file "*hamburg.jp2*" will be input into the *iImage* utility and subsequently transformed into the Fourier domain. This example data file is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\hamburg.jp2*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/hamburg.jp2*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/hamburg.jp2*

The file "*hamburg.jp2*" is in JPEG2000 format and contains a satellite image of the loading docks at the port in Hamburg, Germany.  Start by loading this image into a new *iImage* utility :

1. IDL> `iImage`
2. From the *iImage* menu system select "*File > Open…*".
3. Select the "*hamburg.jp2*" file and hit "*Open*".

The resulting image display should look similar to Fig. 7-20.  Notice the linear and rectangular patterns that are prevalent in this image in both diagonal directions.



Pixel Location:
[258, 290]
Pixel Value:
R: 91
G: 104
B: 94

Pixel Scale:
X:100% Y:100%

Channel: 0

Max: 255

Min: 0

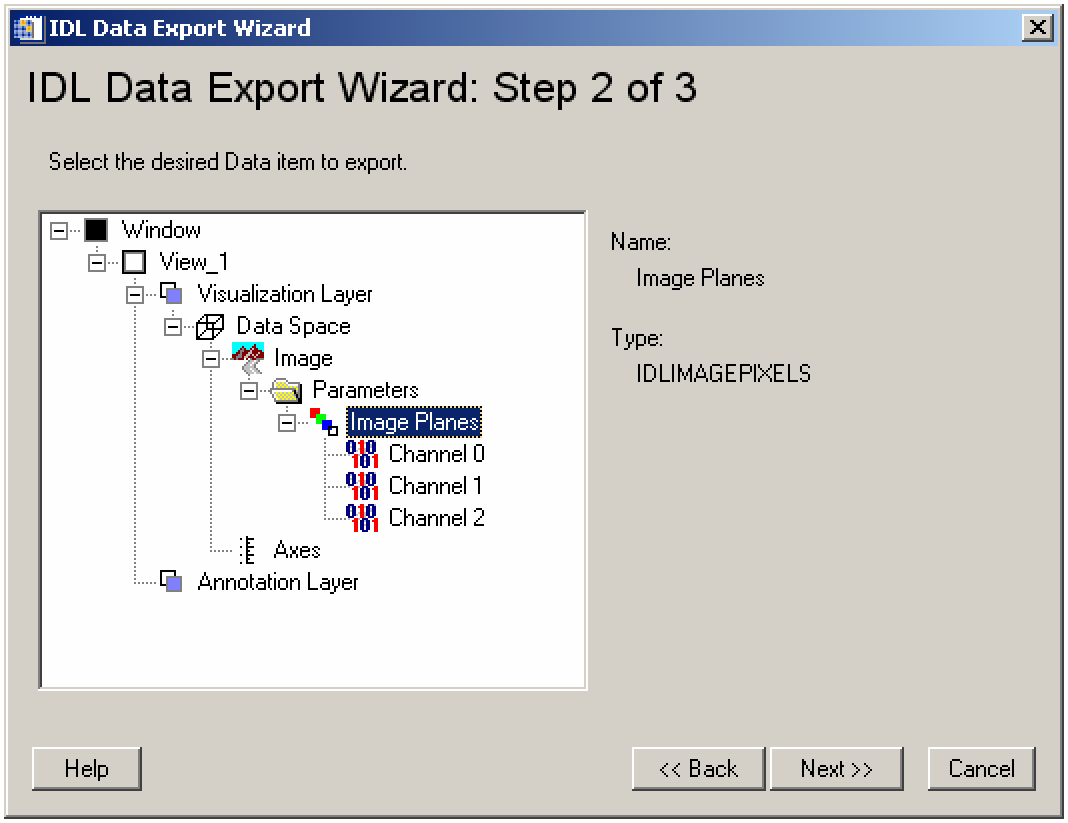
Click & drag to pan view, <Shift> constrains, or use arrow keys

X:258 Y: 290

***Figure 7-20: Image of the port in Hamburg, Germany***

Once the image has been loaded into the *iImage* utility, it can be exported to an IDL variable for processing at the IDL> command prompt. Use the following steps to create a variable for this image at the main IDL level :

4. Select "*File > Export…*" from the *iImage* menu.
5. In Step 1 of 3 of the *IDL Data Export Wizard* select "To an IDL Variable" and press the "*Next >>*" button.
6. In Step 2 of 3 of the *IDL Data Export Wizard* select the "Image Planes" parameter and press the "*Next >>*" button [Fig. 7-21].
7. In Step 3 of 3 of the *IDL Data Export Wizard* change the "IDL Variable Name:" field to "*hamburg*" and press the "*Finish*" button.
8. Once this is accomplished, close the *IDL iImage* window and return to the main IDL Development Environment.



IDL Data Export Wizard

IDL Data Export Wizard: Step 2 of 3

Select the desired Data item to export.

Name:
Image Planes

Type:
IDLIMAGEPIXELS

***Figure 7-21: Step 2 of 3 of the IDL Data Export Wizard***

A new variable named "hamburg" now exists at the main IDL level :

```
9. IDL> HELP, hamburg
   HAMBURG         BYTE      = Array[3, 500, 500]
```

In order to work with this image in the frequency domain it is beneficial to extract the individual color channel images.  This can be accomplished using IDL's standard array subscripting syntax in conjunction with the *REFORM* function, which is used to remove the first dimension (that has a size of one) and return a simple two-dimensional array :

```
10. IDL> r = REFORM (hamburg[0,*,*])
11. IDL> g = REFORM (hamburg[1,*,*])
12. IDL> b = REFORM (hamburg[2,*,*])
13. IDL> HELP, r, g, b
    R               BYTE      = Array[500, 500]
    G               BYTE      = Array[500, 500]
    B               BYTE      = Array[500, 500]
```

**Note:** Remember that the up-arrow and down-arrow keys on the keyboard can be used to perform command recall within IDL, which may be beneficial during these exercises.

Once this is accomplished, the *FFT* routine can be used to transform the image planes into the frequency domain :

```
14. IDL> rFFT = FFT (r)
15. IDL> gFFT = FFT (g)
16. IDL> bFFT = FFT (b)
```

The Fast Fourier Transform decomposes an image into sines and cosines of varying amplitudes and phases. The values of the resulting transform represent the amplitudes of particular horizontal and vertical frequencies. The data type of the array returned by the *FFT* function is complex, which contains real and imaginary parts :

```
17. IDL> HELP, gFFT
    GFFT            COMPLEX   = Array[500, 500]
```

The amplitude is the absolute value of the FFT, while the phase is the angle of the complex number, computed using the arctangent. In most cases, the imaginary part will look the same as the real part.

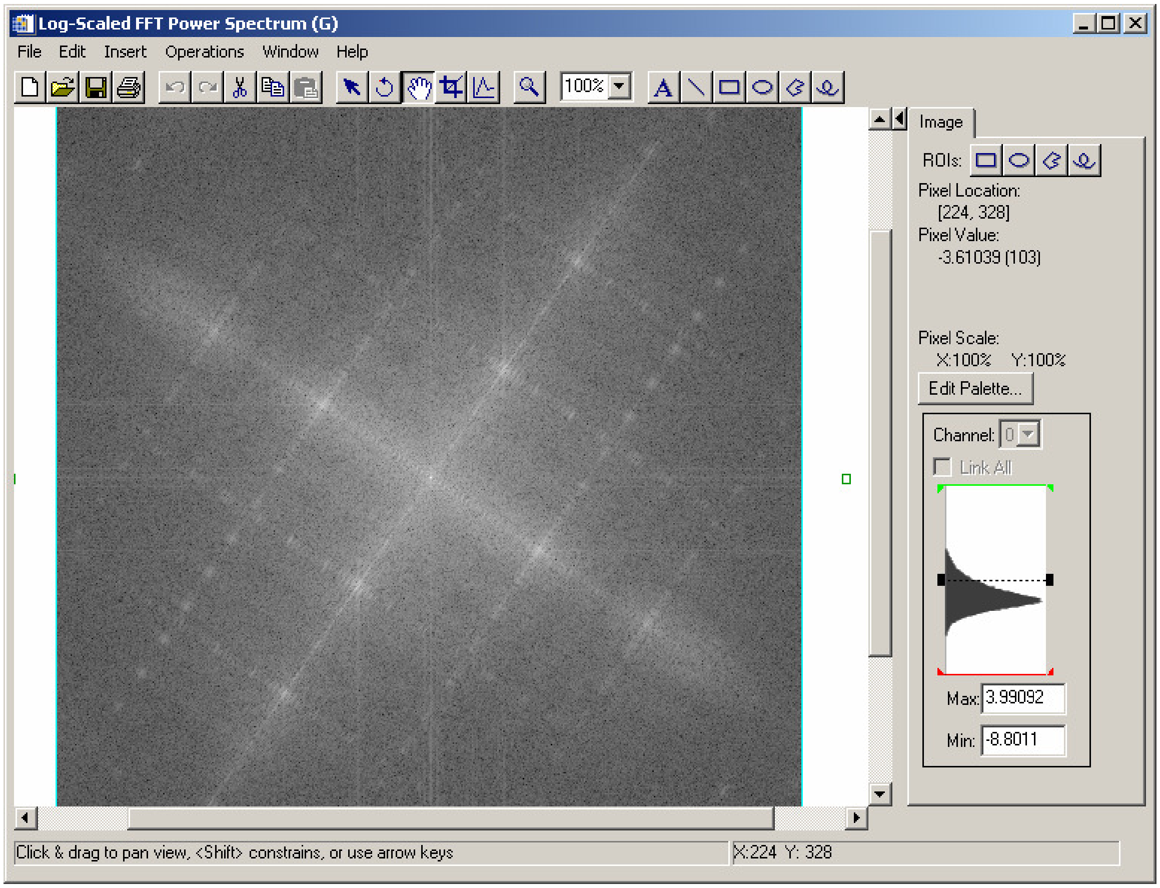
The image information in the frequency domain shows how often patterns are repeated within an image. Within the Fourier domain, low frequencies represent gradual variations in an image, while high frequencies correspond to abrupt variations in the image. The lowest frequencies usually contain most of the information, which is shown by the large peak in the center of the result. If the image does not contain any background noise, the rest of the data frequencies are very close to zero.

The results of the *FFT* function are often shifted to move the origin of the X and Y frequencies to the center of the display. Furthermore, the range of values from the peak to the high frequency noise is usually extreme. Consequently, a logarithmic scale is often utilized in order to visualize the image in the frequency domain. Since the logarithmic scale only applies to positive values, the **power spectrum** should be computed since it is the absolute value squared of the Fourier transform.

Visualize the power spectrum of the Fourier domain image for the green channel by executing the following statements :

```
18. IDL> center = 500 / 2 + 1
19. IDL> gShift = SHIFT (gFFT, center, center)
20. IDL> gPowSpec = ABS (gShift) ^ 2
21. IDL> gScaled = ALOG10 (gPowSpec)
22. IDL> iImage, gScaled, TITLE='Log-Scaled FFT Power Spectrum (G)'
```

The resulting *IDL iImage* visualization window should look similar to Fig. 7-22.



Pixel Location:
[224, 328]
Pixel Value:
-3.61039 (103)

Pixel Scale:
X:100% Y:100%

Channel: 0

Max: 3.99092

Min: -8.8011

Click & drag to pan view, <Shift> constrains, or use arrow keys

X:224 Y: 328

***Figure 7-22: Power spectrum for the green channel image in the frequency domain***

Notice the orientation of spatial patterns within the power spectrum image in both of the diagonal directions (just like the original image).

23. Once finished viewing the power spectrum image, close the *IDL iImage* window.

It may also be beneficial to visualize the power spectrum as a surface. Use the *REBIN* function to sub-sample the power spectrum in order to suppress some of the noise and set the shading for the surface to Gouraud :

24. IDL> `iSurface, REBIN (gScaled, 100, 100), SHADING=1`

The resulting *IDL iSurface* visualization window should look similar to Fig. 7-23.

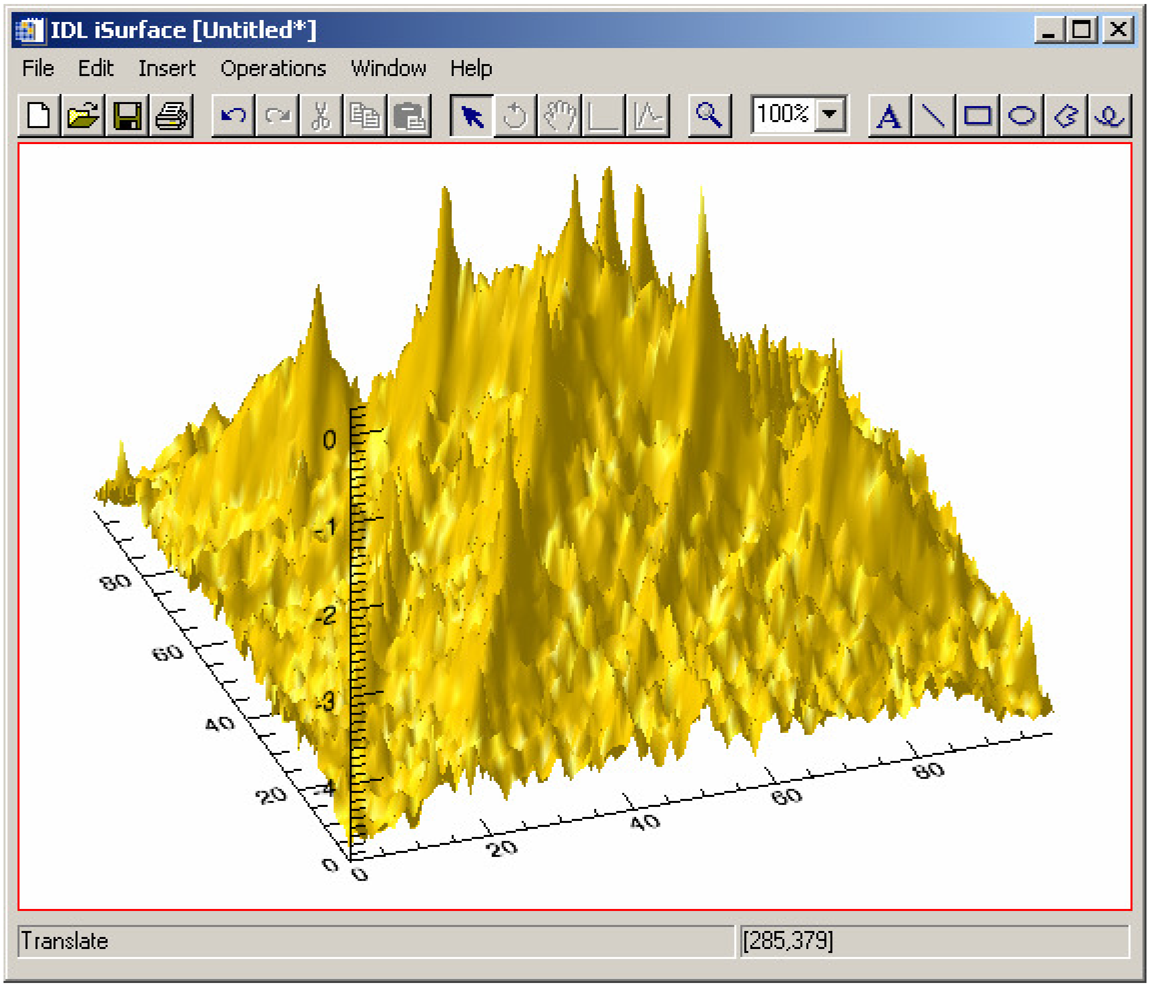


***Figure 7-23: Power spectrum displayed as a surface***

25. Once finished viewing the power spectrum surface, close the *IDL iSurface* window.

Low frequencies within the image tend to contain the most information because they determine the overall shape or patter in the image. High frequencies provide detail in the image, but they are often contaminated by the spurious effects of noise. Consequently, masks can be easily applied to an image within the frequency domain in order to remove noise.

Create a mask for the low spatial frequency components based on the highest values within the power spectrum for the green channel image :

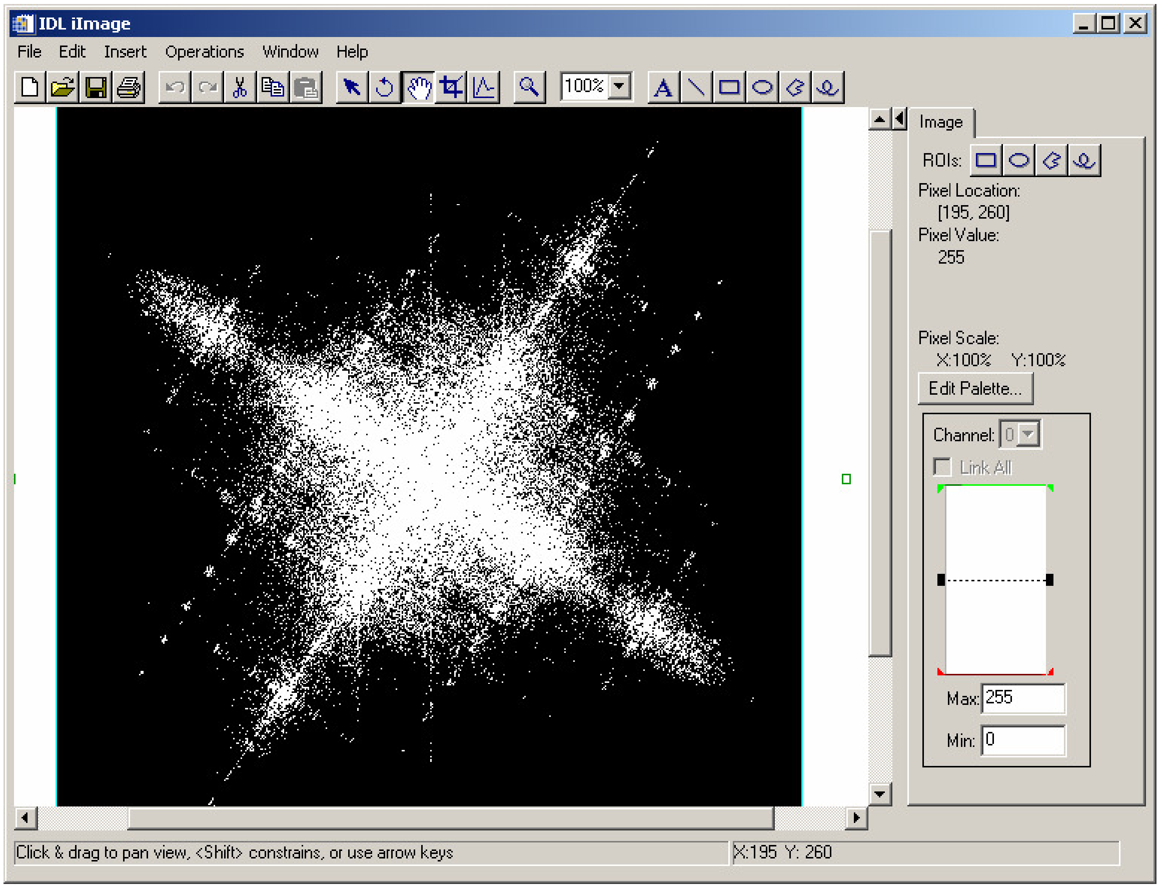
26. IDL> `lsfMask = REAL_PART (gScaled) GT -2.5`

**Note:** The threshold value of –2.5 was arbitrarily selected based on the surface visualization above.

Visualize this mask by loading it into the *iImage* utility :

27. IDL> `iImage, BYTSCL (lsfMask)`

The resulting *IDL iImage* visualization window should look similar to Fig. 7-24.



Pixel Location: [195, 260]
Pixel Value: 255
Pixel Scale: X:100% Y:100%
Channel: 0
Max: 255
Min: 0
Click & drag to pan view, <Shift> constrains, or use arrow keys
X:195 Y: 260

***Figure 7-24: Mask of the low spatial frequency components (white) within the power spectrum for the green color channel***

Notice that the low frequency components are found predominantly in the center of the power spectrum.

28. Once finished viewing the mask image, close the *IDL iImage* window.

In order to remove the high spatial frequency noise from the image, this mask must be applied to the Fourier transform data and then the inverse FFT must be computed. Applying the low spatial frequency mask allows these components to be converted back to the spatial domain during the inverse transform, while the high spatial frequency components are *masked out*.

First, the mask image must be shifted back to the original location of the Fourier transform :

29. IDL> `lsfMask = SHIFT (lsfMask, -center, -center)`

Once this is accomplished, the mask can be applied to the FFT results for the 3 color channels :

30. IDL> `rMasked = rFFT * lsfMask`

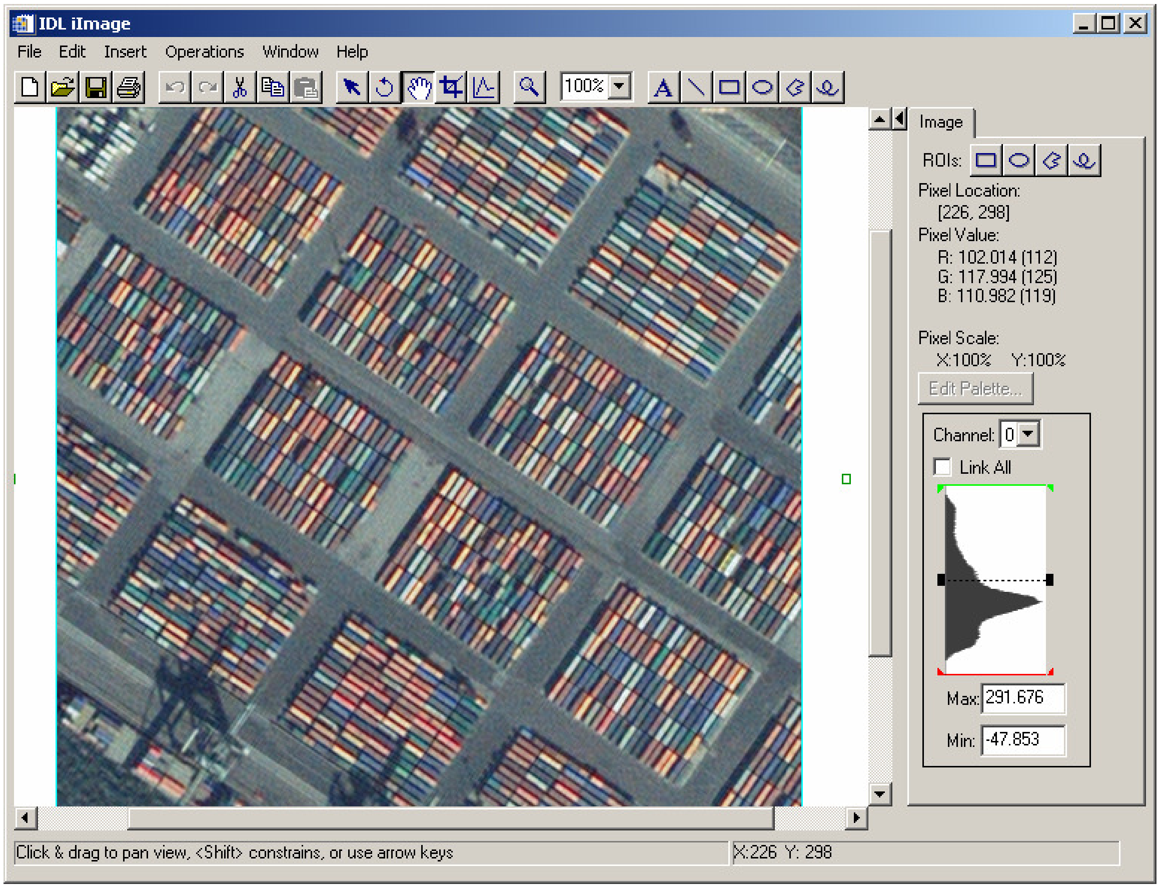
```
31. IDL> gMasked = gFFT * lsfMask
32. IDL> bMasked = bFFT * lsfMask
```

The inverse FFT can be used in conjunction with the *REAL_PART* function in order to convert the images back into the spatial domain :

```
33. IDL> rInvert = REAL_PART (FFT (rMasked, /INVERSE) )
34. IDL> gInvert = REAL_PART (FFT (gMasked, /INVERSE) )
35. IDL> bInvert = REAL_PART (FFT (bMasked, /INVERSE) )
```

The result can be visualized by loading the individual color channel images into the *iImage* utility [Fig. 7-25] :

```
36. IDL> iImage, RED=rInvert, GREEN=gInvert, BLUE=bInvert
```



Pixel Location:
[226, 298]
Pixel Value:
R: 102.014 (112)
G: 117.994 (125)
B: 110.982 (119)

Pixel Scale:
X:100% Y:100%

Channel: 0

Max: 291.676

Min: -47.853

Click & drag to pan view, <Shift> constrains, or use arrow keys
X:226 Y: 298

***Figure 7-25: Result of the inverse FFT after the high spatial frequency components have been masked out (low pass filter)***

37. Once finished viewing the inverse FFT image, close the *IDL iImage* window.

The high spatial frequency components of an image can also be enhanced using masking techniques in the frequency domain. A **circular-cut** (high pass) filter can be created by utilizing the DIST function in IDL and the appropriate threshold value :

```
38. IDL> hsfMask = DIST (500) GE 50
```

Visualize this mask by shifting it into the appropriate location and loading the result into the iImage utility [Fig. 7-26] :

```
39. IDL> iImage, BYTSCL (SHIFT (hsfMask, center, center) ), $
         BACKGROUND=[80,80,80]
```

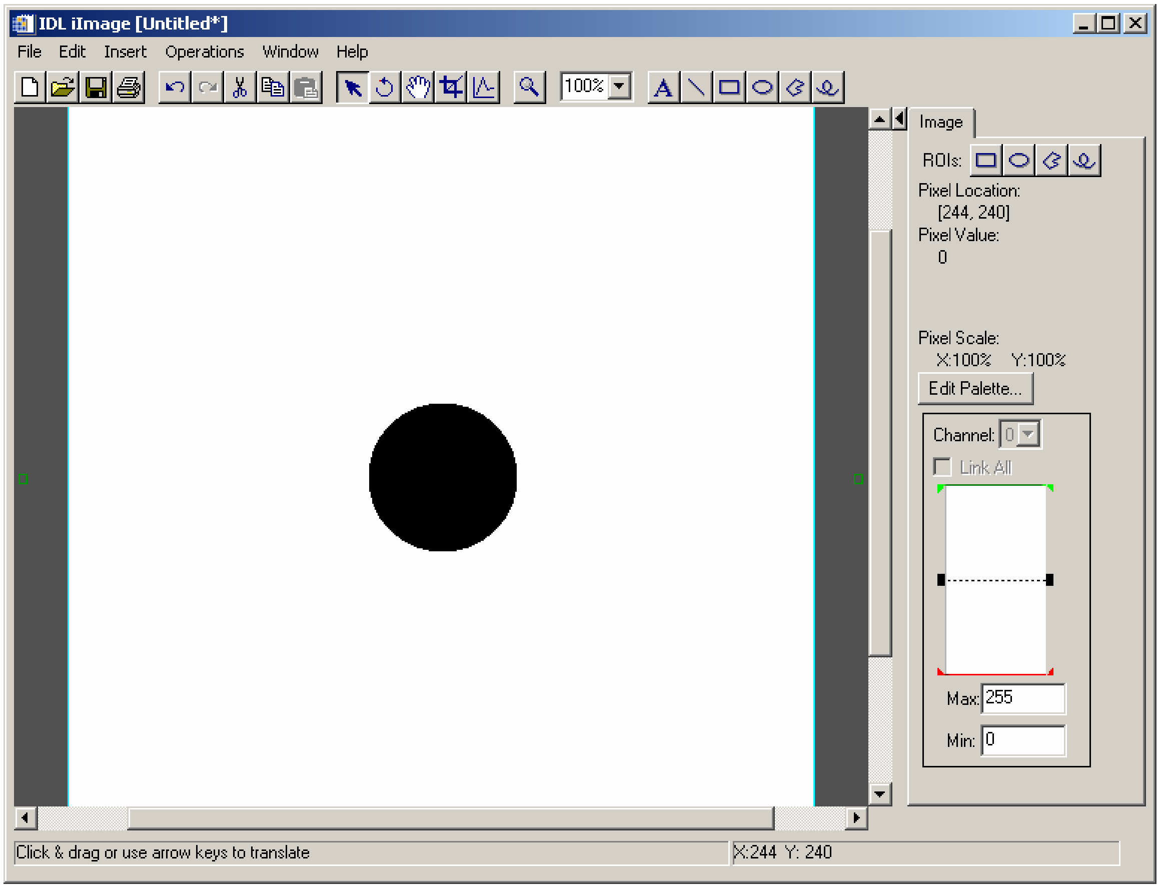


***Figure 7-26: Circular-Cut filter for the high spatial frequency components (white) within the image***

Notice that the high frequency components are found around the outer edges of the transform.

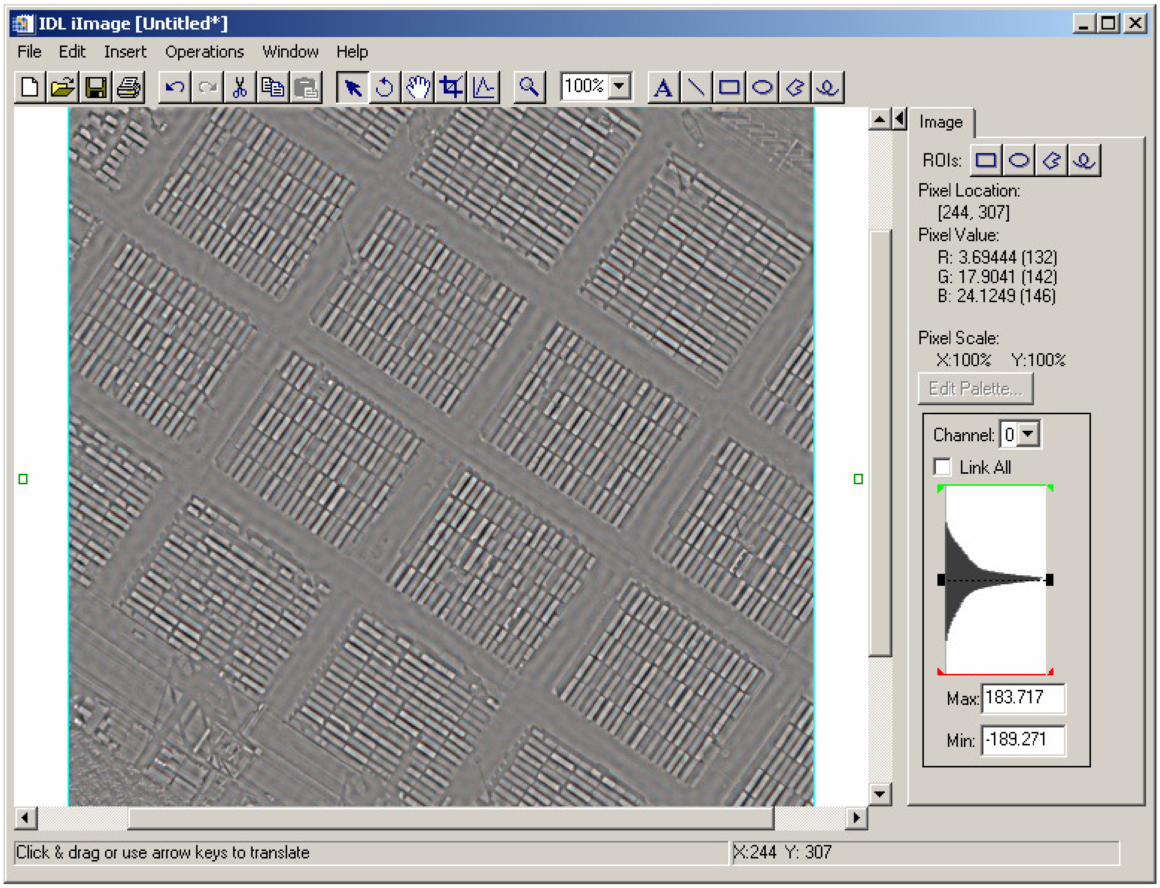
40. Once finished viewing the mask image, close the *IDL iImage* window.

Use the same methodology as before to apply the high pass filter, compute the inverse FFT, and display the result :

```
41. IDL> rMasked = rFFT * hsfMask
42. IDL> gMasked = gFFT * hsfMask
43. IDL> bMasked = bFFT * hsfMask
44. IDL> rInvert = REAL_PART (FFT (rMasked, /INVERSE) )
45. IDL> gInvert = REAL_PART (FFT (gMasked, /INVERSE) )
```

46. IDL> `bInvert = REAL_PART (FFT (bMasked, /INVERSE) )`
47. IDL> `iImage, RED=rInvert, GREEN=gInvert, BLUE=bInvert`

The resulting *IDL iImage* visualization window should look similar to Fig. 7-27.



100%

Pixel Location:
[244, 307]

Pixel Value:
R: 3.69444 (132)
G: 17.9041 (142)
B: 24.1249 (146)

Pixel Scale:
X:100% Y:100%

Channel: 0

□ Link All

Max: 183.717

Min: -189.271

Click & drag or use arrow keys to translate

X:244 Y: 307

***Figure 7-27: Result of the inverse FFT after the low spatial frequency components have been masked out (high pass filter)***

48. Once finished viewing the inverse FFT image, close the *IDL iImage* window.
49. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```

# Chapter 8:  Working with Maps

## Introduction to Mapping

In some cases the data that is being analyzed may be related to a location or area on the surface of the Earth, and it is usually beneficial to visualize this data within a **map projection**.  Map projections are attempts to portray the surface of the earth or a portion of the earth on a flat surface (in this case the computer monitor screen). Every flat map misrepresents the surface of the Earth in some way.  Consequently, no 2-D map can rival a 3-D globe in truly representing the surface of the entire Earth.  Some distortions of conformality, distance, direction, scale, and area always result from this process.  However, a map or parts of a map can show one or more (but never all) of the following :

- True Directions
- True Distances
- True Areas
- True Shapes

Some map projections minimize distortions in some of these properties at the expense of maximizing errors in others.  In addition, some projections are attempts to only moderately distort all of these properties.  A discussion on all of the various map projections and their advantages vs. disadvantages is beyond the scope of this manual.

IDL has several built-in routines that deal with creating, manipulating, and displaying data within map projections.  In addition, the installation of the IDL software package includes three databases that can be useful when mapping at a small scale (i.e. over a large regional area) :

- A low resolution database of continental outlines (located in the "*…/resource/maps/low/*" subfolder of the IDL installation).
- A high resolution database that was adapted from the 1993 CIA World Map database (located in the "*…/resource/maps/high/*" subfolder of the IDL installation).
- ESRI Data and Maps CD-ROM datasets in **Shapefile** format (located in the "*…/resource/maps/shape/*" subfolder of the IDL installation).

**Note:**  The "High Resolution Maps" is an optional item that is not installed by default.  If this item was not installed the user can go back to the software installer and modify the contents of the distribution in order to add this database.

The iTools system within the IDL software package contains a pre-built utility for working with data in a map projection called *iMap*.  The *iMap* utility displays image and contour data that are georeferenced to a particular map projection.  The *iMap* tool gives the user great flexibility in manipulating and visualizing these datasets. Images and contour data can be warped from geographic (latitude / longitude) coordinates to a specific map projection, or the data can be directly displayed onto the chosen projection.  In addition, the user can import ESRI Shapefile data and warp the data to the desired map projection.  The Shapefile format stores nontopological vector geometry and attribute information for the spatial features in an area.  The Shapefile format was created by ESRI, and is widely used in the geographic information systems (GIS) community.  Several predefined Shapefile datasets are provided with the IDL installation, including continents, countries, rivers, lakes, states & provinces, and cities.  The user can also insert a set of longitude and latitude gridlines.

Launch the *iMap* utility by executing the *iMap* procedure at the IDL> command prompt :

1. IDL> `iMap`

Start by inserting some of the vector data from the ESRI Shapefile database included with the IDL software installation :

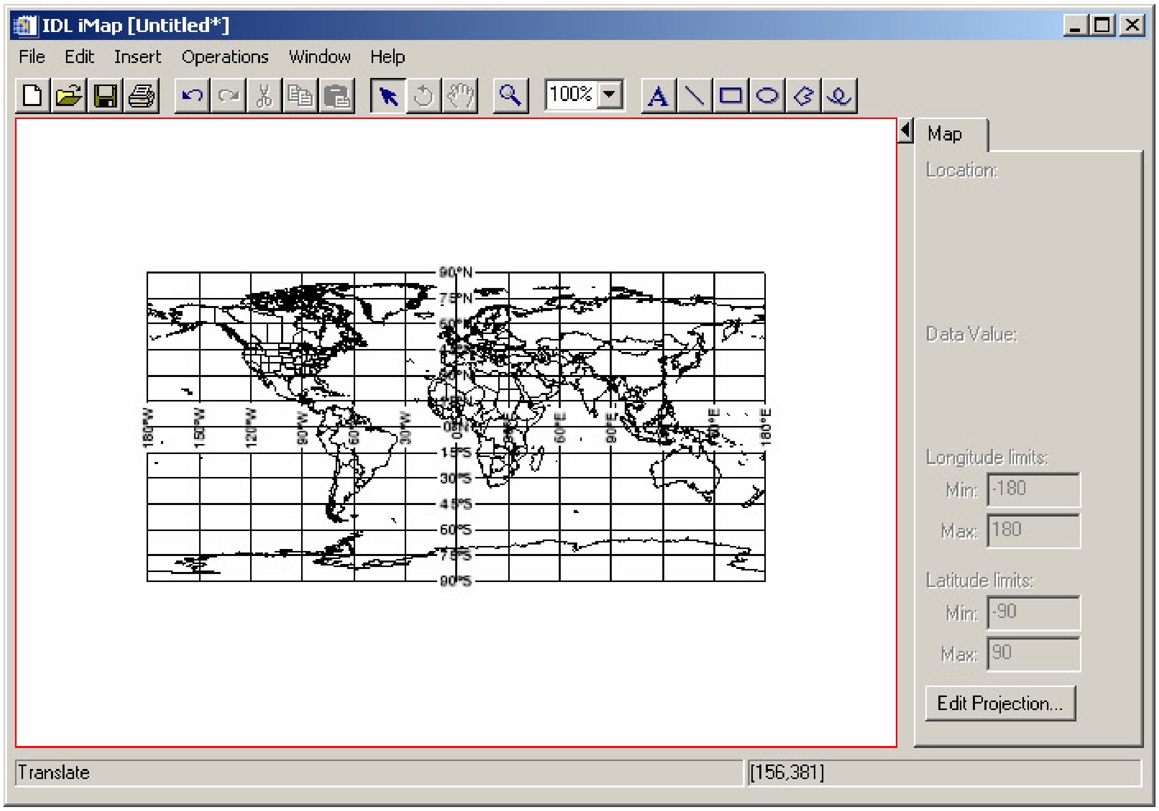
2. From the *IDL iMap* window menu system select "*Insert > Map > Continents*".

Since the iMap tool is currently in its initial state with no active map projection defined, the vector data is displayed in the standard geographic (lat/lon) coordinate system.

Next, insert the low-resolution countries, along with the United States and Canadian province boundaries, and the longitude-latitude grid :

3. Select "*Insert > Map > Countries (low res)*" from the *iMap* menu system.
4. Select "*Insert > Map > United States*" from the *iMap* menu system.
5. Select "*Insert > Map > Canadian Provinces*" from the *iMap* menu system.
6. Select "*Insert > Map > Grid*" from the *iMap* menu system.

The resulting *IDL iMap* visualization window should look similar to Fig. 8-1.



Longitude limits:
Min: -180
Max: 180

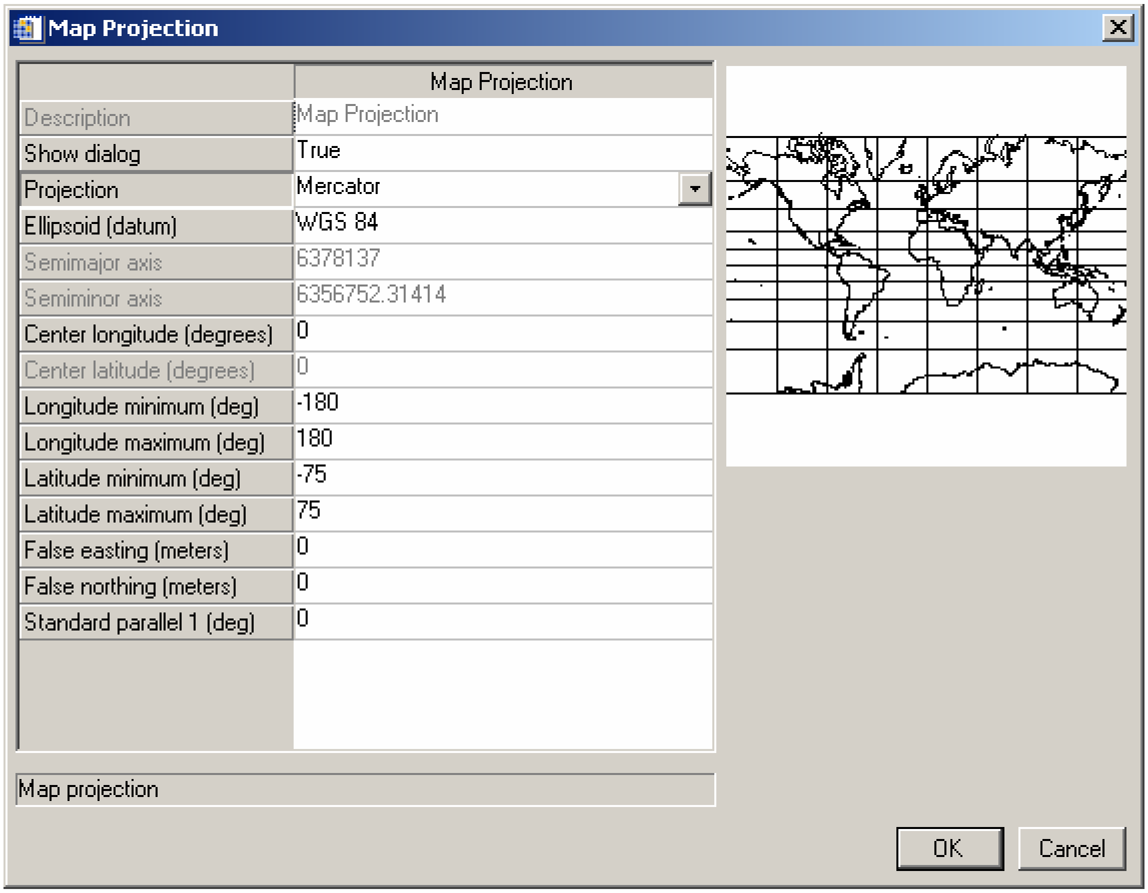
Latitude limits:
Min: -90
Max: 90

[156,381]

***Figure 8-1: Display of some of the ESRI Shapefile datasets included with the IDL installation***

Currently the data is displayed in the geographic lat/lon coordinate system and has not been warped into a specific map projection. The data can be viewed in any of the supported map projections by editing the current map projection within the *iMap* utility. Use the following steps to change the map projection to Mercator with the WGS-84 datum :

7. Press the "*Edit Projection…*" button on the *Map* panel in the bottom right-hand corner of the *IDL iMap* window. The *Map Projection* dialog will appear [Fig. 8-2].
8. Within the *Map Projection* dialog change the "Projection" droplist to "*Mercator*" [Fig. 8-2].
9. Change the "Ellipsoid (datum)" droplist to "*WGS 84*" [Fig. 8-2].
10. Press the "*OK*" button to apply the changes and dismiss the *Map Projection* dialog.



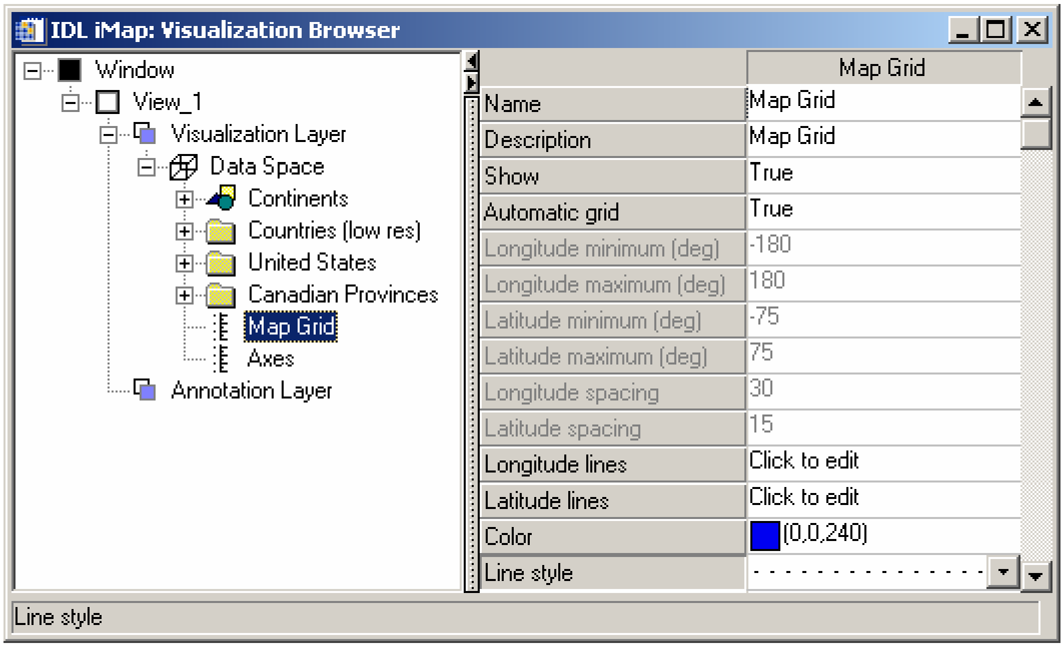
| | Map Projection |
|---|---|
| Description | Map Projection |
| Show dialog | True |
| Projection | Mercator |
| Ellipsoid (datum) | WGS 84 |
| Semimajor axis | 6378137 |
| Semiminor axis | 6356752.31414 |
| Center longitude (degrees) | 0 |
| Center latitude (degrees) | 0 |
| Longitude minimum (deg) | -180 |
| Longitude maximum (deg) | 180 |
| Latitude minimum (deg) | -75 |
| Latitude maximum (deg) | 75 |
| False easting (meters) | 0 |
| False northing (meters) | 0 |
| Standard parallel 1 (deg) | 0 |

Map projection

***Figure 8-2: The map projection selection dialog***

The cursor will change to an hourglass while the transformation is processing. Once the conversion is complete, it may be beneficial to change some of the properties associated with the map objects so they are easier to identify within the visualization.

11. From the *IDL iMap* window menu system, select "*Window > Visualization Browser…*". A dialog entitled "*IDL iMap: Visualization Browser*" will appear [Fig. 8-3].
12. Click on the "*Map Grid*" object so it is selected within the left visualization panel [Fig. 8-3].
13. Click on the ▶ show right panel button to display the property sheets for the selected object [Fig. 8-3].
14. Change the "Line style" property to dashed and the "Color" property to blue for the "*Map Grid*" object [Fig. 8-3].
15. Next, select the "*Canadian Provinces*" object, change the "Color" to green, set "Fill background" to *True*, change the "Fill color" to orange, and set the "Transparency" to "*0*".
16. Select the "*United States*" object, change the "Color" to red, set "Fill background" to *True*, change the "Fill color" to yellow, and set the "Transparency" to "*0*".
17. Close the *Visualization Browser* window.

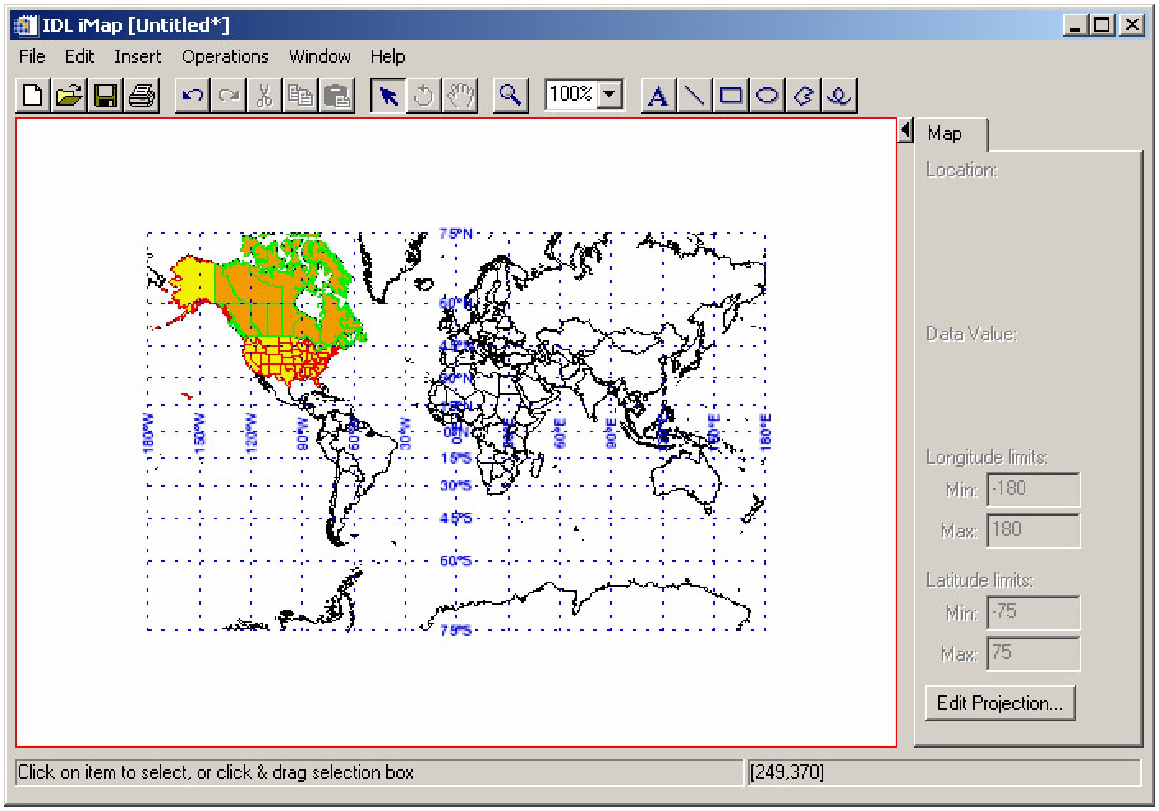


| | Map Grid |
|---|---|
| Name | Map Grid |
| Description | Map Grid |
| Show | True |
| Automatic grid | True |
| Longitude minimum (deg) | -180 |
| Longitude maximum (deg) | 180 |
| Latitude minimum (deg) | -75 |
| Latitude maximum (deg) | 75 |
| Longitude spacing | 30 |
| Latitude spacing | 15 |
| Longitude lines | Click to edit |
| Latitude lines | Click to edit |
| Color | 🟦 (0,0,240) |
| Line style | - - - - - - - - - - - - - - |

Line style

**Figure 8-3: The iMap visualization browser window**

The resulting *IDL iMap* visualization window should look similar to Fig. 8-4.



Location:

Data Value:

Longitude limits:

Min: -180

Max: 180

Latitude limits:

Min: -75

Max: 75

Click on item to select, or click & drag selection box

[249,370]

**Figure 8-4: Display of the data in the Mercator map projection**

18. Once finished viewing the map visualization, close the *IDL iMap* window.

## Displaying an Image within a Map Projection

A map projection establishes the axis type and coordinate conversion mechanism for mapping points on the Earth's surface, expressed in latitude and longitude, to points on a plane according to one of several possible projections. The user can apply a map projection either before or after an image is imported into the *iMap* utility. In the following exercise, the image from the example data file "*colorado.jpg*" will be input into the *iMap* utility and subsequently displayed in the UTM map projection. This example data file is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\colorado.jpg*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/colorado.jpg*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/colorado.jpg*

The file is in JPEG format and contains a hill shade image of a digital elevation model for a portion of the state of Colorado. Start by inputting this image into a new *iMap* utility window :

1. IDL> `iMap`
2. Select "*File > Open…*" from the *iMap* menu system.
3. Select the file "*colorado.jpg*" and press "*Open*".

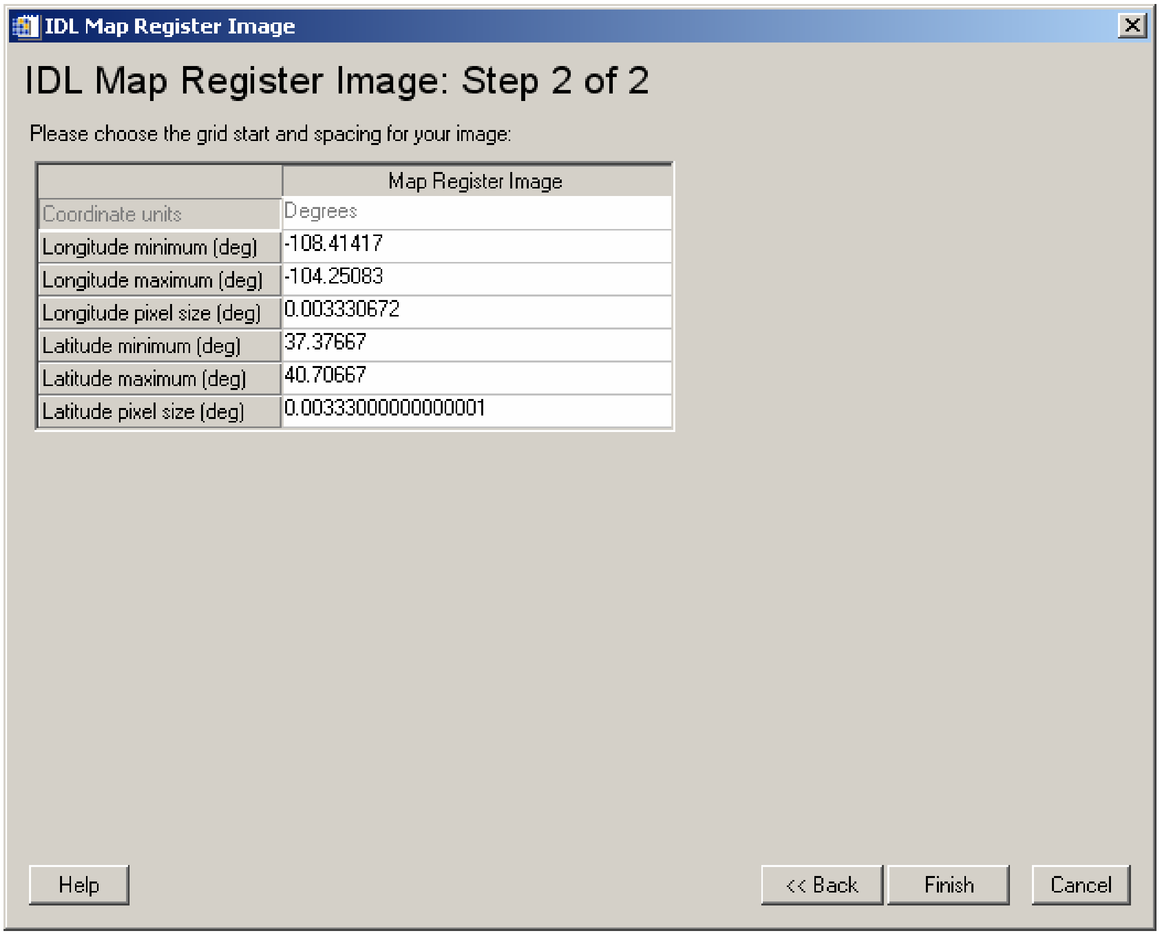
The *IDL Map Register Image* dialog will be displayed, which walks the user through the steps of specifying the geolocation for the image dataset.

4. In Step 1 of 2, select "Degrees longitude/latitude (geographic coordinates)" and press the "*Next >>*" button.
5. In Step 2 of 2, enter the following geographic coordinates [Fig. 8-5] :

```
Longitude minimum (deg) :  -108.41417
Longitude maximum (deg) :  -104.25083
Latitude minimum (deg)  :    37.37667
Latitude maximum (deg)  :    40.70667
```

Once the minimum and maximum values have been entered the appropriate pixel size is automatically computed based on the size of the image (approximately 0.00333 degrees).

6. Press the "*Finish*" button to complete the geolocation definition and return to the *IDL iMap* window.



IDL Map Register Image

IDL Map Register Image: Step 2 of 2

Please choose the grid start and spacing for your image:

| | Map Register Image |
|---|---|
| Coordinate units | Degrees |
| Longitude minimum (deg) | -108.41417 |
| Longitude maximum (deg) | -104.25083 |
| Longitude pixel size (deg) | 0.003330672 |
| Latitude minimum (deg) | 37.37667 |
| Latitude maximum (deg) | 40.70667 |
| Latitude pixel size (deg) | 0.00333000000000001 |

***Figure 8-5: Step 2 of 2 for the IDL Map Register Image wizard***

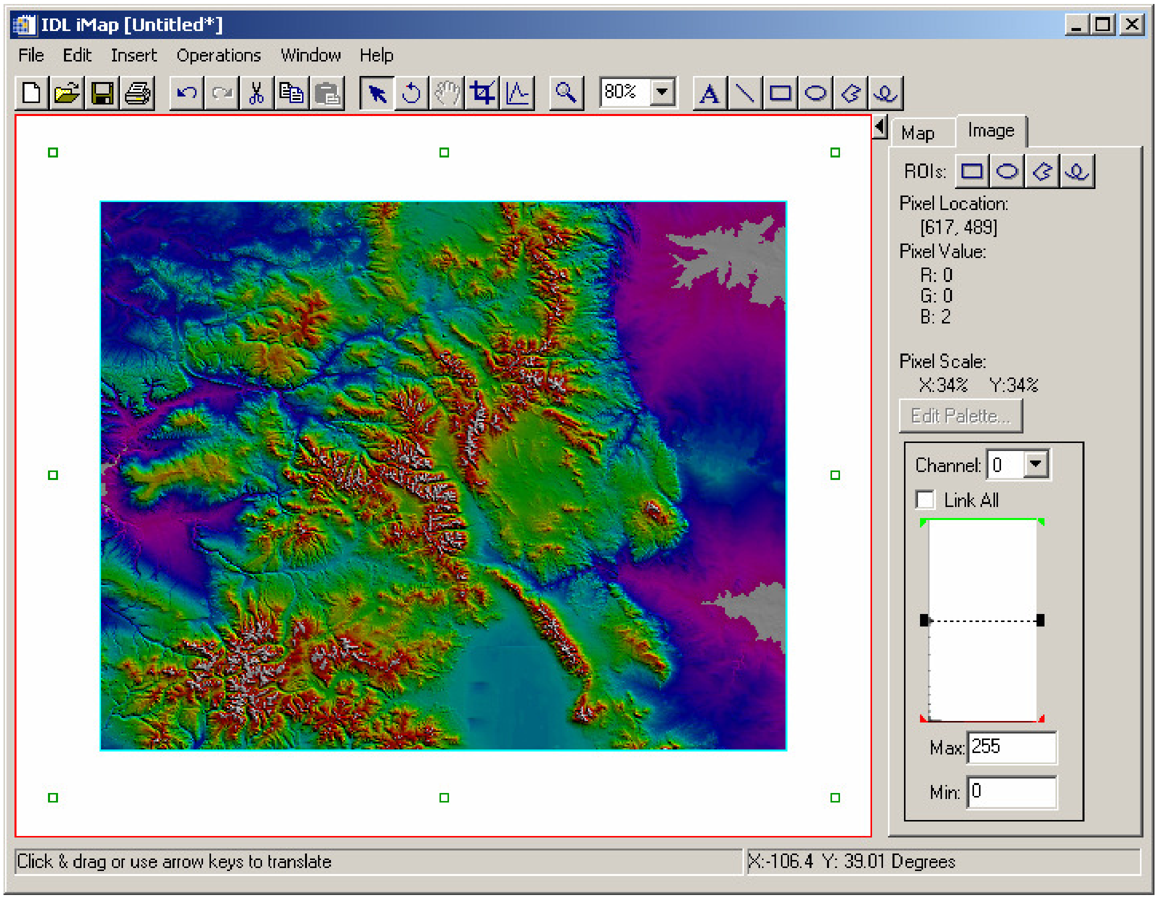
The standard *Image* panel will automatically be placed along the right-hand side within the current tool window since the data being displayed is an image [Fig. 8-6]. This is the same *Image* panel that is present within the *iImage* utility.

7. Maximize the *IDL iMap* window and scale the image so it fits within the current view space.

The "Pixel Location:" and "Pixel Value:" fields within the *Image* panel will show the mouse cursor location within the image and the pixel values in each of the 3 color channels, while the geographic location is displayed in the lower-right hand corner of the *IDL iMap* window [Fig. 8-6].

8. Using the mouse, click on the *Map* panel tab to switch back to the original tool configuration.

Notice that the longitude and latitude locations are also reported in the *Map* panel as the mouse cursor is moved over the image display.



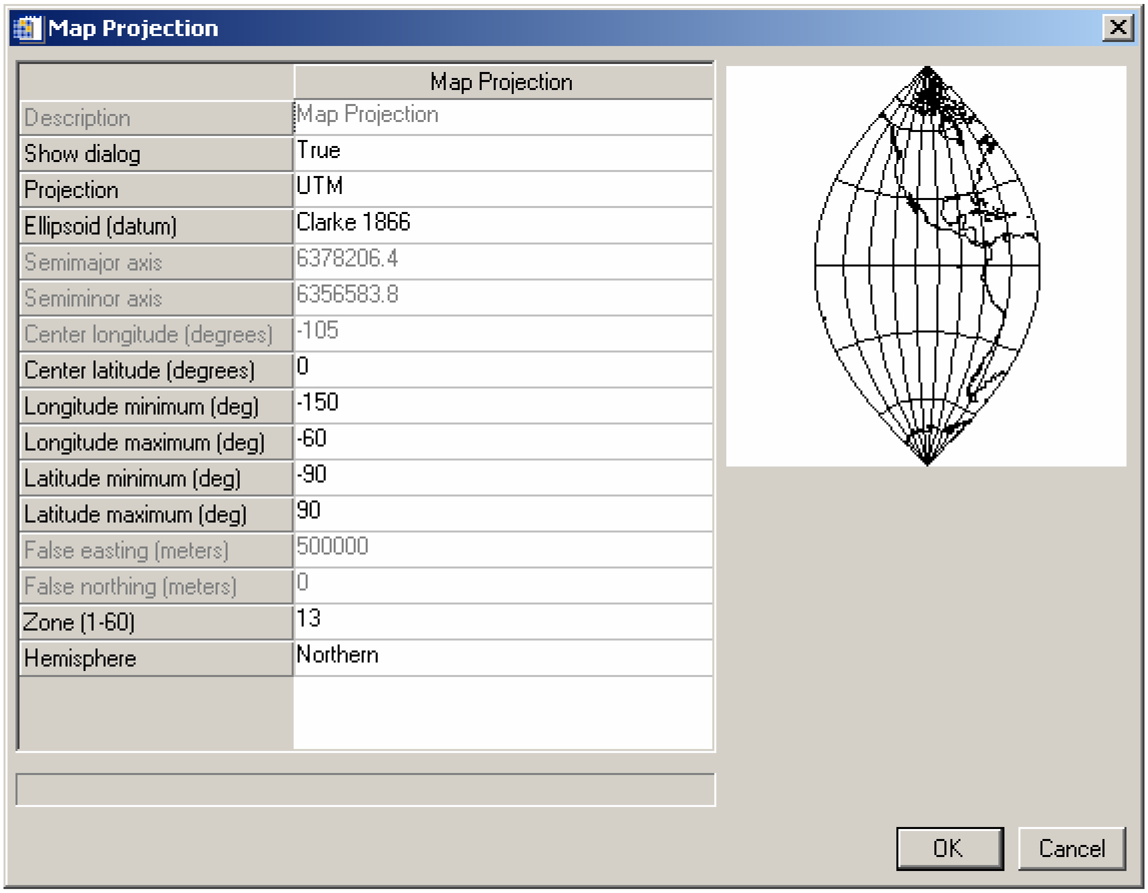
Pixel Location: [617, 489]
Pixel Value: R: 0 G: 0 B: 2
Pixel Scale: X:34% Y:34%
Channel: 0
Max: 255
Min: 0
80%
Click & drag or use arrow keys to translate
X:-106.4 Y: 39.01 Degrees

***Figure 8-6: Display of the image in the geographic coordinate system***

Now that the image has been imported into the iMap utility and the geolocation has been defined, a new map projection for the tool can be defined. In this case, it will be beneficial to visualize the current image within the UTM map projection. The area within the state of Colorado that this image covers is located within UTM zone #13 (North). Use the following steps to change the current map projection to UTM :

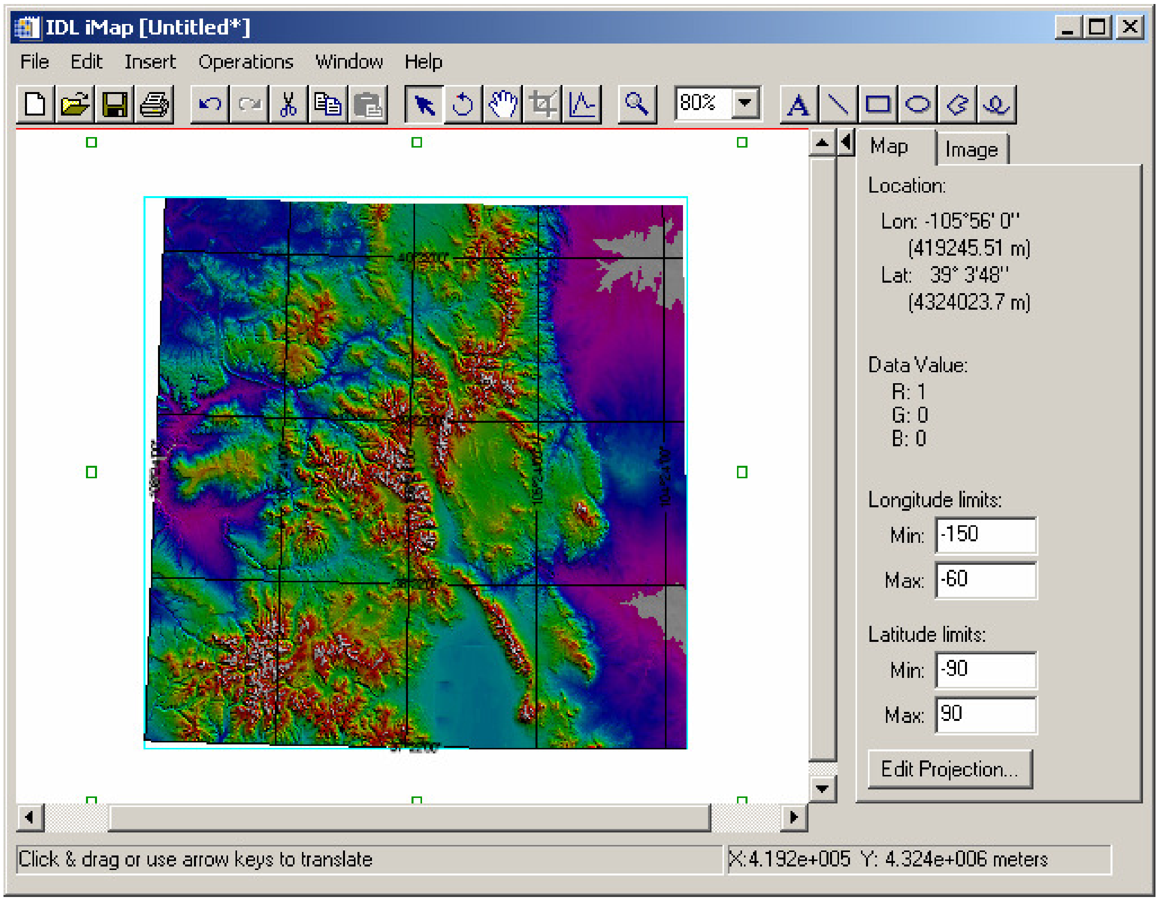
9. Within the *Map* panel, press the "*Edit Projection…*" button.
10. Within the *Map Projection* dialog, change the "Projection" droplist to "*UTM*" [Fig. 8-7].
11. Change the "Zone (1-60)" field to "*13*" [Fig. 8-7].
12. Press the "*OK*" button to apply the map projection and dismiss the *Map Projection* dialog.

The mouse cursor will change to a hourglass while the map projection conversion of the image data is processing. Once the conversion is complete the map projection for the *iMap* tool will now be UTM. The resulting *IDL iMap* visualization window should look similar to Fig. 8-8.



| | Map Projection |
|---|---|
| Description | Map Projection |
| Show dialog | True |
| Projection | UTM |
| Ellipsoid (datum) | Clarke 1866 |
| Semimajor axis | 6378206.4 |
| Semiminor axis | 6356583.8 |
| Center longitude (degrees) | -105 |
| Center latitude (degrees) | 0 |
| Longitude minimum (deg) | -150 |
| Longitude maximum (deg) | -60 |
| Latitude minimum (deg) | -90 |
| Latitude maximum (deg) | 90 |
| False easting (meters) | 500000 |
| False northing (meters) | 0 |
| Zone (1-60) | 13 |
| Hemisphere | Northern |

***Figure 8-7: Defining the UTM map projection for the iMap utility***



Location:

Lon: -105°56' 0"
(419245.51 m)
Lat: 39° 3'48"
(4324023.7 m)

Data Value:
R: 1
G: 0
B: 0

Longitude limits:
Min: -150
Max: -60

Latitude limits:
Min: -90
Max: 90

Click & drag or use arrow keys to translate

X:4.192e+005 Y: 4.324e+006 meters

***Figure 8-8: Display of the image in the UTM map projection***

Notice that the image has been warped to conform to the specifications of the UTM map projection. Also notice that the geographic location that is reported while the mouse cursor is moved over the image is now includes Easting and Northing in units of meters.

13. Once finished viewing the map visualization, close the *IDL iMap* window.

## Mapping Data Programmatically

In addition to the *iMap* utility, IDL also has several routines in its library that can be used to programmatically warp data to and from specific map projections. Once the data itself has been warped into a map projection it can be subsequently displayed in any of the standard iTools such as *iImage* or *iContour*. The four primary routines that can be used to work with maps within the IDL language are :

- *MAP_PROJ_INIT* : Initializes a mapping projection using either IDL's own map projections or the map projections from the U.S. Geological Survey's General Cartographic Transformation Package (GCTP).

- *MAP_PROJ_FORWARD* : Transforms map coordinates from longitude and latitude to Cartesian (x, y) coordinates using a supplied map projection structure.
- *MAP_PROJ_INVERSE* : Transforms map coordinates from Cartesian (x, y) coordinates to longitude and latitude using a supplied map projection variable.
- *MAP_PROJ_IMAGE* : Warps an image (or other 2-dimensional dataset) from geographic coordinates (longitude and latitude) to a specified map projection.

In the following exercise, the DEM data from the example data file "*New_Zealand_DEM.tif*" will be input into IDL and displayed using *iContour*. This example data file is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:**
  *C:\RSI\IDL##\IDL_QS_Files\data\New_Zealand_DEM.tif*
- **UNIX et al.:**
  */usr/local/rsi/idl_#.#/IDL_QS_Files/data/New_Zealand_DEM.tif*
- **Mac OS X:**
  */Applications/rsi/idl_#.#/IDL_QS_Files/data/New_Zealand_DEM.tif*

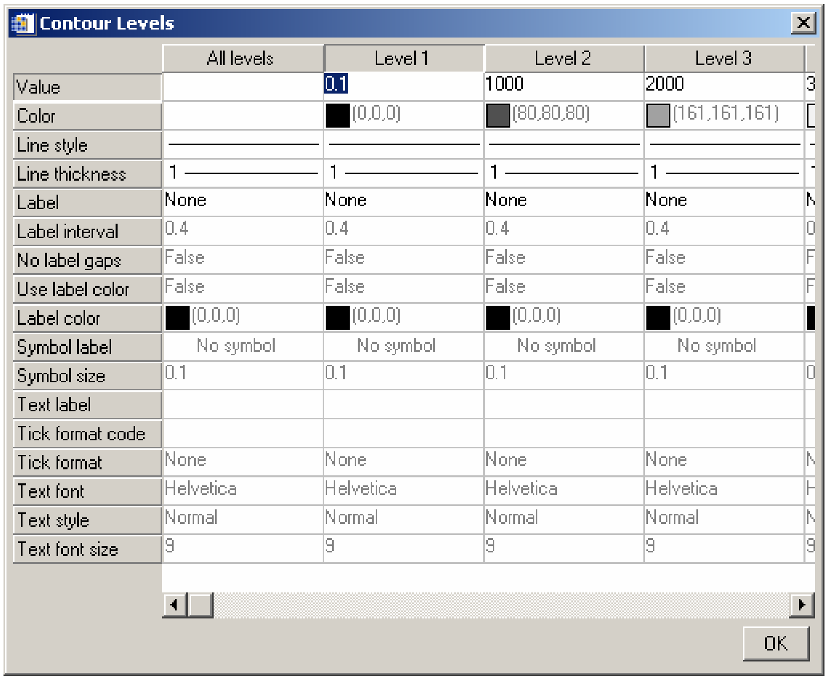
The file is in TIFF format and contains elevation values for both the north and south islands of New Zealand. The elevation data is in units of meters and ranges from sea-level (0 m) up to the highest elevation (3173 m). Start by inputting this data into a new *iContour* utility :

1. IDL> `iContour`
2. Select "*File > Open…*" from the *iContour* menu, select the "*New_Zealand_DEM.tif*" file, and press "*Open*".

A contour object with the default properties will be inserted into the *IDL iContour* window. Since the *iContour* utility automatically creates a certain number of contours at evenly spaced intervals from the minimum data value to the maximum, it is difficult to see the outline of the two islands. In this case, it is beneficial to change the minimum contour value from the default 0.0 (sea level) to an elevation slightly above the ocean in order to see the edge of the land mass. In addition, the data space scaling is anisotropic by default, which leads to a difference between the length of two axes.

3. From the *iContour* menu select "*Window > Visualization Browser…*".
4. Within the *Visualization Browser* window, select the *Data Space* object and change the "Isotropic scaling" property to "*Isotropic*".
5. Next, select the *Contour* object, click on the box to the right of the "Contour level properties" field, and select "*Edit…*".
6. Within the *Contour Levels* dialog, change the "Value" field for "Level 1" from *0* to *0.1* [Fig. 8-9].
7. Press the "*OK*" button to dismiss the *Contour Levels* dialog.
8. Close the *Visualization Browser* window.

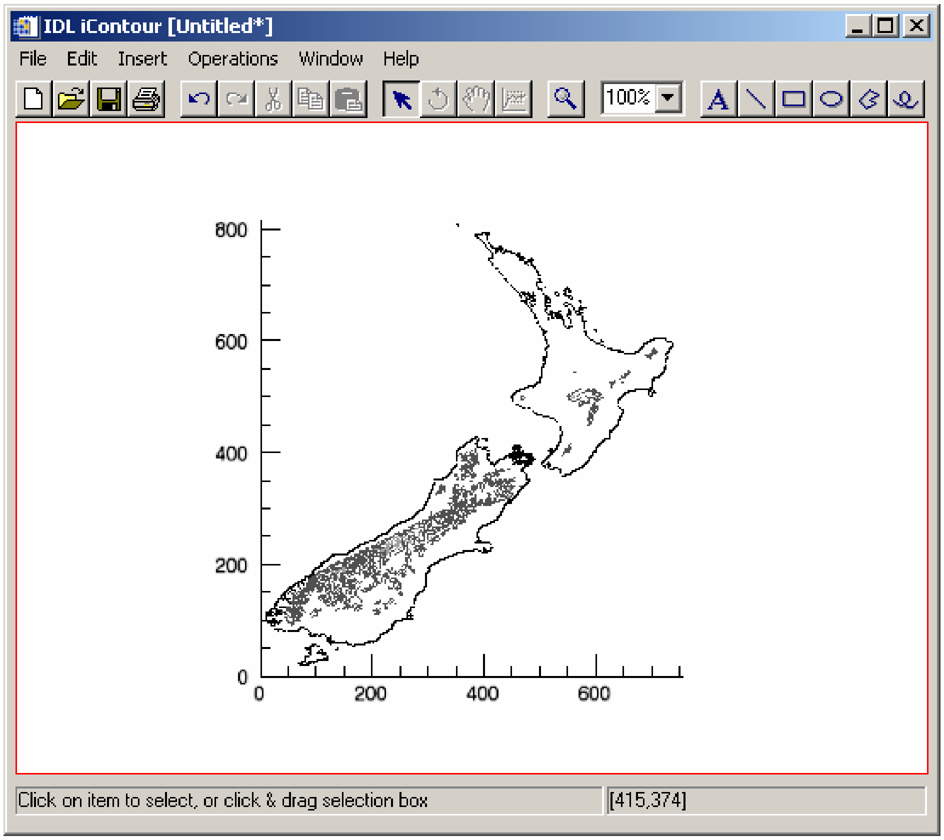
The outline of both the north and south islands of New Zealand should now be readily visible [Fig. 8-10].



Contour Levels

| | All levels | Level 1 | Level 2 | Level 3 | |
|---|---|---|---|---|---|
| Value | | 0.1 | 1000 | 2000 | 3 |
| Color | | ■ (0,0,0) | ■ (80,80,80) | ■ (161,161,161) | [ |
| Line style | ——— | ——— | ——— | ——— | - |
| Line thickness | 1 ——— | 1 ——— | 1 ——— | 1 ——— | |
| Label | None | None | None | None | N |
| Label interval | 0.4 | 0.4 | 0.4 | 0.4 | 0 |
| No label gaps | False | False | False | False | F |
| Use label color | False | False | False | False | F |
| Label color | ■ (0,0,0) | ■ (0,0,0) | ■ (0,0,0) | ■ (0,0,0) | ■ |
| Symbol label | No symbol | No symbol | No symbol | No symbol | |
| Symbol size | 0.1 | 0.1 | 0.1 | 0.1 | 0 |
| Text label | | | | | |
| Tick format code | | | | | |
| Tick format | None | None | None | None | N |
| Text font | Helvetica | Helvetica | Helvetica | Helvetica | H |
| Text style | Normal | Normal | Normal | Normal | N |
| Text font size | 9 | 9 | 9 | 9 | 9 |

*Figure 8-9: Setting a contour level just above sea-level*



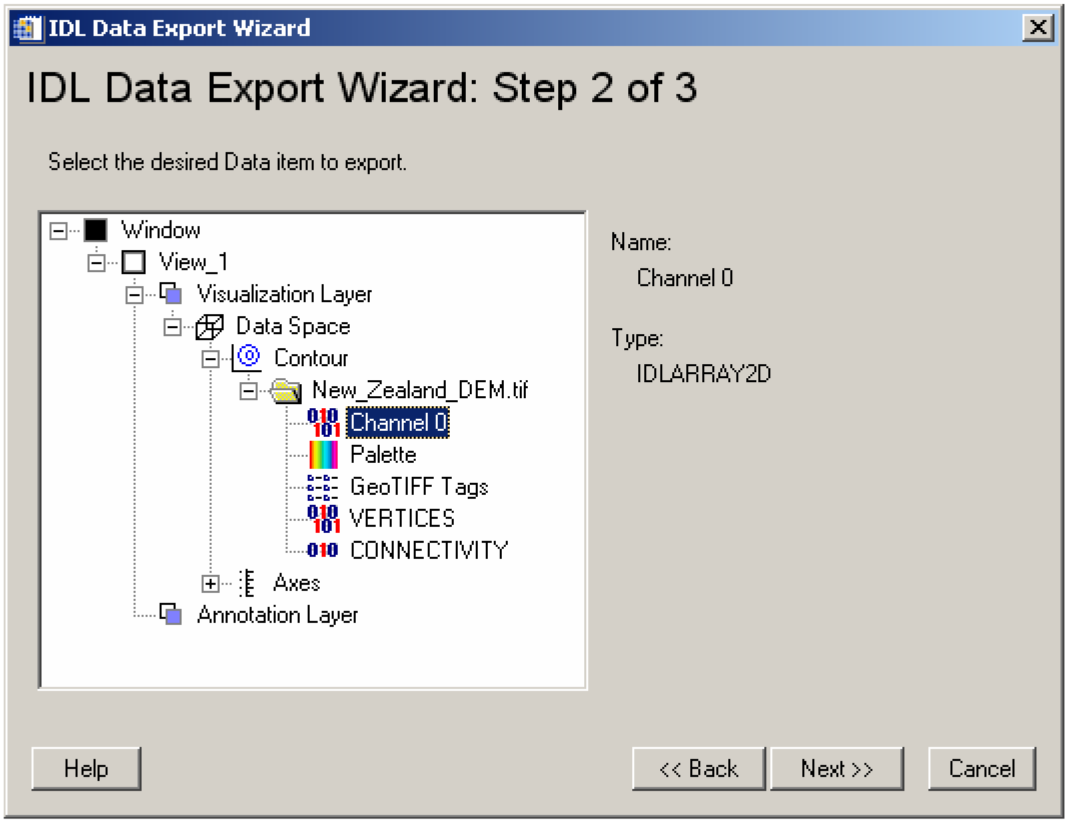
Click on item to select, or click & drag selection box

[415,374]

*Figure 8-10: Contour of the New Zealand digital elevation model data*

Currently the DEM dataset is displayed in the geographic lat/lon coordinate system. The data can be exported to the IDL> command line and warped using the appropriate programmatic mapping routines so it can be displayed in an actual map projection :

9. Select "*File > Export…*" from the *IDL iContour* window.
10. In Step 1 of 3 of the *IDL Data Export Wizard*, select "*To an IDL Variable*" and press "*Next >>*".
11. In Step 2 of 3, select the "*Channel 0*" two-dimensional array under the *Parameters* for the *Contour* object and hit "*Next >>*" [Fig. 8-11].
12. In Step 3 of 3, change the "IDL Variable Name:" field to "*nz*" and press "*Finish*".



IDL Data Export Wizard

IDL Data Export Wizard: Step 2 of 3

Select the desired Data item to export.

Name:
Channel 0

Type:
IDLARRAY2D

***Figure 8-11: Step 2 of 3 of the IDL Data Export Wizard***

13. Once finished viewing the contour visualization, close the *IDL iContour* window.
14. Confirm that the new variable named "*nz*" exists at the main IDL level :

```
IDL> HELP, nz
NZ               INT       = Array[760, 819]
```

Next, create variables for the input parameters to the *MAP_PROJ_INIT* function that define the geographic location of the current dataset. The *MAP_PROJ_INIT* function will be used to define the map projection structure for a Stereographic map projection that is centered over the country of New Zealand.

```
15. IDL> cLon = 172.6
16. IDL> cLat = -40.8
17. IDL> limit = [-47.616, 166.242, -33.967, 178.900]
```

Use the *MAP_PROJ_INIT* function to define a Stereographic map projection with the appropriate parameters :

```
18. IDL> sMap = MAP_PROJ_INIT ('Stereographic', CENTER_LON=cLon, $
         CENTER_LAT=cLat, LIMIT=limit)
```

Next, define the range for the current dataset and warp into the Stereographic map projection using the *MAP_PROJ_IMAGE* function :

```
19. IDL> range = [166.242, -47.616, 178.900, -33.967]
20. IDL> warped = MAP_PROJ_IMAGE (nz, range, MAP_STRUCTURE=sMap)
```

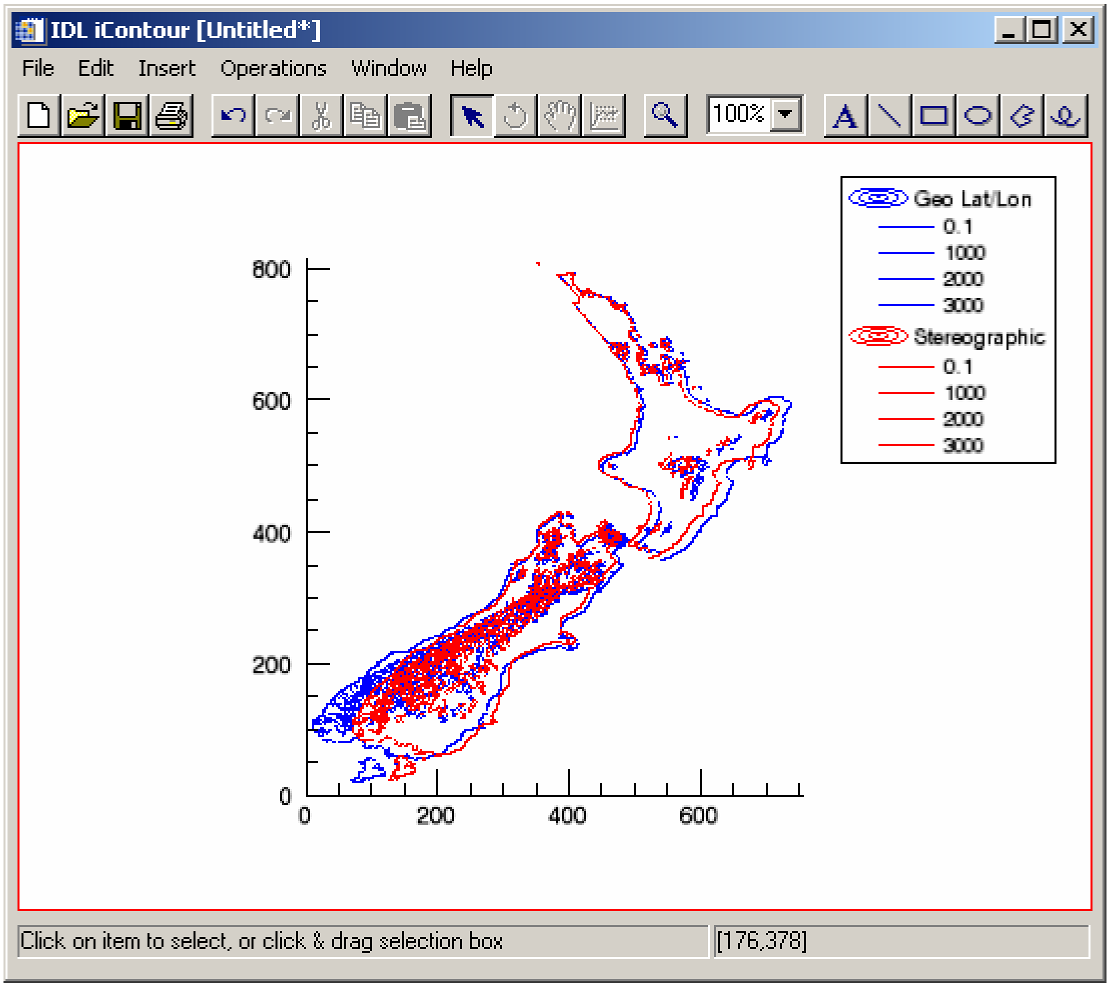
Finally, create a vector of elevation values for the contour levels and load the two datasets into a new *iMap* utility :

```
21. IDL> cValues = [0.1, 1000.0, 2000.0, 3000.0]
22. IDL> iContour, nz, C_VALUE=cValues, COLOR=[0,0,255], $
         NAME='Geo Lat/Lon'
23. IDL> iContour, warped, C_VALUE=cValues, COLOR=[255,0,0], $
         NAME='Stereographic', /OVER
```

Once again, the scaling of the axes are anisotropic by default, so change the scaling and insert a legend into the current visualization :

24. Select "*Window > Visualization Browser…*" from the *iContour* menu system.
25. Select the *Data Space* object and change the "Isotropic scaling" field to "*Isotropic*".
26. While the *Data Space* object is selected, go back to the *iContour* menu system and select "*Insert > New Legend*".
27. Close the *Visualization Browser* window.

The resulting *IDL iMap* visualization window should look similar to Fig. 8-12.



Click on item to select, or click & drag selection box

[176,378]

***Figure 8-12: Contours of New Zealand DEM data in both the geographic lat/lon coordinate system and the Stereographic map projection***

28. Once finished viewing the contour maps, close the *IDL iMap* window.
29. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```

# Chapter 9:  Advanced Graphics

## IDL's Graphical Systems

The IDL software package has 3 distinct graphical systems :

- IDL Intelligent Tools (iTools)
- Object Graphics
- Direct Graphics

All of the graphical visualizations that have been created thus far have been in one of the pre-built iTools utilities (*iPlot*, *iContour*, *iSurface*, *iImage*, *iVolume*, *iMap*).  The three graphical systems are listed above in order of increasing age; the iTools were introduced in IDL version 6.0 (2003), the Object Graphics were introduced in version 5.0 (1997), and the Direct Graphics have been in IDL since the beginning.  There is actually an inherent relationship between the iTools and the Object Graphics because the iTools system is merely a set of interactive (point-and-click) utilities with a graphical user interface that are built on top of the underlying Object Graphics components.  In contrast, the Direct Graphics are a completely separate graphical system that require a very unique syntax.  Each graphical system has distinct advantages in comparison to the others, and this chapter presents an introduction to the Direct and Object Graphics systems.

The Direct Graphics rely on the concept of a **current graphics device**.  The routines in the IDL language that create Direct Graphics will send the graphical output directly to the current graphics device.  The relevant features of the Direct Graphics system are :

- The Direct Graphics send visualizations to a graphics device ("*X*" for X-windows systems displays, "*WIN*" for Microsoft Windows displays, "*PS*" for PostScript files, etc.).  The user switches between graphics devices using the *SET_PLOT* procedure, and controls the features of the current graphics device using the *DEVICE* procedure.
- Once a Direct Graphics mode visualization is drawn to the current graphics device, it cannot be altered or re-used.  This means that if the user wishes to re-create the graphic on a different device, they must re-issue the IDL commands to create the graphic again.
- When the user adds a new item to an existing Direct Graphics visualization, the new item is drawn in front of the existing graphics.
- Some of the primary routines in the IDL language for creating Direct Graphics are :
    - Display Management :  *WINDOW, WSET, WSHOW, WDELETE*
    - Color Tables :  *TVLCT, LOADCT, XLOADCT, XPALETTE*
    - Line Plots :  *PLOT, OPLOT, PLOTS*
    - Contours :  *CONTOUR, IMAGE_CONT*
    - Surfaces :  *SURFACE, SHADE_SURF, SHOW3*

    - Images : *TV, TVSCL, SLIDE_IMAGE, TVRD, XINTERANIMATE*
    - Volumes : *SLICER3, SHADE_VOLUME, PROJECT_VOL, VOXEL_PROJ, POLYSHADE*
    - Maps : *MAP_SET, MAP_GRID, MAP_CONTINENTS, MAP_IMAGE*

In contrast, the Object Graphics use an **object-oriented** programmer's interface to create graphic objects, which must then be explicitly drawn to a destination of the programmer's choosing. The relevant features of the Object Graphics system are :

- Object Graphics are device independent. There is no concept of a current graphics device when using object-mode graphics; any graphics object can be displayed on any physical device for which a destination object can be created.
- Object Graphics are object-oriented. Graphic objects are meant to be created and re-used; the user may create a set of graphic objects, modify their attributes, draw them to a window on the computer screen, modify their attributes again, then draw them to a printer device without reissuing all of the IDL commands used to create the objects. Graphics objects also encapsulate functionality. This means that individual objects include method routines that provide functionality specific to the individual object.
- Object graphics are rendered in three dimensions. Rendering implies many operations not needed when drawing Direct Graphics, including calculation of normal vectors for lines and surfaces, lighting considerations, and general object overhead. As a result, the time needed to render a given object—a surface, say—will often be longer than the time taken to draw the analogous graphic in Direct Graphics.
- Object Graphics use a programmer's interface. Unlike Direct Graphics, which are well suited for both programming and interactive, ad hoc use, Object Graphics are designed to be used in programs that are compiled and run. While it is still possible to create and use graphics objects directly from the IDL> command prompt, the syntax and naming conventions make it more convenient to build a program offline than to create graphics objects on the fly.
- Because Object Graphics persist in memory, there is a greater need for the programmer to be cognizant of memory issues and memory leakage. Efficient design—remembering to destroy unused object references and cleaning up—will avert most problems, but even the best designs can be memory-intensive if large numbers of graphic objects (or large datasets) are involved.
- Some of the primary object classes in the IDL language for creating Object Graphics are :
    - Display Management : *IDLgrWindow, IDLgrScene, IDLgrView, IDLgrViewgroup, IDLgrModel*
    - Color Tables : *IDLgrPalette, IDLgrColorbar*
    - Line Plots : *IDLgrPlot*
    - Contours : *IDLgrContour*
    - Surfaces : *IDLgrSurface*
    - Images : *IDLgrImage*
    - Volumes : *IDLgrVolume*
    - Geometries : *IDLgrPolyline, IDLgrPolygon*
    - Auxiliary : *IDLgrAxis, IDLgrFont, IDLgrLight, IDLgrROI, IDLgrBuffer, IDLgrClipboard, IDLgrPrinter, IDLgrLegend, IDLgrMPEG, IDLgrPattern*

The following exercise will illustrate a simple comparison between the Direct and Object Graphics systems. Start by generating a small two-dimensional array using the *BESELJ* and *DIST* functions :

```
1. IDL> data = BESELJ (SHIFT (DIST (50), 25, 25) / 2, 0)
2. IDL> HELP, data
   DATA            FLOAT     = Array[50, 50]
```

This two-dimensional array can be visualized using the Direct Graphics version of the surface graphic by making a call to the *SURFACE* procedure :

```
3. IDL> SURFACE, data
```

The resulting display window should look similar to Fig. 9-1.

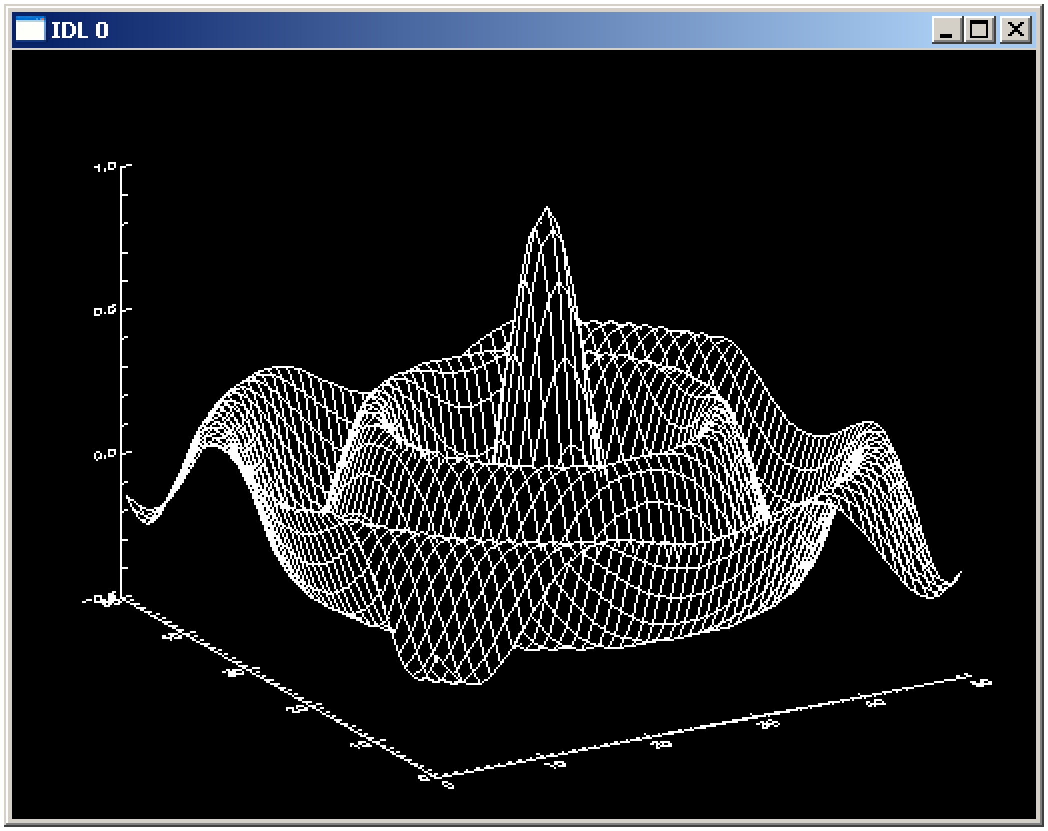


***Figure 9-1: Direct Graphics surface visualization***

One of the primary benefits of the Direct Graphics system is its ability to rapidly render graphics to the current device, especially when the dataset is very large. However, once the visualization is drawn to the display device the user no longer has any control over the graphic.

4. For example, position the cursor over the *IDL 0* window and click-and-drag using the mouse. Notice that the Direct Graphics display window does not provide the ability to interactively manipulate (rotate, translate, zoom, etc.) the current visualization.

Furthermore, the surface visualization is drawn to the display window only once. Consequently, if the display window is resized the scaling of the graphic does not automatically update.

5. Click on the □ maximize button in the upper-right hand corner of the *IDL 0* window, or simply click on the edge of the window and drag.

Notice that the surface visualization does not automatically resize with the change in window size. Moreover, if the user decreases the size of the display window to an area smaller than the original graphic, then re-exposes the area where the graphic first appeared the visualization will not automatically repair itself.

6. Click on the edge of the *IDL 0* window and make it small enough to hide a portion of the surface graphic. Then, drag the edge of the window back out to show the full display area for the original visualization. Notice that the graphic has been erased.
7. Close the current *IDL 0* graphics window.

If the user wants to re-draw the graphic, or modify some property of the visualization, the commands that are used to create the graphic must be re-issued. In addition, aspects of the current graphics device (e.g. the current color lookup table) might need to be modified. For example, in order to change the color of the wire-mesh surface graphic to red, decomposed color must be disabled using the *DEVICE* procedure, the RGB triplet for the color red must be loaded into a color table index using the *TVLCT* procedure, and the call to the *SURFACE* procedure must be re-issued :

8. IDL> `DEVICE, DECOMPOSED=0`
9. IDL> `TVLCT, 255, 0, 0, 1`
10. IDL> `SURFACE, data, COLOR=1`

The surface graphic and axes within the Direct Graphics visualization window should now be red.

11. Once finished viewing the Direct Graphics surface visualization, close the *IDL 0* display window.

Now create the same surface visualization within the Object Graphics system. This can be accomplished by making a call to the *OBJ_SURFACE* procedure :

12. IDL> `OBJ_SURFACE, data`

**Note:** The *OBJ_SURFACE* routine is a custom program that is included with the distribution materials for this IDL Quick Start tutorial. This program is basically a wrapper on top of the appropriate Object Graphics components. If executing this statement results in the error "`% Error opening file.`", then the "IDL_QS_Files/lib/" subfolder that contains this batch file is not included in IDL's path. Please go back to chapter 1 of this Quick Start tutorial and perform the steps delineated in the section entitled "Installing the Tutorial Files" in order to rectify this problem.

The surface graphic visualization will appear within a window that has a bit more functionality than the default Direct Graphics display window [Fig. 9-2]. A series of bitmap buttons across the toolbar expose manipulation functionality (rotate, zoom, pan, select, reset), and the menu system allows the user to export the graphic, print, copy to clipboard, change the background color, etc..

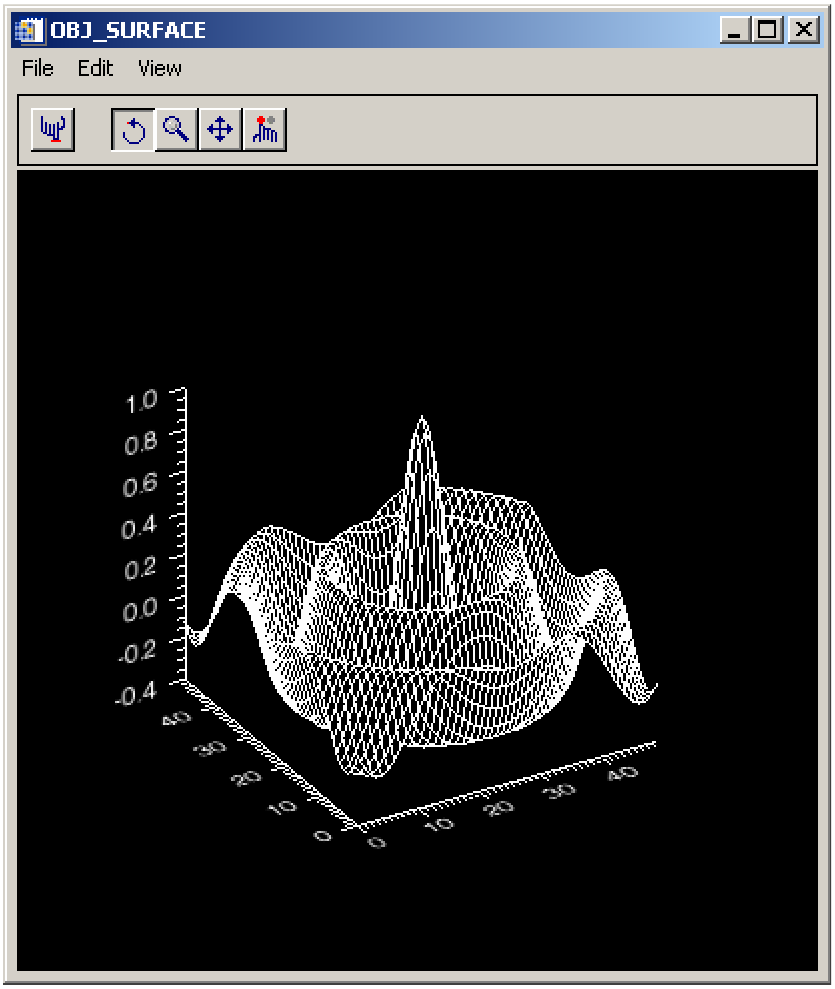


***Figure 9-2: Object Graphics surface visualization***

By default, this Object Graphics display utility is setup in Rotate manipulation mode :

13. Position the cursor within the window and click-and-drag with the mouse in order to rotate the graphic.

One of the primary advantages of the inherent 3-D rendering of the Object Graphics system is the ability to expose robust interactive manipulation capabilities to the user.  In addition to the Rotate manipulation, experiment with the Zoom and Pan capabilities :

14. Click on the 🔍 Zoom button to enter zoom manipulation mode, then click-and-drag with the mouse cursor within the display window.
15. Click on the ✥ Pan button to enter translate manipulation mode, then click-and-drag with the mouse cursor within the display window.

The advantages of the Object Graphics system over the Direct Graphics system, especially when working with 3-Dimensional visualizations, should be readily apparent. However, the Object Graphics still lack the robust interactive graphical properties modification, annotation, ROI, undo/redo, and operations of the iTools graphical system. For example, in order to change the color of the surface graphic to red, the call to the *OBJ_SURFACE* procedure must once again be re-issued :

16. Close the current *OBJ_SURFACE* display window.
17. Re-issue the call to the *OBJ_SURFACE* procedure, but this time set the *COLOR* keyword equal to the color index for red :

```
IDL> OBJ_SURFACE, data, COLOR=1
```

The resulting wire-mesh surface graphic should now be red in color.

18. Once finished viewing the surface visualization, close the *OBJ_SURFACE* window.

The evolution of IDL's graphical systems from Direct Graphics → Object Graphics → iTools has led to the development of a robust environment for data visualization that has the convenience of point-and-click user interaction along with the power and control of a programming language. At this point, users might be asking themselves, "Why would I ever want to use the Direct or Object Graphics systems directly? Why wouldn't I always just use the iTools graphical system?". The following table clearly delineates the primary advantages and disadvantages of the 3 graphical systems :

| GRAPHICAL SYSTEM | ADVANTAGES | DISADVANTAGES |
|---|---|---|
| Direct Graphics | Rapid Rendering | No Built-In User Interaction |
| | Efficient with Large Datasets | Dependant Upon VRAM Resources |
| | Excellent Programmatic Control | Change Code to Change Graphic |
| Object Graphics | Rendering Done in 3-D | Limited Built-In User Interaction |
| | Object-Oriented Design | Computationally Intensive |
| | Excellent Programmatic Control | Can Require Significant Coding |
| Intelligent Tools | Excellent User Interaction | May Expose Too Much Functionality |
| | Pre-Built Tools (iPlot, iImage, etc.) | iTools System Overhead |
| | Customizable and Extendable | Slower When Using Very Large Data |

***Table 9-1: Advantages and disadvantages of IDL's graphical systems***

## Utilizing the Direct Graphics System

The Direct Graphics system sends graphical output to a specific destination, known as a **device**. The output devices that are supported within the Direct Graphics system of IDL are listed in Table 9-2. Useful information about the current Direct Graphics device can be obtained from the *HELP* procedure by setting the *DEVICE* keyword :

1. IDL> `HELP, /DEVICE`

The output from the execution of this statement should be similar to the following :

```
Available Graphics Devices: CGM HP METAFILE NULL PCL PRINTER PS WIN Z
Current graphics device: WIN
    Screen Resolution: 1024x768
    Simultaneously displayable colors: 16777216
    Number of allowed color values: 16777216
    System colors reserved by Windows: 0
    IDL Color Table Entries: 256
    NOTE: this is a TrueColor device
    NOT using Decomposed color
    Graphics Function: 3 (copy)
    Current Font: System,  Current TrueType Font: <default>
    Default Backing Store: None.
```

Since the example above was generated on a Windows computer the default current graphics device is "*WIN*", which is the computer monitor screen. If this command was executed on a UNIX, Linux, or Mac OS X computer the current graphics device should be "*X*", which is short for X-windows.

| DEVICE NAME | DESCRIPTION |
|---|---|
| CGM | Computer Graphics Metafile |
| HP | Hewlett-Packard Graphics Language (HP-GL) |
| METAFILE | Windows Metafile Format (WMF) |
| NULL | No graphics output |
| PCL | Hewlett-Packard Printer Control Language (PCL) |
| PRINTER | System printer |
| PS | PostScript |
| REGIS | Regis graphics protocol (DEC systems only) |
| TEK | Tektronix compatible terminal |
| WIN | Microsoft Windows |
| X | X Window System |
| Z | Z-buffer pseudo device |

***Table 9-2: Direct Graphics Output Devices***

The *SET_PLOT* procedure must be used to select the graphic device to which IDL directs its output. For example, change the current Direct Graphics output device to the PostScript device and obtain information :

```
2. IDL> SET_PLOT, 'PS'
3. IDL> HELP, /DEVICE
```

The output from the execution of this statement should be similar to the following :

```
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
    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: Helvetica TrueType Font: <default>
    # bits per image pixel: 4
    Font Mapping:
         (!3) 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>
```

At this point, if the user were to execute a routine from the IDL library that creates Direct Graphics visualizations the graphical output would actually be directed to this invisible PostScript device and would not appear on the computer monitor screen.

4. In order to create Direct Graphics visualizations and see them on the display, change the current graphics device back to its original setting :

- **Windows:** IDL> `SET_PLOT, 'WIN'`
- **UNIX et al.:** IDL> `SET_PLOT, 'X'`
- **Mac OS X:** IDL> `SET_PLOT, 'X'`

The *!D* system variable within the IDL language also stores useful information on the current Direct Graphics output device in a structure. The user can view these settings by using the *HELP* procedure with the *STRUCTURE* keyword set :

5. IDL> `HELP, !D, /STRUCTURE`

The output from the execution of this statement should be similar to the following :

```
** Structure !DEVICE, 17 tags, length=84, data length=84:
   NAME            STRING    'WIN'
   X_SIZE          LONG               640
   Y_SIZE          LONG               512
   X_VSIZE         LONG               640
   Y_VSIZE         LONG               512
   X_CH_SIZE       LONG                 7
   Y_CH_SIZE       LONG                10
   X_PX_CM         FLOAT          34.1333
   Y_PX_CM         FLOAT          33.3913
   N_COLORS        LONG          16777216
   TABLE_SIZE      LONG               256
   FILL_DIST       LONG                 1
   WINDOW          LONG                -1
   UNIT            LONG                 0
   FLAGS           LONG            328124
   ORIGIN          LONG      Array[2]
   ZOOM            LONG      Array[2]
```

However, the *!D* system variable is read-only and cannot be modified directly by the user. In order to control the settings for the current graphics device (and obtain other useful information) the *DEVICE* procedure must be used. The *DEVICE* procedure controls the graphic device-specific functions for the device currently selected by *SET_PLOT*. An attempt has been made to isolate all device-specific functionality in this procedure.

For example, the use of decomposed color can be turned back on by executing the following statement :

6. IDL> `DEVICE, DECOMPOSED=1`

Once this is accomplished, executing the command "`HELP, /DEVICE`" will show that the current Direct Graphics device is using decomposed (24-bit) color :

7. IDL> `HELP, /DEVICE`

```
Available Graphics Devices: CGM HP METAFILE NULL PCL PRINTER PS WIN Z
Current graphics device: WIN
    Screen Resolution: 1024x768
    Simultaneously displayable colors: 16777216
    Number of allowed color values: 16777216
    System colors reserved by Windows: 0
    IDL Color Table Entries: 256
    NOTE: this is a TrueColor device
    Using Decomposed color
    Graphics Function: 3 (copy)
    Current Font: System,  Current TrueType Font: <default>
    Default Backing Store: None.
```

The primary advantages of the Direct Graphics system is efficient handling of large datasets and very rapid rendering. The following exercise illustrates one of these benefits by executing a small GUI program that utilizes Direct Graphics (and IDL's widget toolkit). This program displays an image of a digital elevation model and allows the user to click on the image and dynamically display X- and Y- line profiles of the DEM data in a very rapid fashion. The data that is used in this exercise is stored in the file named "*DEM.tif*" located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\DEM.tif*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/DEM.tif*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/DEM.tif*

Start by launching the custom program that is built on top of the Direct Graphics system :

8. IDL> `PROFILE_VIEW`

**Note:** The *PROFILE_VIEW* routine is a custom program that is included with the distribution materials for this IDL Quick Start tutorial. This program is basically a wrapper on top of the appropriate Direct Graphics components. If executing this statement results in the error "`% Error opening file.`", then the "IDL_QS_Files/lib/" subfolder that contains this batch file is not included in IDL's path. Please go back to chapter 1 of this Quick Start tutorial and perform the steps delineated in the section entitled "Installing the Tutorial Files" in order to rectify this problem.

A standard native file selection dialog will appear entitled "*Select "DEM.tif" file to open*".

9. Select the file named "*DEM.tif*" and press the "*Open*" button.

A new Direct Graphics display window entitled "*Profile Viewer*" will appear with an image display of the DEM data and two blank plot windows. This program window allows the user to click-and-drag with the left mouse button on top of the image in order to display both an X- and Y- line profile for the elevation data according to the current cursor location [Fig. 9-3]. Moving the cursor around in the image, the user can see how rapidly the Direct Graphics system is able to update the line profile plots.

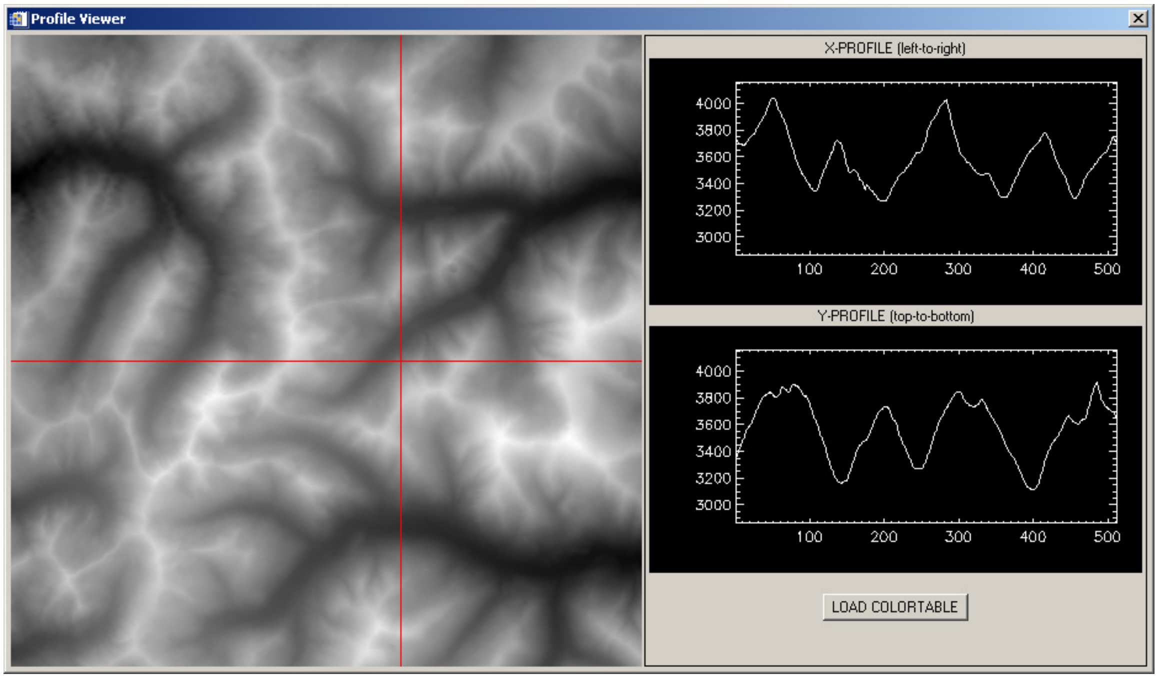


***Figure 9-3: Program illustrating rapid line plot capabilities***

10. Once finished viewing the line profiles, close the *Profile Viewer* window.

The next exercise illustrates another aspect of the Direct Graphics system's rapid rendering by displaying a very large image in a window that allows the user to rapidly move a zoom window around on top of the picture using the mouse. The image that will be used is stored in the file "*St_Louis.jpg*" located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\St_Louis.jpg*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/St_Louis.jpg*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/St_Louis.jpg*

The file is in JPEG format and contains a 2000 x 2000 satellite image of downtown St. Louis, Missouri. Both the Edward Jones Dome and Busch Stadium are readily visible in this image. Start by launching a custom program that is built on top of the Direct Graphics system :

11. IDL> `IMAGE_FULLRES_ZOOM`

**Note:** The *IMAGE_FULLRES_ZOOM* routine is a custom program that is included with the distribution materials for this IDL Quick Start tutorial. This program is basically a wrapper on top of the appropriate Direct Graphics components. If executing this statement results in the error "`% Error opening file.`", then the "IDL_QS_Files/lib/" subfolder that contains this batch file is not included in IDL's path. Please go back to chapter 1 of this Quick Start tutorial and perform the steps delineated in the section entitled "Installing the Tutorial Files" in order to rectify this problem.

A standard native file selection dialog will appear entitled "*Select "St_Louis.jpg" file to open*".

12. Select the file named "*St_Louis.jpg*" and press the "*Open*" button.

A new Direct Graphics display window entitled "*Full Resolution Zoom (click with left or right mouse button)*" will appear with the image graphic. This display window allows the user to click-and-drag with either the left (200 x 200 zoom window) or right (400 x 400 zoom window) mouse button in order to display the current image area at full resolution.

13. Experiment with clicking both the left and right mouse buttons within the image display window [Fig. 9-4].

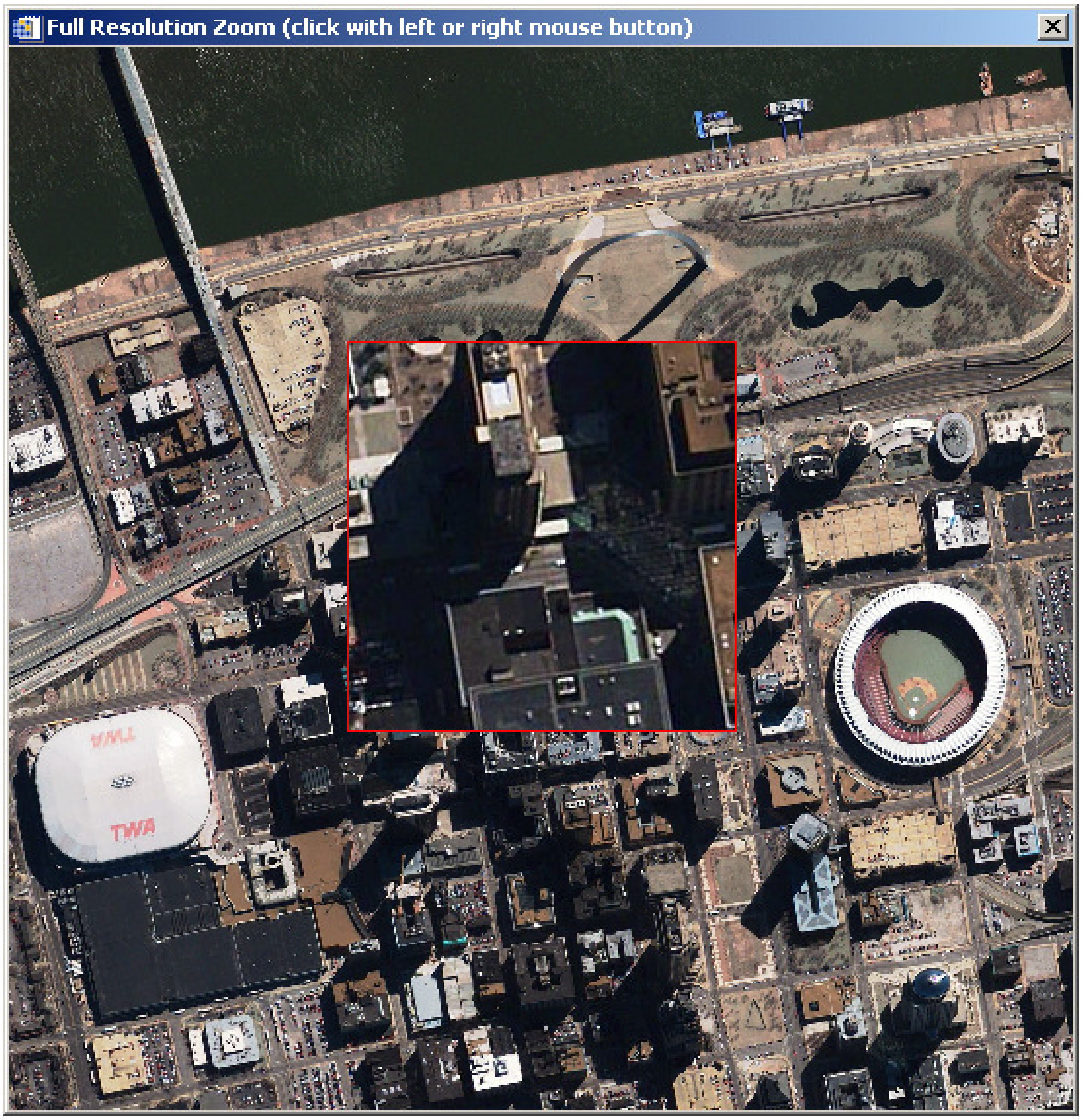

Full Resolution Zoom (click with left or right mouse button)
TWA
TWA

***Figure 9-4: Direct Graphics image display with rapid overlay capabilities***

Notice how rapidly the image display updates as the red zoom box is moved around in the window. When the user moves the zoom box around in the image the *IMAGE_FULLRES_ZOOM* program is utilizing techniques in the Direct Graphics system in order to rapidly display an area surrounding the current cursor position at full resolution within the zoom box, while also repairing the original image display. These operations occur so fast that the user cannot see any delay while the image graphic is being updated.

14. Once finished viewing the image of St. Louis, close the *Full Resolution Zoom* display window.

The following exercise illustrates another powerful aspect of the Direct Graphics system by loading a series of images into IDL and displaying each in a sequential fashion. The delay between the display of each image in the sequence is so short that the visualization has the appearance of an animation (i.e. movie) to the viewer. IDL has a pre-built utility that provides an interface and controls for viewing animations called *XINTERANIMATE*. This utility utilizes the Direct Graphics system to display images in a very rapid fashion.

The images that will be loaded as movie frames into this animation utility are located in the "*frames*" subfolder of the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\frames\**
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/frames/**
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/frames/**

The individual data files found in this "*frames*" subfolder are in BMP format. Start by selecting this folder and inputting the images into the current IDL session. The *DIALOG_PICKFILE* function can be used to display the native file selection dialog. Set the *DIRECTORY* keyword so the dialog prompts the user to select an entire folder instead of individual files :

15. IDL> `folder = DIALOG_PICKFILE (/DIRECTORY)`
16. Within the dialog, select the "*frames*" subfolder and press "*OK*" [Fig. 9-5].

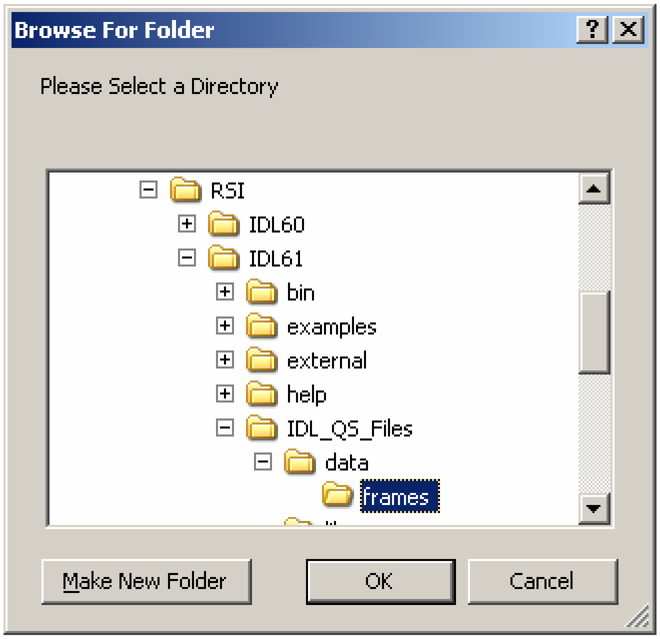


***Figure 9-5: Selecting a directory using the native folder selection dialog***

Confirm that the selected folder was successfully returned as a string into the named variable "*folder*" :

```
17. IDL> HELP, folder
    FOLDER        STRING     = 'C:\RSI\IDL61\IDL_QS_Files\data\frames\'
```

**Note:** The string value for the "*folder*" variable may be different depending on the installation location for the IDL software.

Next, utilize the *FILE_SEARCH* function in order to determine the full path to each of the individual 116 BMP format files located within the "*frames*" subfolder :

```
18. IDL> files = FILE_SEARCH (folder, '*', COUNT=nFiles)
```

Confirm that the number of files located is equal to 116 :

```
19. IDL> HELP, nFiles
    NFILES          LONG      =           116
```

Obtain information on the image data for the first BMP file in the sequence using the *QUERY_BMP* function :

```
20. IDL> query = QUERY_BMP (files[0], info)
21. IDL> HELP, info, /STRUCTURE
    ** Structure <10b5168>, 7 tags, length=40, data length=36,
    refs=1:
       CHANNELS        LONG                 3
       DIMENSIONS      LONG      Array[2]
       HAS_PALETTE     INT              0
       NUM_IMAGES      LONG                 1
       IMAGE_INDEX     LONG                 0
       PIXEL_TYPE      INT              1
       TYPE            STRING    'BMP'
```

```
22. IDL> PRINT, info.dimensions
             320         240
```

The contents of the information structure returned by *QUERY_BMP* illustrate that the image data is of type byte and has a size of 320 columns by 240 rows with 3 color channels. Initialize a new variable named "*frames*" that will be used to store all of the images in the sequence :

```
23. IDL> frames = MAKE_ARRAY (3, 320, 240, nFiles, TYPE=1)
24. IDL> HELP, frames
   FRAMES          BYTE       = Array[3, 320, 240, 116]
```

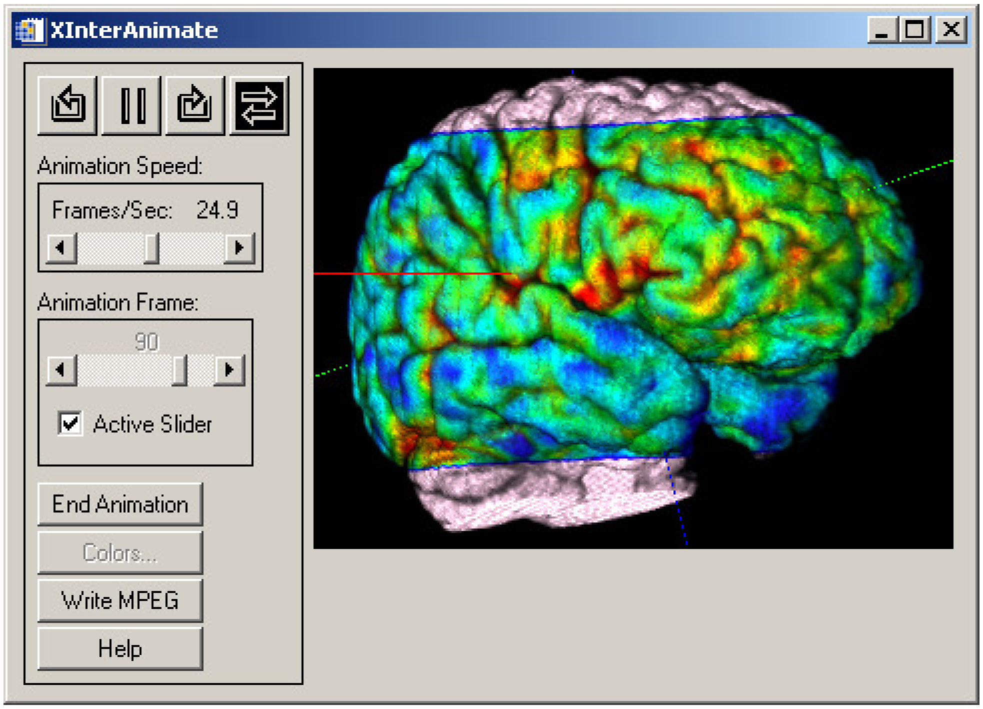
Now that a variable has been initialized within the current IDL session to store all of the images in the sequence, they can be input from their individual BMP format files. Utilize the *FOR* loop control statement to read-in the images from all of the files into the appropriate index of the "*frames*" variable :

```
25. IDL> FOR i=0,nFiles-1 DO frames[*,*,*,i] = READ_BMP(files[i],/RGB)
```

Finally, load the image frames into the *XINTERANIMATE* visualization utility. This is accomplished by calling the XINTERANIMATE procedure 3 times; first to initialize the utility, second to load the individual image frames, and third to actually launch the display window.

```
26. IDL> XINTERANIMATE, SET=[320, 240, 116], /TRACK, /CYCLE
27. IDL> FOR i=0,nFiles-1 DO XINTERANIMATE, FRAME=i, $
          IMAGE=frames[*,*,*,i]
28. IDL> XINTERANIMATE, 50
```

An animation of MRI and PET data of a human brain will be displayed in an interactive display window. The *XINTERANIMATE* utility also has built-in GUI components for controlling the animation. The resulting visualization window should look similar to Fig. 9-6.



Frames/Sec: 24.9

90

☑ Active Slider

***Figure 9-6: Animation of multiple image frames using Direct Graphics***

The extremely fast display speed of the Direct Graphics system can be seen by changing the "Animation Speed:" slider to its highest setting :

29. Click on the vertical bar within the "Frames/Sec:" slider and drag it all the way to the right in order to obtain the maximum setting.

Notice how fast the animation utility is able to update the image display.

30. Once finished viewing the animation, close the *XInterAnimate* window.

## Utilizing the Object Graphics System

The IDL Object Graphics system is a collection of pre-defined object classes, each of which is designed to encapsulate a particular visual representation. These objects are designed for building complex three-dimensional data visualizations. In general, object classes shipped with IDL have names of the form :

IDL*xxYyyy*

where *xx* represents the broad functional grouping (*gr* for graphics objects, *db* for database objects, *an* for analysis objects, *ff* for file format objects, etc.). *Yyyy* is the class name itself (such as *Axis* or *Surface*). For example, the *IDLgrAxis* object provides an encapsulation of all of the components associated with a graphical representation of an axis.

Object Graphics should be thought of as a collection of building blocks. In order to display something on the screen, the user selects the appropriate set of blocks and puts them together so that as a group they provide a visual result. In this respect, Object Graphics are quite different than Direct Graphics. A single line of code is unlikely to produce a complete visualization. Furthermore, a basic understanding of the IDL object system is required (for instance, how to create an object, how to call a method, how to destroy an object, etc.). Because of the level at which these objects are presented, Object Graphics are aimed at application programmers rather than command line users. The syntax involved in working with objects in the IDL language is a bit different than the statements that have been executed thus far in this tutorial. Consequently, a brief introduction to IDL objects and object-oriented concepts is necessary.

IDL objects are actually special variables known as **heap variables**, which means that they are global in scope and provide explicit user control over their lifetimes. Object heap variables can only be accessed via **object references**. In order to perform an action on an object's instance data (such as the modification of attributes), you must call one of the object's **methods**. To call a method, you must use the method invocation operator "->" (the hyphen followed by the greater-than sign). The syntax is :

*ObjRef->Method*

where *ObjRef* is an object reference and *Method* is a method belonging either to the object's class or to one of its superclasses. The method may be specified either partially (using only the method name) or completely using both the class name and method name, connected with two colons :

*ObjRef->ClassName::MethodName*

In order to utilize Object Graphics, the user must build a self-contained hierarchy of the appropriate objects that is subsequently drawn to a destination object. The 5 primary building blocks of an Object Graphics visualization are :

- Destination object : The device (such as a window, memory buffer, file, clipboard, or printer) to which the visualization is to be rendered.
- Scene object : A container that can hold multiple views (if necessary).
- View object : The viewport rectangle (within the destination) within which the rendering is to appear (as well as how data should be projected into that rectangle).
- Model object : A transformation node.
- Atomic Graphic object : A graphical representation of data (such as an axis, line plot, or surface mesh, text annotation, etc.).

For example, in the following exercise a hierarchy of objects will be created that contain a simple line plot visualization, and this hierarchy will be drawn to a display window destination object :

1. Create a simple one-dimensional dataset that can be displayed as a line plot by executing a series of statements at the IDL> command prompt :

   ```
   IDL> x = FINDGEN (720)
   IDL> y = SIN (x * !DTOR) * COS (x * !DTOR / 3)
   IDL> x = x / 450 - 0.8
   ```

2. Create an instance of the *IDLgrPlot* object class for this data using the *OBJ_NEW* function. The object reference for this line plot is returned into the variable named "*oPlot*" :

   ```
   IDL> oPlot = OBJ_NEW ('IDLgrPlot', x, y)
   ```

3. Create an *IDLgrModel* object and execute the "*Add*" method in order to add the line plot object to the model. Once this is accomplished, the model object will contain the plot object :

   ```
   IDL> oModel = OBJ_NEW ('IDLgrModel')
   IDL> oModel -> Add, oPlot
   ```

4. Create an IDLgrView object and add the model to it :

   ```
   IDL> oView = OBJ_NEW ('IDLgrView')
   IDL> oView -> Add, oModel
   ```

5. Create an *IDLgrWindow* destination object, setting the *RETAIN* keyword equal to *2* so that IDL is forced to provide backing store :

   ```
   IDL> oWindow = OBJ_NEW ('IDLgrWindow', RETAIN=2)
   ```

6. Once this is accomplished, the entire object hierarchy can be rendered by executing the *Draw* method with the top object container :

   ```
   IDL> oWindow -> Draw, oView
   ```

The resulting visualization window should look similar to Fig. 9-7.

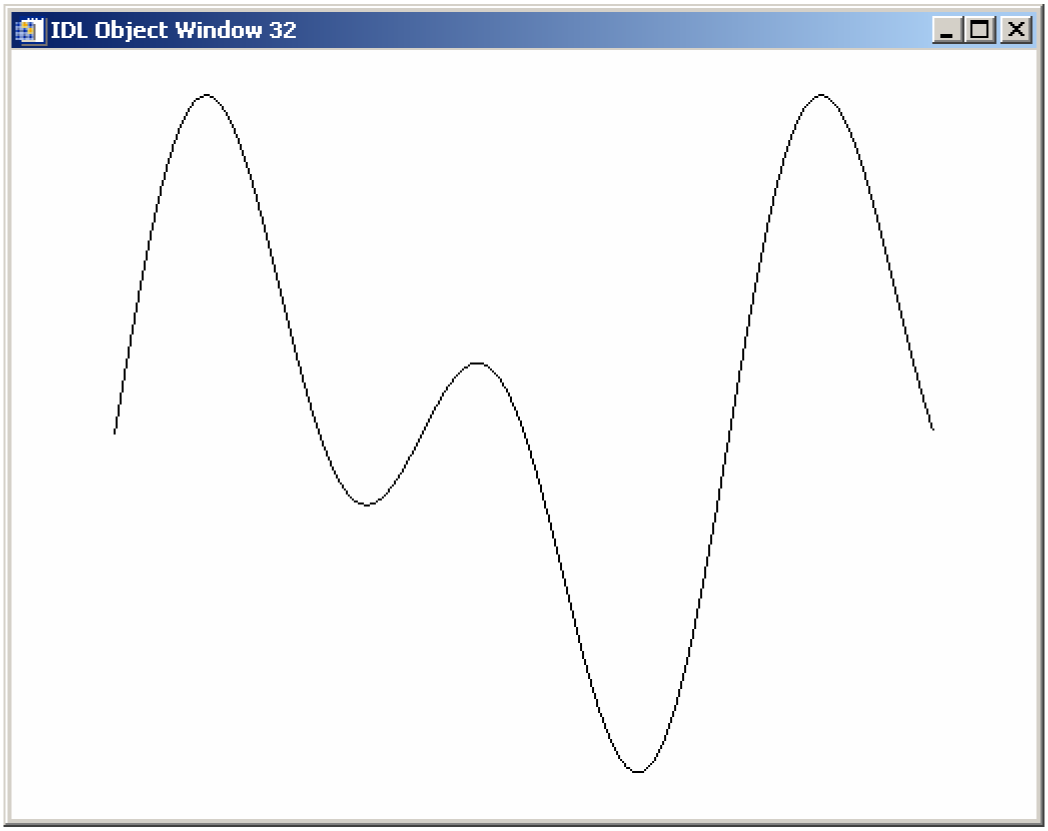


***Figure 9-7: Object Graphics visualization of a line plot***

A simple black line plot will be drawn on top of the white view within the Object Graphics window. Much like Direct Graphics, this is a static display window that does not expose any interactive manipulation capabilities. Furthermore, the visualization does not include axes because instances of the IDLgrAxis class were not explicitly added to the object hierarchy. In order to make modifications to this visualization, such as changing the color and adding an axis, the appropriate object must be created and the SetProperty method must be utilized on the existing objects. For example, the background color can be changed to yellow, the line plot color to blue, and an X axis inserted into the visualization by executing the following statements :

```
7.  IDL> oPlot -> SetProperty, COLOR=[0,0,255]
8.  IDL> oView -> SetProperty, COLOR=[255,255,0]
9.  IDL> oAxis = OBJ_NEW ('IDLgrAxis', RANGE=[-0.8,0.8], /EXACT)
10. IDL> oModel -> Add, oAxis
11. IDL> oWindow -> Draw, oView
```

The resulting visualization window should look similar to Fig. 9-8.

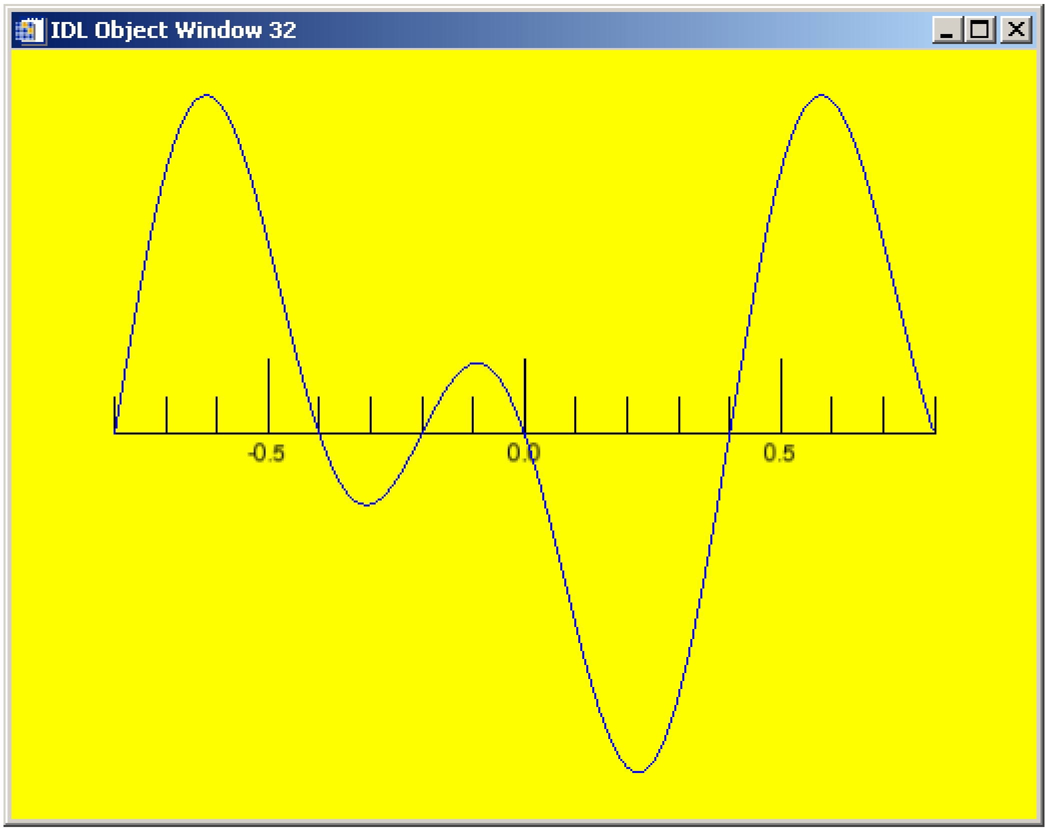


***Figure 9-8: Modification of the graphic colors and insertion of an axis***

Since IDL objects are stored as heap variables, they persist in memory until they are explicitly destroyed. Information on all of the current heap variables that exist within an IDL session can be obtained by executing the HELP procedure with the HEAP keyword set :

```
12. IDL> HELP, /HEAP
    Heap Variables:
        # Pointer: 13
        # Object : 8

    <ObjHeapVar1>    STRUCT     = -> IDLGRPLOT Array[1]
    <ObjHeapVar3>    STRUCT     = -> IDLGRDATA Array[1]
    <PtrHeapVar4>    FLOAT      = Array[3, 720]
    <PtrHeapVar5>    STRUCT     = -> IDL_CONTAINER_NODE Array[1]
    <ObjHeapVar6>    STRUCT     = -> IDLGRMODEL Array[1]
    <PtrHeapVar7>    STRUCT     = -> IDL_CONTAINER_NODE Array[1]
    <ObjHeapVar8>    STRUCT     = -> IDLGRVIEW Array[1]
    <PtrHeapVar10>   STRUCT     = -> IDL_CONTAINER_NODE Array[1]
    <ObjHeapVar11>   STRUCT     = -> IDLGRWINDOW Array[1]
    <PtrHeapVar12>   BYTE       = Array[3]
    <PtrHeapVar13>   BYTE       = Array[3]
    <ObjHeapVar14>   STRUCT     = -> IDLGRAXIS Array[1]
    <PtrHeapVar15>   BYTE       = Array[3]
    <PtrHeapVar16>   DOUBLE     = Array[3]
    <ObjHeapVar17>   STRUCT     = -> IDLGRTEXT Array[1]
    <PtrHeapVar18>   BYTE       = Array[3]
```

```
<PtrHeapVar19>  STRING    = Array[3]
<PtrHeapVar20>  DOUBLE    = Array[3, 3]
<PtrHeapVar21>  OBJREF    = <ObjHeapVar17(IDLGRTEXT)>
<ObjHeapVar22>  STRUCT    = -> IDLGRFONT Array[1]
<PtrHeapVar23>  STRUCT    = -> IDL_CONTAINER_NODE Array[1]
```

The output from the HELP procedure illustrates that a number of objects and pointers have been created and are currently using memory within the current IDL session. In order to destroy all of the heap variables associated with this visualization and free the memory that is currently being used, the OBJ_DESTROY procedure must be used in order to complete the object life cycle :

13. IDL> `OBJ_DESTROY, oWindow`
14. IDL> `OBJ_DESTROY, oView`

Notice that destroying the top level object in a hierarchy (the view object in this case) destroys all of the other objects it contains.  Once this is accomplished, another call to the *HELP* procedure should show that all of the heap variables have been destroyed (and the memory they were using released) :

15. IDL> `HELP, /HEAP`

```
Heap Variables:
    # Pointer: 0
    # Object : 0
```

The number of steps involved in creating an Object Graphics visualization from scratch clearly demonstrate the need to write programs that create the entire object hierarchy that is needed in order to display the graphic.  Fortunately, there are a number of pre-built **utilities** that are included with the IDL software package that help the user visualize data using Object Graphics.

One of these utilities is called *XDXF*, and it is used to visualize geometric shapes stored in AutoCAD DXF format files as 3-Dimensional polygonal objects.  Another more generic utility called *XOBJVIEW* can be used to quickly and easily visualize and manipulate Object Graphics on screen.  In the following exercise, a **geometry** from the example data file "*F-14.dxf*" will be input into both of these utilities.  This example data file is located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\F-14.dxf*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/F-14.dxf*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/F-14.dxf*

This dataset contains the vertices and polygonal connectivity to define the three-dimensional geometry for a F-14 fighter jet.  Start by launching the *XDXF* utility :

16. IDL> `XDXF`

Since no filename was specified, a native file selection dialog will appear entitled "*Select DXF File to Read*".

17. Select the "*F-14.dxf*" file and press the "*OK*" button.

Two separate windows will appear once the data has been input from the DXF file; a display window entitled "*F-14.dxf*", and a window entitled "*XDXF Information*" that displays information on the contents of the DXF file. The user can click-and-drag with the mouse and execute the selected manipulation control on the object within the display window. There is also several pieces of functionality exposed in the menu system built into the *F-14.dxf* window.

18. Once finished viewing the 3-D geometry visualization, close the *XDXF Information* window.

Within the *XDXF* utility, the geometry object is displayed by default with flat shading and the vertex colors that are defined by the DXF file (in this case, all white). There is no way to modify the appearance of the object unless the properties of the underlying polygonal mesh object itself are modified. In order to give the geometry object a more genuine appearance similar to that of a real F-14 fighter jet, the Object Graphics components must be built manually.

In addition to graphical object classes, IDL also contains several file format objects that are used to read and write data to files on disk. The DXF file format is one that is handled using an object class in the IDL software package. The object class is named "*IDLffDXF*", and it can be utilized to input the data from the file "*F-14.dxf*" file. Start by using the *DIALOG_PICKFILE* function to display the native file selection dialog and select the "*F-14.dxf*" file on the harddrive :

```
19. IDL> file = DIALOG_PICKFILE (FILTER='F-14.dxf')
20. IDL> PRINT, file
    C:\RSI\IDL61\IDL_QS_Files\data\F-14.dxf
```

**Note:** The string value for the "*file*" variable may be different depending on the installation location for the IDL software.

Next, create an instance of the IDLffDXF object class and read-in the entity list by executing the *Read* method :

```
21. IDL> oDXF = OBJ_NEW ('IDLffDXF')
22. IDL> status = oDXF -> Read (file)
```

Obtain the contents of the DXF file and filter the result so it only contains "*FACE3D*" entity types by setting the filter argument equal to 10 while executing the *GetContents* method :

```
23. IDL> contents = oDXF -> GetContents (10)
```

Once the contents have been determined, the actual data can be read-in by executing the *GetEntity* method :

```
24. IDL> data = oDXF -> GetEntity (contents)
```

Make sure to destroy the object and free the heap memory :

```
25. IDL> OBJ_DESTROY, oDXF
```

Once this is accomplished, a new structure named “*data*” will exist in the current IDL session :

```
26. IDL> HELP, data, /STRUCTURE
   ** Structure IDL_DXF_POLYGON, 13 tags, length=80, data length=78:
      EXTRUSION       DOUBLE    Array[3]
      VERTICES        POINTER   <PtrHeapVar6>
      CONNECTIVITY    POINTER   <PtrHeapVar7>
      VERTEX_COLORS   POINTER   <PtrHeapVar8>
      MESH_DIMS       INT       Array[2]
      CLOSED          INT       Array[2]
      COLOR           INT            256
      FIT_TYPE        INT             -1
      CURVE_FIT       INT              0
      SPLINE_FIT      INT              0
      DXF_TYPE        INT             10
      BLOCK           STRING    ''
      LAYER           STRING    '0'
```

The vertices and polygonal connectivity can be extracted from the structure by dereferencing their respective pointers :

```
27. IDL> vertices = *data.vertices
28. IDL> polygons = *data.connectivity
29. IDL> HELP, vertices, polygons
   VERTICES        DOUBLE    = Array[3, 4427]
   POLYGONS        LONG      = Array[30372]
```

A new polygon object can be created using the IDLgrPolygon class :

```
30. IDL> oJet = OBJ_NEW ('IDLgrPolygon', vertices, POLYGONS=polygons)
```

Once the polygon object has been created, change some of the properties to modify the appearance of the geometry so it looks more like a grayish-blue metal aircraft :

```
31. IDL> oJet -> SetProperty, AMBIENT=[99,124,180]
32. IDL> oJet -> SetProperty, DIFFUSE=[129,129,154]
33. IDL> oJet -> SetProperty, SPECULAR=[155,155,155]
34. IDL> oJet -> SetProperty, SHININESS=21.2
35. IDL> oJet -> SetProperty, SHADING=1
```

Now that the polygon object has been created and its properties modified, it can be visualized in the generic *XOBJVIEW* utility :

```
36. IDL> XOBJVIEW, oJet, BACKGROUND=[0,0,0]
```

You can rotate the image, zoom in and move it within the window.  The resulting visualization window should look similar to Fig. 9-9.

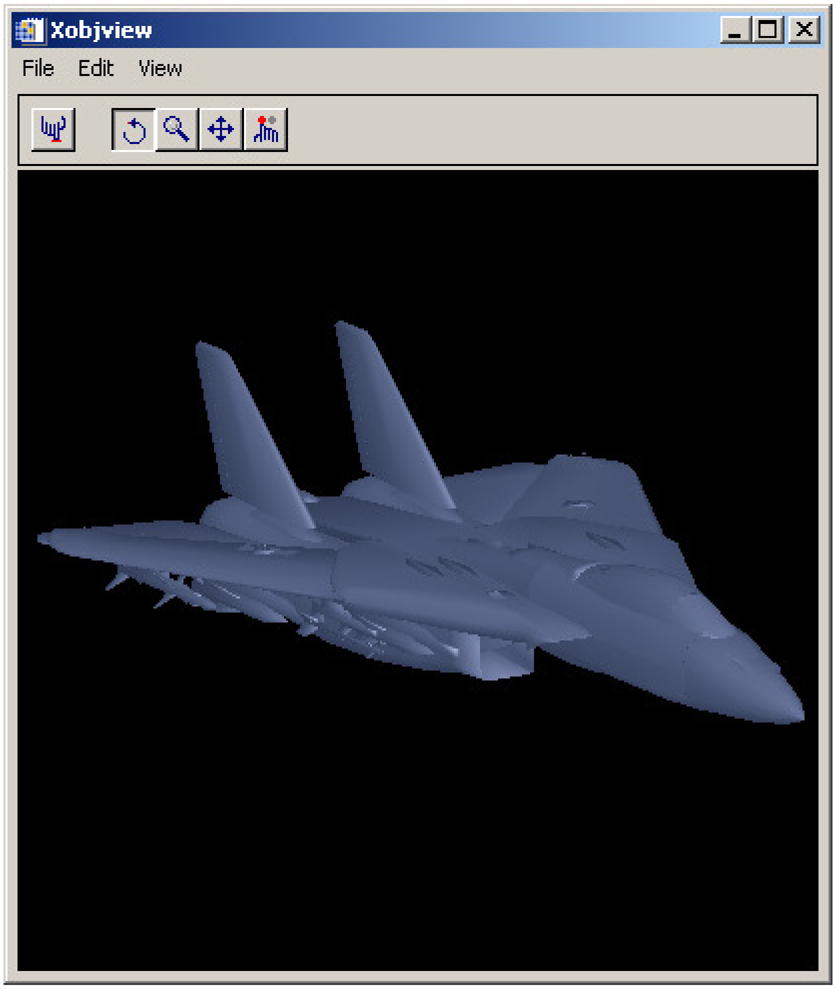


***Figure 9-9: Visualization of F-14 fighter jet geometry from a DXF format file***

37. Once finished viewing the polygon object, close the *Xobjview* window.

Another powerful aspect of the Object Graphics system is that most of the atomic graphic objects have a property named either *ALPHA_CHANNEL* or *BLEND_FUNCTION* that can be used to apply a **transparency** effect to the visualization. This allows the user to create composite visualizations with more than one object that involves an opacity adjustment in order to *see through* certain objects. An example of this methodology is provided in the program named *ALPHA_BLENDING*, which uses 2 images stored in the file "*pictures.sav*" located in the "*data*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\data\pictures.sav*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/data/pictures.sav*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/data/pictures.sav*

The file is in IDL save file format and contains two color photographs from Arches National Park. Start by launching the custom program :

38. IDL> `ALPHA_BLENDING`

**Note:** The *ALPHA_BLENDING* routine is a custom program that is included with the distribution materials for this IDL Quick Start tutorial. This program is basically a wrapper on top of the appropriate Object Graphics components. If executing this statement results in the error "`% Error opening file.`", then the "IDL_QS_Files/lib/" subfolder that contains this batch file is not included in IDL's path. Please go back to chapter 1 of this Quick Start tutorial and perform the steps delineated in the section entitled "Installing the Tutorial Files" in order to rectify this problem.

A standard native file selection dialog will appear entitled "*Select "pictures.sav" file to open*".

39. Select the file named "*pictures.sav*" and press the "*Open*" button.

A new Object Graphics display window will appear that shows a gradual transition from one image to the next and back again [Fig. 9-10].

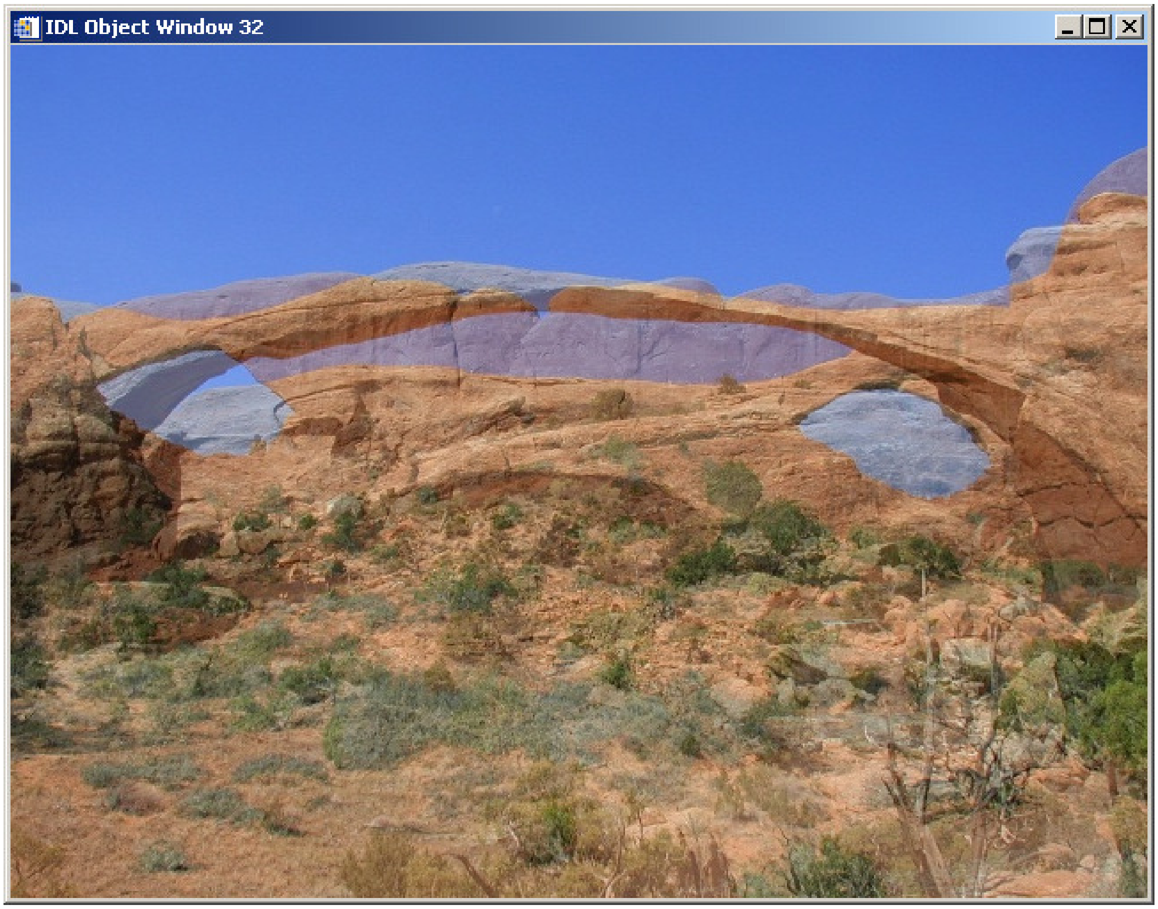


***Figure 9-10: Transparency transition from one image to another***

40. Once finished viewing the images, close the *IDL Object Window 32* window.
41. Before moving on to the next chapter, it is a good idea to reset the IDL session. This can be accomplished by executing the statement :

```
IDL> .reset_session
```

# Chapter 10:  Programming in IDL

## Introduction to IDL Programming

An IDL program consists of one or more IDL commands that are executed in a sequential fashion.  The IDL software integrates a powerful, array-oriented language with numerous mathematical analysis and graphical display techniques. Programming in IDL is a time-saving alternative to programming in compiled computer languages such as FORTRAN or C.  Using IDL, tasks which require days or weeks of programming with traditional languages can be accomplished in hours.  The user can explore data interactively using IDL commands and then create complete applications by writing IDL programs.  Advantages of programming in IDL include :

- IDL is a complete, structured language that can be used both interactively and to create sophisticated algorithms and interactive applications.
- Operators and functions work on entire arrays (without using loops), simplifying interactive analysis and reducing programming time.
- Immediate compilation and execution of IDL commands provides instant feedback and "hands-on" interaction.
- Rapid 2D plotting, multi-dimensional plotting, volume visualization, image display, and animation allow the user to observe the results of computations immediately.
- Many numerical and statistical analysis routines—including Numerical Recipes routines—are provided for analysis and simulation of data.
- IDL's flexible input/output facilities allow the user to read any type of custom data format. Support is also provided for common image standards (including BMP, JPEG, and XWD) and scientific data formats (CDF, HDF, and NetCDF).
- IDL widgets can be used to quickly create multi-platform graphical user interfaces.
- IDL programs run the same across all supported platforms (Microsoft Windows and a wide variety of Unix systems) with little or no modification. This application portability allows the program to easily support a variety of computers.
- Existing FORTRAN and C routines can be dynamically-linked into IDL to add specialized functionality.  Alternatively, C and FORTRAN programs can call IDL routines as a subroutine library or display "engine".

There are several different ways in which a computer program can be created, and IDL supports the development of programs using a wide variety of methodologies. There are 5 primary types of IDL programs :

- Batch Files
- Main-Level Programs
- Named Programs (procedures & functions)
    - Object-Oriented Programs (creating IDL objects and their methods)
    - iTools System Programs

These different types of IDL programs are not necessarily completely distinct from one another, and each may or may not include components from IDL's widget toolkit in order to display a graphical user interface. For example, in order to create objects in IDL the programmer must actually write a series of named procedures or functions. Furthermore, iTools system programming is merely an extension of IDL's object-oriented methodology that focuses on customizing and extending the iTools.

IDL is inherently a **procedural language**, which means that the programmer specifies an explicit sequences of steps to follow to produce a result. A procedural program is written as a list of instructions telling the computer, step-by-step, what to do (e.g. open a file, read in data, perform processing, display result, etc.). Procedural programming is a method of computer programming based upon the concept of the unit and scope (the data viewing range of an executable code statement). It is possible for a procedural program to have multiple levels or scopes, with procedures defined inside other procedures. Each scope can contain variables which cannot be seen in outer scopes.

## Batch Files

A **batch file** contains one or more IDL statements or commands. Each line of the batch file is read and executed before proceeding to the next line. This makes batch files different from main-level programs, which are compiled as a unit before being executed, and named programs, in which all program modules are compiled as an unit before being executed. Batch files are sometimes referred to as **include files**, since they can be used to *include* the multiple IDL statements contained in the file in another program.

In the following exercise, a simple batch file is created and executed using the at symbol ("@") special character. The "@" symbol is either used as an include character within other programs or to signal that batch processing is to be performed. Executing the "@" symbol followed by the path to a batch file on disk will execute all of the statements within the batch file, one line at a time, in a sequential fashion.

1. Start by selecting "*File > New > Editor*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. Within the text editor window, enter the following IDL code :

   ```
   file = filepath('marsglobe.jpg', subdir=['examples', 'data'])
   read_jpeg, file, image
   iImage, image
   ```

3. Once these 3 lines of IDL code have been entered, select "*File > Save As…*" from the menu system.
4. Save the text to a new file named "*batch.pro*" located in the "*output*" subfolder of the Quick Start directory :

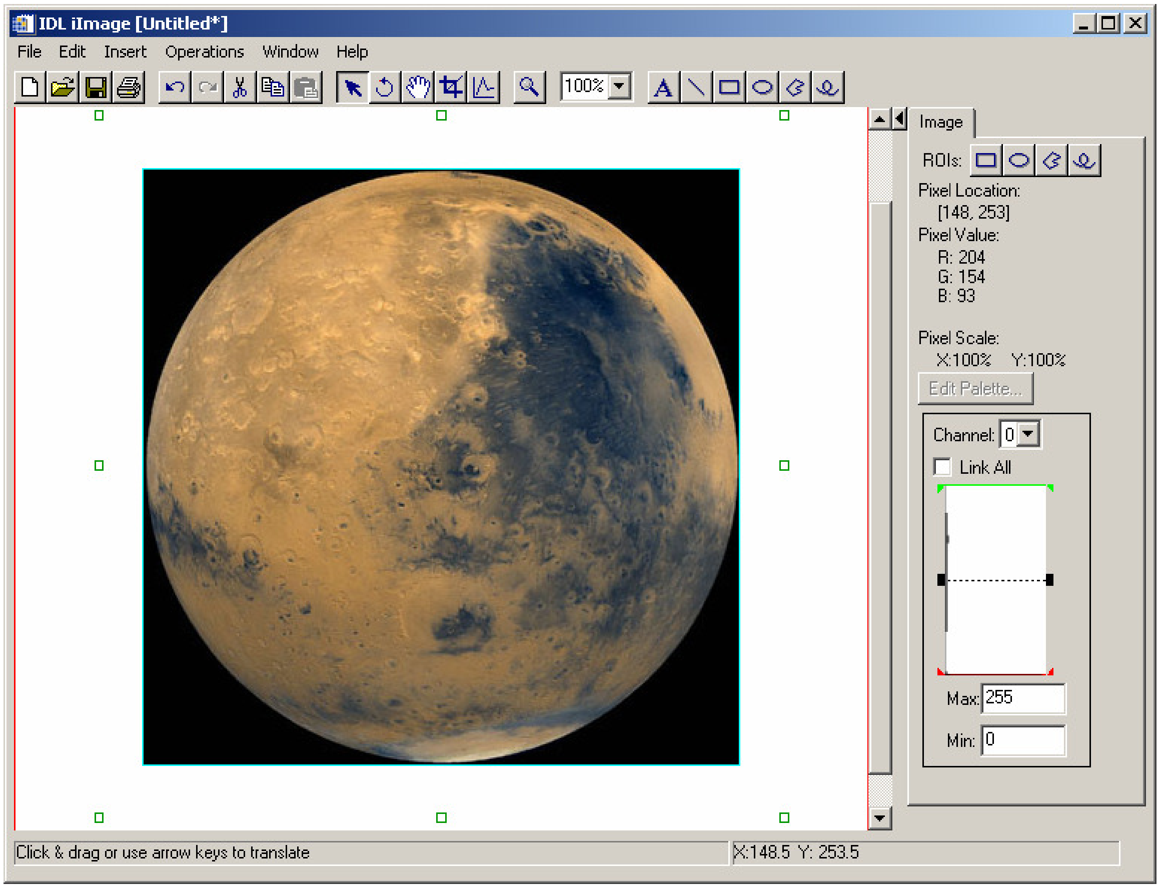
- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\output\batch.pro*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/output/batch.pro*

- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/output/batch.pro*

5. Finally, execute this batch file :

   IDL> `@batch`

This will execute each of the 3 lines from the batch file in sequence as if they were executed individually at the IDL> command prompt. The resulting *iImage* visualization window should look similar to Fig. 10-1.



Pixel Location:
[148, 253]
Pixel Value:
R: 204
G: 154
B: 93

Pixel Scale:
X:100%  Y:100%

Channel: 0

Max: 255

Min: 0

Click & drag or use arrow keys to translate

X:148.5  Y: 253.5

***Figure 10-1: Result of executing batch file program***

6. Once finished viewing the image, close the *IDL iImage* window.
7. Select "*File > Close*" in order to close the text editor window for this batch file.

## Main-Level Programs

**Main-level programs** are entered at the IDL command line, and are useful when you have a few commands you want to run without creating a separate file to contain your commands. Main-level programs are not explicitly named; they consist of a series of statements that are not preceded by a procedure or function heading.

They do, however, require an *END* statement. Since there is no heading, the program cannot be called from other routines and cannot be passed arguments. When IDL encounters a main program as the result of a *.RUN* executive command, it compiles it into the special program named *$MAIN$* and immediately executes it. Afterwards, it can be executed again by using the *.GO* executive command.

The following example creates and executes a small main-level program :

1. At the IDL> command prompt, enter the following :

   ```
   IDL> a = 3
   ```

2. Next, execute the *.RUN* executive command. This changes the command prompt from "IDL>" to "-" :

   ```
   IDL> .RUN
   ```

3. Enter the following 3 statements at the "-" prompt :

   ```
   a = a ^ 2
   PRINT, a
   END
   ```

4. This creates a main-level program, which automatically compiles and executes, resulting in the following output :

   ```
   % Compiled module: $MAIN$.
          9
   ```

5. This main-level program can be run again by executing the *.GO* executive command :

   ```
   IDL> .GO
   ```

6. The result of executing this program a second time is cumulative, resulting in the following output :

   ```
         81
   ```

## Named Programs (Procedures & Functions)

**Named programs** are modules that are given explicit names so they can be called from other programs as well as executed at the IDL command line. Named programs are usually stored in ASCII text files on disk and are given a "*.pro*" extension by convention. There are two different types of named programs in IDL; procedures and functions. The concept of procedures and functions should be familiar since almost all of the statements executed in this tutorial thus far have involved either a procedure or a function (or a combination of both).

Procedures and functions are self-contained modules that break large tasks into smaller, more manageable ones. A **procedure** is a self contained sequence of IDL statements with an unique name that performs a well defined task. A **function** is a self-contained sequence of IDL statements that performs a well-defined task and returns a value to the calling program unit when it is executed. Consequently, all functions must contain a call to the *RETURN* procedure with a specific value (scalar, string, array, structure, etc.) as the argument which is returned to the calling program unit.

Before a procedure or function can be executed, it must be **compiled**. When a system routine (a function or procedure built into IDL, such as *SURFACE*) is called, either from the command line or from another procedure, IDL already knows about this routine and compiles it automatically. When a user-defined function or procedure is called, IDL must compile the program before it can be executed. There are 3 ways in which a procedure or function can be compiled :

- Automatically : If the ASCII text source code file for the program has a filename that is identical to the name of the main program module, the filename ends with a "*.pro*" extension, and this file is found within IDL's path, then the program will be automatically compiled when it is executed. For reference, IDL's path is one of the Preferences found within the IDL Development Environment, and it is also stored in an internal system variable named *!PATH*.
- Interactively : If the source code file for the program is currently open within the text editor of the IDL Development Environment, the user can either select "*Run > Compile*" from the menu system or press the yellow compile button on the toolbar in order to compile the program.
- Manually : The user can explicitly compile a program by executing the *.COMPILE*, *.RUN*, or *.RNEW* executive commands with the appropriate path to the source code file on disk. If the source code file is found within IDL's path or the current working directory, then just the name of the file itself is needed. If the user is working within the IDL Development Environment, the source code will automatically be opened into the text editor window.

In the following exercise, a simple procedure program named "*muscle_view*" will be created, compiled, and executed. This program will open the example data file "*muscle.jpg*" included with the IDL installation, read the image data into IDL, and displays the image in a *FOR* loop that cycles through all of IDL's 41 predefined colortables using Direct Graphics.

1. Start by selecting "*File > New > Editor*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. The first step in writing a procedure is to create the **definition statement**, and this is accomplished using the *PRO* statement in IDL. In this case, the name given to the procedure will be "*muscle_view*", so start by typing the following text in the blank text editor window :

   ```
   PRO muscle_view
   ```

At this point, the name of the current text document is probably [Untitled1*] and it is not being saved to the harddrive, so it may be appropriate to save this text to an IDL source code file on disk.

3. From the main IDLDE menu system select "*File > Save As"*.
4. Save the text to a new file named "*muscle_view.pro*" located in the "*output*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\output\muscle_view.pro*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/output/muscle_view.pro*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/output/muscle_view.pro*

Once this is accomplished, the title bar across the top of the IDL Development Environment should be labeled with "[muscle_view.pro]". Whenever a change is made to the file an asterisk "*" character will be added to the end of the filename, alerting the user that the file contains modifications that have not yet been saved to the file on disk. Changes to the source code file can be saved by selecting "*File > Save"* from the IDLDE menu or by pressing the 💾 button on the IDLDE toolbar.

5. The first step within the program that needs to be entered on the next line of the text editor is the specification of the path to the file that is going to be opened :

```
file = filepath('muscle.jpg', subdir=['examples','data'])
```

6. When writing a program it is always a good idea to include as much error checking as possible. In this case it may be a good idea to make sure that the specified file is a valid JPEG image file and obtain some information about it using the *QUERY_JPEG* function before proceeding :

```
result = query_jpeg(file, info)
```

7. The *QUERY_JPEG* function returns a value of "*1*" into the result if the query was successful and the file appears to be a valid JPEG format image file, otherwise it returns "*0*" upon failure. Consequently, it may be a good idea to terminate the execution of this program at this point using the *RETURN* procedure if the file is not determined to be a valid JPEG image file :

```
if result eq 0 then return
```

8. Now the JPEG image file data can be read into this IDL program unit using the *READ_JPEG* procedure :

```
read_jpeg, file, image
```

9. An IDL graphics window needs to be created to display this image, and the size of this window can be made to match the size of the image using the "*info*" structure returned by the earlier call to *QUERY_JPEG* :

```
window, xsize=info.dimensions[0], ysize=info.dimensions[1]
```

10. Next, the use of color decomposition must be disabled in order to use IDL's built-in colortables on 24-bit displays :

```
device, decomposed=0
```

Now the program is ready to cycle through the colortables and display the image in the Direct Graphics window. In order to accomplish this task, the programmer must make use of the *FOR* control statement. The syntax for declaring a *FOR* loop within the IDL language is :

```
FOR variable = init, limit DO BEGIN
  statements
ENDFOR
```

11. In order to loop through each of IDL's predefined colortables and display the image data insert the following lines of text :

    ```
    for i = 0, 40 do begin
      loadct, i
      tvscl, image
    endfor
    ```

This will cycle through the variable "*i*" set to values 0→40, load the colortable index for the current *FOR* loop iteration value of "*i*" using *LOADCT*, display the image using *TVSCL*, increment the variable "*i*" by a value of 1, and start the next iteration.

12. Once the *FOR* loop has executed, the graphics window that was created can be cleaned up and destroyed by calling the *WDELETE* procedure :

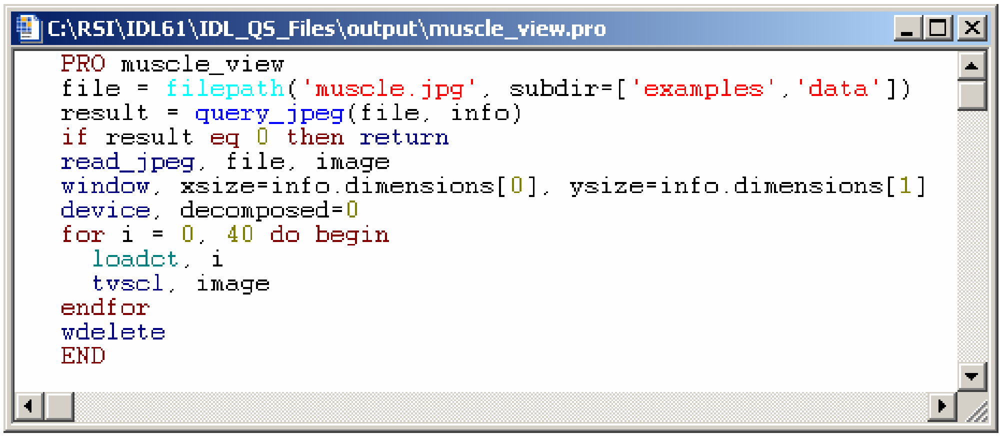
    ```
    wdelete
    ```

13. Finally, the end of the program must be defined by inserting the *END* statement :

    ```
    END
    ```

14. Once all of this text has been entered make sure to select "*File > Save"* or hit the 💾 button on the IDLDE toolbar.

The final program within the text editor window should look similar to Fig. 10-2.



```
PRO muscle_view
file = filepath('muscle.jpg', subdir=['examples','data'])
result = query_jpeg(file, info)
if result eq 0 then return
read_jpeg, file, image
window, xsize=info.dimensions[0], ysize=info.dimensions[1]
device, decomposed=0
for i = 0, 40 do begin
  loadct, i
  tvscl, image
endfor
wdelete
END
```

***Figure 10-2: Completed source code for the "muscle_view" procedure***

The first step in running this IDL program is to compile the procedure so it can be executed within the IDL process.

15. In order to compile the procedure, simply select "*Run > Compile muscle_view.pro"* from the main IDLDE menu system or hit the yellow compile button on the toolbar.  This will automatically call the executive command *.COMPILE* and the output log should read :

```
% Compiled module: MUSCLE_VIEW.
```

If there are any compilation errors an informational message will appear in the output log and a red circular dot will be placed to the left of the line where the error occurs.  Any compilation errors must be resolved before the program can be executed.

16. Once the program is successfully compiled it can be executed by selecting "*Run > Run muscle_view"* from the menu, hitting the blue run button on the toolbar, or simply executing "*muscle_view*" at the IDL> prompt :

```
IDL> muscle_view
```

During execution of the program the "*muscle.jpg*" image will be displayed using all 41 of IDL's built-in colortables [Fig. 10-3].  While the program is executing the *LOADCT* procedure will output the currently loaded colortable to the output log when it's being called within the *FOR* loop.  The total time it takes for this program to execute should be very short, which is a testament to the rapid image display capabilities of the Direct Graphics system and the overall speed of the IDL language.

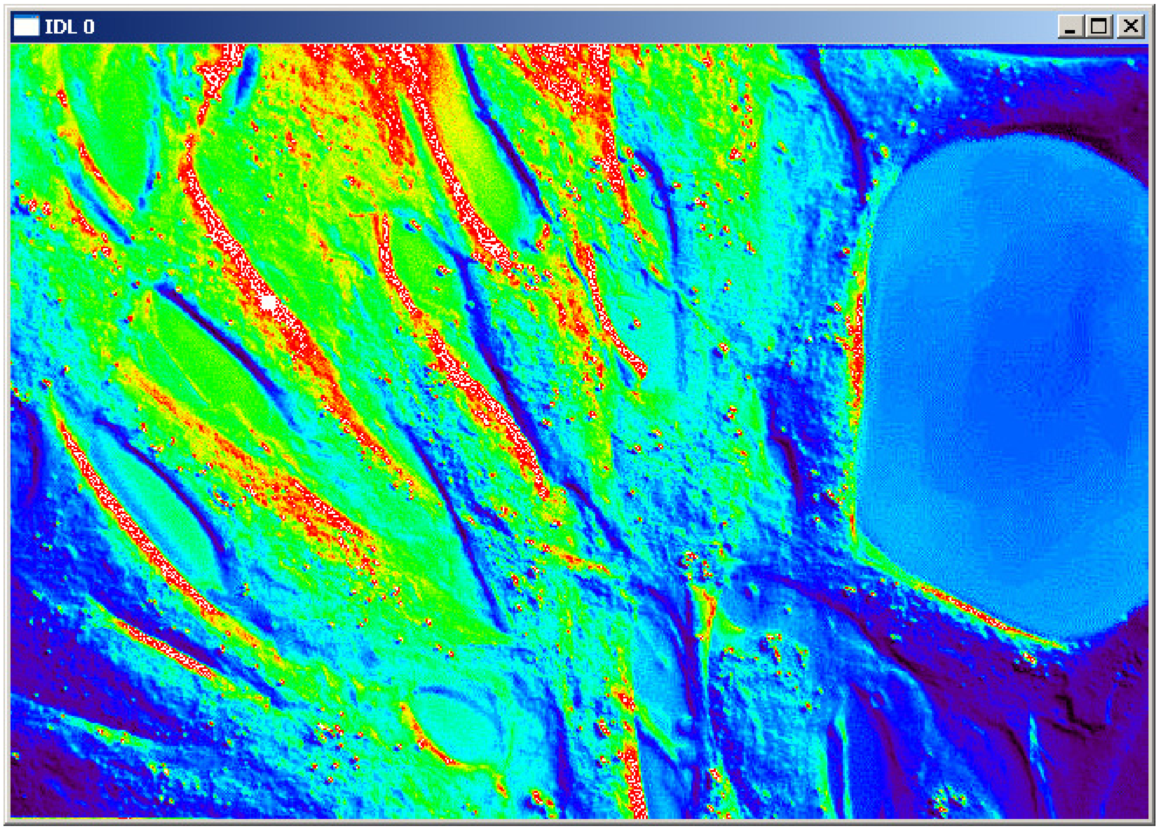


***Figure 10-3: Image display during execution of the "muscle_view" program***

17. Select "*File > Close*" in order to close the text editor window for this procedure.

In the next exercise a simple function will be created that computes the area of a circle given its radius. Functions are slightly different from procedures because they must return a specific value to the calling module. Consequently, functions are particularly useful when performing data processing and analysis. In addition, as previous exercises within this tutorial have illustrated, the calling syntax for executing functions is different than procedures.

18. Start by selecting "*File > New > Editor*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.

Once again, the first step in writing a function is to create the **definition statement**, and this is accomplished using the *FUNCTION* statement in IDL. In this case, the function will need to accept an **argument** in order to allow passing of the input radius value into the program. Furthermore, the user may wish to specify whether they want the calculation performed using single-precision or double-precision floating-point arithmetic. This can be accomplished by specifying a **keyword** that the user can utilize in order to control the behavior of the function.

19. The name given to the function will be "*circle_area*", the input argument should be named "*radius*", and a keyword named "*dbl*" can be declared so the

user has control over the precision. This can be accomplished with the following definition statement :

```
FUNCTION circle_area, radius, dbl=dbl
```

Once again, it is a good idea to save this text to an IDL source code file on disk :

20. From the main IDLDE menu system select "*File > Save As*".
21. Save the text to a new file named "*circle_area.pro*" located in the "*output*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\output\circle_area.pro*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/output/circle_area.pro*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/output/circle_area.pro*

22. The *KEYWORD_SET* function built into the IDL language library is useful for determining the status of a keyword variable and whether or not it was "*set*" by the user. In this case, a variable named "*dbl*" exists within the function and either has a value of *1* if it was set, or *0* if it was left un-set. The *KEYWORD_SET* function can be used in conjunction with an IF … THEN … ELSE code block to perform the necessary calculation using the appropriate IDL system variable for the value of ∏ :

```
if keyword_set(dbl) then begin
  area = !DPI * radius ^ 2
endif else begin
  area = !PI * radius ^ 2
endelse
```

23. Once the area has been computed using the appropriate precision, the "*area*" variable can be returned to the calling program module :

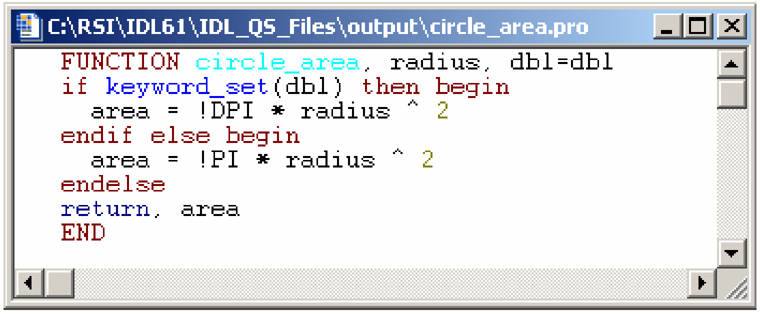
```
return, area
```

24. Finally, the end of the program must be defined by inserting the *END* statement :

```
END
```

25. Once all of this text has been entered make sure to select "*File > Save*" or hit the 💾 button on the IDLDE toolbar.

The final program within the text editor window should look similar to Fig. 10-4.



```
FUNCTION circle_area, radius, dbl=dbl
if keyword_set(dbl) then begin
  area = !DPI * radius ^ 2
endif else begin
  area = !PI * radius ^ 2
endelse
return, area
END
```

***Figure 10-4: Completed source code for the "circle_area" function***

Before the "*circle_area*" function can be executed it must be compiled.

26. In order to compile the function, simply select "*Run > Compile circle_area.pro"* from the main IDLDE menu system or hit the yellow compile button on the toolbar. This will automatically call the executive command *.COMPILE* and the output log should read :

```
% Compiled module: CIRCLE_AREA.
```

If there are any compilation errors an informational message will appear in the output log and a red circular dot will be placed to the left of the line where the error occurs. Any compilation errors must be resolved before the program can be executed.

Once the program is successfully compiled it can be executed by calling it with the appropriate function syntax :

27. IDL> `area = CIRCLE_AREA (7)`

This statement calls the "*circle_area*" function and specifies an input radius value of seven. The named variable "*area*" contains the result of this calculation that is returned from the function :

28. IDL> `HELP, area`

```
AREA            FLOAT     =       153.938
```

In addition, the "*circle_area*" function can be called with the "dbl" keyword set in order to compute the area in double-precision :

29. IDL> `area = CIRCLE_AREA (7, /DBL)`
30. IDL> `HELP, area`

```
AREA            DOUBLE    =        153.93804
```

The "*area*" variable that is returned from the function is now double-precision.

31. Once finished experimenting with the "*circle_area*" function, select "*File > Close*" in order to close the text editor window.

## Object-Oriented Programs

Traditional programming techniques make a strong distinction between routines written in the programming language (procedures and functions in the case of IDL) and the data to be acted upon by these routines. In contrast, object-oriented programming removes this distinction by encapsulating both data and functionality (i.e. routines) into a single entity known as **objects**.

IDL objects may contain data with various data types or organizational arrangement. The routines (i.e. functionality) within an object that act upon this data are called **methods**. Object methods are merely IDL procedures and functions that have special names and are called in a special way. In the IDL programming environment, object data are protected from the rest of the program and are only accessible through the object methods (i.e. IDL object data is always private).

IDL's object system provides support for the following concepts and mechanisms:

- **Classes and Instances** : IDL objects are created as instances of a class, which is defined in the form of an IDL structure. The name of the structure is also the class name for the object. Objects can only be created by calling the *OBJ_NEW* (or *OBJARR*) function with the name of the class structure as the argument, and can only be accessed via the returned object reference.
- **Encapsulation** : Encapsulation is the ability to combine data and the routines that affect the data (known as methods) into a single object.
- **Methods** : Methods are the routines (procedures and functions) that perform specific operations within the object. Method routines are identified as belonging to an object class via a routine naming convention *ClassName::MethodName*.
- **Polymorphism** : Polymorphism is the ability to create multiple object types that support the same operations.
- **Inheritance** : Inheritance is the ability of an object class to inherit the behavior of other object classes. This means that when writing a new object class that is very much like an existing object class, you need only program the functions that are different from those in the inherited class.
- **Persistence** : Persistence is the ability of objects to remain in existence in memory after they have been created, allowing you to alter their behavior or appearance after their creation. IDL objects persist until you explicitly destroy them, or until the end of the IDL session.
- **Object Lifecycle** : The life of an object can be broken into three phases: creation, use, and destruction. Object references are created using one of two lifecycle routines: *OBJ_NEW* or *OBJARR*. The object is used by executing its methods. Finally, the object is destroyed using the *OBJ_DESTROY* procedure.

Objects are implemented in IDL as extensions of data structures. In other words, a data structure and one or more IDL routines are combined to form an IDL object **class**. Object classes are essentially the blueprints used to define individual objects. There are 2 basic steps to creating an object class in IDL :

- Define the object class data structure by creating a named structure. The name of this structure must be the same as the desired name for the object class. For example, the following would be an appropriate structure definition for an object class named *SHOW3OBJ* :

```
namedStructure = {SHOW3OBJ, data:ptr_new()}
```

- Define the methods of the object class using the *CLASSNAME::method* declaration syntax. For example, a method named "*Display*" could be defined for the *SHOW3OBJ* class using the following syntax :

```
PRO SHOW3OBJ::Display
if ptr_valid(self.data) then show3, *self.data
END
```

Once the object class has been defined an instance of the object can be created using the *OBJ_NEW* function :

```
oShow3 = OBJ_NEW ('SHOW3OBJ', dist(100))
```

The *Display* method to this object can then be executed using IDL's method invocation operator ("->"), which is a hyphen followed by a greater-than sign :

```
oShow3 –> display
```

In this example the *Display* method is a procedure, but methods can also be functions. If an object method is defined as a function then its calling sequence will be :

```
result = object -> functionMethod ()
```

Once an object has data assigned to it, the methods automatically have access to this data in a structure called "*self*" that has the same layout as the original object class data structure. Consequently, there is no need to pass the data via arguments and/or keywords in the call to an object method. In addition, the *self* structure can be used to call other methods within an object. For instance, if there was a *LOADCOLOR* method to the *SHOW3OBJ* object class it could directly call the *DISPLAY* method using the *self* structure :

```
PRO SHOW3OBJ::Loadcolor, index
if n_params() NE 1 then return
if !D.N_COLORS GT 256 then device, decomposed=0
loadct, index
self –> display
END
```

When an instance of an object is created from a class using the *OBJ_NEW* function an **object reference** is returned that points to an object **heap variable** [Fig. 10-5]. A heap variable is an area of common memory allocated for a specific use and accessed by way of one or more reference variables. These reference variables are the means through which the object is referenced in IDL.

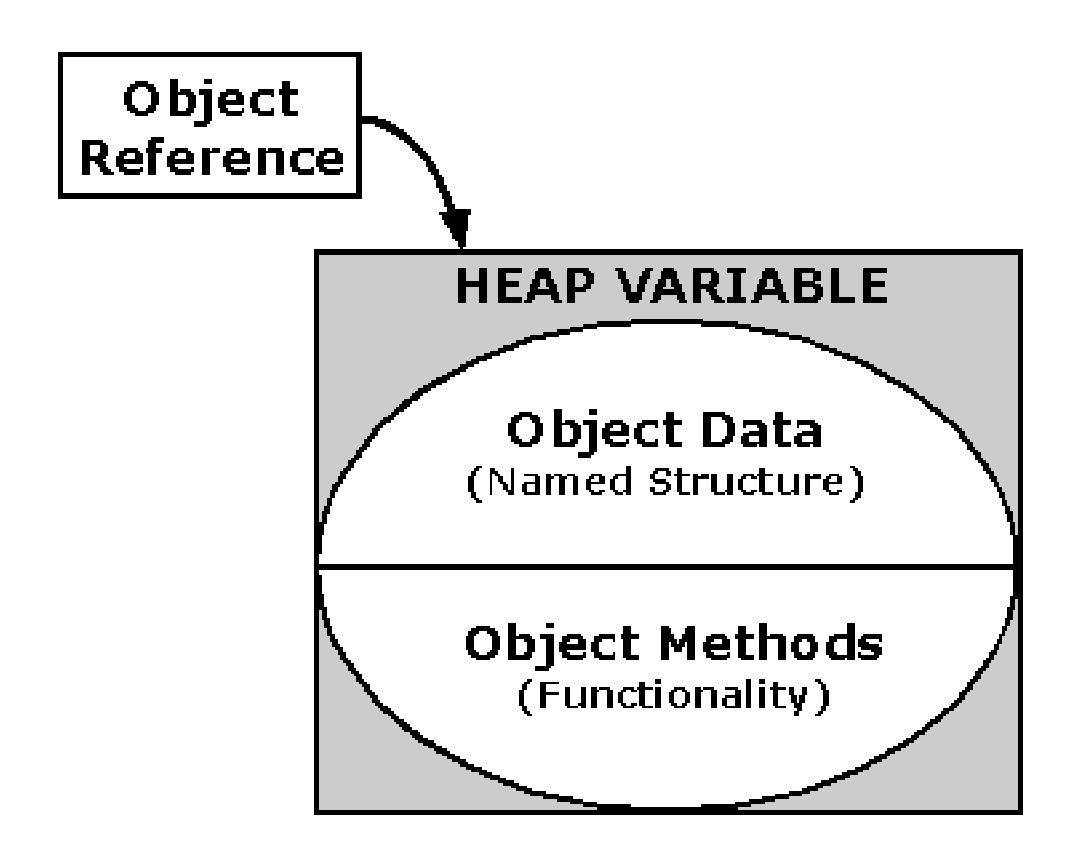


***Figure 10-5: An object heap variable with an object reference. The heap variable encapsulates both the data and methods for the object.***

Consequently, in the hypothetical example above, the variable *oShow3* is not actually the object itself but a reference to the object contained in heap memory. Consequently, the programmer must take care when deleting objects since the destruction of the object reference variable will not cleanup the heap memory occupied by the object. Object heap variables persist in memory until explicitly destroyed by the *OBJ_DESTROY* procedure:

```
OBJ_DESTROY, oShow3
```

Once the *OBJ_DESTROY* procedure is called the object heap variable for this object no longer exists, but the object reference variable *oShow3* still exists and points to a non-existent piece of memory. This is known as a **dangling reference**. Consequently, it is always a good idea to also destroy all object reference variables when destroying an object by either using the *DELVAR* procedure (interactive IDL only) or the *TEMPORARY* function:

```
OBJ_DESTROY, TEMPORARY(oShow3)
```

In order to create an object class in IDL there are a set of programming rules that must be followed when writing the source code. Selecting a name for a custom object class is a very important consideration because it is important not to use the same name as a class already built into IDL. Fortunately, most object classes built into IDL have names that start with the string "*IDL*", so it should be easy to derive a name that does not conflict with one of the built-in classes.

When creating a custom object class in IDL a special naming convention must be used for the source code file name and the procedure/function names within that source code file. As mentioned before, the general form for the procedure/function declaration statements within the object class source code is "*CLASSNAME::method*". In addition, there is also a naming convention for the source code file itself that follows the general form "*CLASSNAME__method*" (Note: "__" is two underscores).

The information necessary for the object must be defined in the object class data. When defining the data of an object, the programmer must determine in advance all of the various data elements that might be necessary to complete all of the functionality for the object. As mentioned earlier, the object class data is defined and stored in an IDL named structure that has the same name as the class itself. This is the first step in creating an object class and it occurs within a special procedural method called "*DEFINE*". Consequently, in order for IDL to create an instance of an object with *OBJ_NEW* an IDL source code file for this object with the name "*CLASSNAME__define.pro*" must be found in the *!PATH* (or manually compiled within the current session of IDL). The define method is also the only method which uses the two underscores ("__") in the procedural declaration statement as opposed to the two colons ("::"). For example, the source code to define the *SHOW3OBJ* object class above would look like:

```
PRO SHOW3OBJ__define
namedStructure={SHOW3OBJ, data:ptr_new()}
END
```

This source code needs to be saved in an ASCII text file called "*show3obj__define.pro*" in order for the object class to be used. The only purpose of the *__define* procedure is to create the named structure that will contain the data for the object class.

Once the data has been defined for an object class, the next step is to create the methods that will define the functionality. These methods can be stored in separate "*CLASSNAME__method.pro*" ASCII source code files, or they can all be stored in the main "*CLASSNAME__define.pro*" file as long as the *__define* procedure is the last module that occurs within the file.

There are also 2 other methods that must be included with every object class: *INIT* and *CLEANUP*. The 3 main object methods (*DEFINE, INIT, CLEANUP*) must be defined and compiled within IDL before an instance of the object can be created with *OBJ_NEW*. The *INIT* method is an IDL function that is called by *OBJ_NEW* when the object is created. The job of the *INIT* method is to take any arguments and keywords passed from the user and perform any initialization necessary for the object class. The process of initialization usually involves filling-in the members of the object data structure with the data passed by the user in the call to *OBJ_NEW*. Finally, the *INIT* method should return *1* if successful in initializing the object and *0* if the initialization failed. In contrast, the *CLEANUP* method is called when the object reference is destroyed (when *OBJ_DESTROY* is executed)) and should clean up any pointers, objects, or other miscellaneous data in an effort to avoid memory leakage.

The following exercise completes the definition of the *SHOW3OBJ* object class that has been discussed herein by writing its IDL source code.

1. Start by selecting "*File > New > Editor*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. Immediately select "*File > Save As"* and save this to a file called "*show3obj__define.pro*" located in the "*output*" subfolder of the Quick Start directory :

- **Windows:**
  *C:\RSI\IDL##\IDL_QS_Files\output\show3obj__define.pro*
- **UNIX et al.:**
  */usr/local/rsi/idl_#.#/IDL_QS_Files/output/show3obj__define.pro*
- **Mac OS X:**
  */Applications/rsi/idl_#.#/IDL_QS_Files/output/show3obj__define.pro*

3. Type out the source code for this object class in the IDLDE text editor. There will be a total of 5 methods defined for the *SHOW3OBJ* object class: *DEFINE*, *INIT*, *CLEANUP*, *LOADCOLOR*, and *DISPLAY*. Make sure to remember the rule of placing the *__define* procedure at the bottom of this file :

```
FUNCTION SHOW3OBJ::Init, data
self.data = ptr_new(data)
if !d.n_colors gt 256 then device, decomposed=0
loadct, 0, /silent
return, 1
END

PRO SHOW3OBJ::Cleanup
ptr_free, self.data
print, 'SHOW3OBJ object successfully destroyed.'
END

PRO SHOW3OBJ::Display
if ptr_valid(self.data) then show3, *self.data
END

PRO SHOW3OBJ::Loadcolor, index
loadct, index
self -> display
END

PRO SHOW3OBJ__define
namedStructure = {SHOW3OBJ, data:ptr_new()}
END
```

4. Once all of this source code has been entered into the text editor, save the file and then compile the object class by pressing the yellow compile button. This should report the following in the IDL Output Log :

```
IDL> .COMPILE show3obj__define.pro
% Compiled module: SHOW3OBJ::INIT.
% Compiled module: SHOW3OBJ::CLEANUP.
% Compiled module: SHOW3OBJ::DISPLAY.
% Compiled module: SHOW3OBJ::LOADCOLOR.
% Compiled module: SHOW3OBJ__DEFINE.
```

Now that the object class is compiled within the current session of IDL it is ready to be used. Initialize an instance of the *SHOW3OBJ* object class with data created by the *DIST* function :

5. IDL> `oShow3 = OBJ_NEW ('SHOW3OBJ', DIST (100) )`

Next, call the *Display* method for the object :

6. IDL> `oShow3 -> Display`

The resulting *IDL 0* visualization window should contain a composite 3-D visualization including an image, wire-mesh surface, and contour plot produced by IDL's built-in *SHOW3* routine.

The *Loadcolor* method can also be executed to load IDL's Rainbow colortable and automatically redisplay the graphic:

7. IDL> `oShow3 -> Loadcolor, 13`

The resulting visualization window should look similar to Fig. 10-6 :

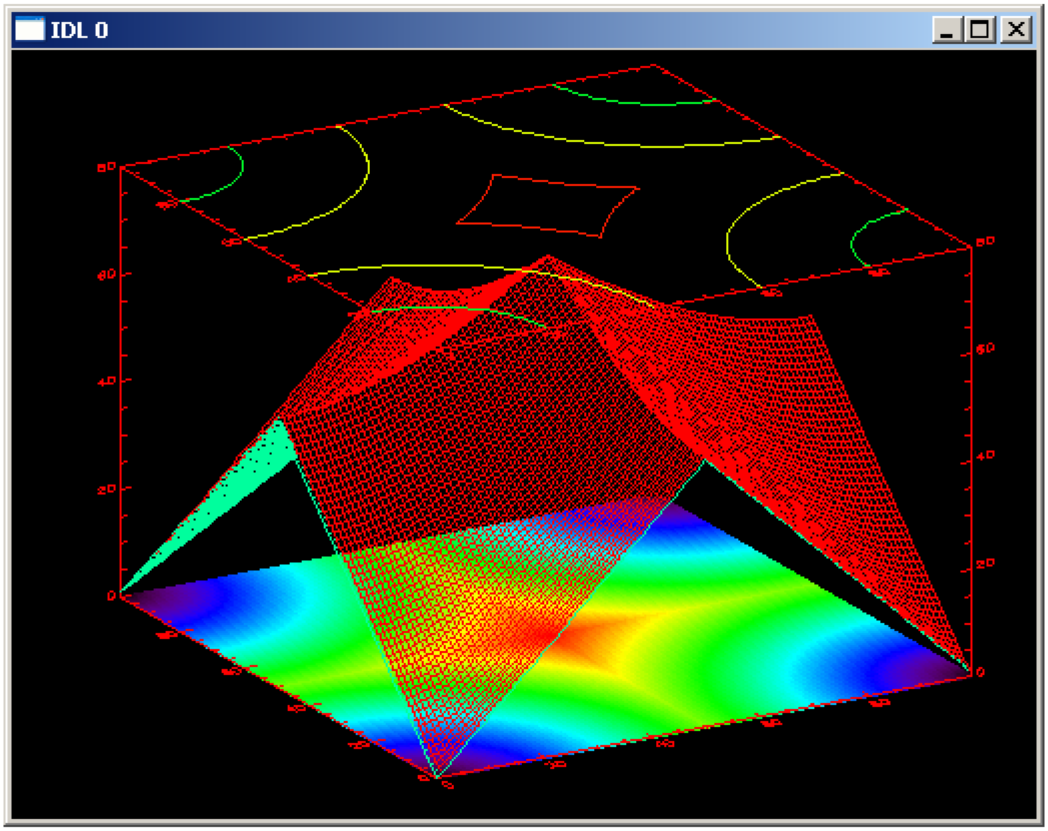


***Figure 10-6: Result of invoking the "Loadcolor" method of the "SHOW3OBJ" object class***

Once the user is finished working with the object it is important to destroy the heap variable and object reference:

8. IDL> `OBJ_DESTROY, TEMPORARY (oShow3)`
   ```
   SHOW3OBJ object successfully destroyed.
   ```
9. Once finished experimenting with the "*SHOW3OBJ*" object, select "*File > Close*" in order to close the text editor window.

## iTools System Programs

**iTools programming** is a special form of object-oriented programming that is used to control, customize, and extend the tools built into the iTools system. The IDL Intelligent Tools, or *iTools*, are applications written in IDL to perform a variety of data analysis and visualization tasks. iTools share a common underlying application framework, presenting a full-featured, customizable, application-like user interface with menus, toolbars, and other graphical features.

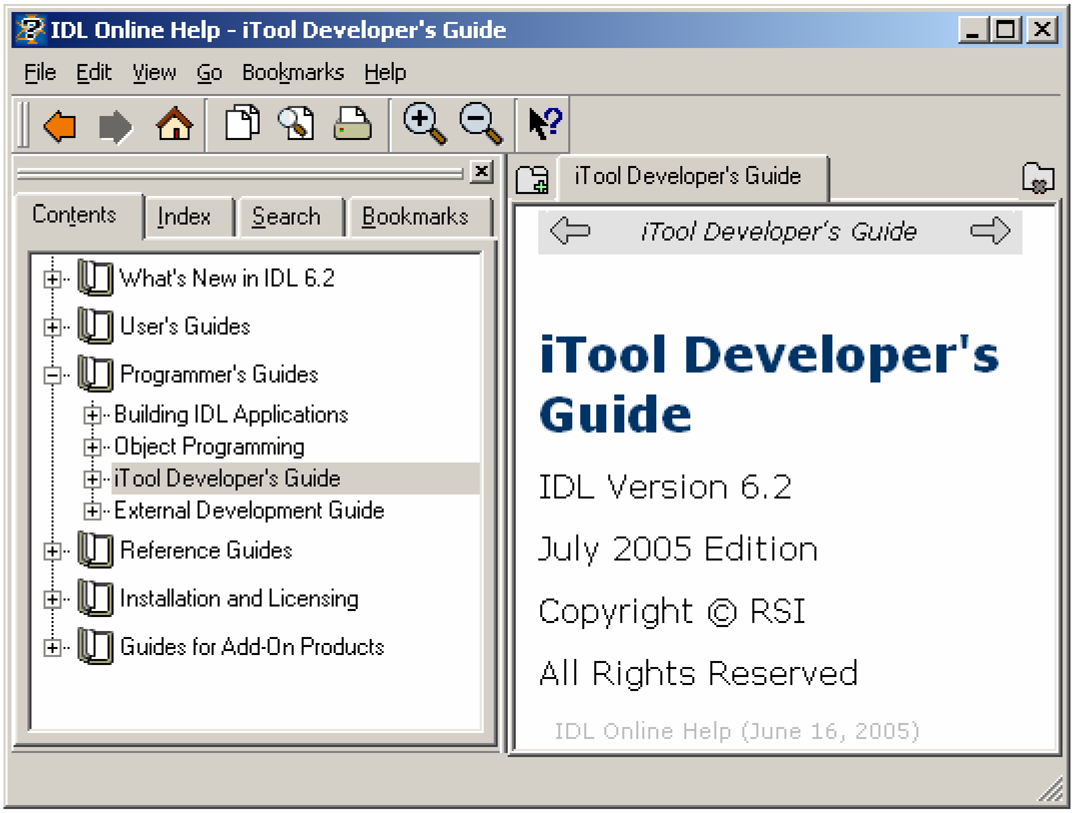
However, iTools are much more than just a set of pre-written IDL programs. Behind the iTool system lies the IDL Intelligent Tools Component Framework — a set of object class files and associated utilities designed to allow you to easily extend the supplied toolset or create entirely new tools of your own. The iTools component framework is a set of object class definitions written in the IDL language. It is designed to facilitate the development of sophisticated visualization tools by providing a set of pre-built components that provide standard features including :

- Creation of visualization graphics
- Mouse manipulations of visualization graphics
- Annotations
- Management of visualization and application properties
- Undo/redo capabilities
- Data import and export
- Printing
- Data filtering and manipulation
- Interface element event handling

In addition, the iTools component framework makes it easy to extend the system with components of your own creation, allowing you to design a tool to manipulate and display your data in any way you choose. Programming in the iTools system allows the user to create their own :

- iTool
- Visualization
- Operation
- Manipulator
- File Reader
- File Writer
- Graphical User Interface

A discussion of programming within the iTools system is beyond the scope of this tutorial. For more information please consult the *iTool Developer's Guide* documentation manual included with the IDL online help system [Fig. 10-7].



iTool Developer's Guide

IDL Version 6.2

July 2005 Edition

Copyright © RSI

All Rights Reserved

IDL Online Help (June 16, 2005)

***Figure 10-7: The iTool Developer's Guide documentation manual***

## Creating Graphical User Interfaces

IDL allows you to construct and manipulate graphical user interfaces using **widgets**. Widgets (or *controls*, in the terminology of some development environments) are simple graphical objects such as pushbuttons or sliders that allow user interaction via a pointing device (usually a mouse) and a keyboard. This style of graphical user interaction offers many significant advantages over traditional command-line based systems.

IDL widgets are significantly easier to use than other alternatives, such as writing a C language program using the native window system graphical interface toolkit directly. IDL handles much of the low-level work involved in using such toolkits. The interpretive nature of IDL makes it easy to prototype potential user interfaces. In addition to the user interface, the author of a program written in a traditional compiled language also must implement any computational and graphical code required by the program. IDL widget programs can draw on the full computational and graphical abilities of IDL to supply these components.

The style of widgets IDL creates depends on the windowing system supported by your host computer. Unix hosts use Motif widgets, while Microsoft Windows systems use the native Windows toolkit. Although the different toolkits produce applications with a slightly different look and feel, most properly-written widget applications work on all systems without change.

IDL graphical user interfaces are constructed by combining widgets in a treelike **hierarchy**. Each widget has one parent widget and zero or more child widgets. There is one exception: the topmost widget in the hierarchy (called a **top-level base**) is always a base widget and has no parent.

Programs that use widgets are **event driven**. In an event driven system, the program creates an interface and then waits for messages (events) to be sent to it from the window system. Events are generated in response to user manipulation, such as pressing a button or moving a slider. The program responds to events by carrying out the action or computation specified by the programmer, and then waiting for the next event. Because of widget applications' event-driven nature, creating applications that use widgets is fundamentally different from creating non-widget programs.

The following exercise creates a very simple graphical user interface application with the appropriate event handling sub-program. This program displays a simple GUI with a button labeled "*Display Image*" that can be pressed by the user in order to generate an event that display an image from an example file included with IDL.

1. Start by selecting "*File > New > Editor*" from the main IDL Development Environment menu system. This will open a new blank text editor window in the document panel of the IDLDE.
2. Immediately select "*File > Save As"* from the menu system and save this document to a file called "*simple_gui.pro*" located in the "*output*" subfolder of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\output\simple_gui.pro*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/output/simple_gui.pro*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/output/simple_gui.pro*

3. Type out the source code for this widget program into the IDLDE text editor :

```
PRO simple_gui_event, event
widget_control, event.top, get_uvalue=state
tv, state.image, true=1
END

PRO simple_gui
tlb = widget_base(/row, title='Simple GUI')
  subBase=widget_base(tlb)
    button = widget_button(subBase, value='Display Image')
  draw = widget_draw(tlb, xsize=512, ysize=512)
widget_control, tlb, /realize
file = filepath('elev_t.jpg', subdir=['examples', 'data'])
read_jpeg, file, image
device, decomposed=1
state = {image:image}
widget_control, tlb, set_uvalue=state
xmanager, 'simple_gui', tlb
END
```

This program contains 2 modules; the main GUI creation procedure named "*simple_gui*", and the event handler procedure called "*simple_gui_event*". Within the “*simple_gui*” procedure the layout of the widget hierarchy is defined, the GUI is realized (i.e. displayed) to the screen, the image is read-in from a JPEG file on disk and stored in a state structure, and the *XMANAGER* routine is called in order to manage the event callback distribution. The “*simple_gui_event*” procedure is called when the user presses the “*Display Image*” button, and it obtains the state structure for the application and displays the image using the *TV* routine.

4. Once the source code for the program has been entered, save the changes by pressing the 💾 button on the IDLDE toolbar.
5. Compile the program by pressing the yellow compile button.
6. Finally, run the program by pressing the blue run button.
7. Once the GUI for the program is displayed, use the mouse to press the “*Display Image*” button [Fig. 10-8].

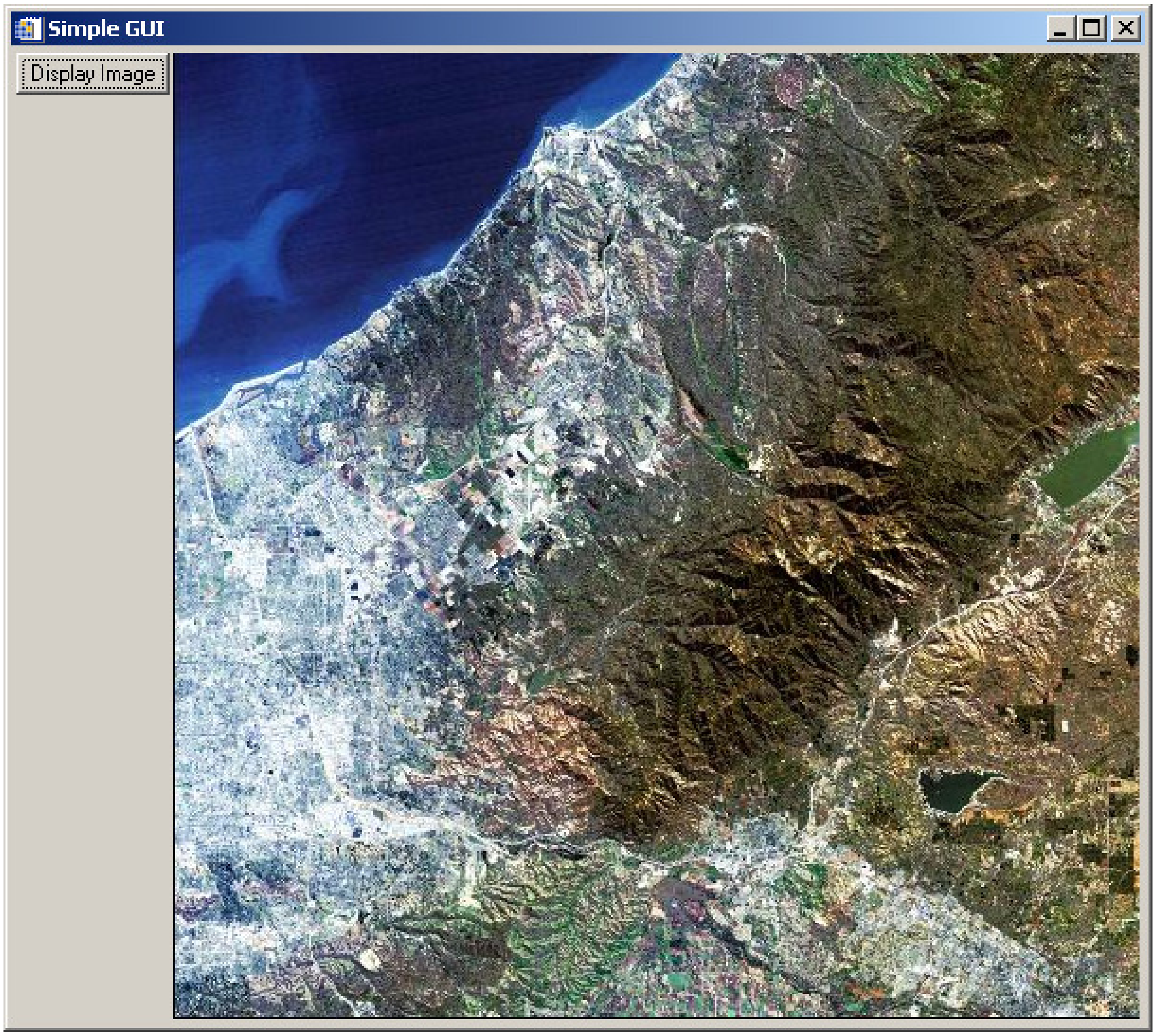


***Figure 10-8: Execution of the “simple_gui” widget program***

8. Once finished viewing the simple widget application, close the *Simple GUI* window.

## Distributing IDL Programs

Once an application has been written in IDL, the user may wish to distribute this program to friends, colleagues, or their own customers. IDL programs can be stored and distributed as source code in ASCII text files with a "*.pro*" extension, or they can be compiled within a development copy of IDL and saved to runtime binary files with a "*.sav*" extension. Creating a runtime binary save file for an application allows the programmer to create a distributable version of their application that does not contain the original source code, thereby protecting the intellectual property rights of the developer. Furthermore, the runtime binary save file version of a program can be executed in the runtime environment of IDL, whereas a source code file requires that the end-user have a full development copy of the IDL software package in order to compile the program before execution. Creation of a runtime binary save file version of a program involves using either the *Projects* built into the IDL Development Environment, or the *.COMPILE*, *RESOLVE_ALL*, and *SAVE* routines built into the IDL language library.

IDL has an architectural (and distribution) paradigm that is very analogous to Java since they are both interpreted computer languages. There are basically 3 different varieties of the IDL software package that can be used to execute programs written in the IDL language :

- IDL (full developer's copy)
- IDL Runtime
- IDL Virtual Machine

The primary difference between these varieties of IDL is that the full developer's copy allows the user to create, edit, modify, compile, execute, and save IDL programs. In contrast, both the IDL Runtime and IDL Virtual Machine versions of IDL can only be used to execute pre-compiled runtime binary versions (**.sav* file) of a program. Consequently, users of the IDL Runtime or the IDL Virtual Machine cannot modify the IDL programs that are being executed.

In the following exercise, the "*muscle_view*" program that was created earlier will be compiled and saved out to a runtime binary file on disk that can be executed in either IDL Runtime or the IDL Virtual Machine. This can be accomplished with the *SAVE* procedure, which is used to save either data variables or the compiled routines within the current software session to IDL's proprietary binary save file format. The IDL save file format is encrypted (and is not documented) so it is impossible to reverse-engineer a program stored in a "**.sav*" file and obtain the original source code. Calling the *SAVE* procedure with the *ROUTINES* keyword set will save all currently compiled user-defined procedures and functions within the current IDL session. Consequently, it is a good idea to reset the IDL session and start fresh so that none of the other programs that have been compiled thus far are included in the save file that is created :

1. IDL> `.reset_session`

Next, compile the "*muscle_view*" program that was created in the earlier exercise by utilizing the *.COMPILE* executive command :

2. IDL> `.COMPILE muscle_view`

   ```
   % Compiled module: MUSCLE_VIEW.
   ```

The Runtime and Virtual Machine versions of the software are basically the IDL interpreter provided in a series of library files. This interpreter includes all of the internal system routines within the IDL language. However, it does not include the routines within the IDL library that are written in IDL itself and are distributed as source code **.pro* files in the "*lib*" subfolder of a developer's copy installation. For example, the *FILEPATH* function from the IDL library that this "*muscle_view*" program utilizes is actually distribute with the software in the following source code file :

- **Windows:** *C:\RSI\IDL##\lib\filepath.pro*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/lib/filepath.pro*
- **Mac OS X:** */Applications/rsi/idl_#.#/lib/filepath.pro*

When the "*muscle_view*" program is executed within a full developer's copy of IDL, the "*filepath.pro*" source code file is automatically located and compiled on-the-fly because the "*lib*" subfolder of the installation is part of the default path. Since the Runtime and Virtual Machine versions of IDL cannot compile programs from their ASCII text source code, these routines cannot be distributed in their original **.pro* file form. Consequently, the programmer must be sure to resolve and compile all of the external IDL routines that their program utilizes that are not internal system routines built into the interpreter libraries. Fortunately, IDL contains a convenient procedure that can be used by the IDL programmer in order to resolve all of these dependencies named *RESOLVE_ALL* :

3. IDL> `RESOLVE_ALL`

   ```
   % Compiled module: RESOLVE_ALL.
   % Compiled module: LOADCT.
   % Compiled module: FILEPATH.
   % Compiled module: PATH_SEP.
   % Compiled module: UNIQ.
   ```

Notice that the *RESOLVE_ALL* procedure will locate and compile not only the *LOADCT* and *FILEPATH* routines from the IDL library that this program explicitly calls, but will also resolve any subsequent routines that these programs happen to call that are not already compiled (in this case, *PATH_SEP* and *UNIQ*). Under initial inspection, it may not be obvious to the programmer that this "*muscle_view*" program relies on the *PATH_SEP* and *UNIQ* routines from the external IDL library because they do not explicitly appear within the program, and this is the benefit of utilizing the *RESOLVE_ALL* procedure when creating distributable applications.

Finally, the *SAVE* procedure can be called with the *ROUTINES* keyword set in order to save all of the currently compile procedures and functions to an IDL runtime binary save file. Use the *FILE* keyword to specify the appropriate output filename, which is the name of the primary program module followed by a *.sav* extension :

4. IDL> `SAVE, /ROUTINES, FILE='muscle_view.sav'`

This operation will create a new file named "*muscle_view.sav*" within the current working directory. The current working directory should be the "*output*" subfolder of the of the Quick Start directory :

- **Windows:** *C:\RSI\IDL##\IDL_QS_Files\output\muscle_view.sav*
- **UNIX et al.:** */usr/local/rsi/idl_#.#/IDL_QS_Files/output/muscle_view.sav*
- **Mac OS X:** */Applications/rsi/idl_#.#/IDL_QS_Files/output/muscle_view.sav*

5. Navigate to this "*output*" folder and attempt to locate the file named "*muscle_view.sav*". If this file is not found within the "*output*" folder, then execute the following IDL statements in order to determine the current working directory and locate the "*muscle_view.sav*" file :

```
IDL> CD, CURRENT=current
IDL> PRINT, current
```

6. Once the "*muscle_view.sav*" file has been located, it can be executed within IDL Runtime using the appropriate execution methodology based on operating system :

- **Windows:** Simply double-click on the "*muscle_view.sav*" file.
- **UNIX, Linux, & Mac OS X:** At a shell or X11 terminal prompt, navigate to the folder that contains the "*muscle_view.sav*" file and execute the following command :

```
idl –rt=muscle_view.sav
```

Once this is accomplished a new IDL process will launch, the "*muscle_view*" program will be executed, and the IDL process will shut-down.

7. Finally, the same program can be executed within the IDL Virtual Machine by performing the following steps :

- **Windows:** Select "*Start > Programs > RSI IDL #.# > IDL Virtual Machine*".
- **UNIX, Linux, & Mac OS X:** At a shell or X11 terminal prompt, navigate to the folder that contains the "*muscle_view.sav*" file and execute the following command :

```
idl –vm=muscle_view.sav
```

8. The IDL Virtual Machine splash screen will appear, and the user must click on this splash screen in order to continue.
9. On Windows, the user will be prompted with a dialog to select the "*muscle_view.sav*" program file on the harddrive.

Once this is accomplished a new IDL process will launch, the "*muscle_view*" program will be executed, and the IDL process will shut-down.

A discussion on the creation of distributable applications, utilization of the Projects built into the IDL Development Environment, and the exact differences between IDL Runtime and IDL Virtual Machine is beyond the scope of this tutorial. For more information please consult the *Building IDL Applications* documentation manual.

# Appendix A:  IDL Code Tuning

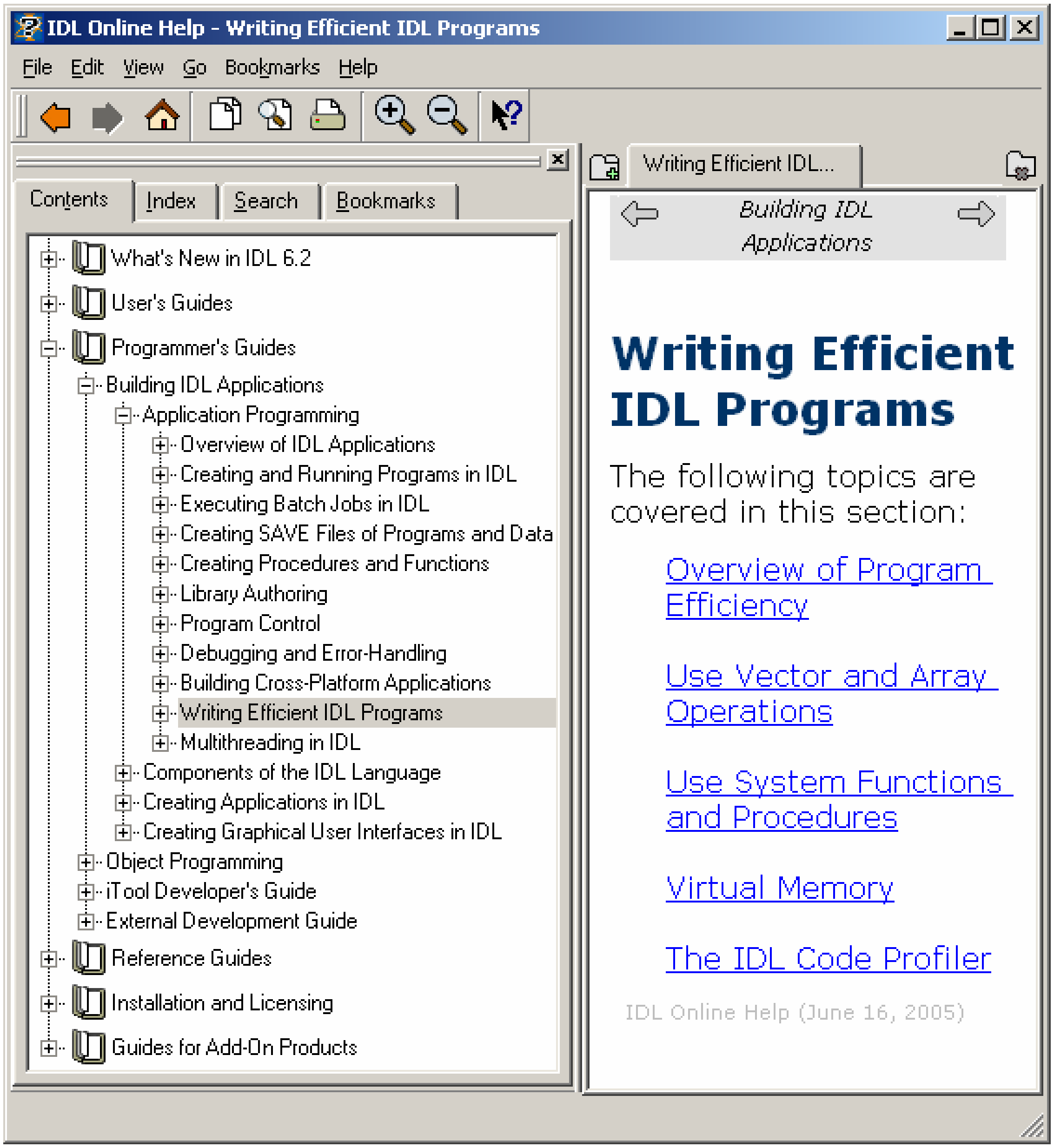
## Writing Efficient IDL Programs

Knowledge of IDL's internal design and implementation, along with careful memory management, can be exploited to greatly improve the efficiency of IDL programs.  In IDL, complicated computations can be specified at a high level.  Therefore, inefficient IDL programs can suffer severe speed penalties — perhaps much more so than with compiled programming languages.

Techniques for writing efficient programs in IDL are identical to those in other computer languages, with the addition of the following simple guidelines :

- Use vector and array operations rather than loops wherever possible.
- Try to avoid loops with high repetition counts.
- Use IDL system functions and procedures wherever possible.
- Access array data in machine address order (IDL is row-major).
- Pay attention to expression evaluation order.
- Avoid *IF … THEN … ELSE* code block statements if possible, especially within loops.
- Use only the highest precision (variable data types) necessary during computations.
- Eliminate invariant expressions.
- Make use of the *TEMPORARY* function.
- Utilize the IDL Code Profiler.

Attention also must be given to algorithm complexity and efficiency, as this is usually the greatest determinant of resources used.  For a more detailed discussion on the subject of writing efficient IDL programs (along with some code examples), please consult the *Building IDL Applications* documentation manual included with the IDL online help system [Fig. A-1].


IDL Online Help - Writing Efficient IDL Programs
File Edit View Go Bookmarks Help
Contents
Index
Search
Bookmarks
Writing Efficient IDL...

- What's New in IDL 6.2
- User's Guides
- Programmer's Guides
  - Building IDL Applications
    - Application Programming
      - Overview of IDL Applications
      - Creating and Running Programs in IDL
      - Executing Batch Jobs in IDL
      - Creating SAVE Files of Programs and Data
      - Creating Procedures and Functions
      - Library Authoring
      - Program Control
      - Debugging and Error-Handling
      - Building Cross-Platform Applications
      - Writing Efficient IDL Programs
      - Multithreading in IDL
    - Components of the IDL Language
    - Creating Applications in IDL
    - Creating Graphical User Interfaces in IDL
  - Object Programming
  - iTool Developer's Guide
  - External Development Guide
- Reference Guides
- Installation and Licensing
- Guides for Add-On Products

*Building IDL Applications*

Writing Efficient IDL Programs

The following topics are covered in this section:

- Overview of Program Efficiency
- Use Vector and Array Operations
- Use System Functions and Procedures
- Virtual Memory
- The IDL Code Profiler

IDL Online Help (June 16, 2005)

***Figure A-1: The discussion on Writing Efficient IDL Programs***

# Appendix B:  External Development

## IDL Calling External Software

There are a number of different approaches used to link IDL to other software. Some of these methods are simple, while others are quite complex.  Some methods work on all platforms supported by IDL, while others depend on operating services and are not supported on all platforms (e.g. ActiveX is a Windows only technology). Moreover, sometimes a method is implemented differently on the various operating systems that are supported by the IDL software package.

The various methods for calling external software components from IDL are :

- The CALL_EXTERNAL method
- The LINKIMAGE method
- COM Object method (Windows only)
- The IDL-Java Bridge

The ***CALL_EXTERNAL*** function allows IDL to link to C or FORTRAN code in a very tight, but somewhat limited way that is far easier to use and implement that the *LINKIMAGE* procedure.  With *LINKIMAGE*, a programmer needs to write C code similar to the system routines built into IDL.  This methodology is more robust than *CALL_EXTERNAL*, but requires more detailed knowledge of IDL internals and, inevitably, more coding.  The *CALL_EXTERNAL* method is simpler and quicker to develop.  The basic concept of the *CALL_EXTERNAL* method is to create an external shareable code resource from your existing C or FORTRAN code, and then dynamically load it into the IDL process space at run-time.  The external shareable code resource created is referred to differently on each operating system.  Under UNIX, the shareable code is known as a shared object library; on Windows platforms it is a 32-bit dynamic link library (DLL).  The form that the shareable code resource is in (i.e. what platform you are on) does not affect the way it is accessed using the *CALL_EXTERNAL* function.  The code in the external shareable resource is dynamically linked into the IDL process the first time it is referenced in the IDL session, and every time you want to use this code within IDL you have to "*wrap*" it with a call to the *CALL_EXTERNAL* function.

In contrast, the ***LINKIMAGE*** procedure provides a way to load external C code into IDL and declare it in a manner so it appears as if it were part of the IDL internals. That way, you can execute your external code as if it were an IDL routine built into IDL, with the same calling sequence syntax as any other IDL function.  An obvious caveat is that you must declare it with a name that does not conflict with any of the routine names already built into IDL.  The *LINKIMAGE* procedure must "*link in*" this code in any given IDL session before the routine can be used, so it is a good idea to perform this linking in an IDL startup file.

**COM** (Component Object Model) objects are a specification and implementation for building software components that may be used to build programs or to add functionality to existing programs running on the Windows platform. COM components are written in a variety of programming languages (although most are written in C++) and are able to be utilized in a program at run time without having to recompile the program. In IDL, COM objects, regardless of type or method of creation, are treated as IDL objects. IDL will then internally recognize this COM-based object and will route the method calls to the internal COM subsystem for dispatching. COM Objects can be used in 2 ways when calling an external resource from an IDL program :

- Using the *IDLcomIDispatch* object within your IDL program to instantiate a desired COM object by using a provided class or program ID. This method is ideal for COM objects that do not utilize a graphical-user interface.
- Using the *WIDGET_ACTIVEX* function to embed an ActiveX control in an IDL widget hierarchy. In this scenario IDL would be acting as an "*ActiveX Container*".

The **IDL-Java Bridge** allows users to access Java objects within their IDL code, which enables you to take advantage of functionality provided by Java, including Java I/O, networking, and third party functionality. The *IDLjavaObject* class instantiates a desired Java object using the object's class name. An instance of this object within IDL allows you access methods and data members (properties) of the desired Java object. To the IDL user, an instance of the *IDLjavaObject* class behaves just like any other IDL object. When an instance of the *IDLjavaObject* class is created, the IDL-Java bridge connects that instance to a Java object. This initial connection starts a Java session. In IDL, you can monitor the session through the *IDLJavaBridgeSession* object. This object can be used to handle any exceptions (caused by the Java object) within IDL. Currently, the IDL-Java bridge is supported on the Windows, Linux, Solaris, and Macintosh platforms supported in IDL.

## External Software Calling IDL

The various methods for calling IDL from external software are :

- The Callable IDL method
- The Remote Procedure Call (RPC) method (UNIX only)
- The IDLDrawWidget ActiveX Control (Windows only)
- ION (IDL On the Net) plugin for IDL
- IDL Export Bridges

IDL is packaged in a shareable form that allows other programs to call IDL as a subroutine using a mechanism known as **Callable IDL**. This shareable portion of IDL can be linked into your own programs written in C, C++, FORTRAN, etc.. With this mechanism, all the functionality of the IDL interpreter is usable and presented to the programmer as a library of commands. The actual functionality is linked into the external program at run-time via the shared resource, such as a dynamic link library on Windows. For example, within a C program, IDL statements can be executed using *IDL_ExecuteStr* :

```
      IDL_ExecuteStr("tmp = indgen(100)");
      IDL_ExecuteStr("plot, sin(tmp)");
```

This code creates an IDL variable called "*tmp*" and plots the sine of "*tmp*", just as if these commands were executed directly at an IDL> command prompt.

The **Remote Procedure Call** mechanism allows IDL to be run as an RPC server on UNIX platforms. In other words, your program runs as an RPC client. It can execute procedure and function calls as if it were in the same memory space as the server. However, these two programs may be on entirely separate machines connected via a network. The commands are actually executed on the server machine. Variables can be created on the client and sent to the server, and variables created on the server can be sent to the client. With this method, you have access to the power of the server machine from a client machine.

IDL has been built into an **ActiveX Control**, which allows programs enabled with the ActiveX technology to access IDL functionality. This allows IDL functionality to be embedded into Windows applications and programming environments such as Visual Basic and Visual C++. When the ActiveX control is embedded, all the analysis and graphics capabilities of IDL become available to the application. The ActiveX control in IDL is presented in the form of the *IDLDrawWidget* object class. ActiveX is a technology that defines a standard way of embedding and controlling foreign components into applications on Window platforms. ActiveX is a Microsoft term for this set of technologies and services that are all based on the common object model (COM). There are several variations of ActiveX including ActiveX Documents, ActiveX Scripting, and ActiveX Controls. All variations of ActiveX are an outgrowth of Microsoft's Object Linking and Embedding (OLE), which defines a common interface for including foreign software components in applications.

**ION** (IDL On the Net) is a plugin for IDL. ION is a family of products that allow you to run your IDL applications in a networked environment, giving Intranet or Internet users access to IDL visualization and analysis. The primary design of this plugin is focused on a client-server environment that involves a web browser. There are 2 versions of the ION plugin. "ION Script" is a tool that includes a powerful tag-based language (similar to HTML) to publish IDL visualizations, analyses, and interactive applications on a webpage. "ION Java" can be used to create either Web-based, or entirely self-contained, distributable Java applications. ION Java combines both IDL and Java into a single, powerful tool for building client-server Java applications and Web applets. ION Java includes a low-level Java class library, pre-built Java applets, and mid-level component classes that provide you with the ability to create sophisticated Java applications that are driven by IDL.

The **IDL Export Bridges** provide the ability to easily export an IDL object as a native Java or ActiveX/COM object for use in external environments (such as Visual C++ and Visual Basic). When exported, an IDL object will appear like a native object in the target environment and, as such, doesn't require the user to know IDL syntax or operating methodologies. The user interacts with the objects using native methodologies and syntax. The bridge technology will convert data between the native formats and IDL as well as dispatch the method calls being made to the target IDL object. While the classic IDL execute string methodology is still available, for non-IDL users this technology provides a rapid method to use and deploy functionality developed in IDL.

# Appendix C:  Other Training Courses

## Training Courses Available from ITT

ITT offers introductory, intermediate and advanced courses in both IDL and ENVI. Classes are held year round at a variety of convenient locations and can be customized to meet your unique requirements.  In addition, we can come to your facility and perform custom on-site training as well.  Our expert instructors focus on your goals and how you can best utilize ITT's tools to achieve them.  All classes are taught with hands-on instruction by a team of skilled professionals.

The course that are currently offered for the IDL software package include :

- **Introduction to IDL** :
  - Scientists and programmers starting to use IDL for exploring their data become immediately more productive with this three-day course. Beginning with the basic concepts of variables and line plotting, the course takes students through file manipulation, programming methods, interactive data visualization and analysis techniques.  Users are introduced to the IDL Development Environment (IDLDE) and IDL's advanced mathematical and image processing capabilities.  This class is perfect for new users of IDL.
- **Intermediate Programming with IDL** :
  - Standard techniques are presented for building IDL programs to perform custom analyses that can be easily used by others.  Topics include advanced visualization techniques, use of built-in data analysis and image processing routines, development of applications employing a graphical user interface, introduction to object-oriented programming, introduction to the IDL Object Graphics system and a discussion of linking with external programs using the CALL_EXTERNAL function.
- **Advanced Topics in IDL** :
  - In this course a significant IDL application is planned, designed and built.  The application employs a user interface that allows interactive manipulation of objects in a graphical scene.  Emphasis is placed on visualization using the newest features of the IDL Object Graphics System and the IDL Widget Toolkit.  This course is designed for the scientist or programmer wanting to increase the visual impact and usability of their IDL applications for customers or colleagues.
- **iTools Programming** :
  - Learn how to work within the iTools Component Framework to build your own iTool.  Construct and register iTools visualizations, file readers and writers, manipulators and operations.  Learn how to modify the default iTools user interface, construct your own iTools user interface and embed an iTool in an existing IDL widget application. This class includes a review of object-oriented programming in IDL,

  - including working in the IDL Object Graphics system, as well as an overview of the iTools system architecture. This class is intended for advanced users of IDL.
- **Medical Image Processing with IDL** :
  - The course covers image processing with IDL. Emphasis on hands-on demonstration of image processing techniques commonly used in medical research. Explanation of how to use IDL's IDLffDICOMex class to read, write and query DICOM-format files, through several examples and exercises.