IRIS Explorer User’s Guide
Release 3

Document Number NP2872
ISBN 1−85206−110−3

Introduction

At the turn of the century, someone defined an inventor as a "person who makes an ingenious arrangement of wheels, levers and springs, and believes it civilization."

Footnote: *The Enlarged Devils's Dictionary,* by Ambrose Bierce (Penguin Books, 1967)

Now, nearly a hundred years later, it could be said that an inventor is a person who concocts an ingenious arrangement of computer code and believes it intuitive. Since everyone's intuition has its off−days, the *IRIS Explorer User's Guide* provides a few tips for using IRIS Explorer™, a decidedly "ingenious arrangement."

IRIS Explorer is a system for creating powerful visualization *maps*, each of which comprises a series of small software tools, called *modules.* A map is a collection of modules that carries out a series of related operations on a dataset and produces a visual representation of the result.

IRIS Explorer has three main components:

- the Map Editor, which is a work area for creating and modifying maps
- the DataScribe™, which is a data conversion utility for moving data between IRIS Explorer and other data formats
- the Module Builder, which lets you create your own custom modules (described in the *IRIS Explorer Module Writer's Guide*)

The IRIS Explorer System

To understand how these components interact in the system, think of a factory. The purpose of a factory is to take raw materials and fashion them into an end product, according to a specific design. The raw materials are fed into an assembly line at one end, go through a number of alterations and manipulations as they pass through the machines on the factory floor, and then come out at the other end in the form of a finished product. The product is inspected for the qualities stipulated in the design. If they are not present, or not satisfactory, the machines on the floor can be adjusted until the quality of the product meets requirements.

The factory floor is not the only important area in a factory. To make products, the machines need raw materials that are suitable for the task at hand. The materials may be delivered in usable condition, or they may need processing to get them into a form that the machines can accept. So the materials controller is important.

The factory also needs machines, a different one for each process that takes place on the floor. Some machines may be standard models, but for certain tasks, the factory needs custom−built machines. The machine shop, then, is also very important.

The way in which IRIS Explorer operates is analogous to the workings of this factory. Think of IRIS Explorer itself as the factory. The Map Editor corresponds to the factory floor, and the modules to the machines. The raw materials are the data, fed in at one end and passing through the system into the visualization unit, the inspection area. Here, the finished product, which is the visual object or image, is inspected and, if necessary, tweaked. A little more color here, a slightly smoother isosurface there, and the object is ready for its destiny — to be interpreted.

Alongside the Map Editor is the Module Librarian, where the modules are stored. These correspond to functioning machines, a variety of which are brought out of storage and reconnected each time a new design is implemented on the factory floor. So specific modules are launched in the Map Editor and wired together into a map.

Raw materials are critical to the manufacturing process. The IRIS Explorer factory uses data as the manufacturing unit. It makes some of its own raw materials, in the form of generated lattices. But many designs will require materials from elsewhere, external data that needs to be picked over and cleaned up, sorted through and carefully sieved, before the machines will accept it. The DataScribe performs this task. It accepts external data and lets the user build a module that converts the data into IRIS Explorer lattices, which other IRIS Explorer modules can process.

With data coming in and modules firing, the IRIS Explorer factory produces many interesting and informative geometric objects. However, the day may come when no modules are available in the Module Librarian to fulfill a part of the map design. This calls for a trip to the machine shop, the Module Builder. Here, you can build a module that will do exactly what you want it to do when it is wired into the map. It looks just like the other modules, with the same type of control panel and controls, and it can be stored and used again. Thus the scope of the factory has been enlarged.

However, large factory floors have a disadvantage. They can be noisy and messy, with lots of wires snaking here and looping there, and they tend to be crowded, on the principle that an empty space is immoral. Some machines may be running at less than their full capacity if a couple of their features are not needed for a particular design. Likewise with the IRIS Explorer Map Editor. When you create large maps, you might be tempted to think less of visualization systems and more of a bowl of noodle soup.

The IRIS Explorer factory has a way to cope with this. The manager scopes out a group of machines that for some reason are logically related, probably because they carry out closely related tasks, and selects certain of their functions to wire into a new control panel. The machines themselves are relegated, still intact, to the basement and the connecting wires come through the factory floor to the administrative, or group, control panel. In one easy move, the factory floor has gained a large open space, the assembly line has increased in efficiency, and the individual machines in the group are unscathed — simply out of sight until fetched up from the basement again. If you plan carefully, you could end up with one sleek control panel on the visualization unit and a completely clean and quiet floor.

The analogy should be clear. You can encapsulate a collection of modules in a group and gain all the benefits of a streamlined work area without sacrificing the integrity of the individual modules.

IRIS Explorer is a powerful and versatile system. The analogy of the factory necessarily simplifies its structure, but the examples in Chapter 1 give you a sense of what can be accomplished.

To use IRIS Explorer effectively, you should know what data you have, but you need no blueprint for what kind of visualization you want to produce. The great strength of IRIS Explorer is that you can innovate without retooling, and the design of your map reveals itself as you go. As with any exploration, you could end up with a map that takes you into uncharted realms of discovery and shows you implications of your data that you would not have thought existed.

The inventors of today have a distinct advantage over their predecessors. An ordinary factory floor is limited by the constraints of matter: the IRIS Explorer factory floor is limited only by the power of your imagination.

How to Use This Guide

The *IRIS Explorer User's Guide* describes how to use IRIS Explorer, which is a system for creating custom visualization programs and applications.

This guide is written for computational scientists and engineers in the fields of fluid dynamics, chemistry, meteorology, cosmology, physics, and mathematics. It addresses high−level users who are experts in their fields but are not programmers, and who will use the visual interface and tools in IRIS Explorer to create their models.

The guide assumes you have a basic understanding of:

- the UNIX® operating system, and can create data files
- the X Window System™ and Motif™, and can use a mouse

Contents of This Guide

The *IRIS Explorer User's Guide* has seven chapters and two appendices. They cover the following topics:

- **Chapter 1, "Getting Started",** provides five sample maps that visualize data from different scientific fields.
- **Chapter 2, "Working with the Map Editor",** explains how to use the Map Editor and Module Librarian to create maps.
- **Chapter 3, "Using Modules",** gives detailed instructions for using some of the more useful or complex IRIS Explorer modules, including loop controllers.
- **Chapter 4, "Editing Control Panels and Functions",** describes the Control Panel and the Parameter Function (P−Func) Editors, used to create module control panels and parameter functions.
- **Chapter 5, "Creating Groups and Applications",** describes how you can encapsulate a collection of modules into a single group and create an application from it.

- **Chapter 6, "Scripting",** explains how to write scripts in the IRIS Explorer Skm language.
- **Chapter 7, "Using the DataScribe",** explains how to convert external data into and out of IRIS Explorer data types through a DataScribe interpreter module.
- **Appendix A, "Configuring Your IRIS Explorer Environment",** describes how each of the IRIS Explorer menu options works.
- **Appendix B, "The Skm Language",** provides information on basic system configuration.

Some of the examples are complete, and some are fragmentary. Maps for all the examples are included with the IRIS Explorer software.

The companion volume to this guide is the *IRIS Explorer Module Writer's Guide*, which provides information on how to build your own modules with the Module Builder. In addition, documentation for all IRIS Explorer modules is provided on−line as *man* and *help* pages for each module. Similarly, the specification for each of the IRIS Explorer Application Programming Interface (API) routines is available via *man*. More information on this is provided in the on−line *IRIS Explorer Reference Pages*.

All of the on−line documentation is available in hard−copy form as the three volumes:

- *IRIS Explorer User's Guide*,
- *IRIS Explorer Module Writer's Guide*,
- *IRIS Explorer Reference Pages*, which contains all the information from the *man* pages for the modules and API routines.

Contact your IRIS Explorer supplier for more details.

Chapter 1

Getting Started

This chapter introduces you to IRIS Explorer using five sample maps, each from a different scientific field. The examples cover:

- image processing
- numerical mathematics
- molecular chemistry
- atmospheric physics
- computational fluid dynamics

1.1. Overview

You can create maps for all occasions, for example, to generate a 3D image of vortices around hydro−electric turbines, or to visualize carbon dioxide concentration as a function of geographic location. These activities involve a number of separate tasks such as reading the data into the map, colormapping a gradient, extracting a 2D slice of a 3D dataset and visualizing the result. The modules in the map carry out tasks like these.

The five maps presented here use authentic data to illustrate the workings of IRIS Explorer. Each map deals with a specific issue, and the modules provide ways of altering or enhancing the data so as to make its significance clear, and perhaps to elicit correlations that might otherwise have gone unnoticed.

The maps are complete. You need only launch, or open, the map in the Map Editor, and then try varying the parameters as suggested in each example. Chapter 2 explains how you set about creating your own maps. **Table 1−1** defines some of the terms you will encounter when you start IRIS Explorer.

Term	Meaning
Map Editor	Work area for creating and modifying maps
Module Librarian	Lists maps and modules available for use
Module	Performs an algorithmic function on the data passing through it
Control Panel	Gives the user access to the module functions
Widgets	Dials, sliders, buttons, and text slots for changing parameters
Map	A collection of modules that carry out a series of related operations on a dataset
Image	A 2D array that can be viewed using the *DisplayImg* module
Geometric Object	A collection of points, lines, polygons or other shapes that can be displayed by the *Render* module

Table 1−1 Some Components of IRIS Explorer

1.2. Launching a Map

You can open maps in two ways: from the command line in the UNIX shell, and from the Module Librarian in IRIS Explorer.

1.2.1. Opening Your First Map

If IRIS Explorer is not running, you can use the command line in UNIX when you open a map for the first time. Change to */usr/explorer/maps* and type the IRIS Explorer command at the % prompt:

```
cd /usr/explorer/maps
explorer −map mapname.map
```

The *mapname.map* is the name of the map you want to open, for example, *chemistry.map*;. The name of each map shown in this chapter is given in the introduction to the map.

IRIS Explorer starts up and the map appears in the Map Editor. All the modules are *selected* (highlighted in white), and you can move the entire map by dragging on the title−bar of a single module.

Click on the background of the Map Editor to deselect the modules in the map before you experiment with individual modules. The highlighting disappears.

1.2.2. Opening Subsequent Maps

Once IRIS Explorer is running, you can use the Module Librarian to open a map. Follow these steps.

1. To clear the Map Editor of modules, click on the Edit menu and select "Destroy All."
2. Go to the Module Librarian next to the Map Editor and use the vertical scroll bars to locate the maps, which are shown in blue.
3. Select the map you want by placing the cursor on the map name and clicking the right mouse button, then selecting "Launch" from the pop−up menu.

   You can also hold down the left mouse button, drag the map over to the Map Editor and release the button.

When you release the mouse button, the map appears, module by module, on the screen. The menu bar displays a number in angle brackets, for example, `<5>`, which tells you initially how many modules are in the map; the number decreases as modules appear on screen, indicating how many modules remain to be launched.

Quitting IRIS Explorer

To quit IRIS Explorer, select "Quit" from the Admin menu in the Map Editor.

1.2.3. Opening Modules

You can expand the mini control panel (shown in **Figure 1−1**) of any module into the full−scale control panel by clicking on the square *Maximize* button at the right top corner of the control panel. When you do this, the parameter widgets and their values are displayed for easier manipulation.

Figure 1−1 Module Control Panel

1.3. Image Processing

This example illustrates how you can use IRIS Explorer modules to process an image. The *KoreaContour* map takes an aerial image of a region of the Korean landscape, renders it in 3D, and overlays a contour map showing altitudes. The resulting image is displayed on−screen. The map file is */usr/explorer/maps/KoreaContour.map*.

1.3.1. Opening the Image Map

To open the *KoreaContour* map, displayed in **Figure 1−2**, type

```
explorer −map koreaContour.map
```

The seven modules are connected to one another by blue wires. The two modules on the left read in the image files, and the module on the far right displays the processed image. In between these are four modules that manipulate the image data.

Figure 1−2 Processing an Image

The seven modules in the map have the following functions:

- *ReadImg<2>* reads in the first image file (2D aerial image of a region of Korea).
- *ReadImg* reads in the second image file (altitude data for the same region).
- *DisplaceLat* displaces each node of the aerial image in the vertical direction according to the local value of the altitude.
- *DisplaceLat<2>* performs the same function as *DisplaceLat*, but with greater vertical distortion.
- *LatToGeom* creates polygons of the displaced data from *DisplaceLat*.
- *Contour* creates contours colormapped according to height, using the displaced data from *DisplaceLat<2>*.
- *Render* displays the output from *LatToGeom* and *Contour* as the surface with overlaid contours.

1.3.2. Experimenting with the Image Map

Both the aerial image and the altitude data go through *DisplaceLat* (see **Figure 1−3**), which displaces the image data so that topographical features become 3D.

- You can regulate the scale of the 3D rendering by turning the Scale dial on the module control panel.

  The result is a 3D version of the original 2D image, showing topographical features of the Korean countryside.

Figure 1−3 *DisplaceLat* Control Panel

The output of *DisplaceLat<2>* passes into *Contour* (see **Figure 1−4**), which creates a series of lines at varying heights above sea level just as in a topographic map. The scale factor for *DisplaceLat<2>* should be greater than that for *DisplaceLat*, to make the contour lines visible.

Figure 1−4 *Contour* Control Panel

To set the relationship between scale factors, the Scale parameter output port on *DisplaceLat* is wired to the Scale parameter input port on *DisplaceLat*<2>.

- You can alter the number of contour levels by moving the slider widget on the module control panel.
- You can also change the altitude range for which contours are shown by resetting the minimum and maximum levels.

The result is shown in *Render* (see **Figure 1−5**). You can open the *Render* window by clicking on the *Maximize* button and rotate the object in the window.

- To move the object, place the cursor on the object, hold down the left mouse button, and drag the object around. It continues to rotate when you release the mouse button.
- To stop the object, click on the background of the *Render* window.

Figure 1−5 The Contour Map in *Render*

For more information on using the *Render* menus to manipulate objects, see **"Visualizing Data"** in Chapter 3.

1.4. Numerical Mathematics

This example illustrates how to use IRIS Explorer modules to read in a short program script describing a mathematical function and visualize the data in the *Render* window. The mathematical function describes diffusive heat flow. It is a finite difference

Footnote: For more information on finite difference techniques, see Mitchell, A.R., and Griffiths, D.F., "The Finite Difference Method in Partial Differential Equations," John Wiley and Sons, 1980.

stencil for solving problems in parabolic differential heat conduction.

The form of the equation in this example is dimensionless; it serves as a normalized thermal model. A map using a

The form of the equation in this example is dimensionless; it serves as a normalized thermal model. A map using a specific form of such an equation with actual data can be used, for example, to test the conductivity of steel.

1.4.1. Opening the Heat−flux Map

To open the "heat−flux" map (see **Figure 1−6**), type

```
explorer −map heat−flux.map
```

Figure 1−6 Map for a Heat Flux

The map contains six modules, which have these functions:

- *LatFunction* reads in a program script that sets up a heat source and a heat sink, and generates the time−varying temperature distribution around them.
- *DisplaceLat* displaces the 2D lattice temperature data to create a 3D visualization of the way in which the temperature changes over time.
- *GenerateColormap* colormaps the temperature lattice by value.
- *LatToGeom* converts the displaced lattice data into a surface geometry.
- *Legend* produces a scale showing the temperature gradient.
- *Render* displays the surface and the legend.

Note: A *lattice* is an IRIS Explorer data type, which is used to store an ordered array of data. For more information, see **"Understanding IRIS Explorer Data Types"** in Chapter 2.

1.4.2. Experimenting with the Heat−flux Map

LatFunction reads in a short program written in the Shape language, which is described in detail in Chapter 10, "Module Prototyping with Shape" in the *IRIS Explorer Module Writer's Guide*. It is a C−like language that has capabilities for operating on whole lattices with a single statement. The program is saved in a file called *heat−flux* in */usr/explorer/maps* and can be run as often you choose.

The data passes from *LatFunction* to *Displacelat*, which displaces the 2D data into a 3D array to show the heat source as a peak and the heat sink as a trough. You can change the degree to which the peaks project from the surface by turning the dial on *Displacelat*. This causes the data to be displaced out of the plane to a greater or lesser degree.

GenerateColormap colormaps the heat−flux data according to temperature value. You can change the colors associated with each temperature value by using the option menus on the control panel. Refer to **"Creating Colormaps"** in Chapter 3 for more information.

LatToGeom uses the heat−flux and colormap data (which are both IRIS Explorer lattices) to create geometry for display in *Render* (see **Figure 1−7**). *Render* can only display geometry, and all lattice data must be converted first.

Figure 1−7 Visualization of the Flux Peaks

Once the data is displayed in *Render*, you can change the background color of the window by selecting "Edit Background Color" from the Viewing menu. When the Background Color Picker appears:

- Click anywhere on the color wheel to select a color
- Drag the slider to increase the intensity of the selected color

1.5. Molecular Chemistry

This example illustrates how you can use IRIS Explorer modules to construct a 3D model of a complex organic molecule and then calculate the optimal radius and likely path for a probe atom investigating the target molecule.

1.5.1. Opening the Chemistry Map

To open the "chemistry" map (see **Figure 1−8**), type:

```
explorer −map chemistry.map
```

Figure 1−8 Visualizing a NutraSweet Molecule

The map takes configuration data for a NutraSweet® molecule and creates a ball−and stick representation of it and a dot surface that shows the region that is accessible by a probe atom.

The map contains seven modules:

- *FileList* lets you select which of the available chemistry molecule data files you want to visualize.
- *ReadPDB* reads in the molecular configuration data.
- *BallStick* creates the ball−and−stick representation of the molecule.
- *AtomicSurf* calculates a solvent−accessible surface around the molecule. It uses the USURF

  Footnote: Written by Joseph J. Moon and published under the auspices of the Quantum Chemistry Program Exchange (QCPE), Indiana University. Program #566

  program to define a solvent−accessible surface around the molecule.
- *GenerateColormap* produces a colormap which is used to color the atoms in the molecule and the dot surface.
- *LatToGeom* converts the surface data in IRIS Explorer lattice format into geometry.
- *Render* displays the ball−and−stick geometry of the molecule and the solvent−accessible surface data as a dot surface.

1.5.2. Experimenting with the Chemistry Map

The molecular configuration data is read in by *ReadPDB*. The input file is in the Brookhaven Protein DataBase (PDB) format, a commonly used format for saving descriptions of proteins, including atom positions and properties, and bond locations.

BallStick (see **Figure 1−9**) receives the data in pyramid form and generates sphere−and−cylinder geometry. (The pyramid is an IRIS Explorer data type.) The radius of each sphere is calculated according to the van der Waals' radius of the corresponding atom.

- You can increase or decrease the radii of all spheres in *BallStick* proportionately by turning the dial on the module control panel.

Figure 1−9 *BallStick* Control Panel

The molecule data also passes into *AtomicSurf* (see **Figure 1−10**). It uses the van der Waals' radius of each atom and adds a solvent probe radius to it. The surface points in this "solvation layer" represent the positions in which a solvent probe would be in contact with the target molecule.

Figure 1−10 *AtomicSurf* Control Panel

- You can change the probe radius for which the solvent−accessible surface is calculated by moving the slider on the control panel. The smaller the value, the finer the surface.

  A value of 1.5 Angstroms approximates the radius of a water molecule, a commonly used probe. Probes that are too small or too large are not useful, particularly as the probe size approaches that of the target molecule.
- You can also change the surface dot density by turning the control panel dial. The larger the value, the larger the number of dots produced.

  It is best to use lower densities for large molecules, otherwise the visualization becomes very cluttered. A density of about 10 dots/Angstrom^2 works well for molecules which have sizes comparable to that of NutraSweet.

GenerateColormap creates a default colormap for the molecule based on atomic number. The domain is set at a minimum of 0 and a maximum of 16 to accommodate the atoms, which are hydrogen, carbon, oxygen, and nitrogen.

It also colormaps the dots in the solvent−accessible surface according to the parent atom of each dot. This provides you with information about which atoms a given probe will touch as it traverse the surface of the molecule.

- You can change the default color settings by using the option menus on the control panel. Refer to **"Creating Colormaps"** in Chapter 3 for more information.

You can use the capacities of *Render* to look at the NutraSweet molecule in great detail (see **Figure 1−11**). You can:

- *rotate* the molecule by holding down the left mouse button and dragging the molecule around with the cursor
- *translate* the molecule by holding down the middle mouse buttons and dragging the molecule around using the cursor
- *zoom* in and out on the molecule by holding down both the left and middle mouse buttons at the same time and moving the cursor up and down on the screen
- use the Lights menu to control the various light sources and highlight less distinct areas of the dot surface.

Figure 1−11 The Molecule in *Render*

For more information on using *Render*, read **"Visualizing Data"** in Chapter 3.

You can use *FileList* to select another chemistry molecule to examine.

1.6. Atmospheric Physics

This example illustrates how you can use IRIS Explorer modules to visualize atmospheric data. Intense storms can produce tornadoes, high winds, and hail, and it is possible to simulate these storms by integrating a set of mathematical flow equations. These equations can, for example, predict values for the wind speed and direction, air temperature, humidity, pressure, and water content every 5 to 10 seconds on a lattice of grid points 500 to 1000 meters apart.

The data in the example map is taken from a simulation of a single severe storm made by the storm group at the University of Illinois

Footnote: Dr. Robert Wilhelmson, NCSA Research Scientist and Professor of Meteorology in the Department of Atmospheric Sciences (DAS) at the University of Illinois at Champaign−Urbana and Dr. Lou Wicker, post−doc, and Crystal Shaw, research programmer, both at NCSA and DAS.

. The map shows an isosurface of rain density and a volume rendering of air buoyancy. An isosurface is a surface which passes through all points in a 3−D dataset where the data has a particular value. The data is in the form of a uniform lattice.

1.6.1. Opening the Volume Map

To open the "volume" map (see **Figure 1−12**), type:

```
explorer −map volume.map
```

Figure 1−12 Simulating an Evolving Storm

The map contains eight modules:

- *ReadLat* reads in the rain density data.
- *ReadLat<2>* reads in the water buoyancy data.
- *ReadLat<3>* reads in the colormap for the water buoyancy data.
- *IsosurfaceLat* generates an isosurface of the rain density.
- *WireFrame* creates a frame that defines the boundaries of the storm data.
- *GenerateColormap* modifies the colormap for the buoyancy data.
- *VolumeToGeom* converts the volume data to geometry for *Render.*
- *Render* displays the rain density isosurface and the water buoyancy volume.

Note: For volume rendering techniques to display the volume as shown in this map, you require alpha blending hardware. Some workstations do not have this hardware, but they can display the volume adequately if the Splat type on *VolumeToGeom* is set to "Point" (see **"Experimenting with the Volume Map"** below).

1.6.2. Experimenting with the Volume Map

This map uses both surface rendering (the isosurface) and volume rendering (the haze) in one visualization to show the relationship between a number of variables which have been calculated in 3−D space during the modelling of the evolution of the storm.

IsosurfaceLat (see **Figure 1−13**) calculates an isosurface from the density data and outputs it as geometry. The threshold value is the density value for which an isosurface is created.

- You can change the threshold value by turning the dial on the control panel, and thereby change the shape of the isosurface. This indicates how the rain density varies throughout the storm space.

Figure 1−13 *IsosurfaceLat*Control Panel

VolumeToGeom volume−renders the buoyancy data as a hazy cloud surrounding the isosurface. The entities that make up the cloud are called splats, small planar shapes that simulate fuzzy blobs on screen.

- You can control the splat configuration by changing the splat type and size on the module control panel.

Rendering splats can may take a long time. You can set and adjust an error tolerance to get a balance between rendering time and splat quality.

- To set the error tolerance, use the dial on the control panel. A low error value produces an accurate volume display, but the display takes longer to generate.

ReadLat<3> reads the colormap settings for the volume rendering of the water buoyancy data into *GenerateColormap*. The current setting for the colormap shows buoyant air, containing little water, in red and the saturated air in blue.

Once the data is in *Render*, you can analyze it in detail. **Figure 1−14** displays the reflectivity from the storm

Footnote: R.B. Wilhelmson, B.F. Jewett, C. Shaw, L.J. Wicker, M. Arrott, C.B. Bushell, M. Bajuk, J. Thingvold, and J. B. Yost, "A Study of the Evolution of a Numerically Modeled Severe Storm," International Journal of Supercomputer Applications, Volume 4, No. 2, Summer, 1990 pp. 20−36.

. The blue−white surface encloses the area containing most of the large drops and hail. Several light sources illuminate the surface. You can see the overhang, observed by weather radar in many severe storms, at mid−level where the reflectivity surface comes out toward the viewer.

Figure 1−14 Storm Data in *Render*

Above the storm, there is a region of many colors volumetrically rendered from the buoyancy, a quantity used in the equations for vertical velocity. It represents the instantaneous effects of temperature, moisture, and water mixed with ice on the velocity acceleration, and is large at and above the top of the storm.

A wavy appearance is evident, indicating the presence of strong gravity waves. Yellow/green indicates relatively high buoyancy, blue indicates relatively low buoyancy. The effect is roughly similar to throwing a rock into a pond, but here it is the storm growing into the upper atmosphere.

You can change the opacity values on the colormap to affect the look of the volume rendering (see Chapter 3, **"Creating Colormaps"**).

- You can lower the opacity of a range of colors to make them more transparent, and let you see other aspects more clearly.
- You can increase the opacity of a color to emphasize a feature.

1.7. Computational Fluid Dynamics

This example illustrates how you can use IRIS Explorer modules to study the dynamics of air flow over the nose of an aeroplane. The "cfd" map shows variations in air density surrounding the surface of the aeroplane.

The map generates an air flow field around the plane, extracts 2D slices of the volume, and colormaps them by density value, thus providing a means of examining different areas of the field.

The CFD group at Silicon Graphics created the plane from a model aeroplane by a digitized 3D scanning process, and the air flow data was generated from NASA's fluid dynamics program, *ARC3D*

Footnote: US Public Domain Software. For more information, see Pulliam, T. and Steger, J., "Implicit Finite−Difference Simulations of Three−Dimensional Compressible Flow," AIAA Journal, Vol 18, No. 2, Feb. 1980, pp. 159−167.

.

1.7.1. Opening the Cfd Map

To open the "cfd" map (see **Figure 1−17**), type:

```
explorer −map cfd.map
```

The map contains nine modules:

- *ReadLat* reads in the air density data file.
- *IsosurfaceLat* generates an isosurface from the air density data.
- *OrthoSlice* extracts the density near the plane's surface.
- *ProbeLat* creates non−orthogonal slices of the air density data.
- *TransformGen* sets the orientation for the *ProbeLat* slice.
- *PyrToGeom* converts the slice data from *ProbeLat* to geometry.
- *GenerateColormap* creates a colormap for the density data.
- *LatToGeom* converts the slice data from *OrthoSlice* to geometry.
- *Render* displays the isosurface and the slices.

1.7.2. Experimenting with the Cfd Map

OrthoSlice (see **Figure 1−15**) operates on the air density data to produce a curvilinear grid depicting the density field around the plane. The slice number is 1 (the minimum) and this index increases in a direction normal to the plane's surface. For *J = 1*, the grid actually lies along the surface of the plane.

Figure 1−15 *OrthoSlice* Control Panel

- You can change the slice number while holding the axis constant, and see the grid move out from the plane surface.

  As you vary the slice numbers, you get information about air densities at the plane surface and in its vicinity. High air densities occur at the nose when there is a massive deceleration of fluid. As the air flows away from the leading edge and accelerates over the wing, the density drops.
- You can also vary the axis. If you select the *K* axis, the slice is taken down the longitudinal axis of the wing and produces a cross−section of the wing.

ProbeLat and *IsosurfaceLat* complement each other. *IsosurfaceLat* generates an isosurface from the air density data. This tells you how all the air of a given density is distributed around the plane.

- You can change the threshold (the value at which the isosurface is calculated) by turning the dial on the control panel.

ProbeLat (see **Figure 1−16**) probes the curvilinear flow field along an arbitrary axis. The slice shows all density values in that region, colormapped by value.

Figure 1−16 *ProbeLat* Control Panel

- You can control the area of the slice by turning the Clip Size dial on the control panel.
- You can also change the form the probe takes, by selecting a type from the "Probe Type" option menu. In **Figure 1−17**, the selected probe type is a paddle.

Figure 1−17 Different Levels of Air Density

The axis along which the plane is cut is controlled by *TransformGen*. You can slice in any direction you choose by using the widgets on the *TransformGen* control panel to set a different axis. You can alter the orientation of the probe by using the mouse to move the object in *TransformGen*'s window.

GenerateColormap colormaps the grid produced by *OrthoSlice* according to its air density values. It is a scalar field, with one value at every point. The domain in *GenerateColormap* is set to a density minimum of about 0.7 and maximum of 1.1. This colormap is also used to color the probe surface according to air density values.

PyrToGeom and *LatToGeom* convert pyramid and lattice data respectively to geometry which can be displayed by *Render.*

Once the data is displayed in *Render* (see **Figure 1−18**), you can enhance the visualized data by using the *Render* menus. For example, click on the background of the *Render* window to bring up the *Render* pop−up menu. Select the "Draw Style" option, and then select "wireframe" from the cascading menu. To return all the surfaces to their original state, bring up the menu again and select the "as is" option.

Figure 1−18 The Rendered Flow Field

The grid, isosurface, and probe surface are separate objects, which can be separately treated in *Render.* For example, you could make the probe surface transparent, which would reveal the features behind it.

Chapter 2

Working with the Map Editor

This chapter explains, step by step, how to use IRIS Explorer tools to create and run maps. It describes how you:

- launch modules from the Module Librarian
- wire modules into a computational network, or map
- use the map to visualize numerical data
- select modules with compatible data types
- change the values of module parameters
- set parameter relationships in the Parameter Function Editor
- incorporate modules on remote systems into a map
- fix some common problems

2.1. Overview

In the Map Editor, you create an IRIS Explorer *map.* A map is a collection of *modules* of different kinds, connected in sequence, which process numerical data to accomplish a task. A module is a processing unit in a map. Each module accepts data, acts on it, and sends the result to the next module downstream. The Map Editor is the environment in which maps are created and executed, or "fired."

Maps can be used to perform a variety of tasks, for example, generating a visual image from a specific dataset. A patient may arrive at a medical clinic with an injured leg. The doctor generates a 3D CT scan, which shows a compound fracture of the femur. The CT scan data can be fed into an IRIS Explorer map and visualized as a 3D or "volume" image of the bone. The doctor may extract a 2D "slice" from the CT scan data along a plane close to the break in the bone. Then, if necessary, another slice can be taken from a slightly different angle for further elucidation. In this way, the doctor acquires a unique understanding of the fracture, which can help when deciding on an appropriate treatment.

Data is read into IRIS Explorer by the first modules in a map sequence, and the succeeding modules determine what form the results will take, and for what range of data values. For example, you can pinpoint areas where the bone is badly damaged by coloring the image according to bone density values.

The Module Librarian displays available maps and modules. You can launch single modules from the Module Librarian, arrange them in the Map Editor, and connect them to one another by their *data input* and *output ports.* These ports allow the flow of data between modules, the connections appearing as wire−like blue lines. Maps are usually arranged so that the data stream flows from left to right across the screen. You can also launch complete maps.

The Map Editor and Module Librarian menus provide you with options for manipulating the modules in a map. The modules themselves are accessible through their *control panels***.**These are rectangular panels, displayed on−screen, which contain *widgets*, the means of adjusting module parameters.

2.2. The Map Editor

The Map Editor (see **Figure 2−1**) is the work area in which you assemble modules for the purpose of organizing them into an operational map. A map is analogous to an application, in that it comprises a series of algorithms performing mathematical operations on data. Each module represents one discrete operation on the data.

2.2.1. Opening the Map Editor

To bring up the Map Editor, type `explorer` at the shell prompt in the shell window.

The Module Librarian and the Map Editor appear.

2.2.2. Quitting the Map Editor

To exit from the Map Editor, open the Admin menu and select *Quit*. When you do this, you automatically quit IRIS Explorer.

2.2.3. Editing Maps and Modules

You can select, copy, duplicate, delete, move, single or multiple modules by using the Map Editor. The modules may be free−standing or already wired into a map.

Use the Map Editor pull−down menus to:

- duplicate, cut, copy, paste, destroy, enable, disable, disconnect and reconnect modules (Edit menu). The operation is applied to **all** selected modules. In **Figure 2−1** only the *Render* module is highlighted as selected.

  Some of the Edit menu options have keyboard shortcuts listed next to them.
- group modules together, and open or close groups (Group menu).
- control layout within the Map Editor, such as of the connecting wires and the Log window (Layout Menu).

Alternatively you can use the drop icons at the bottom of the Map Editor window for some module operations such as destroying, disabling/enabling and disconnecting/reconnecting, see **Figure 2−1**.

Once a map is set up in the Map Editor, you can run it so that the modules fire and send data to one another. You can also incorporate several modules into a group or an application (see **Chapter 5, "Creating Groups and Applications"**).

The object visualized in **Figure 2−1** is a curved torus, generated by *GenLat*.

Figure 2−1 The Map Editor

2.3. The Module Librarian

The Module Librarian (see **Figure 2−2**) is a file browser and selector that provides access to IRIS Explorer modules and maps through its scrolling columns and Librarian shelf. Each Module Librarian has the name of its host system on the title bar.

Use the Module Librarian and its menus to:

- launch and save modules or maps in the Map Editor (File menu)
- reorganize modules into categories (Categories menu)
- store modules on the Librarian shelf (Display menu)
- add remote hosts if you are using a map distributed over several systems (Hosts menu)

Figure 2−2 The Module Librarian

2.3.1. Launching a Module

The act of bringing a module or map from the Module Librarian into the Map Editor and activating it is called *launching*. To launch a module or map, you can:

- Place the cursor on a module or map in a Module Librarian column and drag it to the Map Editor while holding the left mouse button down. Then drop it in place (the "drag and drop" technique).
- Place the cursor on the right side of the module or map in the Module Librarian and hold down the right mouse button. The Launch menu appears (see **Figure 2−3**). Select "Launch."

Figure 2−3 Module Launch Menu

- Drag and drop a module from the shelf of the Module Librarian (see **"Storing Modules on the Shelf"**).

- Use the "Open" option on the File menu. A file browser appears from which you can select a module or map (see **"File Browsers"**).

Maps in the Module Librarian are colored blue, modules are beige.

Finding a Module in the Librarian

Use the scroll bars in the Module Librarian columns to scroll up or down through the lists of available modules and maps. To scroll easily through the module categories:

- Put the cursor on the list and type the first letter or two of the module name. For example, if you want to launch the Render module, place the cursor on the list and type **`re`**. The list scrolls down to the first module name starting with **`Re`**. The Librarian is not case−sensitive, so you could also type **`RE`**.
- Place the cursor in the scrolling channel above or below the scroll bar and click on the mouse. The left mouse button moves you up or down one page at a time. The middle mouse button moves you up or down to the cursor position.

Placing the Module In the Map Editor

If you use the file browser to launch a module from a directory, the module comes up in the current launch position, which is the active point in the Map Editor. You can change the launch position by clicking the left mouse button on another spot in the Map Editor background before you launch the module. The next module is launched at that site.

To launch a module or a map from a remote system, select the remote host Librarian from the Hosts menu by clicking on the host name and then selecting the maps or module you want. You must first install the host in your host list, using the "New Host" option on the Hosts menu.

You can launch as many copies of a module as you like, although some modules might require special window server resources

Footnote: To launch many copies of modules that require a large number of color entries in the window server, such as *GenerateColormap*, **"Running the X Server "**.

. If you have more than one copy of a module in a map, the second copy will have the number **`<2>`** after its name. The third will have **`<3>`**, and so on.

2.3.2. Launching a Map

Maps, in the Module Librarian, can be identified as blue. You can launch a new map in the Map Editor by the "drag and drop" method from the Module Librarian (see **"Launching a Module"**).

If the map you want is not listed in the Module Librarian, you can select "Open" from the Module Librarian File menu. Use the file browser to find the directory and then choose the map.

The top left module in a map is always launched at the launch point.

If you want to start IRIS Explorer and launch a particular map at the same time, you can use the UNIX command line option *−map* with a filename.

Type:

```
explorer −map filename.map
```

The Map Editor opens and the map is launched and displayed automatically.

Note: If you are not in the directory where the map resides, either change to that directory before typing the command, or use the full pathname of the map, for example, */usr/explorer/maps/cfd.map*.

2.3.3. Saving Maps and Modules

Use the Module Librarian File menu to save your maps. You must save all IRIS Explorer maps with the filename extension *.map*; otherwise, IRIS Explorer will not be able to find the file when you try to open it again.

Click on the module title bar to select a module. The title bar and access pads will be highlighted in white. (See **"Selecting a Module"**).

A file browser comes up, and you are required to enter a filename for the map, for example, *NewMap.map*.

From the File menu you can select:

- "Save All," which saves everything in the Map Editor, or
- "Save Selected," which saves only the highlighted modules into the map

Modules can be saved only as part of a map, but you can save incomplete maps and bring them up later for further work.

2.3.4. Removing Maps and Modules

You can remove some or all modules from the Map Editor at a time.

To remove the current map completely, select "Destroy All" from the Map Editor Edit Menu.

To destroy only some of the modules in the map, highlight the modules (see **"Selecting a Module"**) and select "Destroy" from the Map Editor Edit menu.

To remove a single module, select "Destroy" from the module pop−up menu (see **"Using the Pop−up Menu"**).

To bring up a new version of the module, launch the module again from the Module Librarian. The new version of the module will be numbered **`<2>`** (or **`<3>`**, if it is the third version to be launched).

Example 2−1 Launching a Map

This example illustrates how to launch an existing map from the Module Librarian and carry out some simple operations.

1. In the Module Librarian, find the map called *simple*.
2. Drag and drop the *simple* map into the top left corner of the Map Editor. The map opens to show the control panel of each module in the Map Editor.

   Figure 2−4 shows the map you should see on your screen. It visualizes the data in the file */usr/explorer/data/lattice/testVol.lat*.

Note: On workstations without two−sided lighting the isosurface in *Render* may be dark. Change the Flip Normal? option on the *IsosurfaceLat* control panel from `No` to `Yes` to display the geometry properly.

Figure 2−4 A Simple Map Example

The *simple* map is a three−module map, consisting of:

- *ReadLat*, which reads in a file containing data for a lattice (lattices are discussed in **"Defining the Data Types"**)
- *IsosurfaceLat*, which creates an isosurface, a surface of uniform value in the test volume
- *Render*, which visualizes the selected isosurface of the lattice

All the module control panels are displayed with their wiring connections to one another visible. There is also a full−scale control panel for *Render*'s volume visualization. For more information on control panels, see **"Resizing the Control Panel"**.

When you first launch a map, all the module title bars and access pads are highlighted in white, to show they are selected. You can move the entire map by dragging on one title bar.

Before you can move individual modules, you must deselect them. Do this by clicking on the background of the Map Editor. A module is reselected if you click on the title bar.

You can:

- Click on the square *Maximize* button to bring up the full−scale control panel of *IsosurfaceLat*.
- Turn the dial on the *IsosurfaceLat* module control panel to change the threshold value of the isosurface, and hence, change the shape of the object displayed in *Render*.
- Experiment with the menus on the *Render* window menu bar and the decoration icons around the border.

For more information on using *Render*, see **Chapter 3, "Using Modules"**.

2.4. Modules

Modules are the individual units in a map. They are the mathematical engines, powered by data, that make maps work. Although their functions vary, they all share a basic structure (see **Figure 2−5**). Each module has a visible form, its *control panel*, by means of which you can direct the *internal* (and invisible) engine, or *core*.

The internal portion of the module consists of:
- module wrappers, which mediate between the core and the control panel
- the module core itself, which consists of a mathematical algorithm. Each module has a different algorithm, hence it carries out a different function in IRIS Explorer.

See the *IRIS Explorer Module Writer's Guide* for more information on the module core.

Figure 2−5 General Structure of a Module

The visible portion, or control panel, which appears in the Map Editor, is a rectangular window that contains:
- *input* and *output ports*, which allow the module to accept and pass on compatible data types. See **"Understanding IRIS Explorer Data Types"** for an explanation of data types.
- *parameters*, scalar data values which can be set and altered by widgets in the control panel. See **"Setting Module Parameters"** for a full description.

The control panel has three forms, which are described in **"How Modules Work"**. You can direct the activity in the module core through the ports and widgets on the module control panel.

2.4.1. What Modules Do

Modules accept data on their input ports, and pass it along via their output ports after they have modified it. When a module acts on data, or executes, it is said to *fire*. A module may be turned off, or disabled; and it can be fired on a computer other than the one where you are running the Map Editor.

General Module Functions

IRIS Explorer modules can be grouped according to their general function (see **Table 3−1** in Chapter 3). They can:
- read in data files (input modules, such as *ReadLat*)
- generate data from other data, such as extracting a planar slice or computing a numerical result from a volumetric dataset (feature extraction and analysis modules, such as *DisplaceLat*)
- process data to create a visual form (geometric representation modules, such as *IsosurfaceLat*)
- display the geometric representations on−screen (the *Render* module)
- write data to disk files (output modules, such as *WriteLat*)

Of these, the *Render* module stands on its own. It is the means whereby the data passing through the map is visualized and is likely to be an integral component of most maps. Once the data has been visualized as an object in the*Render* window, you can use the graphic capabilities of *Render* itself to enhance certain aspects of the object and to view it from various angles.

The other modules operate on the data in various ways to produce a geometric object that makes it easier to understand, interpret, and extrapolate from the original data. *Render* and some of the more commonly used modules are described fully in Chapter 3.

Specific Module Functions

To decide which modules you want in a map, you need to have more information about what each one does, what data it accepts, and what it produces for other modules to use, as well as what parameters it can have.

You can find information in these places:

- All the available IRIS Explorer modules are listed in the Module Librarian. You can move them from one category to another in the Librarian. For details, see **"Organizing the Module Librarian"**.
- To see the modules listed by function, refer to **Table 3−1** in Chapter 3.
- To see the modules listed by the data types they accept as input and produce as output, refer to **Table 2−1** through **Table 2−3**.
- To find out in detail what the module does, see the module definitions in *IRIS Explorer Reference Pages*, or use the module Help option (see **"Getting Help"**).

2.4.2. How Modules Work

When you launch a module from the Module Librarian, the module control panel (see **Figure 2−6** through **Figure 2−8**) appears in the Map Editor. Module control panels are rectangular windows that give you access to the capabilities of the module core via two main channels:

- the input and output ports, which allow you to pass data into and out of a module, and between modules in a map (see **"Wiring Module Ports Together"**). They appear on the *micro* and *Diminutif*™ forms of the control panel .
- the widgets, including buttons, dials, and sliders, which allow you to set module parameters (see **"Setting Module Parameters"**). They appear on the Diminutif and full−scale forms of the control panel.

The control panel also has:

- a title bar, which contains the module name and access pads for the input and output ports (see **Figure 2−6**).
- sizing buttons, for enlarging or closing down the control panel.
- a pop−up menu, which provides access to the module help files, parameter function editor, and other control options (see **Figure 2−9**).

Figure 2−6 Micro Control Panel/Title Bar

Selecting a Module

Before you can move or save a module, or use any of the Edit Menu operations on it, you must select it.

To select a single module, click on the module title bar with the left mouse button. The title bar and both access pads are highlighted. You can move the module around the Map Editor by holding down the left mouse button and dragging the mouse.

To select more than one module at a time, you can "lasso" them by clicking in the background of the work area, holding down the left mouse button, and sweeping out a rectangle that encloses the desired modules. When you release the mouse button, all modules that are completely enclosed by the lasso will be selected. Alternately, you can hold down the **`<Shift>`** key and click on each module in turn.

To select all the modules currently in the Map Editor, open the Edit menu and click on "Select All."

Use the **`<Shift>`** key with the "sweep" gesture to add more modules to an already selected group.

To deselect all selected modules, move the cursor off the title bar and click any mouse button. To deselect one module, hold down the **`<Shift>`** key and click on the module.

Resizing the Control Panel

The sizing buttons let you switch between the different forms of the module control panel. When you first launch a module, it appears in the Diminutif form (see **Figure 2−7**), which shows the title bar, port access pads, and general widget layout.

Figure 2−7 Diminutif Control Panel

- Click on the *Minimize* button to fold the panel into the *micro* form (see **Figure 2−6**). This saves space in the Map Editor. You can toggle back and forth between the Diminutif and micro control panels.
- Click on the *Maximize* button to open the *full−scale*, or maxi, control panel (see **Figure 2−8**); the Diminutif panel remains in the map. You can also open the maxi control panel from the micro control panel.

Figure 2−8 Full−scale Control Panel

The full−scale panel shows details of the widgets and the parameter values. You can move it outside the Map Editor window and enlarge it as much as you wish. You can also use the *Maximize* and *Minimize* buttons to open the maxi control panel to full−screen size or to iconify it.

The maxi control panel shows the names, ranges, and current values for widgets. It also has a menu bar with a Help menu, but does not show the ports.

To close a maxi control panel, click on the left button of the title bar and select "Close" from the Window menu, or double−click quickly on the left button of the title bar. The full−scale panel disappears but the Diminutif control panel remains in the network.

If you are working with a group of modules, use the "Select All" option from the Map Editor Edit menu and then choose "Make Micro" or "Make Mini" from the Layout menu to swap between these forms. You have to open each full−scale panel individually.

Note: If the maxi control panel disappears under other windows on the screen, you can pop it up again by clicking on the *Maximize* button on the Diminutif or micro control panel.

Using the Pop−up Menu

The module pop−up menu (see **Figure 2−9**) appears when you click on the title of the module control panel with the right mouse button. You can choose any one of these options:

Fire Now	The module fires and sends processed data to the next module downstream.
Duplicate	Makes a copy of the module, which you can wire into the map.
Disable	Temporarily deactivates the module. A disabled module will not fire until you enable it again. However, you can still set its widgets.
P−Func Editor	Opens the Parameter Function Editor, which lets you establish relationships among module parameters. It is described in **"Using the Parameter Function Editor"**.
Show Log	Brings up the module Log window, which displays all the module output resulting from the current map operation (see **"Logging Module Output"**.)
Help	Explains the purpose of the module and lists its ports and parameters.
Credits	Brings up information about the module’s author.
Replace	Replaces the module with a newly−launched module with the same connections and parameter settings.
Interrupt	Interrupts a running user function without destroying the module. You may get unpredictable results from a map if you halt a module when the map is still firing.
Destroy	Completely removes the module from the map and the Map Editor.

Figure 2−9 Module Pop−up Menu

Using Drag−and−Drop Operations

The Drop Icons at the bottom of the Map Editor window can also be used to perform operations on a single module or many selected modules by dragging a module onto the icon. Press the middle mouse button over the module control title bar. This will show the "drag cursor", a representation of a module. Keeping the middle mouse button pressed, drag the cursor to the required icon and release to perform the operation.

If the module dragged has been previously "marked for selection" then the operation is applied to **all** the selected modules, otherwise the operation is only applied to the dragged module.

Destroy, *Replace*, *Help*, *Show Log* and *Disable/Enable* icons provide the same functionality as above options in the module pop−up menu, but can be applied to many modules. *Disconnect* and *Reconnect* icons provide an alternative to using the options on the edit menu.

Getting Help

Each module has a Help window associated with it, which explains the purpose of the module and lists its input ports, output ports, and parameters.

To open this window, click on the title bar of the control panel with the right mouse button and select "Help" from the pop−up menu (see **Figure 2−9**).

To close the Help window, click on the left button of the title bar and select "Close" from the Window menu, or double−click quickly on the left button.

The modules are listed alphabetically and described in detail in the *IRIS Explorer Reference Pages* and by −−> −−their individual Help files and *man* pages.

Replacing Modules

Replace provides a convenient one step method of disconnecting, destroying, re−invoking and reconnecting a module! This is particularly useful when developing a module. The module can be replaced, with the new version of the executable, in one easy step. The new module is re−invoked with all the old's existing connections, P−Func and Widget values. There are two ways you can instigate a replace:

- you can drag a module to the "Replace" icon, located at the bottom of the Map Editor, to replace it or if it has been selected, replace it together with all other selected modules.
- You can also use the "Replace" option on the control panel's pop−up menu to affect a single module.

Example 2−2 Creating an Incomplete Map

This example illustrates the first steps in creating a map.

1. Launch these modules from the Module Librarian by dragging and dropping each one in turn into the Map Editor:
   - a *ReadLat*module (to read data in a lattice format)
   - a *Contour* module (to create a contour plot of the lattice data)
   - a *GenerateColormap* module (to colormap the contours according to value)
   - a *Render* module (to visualize the contoured data)

   You now have the basic components of a map in the Map Editor.
2. Save these modules as a map by selecting "Save All" from the Module Librarian File menu. This will pop up a file browser.
3. When the file browser appears, in the "Selection" type−in slot, enter the name of the file in which to save the map. You can change directory, by editting the "Filter" type−in slot.

   Note: A map name must have *.map* as a suffix for IRIS Explorer to recognize and treat it as a map, for example, *incomplete.map*.

   When you have typed in a name, click on the OK button.
4. Now destroy the modules in the Map Editor by selecting the "Destroy All" option from the Map Editor's "Edit" menu. This will allow the saved map to be loaded onto a "clean slate".
5. To load the saved map, select the "Open" option, from the "File" menu in the Module Librarian. When the file browser appears, in the "Selection" type−in slot, enter the name of the map you saved previously

   You can add the map to the list in the Module Librarian by editing your personal configuration file. This procedure is described in **Appendix A, "Configuring Your IRIS Explorer Environment"**.

To launch this map and connect the modules, go to **"Wiring Modules Together"**.

2.5. Wiring Module Ports Together

Modules communicate by passing data from one to the next through their input and output ports (see **Figure 2−10** and **Figure 2−11**). Once you have modules in the Map Editor, you can connect them together individually or severally, disconnect them, change your mind about connecting them, and alter the appearance of the connecting wires.

Figure 2−10 Module Input Ports

Figure 2−11 Module Output Ports

Each module control panel has two port access pads, one for input ports and parameters on the left of the control panel, and one for output ports and parameters on the right. A single module can have several input and/or output ports.

Data port names are listed first on the port menu, parameter port names are listed next, followed by synchronisation port(s). The port name is followed by the IRIS Explorer data type that it can accept; that is, "Lattice," "Pyramid," "Geometry," "Parameter," and "Pick." Parameter ports are associated with widgets. If the port is optional, the type is followed by "(Opt)."

A connection is required on a module port unless the port is marked as optional (*Opt*). For example, *GenerateColormap* has an optional input port. A module will not operate properly in a map unless all its required ports are connected to compatible ports on other modules and are receiving data.

A module with parameter−based widgets always has an input and an output port for each parameter. For example, *BlendImg* has two data input ports and one parameter input port for "Blend" (see **Figure 2−10**). It has one data output port and a corresponding parameter output port for "Blend" (see **Figure 2−11**). Parameters are passed through the output ports and you can set and connect them so as to control widgets on other modules.

The synchronisation ports *Fire* and *Firing Done* are present on all modules, and as their names suggest can be used to control a firing sequence. See **Using the Synchronization Ports**. A *Firing Done* output port can be connected to a *Fire* input port. Loop controller modules have an additional output port *Loop Ended*. See **Constructing Loops**.

2.5.1. Making a Compatible Connection

To select an input or output port, click on the associated access pad using the right mouse button. A list of the input or output ports and parameters associated with that module appears (see **Figure 2−10** and **Figure 2−11**).

IRIS Explorer makes use of five data types: Lattice, Pyramid, Geometry, Parameter, and Pick. They are explained in **"Understanding IRIS Explorer Data Types"**. You can connect only input and output ports that accept the same data type.

For example, you can connect an output port called "Colormap − − Lattice" to the input port called "Input − − Lattice" but not to one called "Input − − Pyr." Only an output port of type "Pyramid" is compatible with the latter.

When you click on an output port to select it, all the compatible input ports on other modules in the Map Editor are highlighted in green. Likewise, when you select an input port, all compatible output access pads light up in a different shade of green. Incompatible ports are grayed out, so they cannot be selected.

Since a lattice can assume many forms, the Map Editor does type−checking for types of lattices and lets you connect only those modules that accept and produce the same type of lattice. For more information on lattices, see Chapter 3, "Using the Lattice Data Type" in the *IRIS Explorer Module Writer's Guide*.

2.5.2. Wiring Modules Together

Wiring modules together is a simple process. You can wire input to output port or output to input port. **Figure 2−12** shows a connection from the output port of *Contour* to an input port of *Render.*

To make the connections, follow these steps:

1. Click on the access pad of the first module and select a port from the cascading menu. The access pads on other modules in the Map Editor that have compatible ports are highlighted and you are in wiring mode.
2. Click on one of the highlighted pads and select a matching port from the port menu.

   Once you click on the second port, the highlight disappears and a blue wire is shown running between the ports, signifying that a connection has been made.

To cancel an incomplete connection, when only one port has been selected, click on the background of the Map Editor.

You exit the wiring mode automatically once a connection has been made.

Figure 2−12 Making Connections between Modules

2.5.3. Connecting Several Modules

You can wire more than one input port to a given output port, and vice versa, provided that the data types on the ports are compatible. This is called "fanning" connections out and in.

Figure 2−13 shows a *ReadLat* module with two connections on its output port, one to the input port of *Contour* and one to the input port of *Isosurface.* To make these connections:

1. Select the output port of the module you want to connect (*ReadLat* in **Figure 2−13**).
2. Select the input port of the second module (*Isosurface* in this case). A connection is made between the first and second modules *(ReadLat* and *Isosurface*).
3. Select the input port of the third module (*Contour* in this case).
4. Then select the output port of the first module (*ReadLat*) again. This port now has one other connection to it.
5. The cascaded port menu shows the names of the output module and port already connected, below a <*Connect*> bar. Click on the <*Connect*> bar to add the new output port to the list (in this case, *DiffLat*). The highlighting disappears, indicating that the connection has been made (see **Figure 2−13**). The number following the data type on the port menu indicates how many connections the port has.

Figure 2−13 Making Multiple Connections

To connect an input port to the output ports on several different modules, follow the same procedure.

2.5.4. Disconnecting and Reconnecting Modules

You may wish to disconnect a module or a number of modules completely from the map, or you may wish to simply break a single connection, between two modules. To break a single connection between two modules, follow these steps:

1. Choose either of the modules in the linkage and click on the access pad of the wired−up port.

   The cascaded port menu shows the name of the other module port connected to it (see **Figure 2−13**).
2. Select the connected port by clicking on its name (not on the <*Connect*> bar).

The connection between the modules is broken and the blue wire disappears. This method is useful if you wish to break only one connection.

If you wish to break all a module’s connections then it is easier to "disconnect" the module, rather than breaking each connection in turn. There are two ways to disconnect a module:

- you can use the "Disconnect" sub−menu from the Map Editor "Edit" menu to break the connections of a selected module or a number of selected modules. There are three options to the "Disconnect" sub−menu:

  - *All* − breaks all connections
  - *Input* − breaks input connections only
  - *Output* − breaks output connections only
- you can drag a module to the "Disconnect" icon. This operation breaks both input and output connections. As for all the drop icons, if the module has been selected, the operation is applied to all selected modules.

When you use "Disconnect", IRIS Explorer will "remember" a module's connections so that they can be re−wired later if desired using "Reconnect". If another disconnect operation is applied to it, only the connections most recently broken will be "remembered".

The "Reconnect" operation, using either the Map Editor "Edit" menu, or the "Reconnect" drop icon can be applied to a module after a "Disconnect". This operation will attempt to restore its connections to the state prior to the last "Disconnect". If an individual connection to another module cannot be re−instated, for example the other module has been destroyed, then that connection will be skipped. If a connection has already been re−instated, for example the user has already manually wired it up, then again the connection will be skipped. The operation will also work on a module that has been destroyed and re−launched.

2.5.5. Wire Routing

You can change the direction, or slant, of the wires that connect the modules in a map using "Wires" on the Map Editor Layout menu. Each option to "Wires" changes the alignment of the wires between modules, as their titles indicate.

To change the slant of the connecting wires, choose one of the three options, "Pt to Pt," "Right Angle," or "Diagonal." Click on each one in turn to see its effect.

It is an art to keep the wires in a map distinct and disentangled. However, no damage results from their being crossed or threaded underneath a module. The map merely becomes a little harder to read.

You can create more aesthetic maps by collapsing several modules into a group module. In this case all the selected modules disappear and are replaced by one control panel of your own design. For more details, refer to **Chapter 5, "Creating Groups and Applications"**.

Example 2−3 Wiring Modules Together

This example illustrates how to connect modules to make a functioning map (see **Figure 2−14**). In the previous example, you saved a set of four modules.

Figure 2−14 Example of Map Connections

Launch the map by dragging and dropping it in the Map Editor. Then wire the modules together and set widgets as follows:

1. Connect the Lattice output port of *ReadLat* to the "Input −− Lattice" port of the *Contour* module.
2. Type a filename into the file browser of the *ReadLat* module. A good lattice data file is in */usr/explorer/data/lattice/testVol.lat*.
3. Open the full−scale control panel for *Contour* by clicking on the square *Maximize* button (for more information, see **"Resizing the Control Panel"**).
4. Set the minimum and maximum values for *Contour* by turning the dials to 1.0 and 2.0.
5. Connect the Lattice output port of *GenerateColormap* to the "Colormap − − Lattice (Opt)" port of *Contour*.

   Notice that *Contour* has two connections on its input pad, but they go to two separate input ports.
6. Connect the "Contours −− Geometry" output port of the *Contour* module to the "Input −− Geometry" input port of the *Render* module. An object appears in the *Render* window representing contour curves of equal value through the data in the lattice file.
7. Move the slider on *Contour* to create more sets of contour lines.

8. Break the connection to *Render.* Notice that the contour lines disappear. Connect it up again and they reappear.
9. Disconnect *GenerateColormap* from *Contour* and see what happens.

If you want to keep the finished map, remember to save it again.

2.6. Firing Modules

When a module is activated, it fires and the user function operates on the data it has received from its input ports. As the module fires, its title bar turns yellow, and stays yellow until the module has completed execution. The color change is called *execution highlighting.* (This highlighting can be turned off by toggling the "Exec Hilite" option on the Layout menu). This might take a fraction of a second or several minutes, depending on the complexity of the user function and the amount of data it is processing.

Firing can occur at several different stages of the map−building process. A module fires when:

- it is launched in the Map Editor, provided it has all the data it needs on its input ports
- modules upstream of it in the map fire, sending data to it
- a module upstream of it in the map fires, that has a connection to its *Fire* port.
- one of its parameter values is changed
- you choose "Fire Now" from the module pop−up menu
- you connect it to an upstream module which has already fired and produced data on its output port
- you connect its *Fire* port to an upstream module which has already fired
- it is triggered by an external event; for example, data arrives on a port from an external file or process

2.6.1. Forcing a Firing

You can force a module to fire by:

- Selecting "Fire Now" from the module pop−up menu.

  This is useful for modules that have no parameters, for example, many of the image−processing modules, or if you do not want to change a parameter value.
- Changing the value of a module parameter.

  You change the value of a parameter by moving the widget that controls the parameter of interest. You can turn a dial, move a slider up or down, or switch a button off or on (see **"Setting Module Parameters"**).

Each time you alter a parameter or change a connection in a map, in effect feeding in more or altered data values, the affected modules automatically fire again. This can cause a cascade of firings downstream of the module.

For an in−depth explanation of the mechanism behind module firing, see Appendix B of the *IRIS Explorer Module Writer's Guide.*

2.6.2. Using the Synchronization Ports

When a module fires successfully a connection from its *Firing Done* port to a downstream module's *Fire* port can cause the downstream module to fire in exactly the same way that sending new data to the module would cause it to fire.

You can therefore use these synchronization ports to

- control a sequence of module firings rather than allowing the module firing mechanism to determine the ordering. Normally this is controlled by the data flow, therefore these ports are useful when there is no need for any data to be transferred between modules. For example wiring modules as shown in **Figure 2−15** will cause module firings to be sequential in the order D,C, B, when A is fired. Omitting the *Firing Done* to *Fire* port connections would cause the modules B, C and D to attempt to fire simultaneously.
- control successive firings of a module, for example a module that sends out a sequence of data on one of its output ports. The *AnimateCamera* module sends out a series of cameras, usually to *Render.* A new camera will be sent each time an upstream module that is connected to its *Fire* port, fires. Modules that are designed to fire repeatedly

are usually wired into a loop, rather than a map with a linear dataflow. See **Constructing Loops** for further discussion.

Figure 2−15 Controlling Module Firing with Fire Ports

2.6.3. Temporarily Disabling a Module

There are a number of ways in which module(s) can be disabled:

- you can use the "Disable" option from the Map Editor "Edit" menu to disable a selected module or a number of selected modules temporarily. The "Enable" option reactivates it or them again.
- you can drag a module to the "Disable" icon, located at the bottom of the Map Editor, to disable it or if it has been selected, disable it together with all other selected modules. The "Enable" icon can be used to reactivate it or them again.
- You can also use the "Disable/Enable" toggle option on the control panel's pop−up menu to affect a single module.

Disabling a module is useful if:

- You want to modify several module parameters without having the module fire every time you make a change.
- Your map contains a module that takes several minutes to finish firing.

Each time you made a connection to the module or altered a parameter, you would have to wait several minutes while it fired before you could continue. You can avoid the wait by disabling the module while you complete your operations, then enable it again when you are ready to fire the complete map.

If you disable a module that is connected in sequence with other modules, it will not send new data to the modules that follow it. If those modules rely entirely on the disabled module for data, they also cease to fire. The flow of data will be blocked, along this branch at least, until you reenable the disabled module.

2.7. Logging Module Output

When modules fire in the Map Editor, they often generate informative text messages. For example, *PrintLat*, *PrintPyr*, and *PrintPick* all output text strings describing their data. Other modules may also produce messages in the course of a map cycle, all of which are logged by IRIS Explorer. You may be interested in seeing all generated messages, or only those put out by a particular module.

2.7.1. Logging Map Data

The Map Editor Log window captures all the messages generated by modules in the Map Editor. It is a read−only scrolling text window that typically appears below the Map Editor (see **Figure 2−16**). Messages in the window scroll upward, with new lines inserted until the size limit of the Log is reached. After that, lines are deleted. The Log lists the output from each module as it occurs, with the name of the generating module as a prefix to each line.

Figure 2−16 The Map Log Window

You can hide or display the Map Editor Log by toggling the "Log Window" option on the Map Editor Layout menu.

2.7.2. Logging Module Data

To see text messages for an individual module without having to reviewing the entire Map Editor Log, you can open and examine a Module Log (see **Figure 2−17**). Each Module Log displays messages generated by that module alone.

Open the Module Log by selecting "Show Log" from the module pop−up menu (see **Figure 2−9**).

Figure 2−17 Module Log Window

2.7.3. Reviewing Log Messages

You can cut and paste the information displayed in the Log windows into other windows. By selecting the "Editable Log" option from the Log window pop−up menu (see **Figure 2−18**) you can type further text into the window, if, for example, you wish to annotate the output. The menu appears when you click the right mouse button in the Log window. To change the Log back to a read−only window (the default), select the "Un−Editable Log" option. The other two options let you clear the window and save its contents.

Figure 2−18 Log Window Pop−up Menu

You can also send the log output to the UNIX shell from which you launched IRIS Explorer. To have text messages sent to `standard output`, start IRIS Explorer with this command:

```
explorer −debug printAlert
```

If you use this option, you can still use the Log windows for reviewing text messages.

2.8. Understanding IRIS Explorer Data Types

In IRIS Explorer, the data that modules accept and pass on is characterized by the *data type*. The data type is the form that data takes in IRIS Explorer. There are five IRIS Explorer data types: lattice, pyramid, geometry, parameter, and pick.

All data is passed through the module input and output ports, each of which accepts or outputs only one data type. Modules may, however, have several ports with a different data type on each one.

2.8.1. Defining the Data Types

The IRIS Explorer data types are:

Lattice	A series of arrays, in one or more dimensions, that hold dimension variables, nodal data values, and coordinate values. For example, data that defines a volume, such as a 3D CT scan of a bone, is in lattice format. This is the most commonly used data type.
Pyramid	A hierarchy of lattices, with information on how the lattices are connected. The pyramid data type is used to manipulate finite element data and molecular data. For example, data describing a complex organic molecule and a finite−element mesh of a car body are both in pyramid format.
Geometry	A collection of geometric primitives, including polygons, lines, and points, used to build a visual representation of a dataset. For example, a contour surface from a CT scan is a collection of polygons. The vertices of each polygon can have associated color and surface normals. Data is displayed in the *Render* module in geometry format.
Parameter	A parameter holds a scalar value: an integer, real number, or string. The value of a parameter input to a module is controlled by a widget on the module control panel . You can also perform arithmetic functions on them using the Parameter Function Editor (see **Chapter 4**). Parameters are used to set values for slicing planes, error tolerances, quantities, and filenames.
Pick	A specialized data type that lets you select an area of an object in the display window of a render or image−display module and collect data about the selected area. This information can be fed to another module such as a query module.

It is possible to define new data types in IRIS Explorer, and you may find that the input and output port menus of some modules list data types in addition to those described here. Those are user−defined data types. They work exactly as IRIS Explorer data types do, handling data in a specific format.

The structure of the lattice data type is discussed in more detail in **Chapter 7, "Using the DataScribe"**. All the data types are described fully in the *IRIS Explorer Module Writer's Guide*.

2.8.2. Using the Data Types

The IRIS Explorer data types represent groups of data. A given instance of a data type on a port can be generic or highly specific. For example, a module may have an input port of Lattice type defined so that it will accept a wide variety of lattices of different sizes, or so that it will accept only 2D scalar lattices of real numbers with uniform coordinate mapping.

For example, the *Contour* module accepts a Lattice data type on its input port, but the lattice must have two or three dimensions. If you try to pass it data in the form of a 1D lattice, you get an error message.

The module definitions give the data type specifications for the ports on existing modules. When you build a module, you can define each port as you like. The more general the specification for a port, the more modules you can connect to it (but the harder the module is to write).

If you have any doubts about port compatibility when you connect modules into a map, you can check the data types acceptable to each module by:

- calling up the Help window from the module pop−up menu
- reading the module definitions in the *IRIS Explorer Reference Pages*

2.8.3. Looking at Module Data Types

Certain modules can accept a pyramid or lattice data type and output a lattice or geometry data type after performing a data conversion. These include *PyrToLat*, *LatToGeom*, and *Contour*. Modules with several input ports may be able to accept two or more different data types, one on each port. For example, *Streakline* has input ports for a lattice and a pick and an output port for geometry. *Render* has input ports for geometry and parameter data and output ports for lattice, geometry, and pick data.

Mixed Data types

Table 2−1 lists some of the IRIS Explorer modules according to the data type (Lattice, Pyramid, Geometry or Pick) they can accept on an input port.

Some modules that accept lattices are further defined by whether they can accept 1D, 2D, or 3D lattices, or a combination. For details, refer to the input port specifications of each module in the *IRIS Explorer Reference Pages*.

Input Port Data	Modules
Lattice	Ball, BallStick, BoundBox, ChannelSelect, ColorXform, Contour, CropLat2D, CropLat3D, DiffLat, DisplaceLat, GenerateColormap, Gradient, Graph, Histogram, InterpLat, Interpolate, IsosurfaceLat, LatFunction, LatToGeom, LatToPyr, Legend, MagnitudeLat, Mixer, MultiSlice, OrthoSlice, PrintLat, ProbeLat, PyrToGeom, SampleCrop, ScaleLatNode, Shell, Slice, SubSample, Transform, Triangulate2D, Triangulate3D, VectorGen, VolumeToGeom, WireFrame, WriteLat
Pyramid	AtomicSurf, BallStick, BoundBoxPyr, CullPyr, CropPyr, IsosurfacePyr, PrintPyr, PyrToGeom, PyrToLat, Transform, WritePyr
Geometry	Render, WriteGeom
Pick	Annotation, Ball, CropLat2D, CropLat3D, InterpLat

Table 2−1 Input Data Accepted by Modules

Table 2−2 lists some IRIS Explorer modules according to the data type (Lattice, Pyramid, Geometry or Pick) they can produce on an output port.

Output Port Data	Modules
Lattice	AtomicSurf, ChannelSelect, ColorXform, CropLat2D, CropLat3D, DisplaceLat, DisplayImg, GenerateColormap, GenLat, Gradient, Histogram, Interpolate, LatFunction, MagnitudeLat, Mixer, OrthoSlice, PyrToLat, ReadLat, ReadPlot3D, Render, SampleCrop, ScaleLatNode, Shell, Slice, SubSample, Transform,

	TransformGen, WaveFormColormap
Pyramid	CullPyr, CropPyr, LatToPyr, ProbeLat, ReadPyr, ReadPDB, Transform, Triangulate2D, Triangulate3D
Geometry	Annotation, Ball, BallStick, BoundBox, BoundBoxPyr, Contour, CropLat2D, CropLat3D, DrawText, IsosurfaceLat, IsosurfacePyr, LatToGeom, Legend, MultiSlice, PyrToGeom, ReadGeom, VectorGen, VolumeToGeom, WireFrame
Pick	Render

Table 2−2 Output Data Produced by Modules

Note: All modules that have input ports for parameters also have output ports for them.

Image Processing Modules

The image processing modules, listed in **Table 2−3**, manipulate lattices. They accept only the lattice data type on their input ports and/or produce it on their output ports.

Modules		
AbsImg	AddImg	AndImg
BlendAlphaImg	BlendImg	BlurImg
CompassAngleImg	CompassDirImg	CompassImg
ComplementImg	DisplayImg	DivImg
ExpImg	ForwardFFTImg	FourierConjgImg
FourierCrossCorrImg	FourierDivImg	FourierExpFltImg
FourierGaussFltImg	FourierMagnitudeImg	FourierMergeImg
FourierMultImg	FourierPhaseImg	FourierPwrImg
GaussBlurImg	GrayScaleImg	HistEqualImg
HistNormImg	HistScaleImg	InverseFFTImg
LaplaceEdgeImg	LogImg	MaxFltImg
MaxImg	MedianFltImg	MinFltImg
MinImg	MultImg	NegImg
OrImg	PowerImg	RankFltImg
ReadImg	RobertsEdgeImg	RotZoomImg
SGIPaletteImg	SharpenImg	SobelEdgeImg
SqRootImg	SquareImg	SubtractImg
ThreshImg	WriteImg	XorImg
ZoomImg		

Table 2−3 Image Processing Modules

2.9. Setting Module Parameters

Module parameters allow you to set and change scalar values for each module. For example, a parameter called "Min Range" that accepts integers may let you set the minimum value for a range of temperature values for the module.

Parameters are controlled through *widgets* on the module control panel. Widgets are mechanisms for regulating a module’s parameter values. The parameter names are given on the module’s input port menu and on the full−scale control panel. A single control panel may contain a number of widgets, each controlling a different parameter.

Widgets include buttons, sliders, dials, text slots, scrolled lists, radio buttons, option menus, and file browsers, as well as drawing areas.

2.9.1. Buttons

Buttons are rectangular or diamond−shaped widgets that let you select a value or switch a specific parameter on or off. They come in two styles (see **Figure 2−19**):

- The one−shot, or single, button can be on or off. It is usually oblong and contains a label on the button itself.
- Diamond−shaped radio buttons offer a set of choices, only one of which is valid at a given time. They have a colored center when on, and are convex when off.

Use the left mouse button to click the buttons on and off.

Figure 2−19 Button Widgets

2.9.2. Sliders

Sliders (see **Figure 2−20**) are either horizontial or vertical bars that let you increase or decrease the value of a parameter in discrete steps. Sliders generally have integer values.

Each slider has:

- a title giving the name of the parameter
- a "thumb" that moves from one end of the bar to the other allowing the sliders value to be changed
- text slots showing the minimum, maximum, and current values of the parameter.

Figure 2−20 Slider Widgets

You can change any one of these values by clicking on the text slot to highlight it, typing in a new value, and pressing **`<Enter>`**.

To change a parameter value, place the mouse cursor on the "thumb" and hold the left mouse button down while you push or pull the slider along its bar. The current value changes as the thumb moves. When you release the mouse button, the new value is sent to the module.

To move the bar to the cursor position, place the cursor in the slider slot and click on the middle mouse button.

2.9.3. Dials

Dials are circular widgets that allow the continuous adjustment of a parameter value between set limits (see **Figure 2−21**). Like sliders, each dial has a title and slots showing minimum, maximum, and current values. You can type new values into any slot (remember to press **`<Enter>`**), and you can also change the current value by turning the dial pointer. Dials generally have float values.

Figure 2−21 Dial Widgets

The dials have a vernier action, in that you can use the pointer for coarse settings. Place the cursor on the pointer and push it clockwise or counterclockwise while holding the left mouse button down. The values change linearly from minimum to maximum through a single revolution of the pointer.

To make a fine adjustment to the parameter value, place the cursor on the dial center and push it clockwise or counterclockwise. The values increment or decrement in minute steps, and the inner dial revolves 50 times for one revolution of the pointer. This is useful for fine−tuning adjustments to a parameter that spans a large range of values, or one that increments its value in minute steps.

2.9.4. Text Type−in Slots

Text type−in slots (see **Figure 2−22**) accept alphanumeric characters.

Figure 2−22 Text Type−in Slots

To type in data values or filenames, click once or twice in the slot until the upright text cursor appears. If necessary, the slot scrolls to the left to give you enough room for your text.

Text type−in slots appear in some input modules as part of a file browser widget (see below).

2.9.5. File Browsers

The file browser widget appears in modules that read or write files from or to the file system. To use the file browser, you must have the full−scale control panel of the module open (see **Figure 2−23**).

Figure 2−23 A File Browser

The file browser has two windows:

- The top one, Contents, displays the contents of the current directory, including files and subdirectories.
- The bottom one, Selection, shows the current selection (file or subdirectory).

You can list the files in any subdirectory on your system. Rather than having to type a filename into a text slot on the module, you can simply select the file from the file browser. You can move up a directory hierarchy by selecting the "../" entry (the first one in the list).

To select a file or directory from the Contents list, double−click on the name.

If your selection is a directory, the file browser displays its contents in the Contents window and its path and name in the Selection window. If it is a file, it sends the filename to the module. The filename appears in the Diminutif control panel text slot and the module fires.

To open a file, you can also click once on the filename to select it, and then click on the *OK* button. The filename appears in the text type−in slot of the Diminutif control panel, and the module fires.

You can use the file browser to traverse directories in a completely different directory tree, in two ways:

- Simply type the directory or filename into the text slot. This takes you directly to your destination.
- Use the Current Directory option menu (see **Figure 2−23**). Each directory entry in this hierarchy contains a cascading menu for all its contents.

  Click on the Current Directory option menu, choose the item you want from each menu, and click on it to get to the next menu level.

2.9.6. Option Menus and Scrolled Lists

Option menus and scrolled lists let you select one or more items from a group of options.

Option menu buttons present a menu from which you can choose one option at a time. The option may be a value, a range of values, or a property, such as uniform or curvilinear (see **Figure 2−24**). The option menu buttons are rectangular, with a small raised bar on the right.

Figure 2−24 Option Menu Widgets

Click anywhere on the button to display the menu and select the item you want.

A scrolled list widget lets you select one or more items from a list of options that may extend beyond the current list window. You may be able to choose one option or several options at a time, depending on how the properties of the particular list were defined when the module was created.

Click on an item to select it. If the scrolled list allows you to select more than one item, click on all the items you want to select.

Figure 2−25 Scrolled List Widget

2.9.7. Drawing Areas

A drawing area is a blank rectangle on the control panel in which an image or visual object can be displayed (see **Figure 2−26**). For example, the modules *DisplayIng* and *Render* have drawing areas on their control panels.

Figure 2−26 A Drawing Area Widget

You can control the size, appearance, and orientation of the subject in the drawing area by using the widgets on the control panel. These vary from module to module, depending on the module's function.

The window may take up varying amounts of space on the control panel. When you enlarge the control panel, the drawing area may be enlarged proportionally. Click on the *Maximize* button of the full−scale control panel to open up the control panel to full−screen size.

Example 2−4 Experimenting with Widgets

This example illustrates how you can affect an object in *Render* by using widgets to change parameter values. Launch the map named *widgets* (see **Figure 2−27**) by dragging and dropping it in the Map Editor. It contains five modules:

- *ReadLat* reads in a lattice data file
- *Contour* generates a contour surface of the volume
- *IsosurfaceLat* generates an isosurface of the volume
- *WireFrame* generates a box that defines the boundaries of the volume
- *Render* visualizes the objects in its window

Note: On workstations without two−sided lighting the isosurface in *Render* may be dark. Change the Flip Normal? option on the *IsosurfaceLat* control panel from `No` to `Yes` to display the geometry properly.

1. Open the full−scale control panels for *Contour*, *IsosurfaceLat*, and *Render* by clicking on the *Maximize* button.
2. Change the dial setting on *IsosurfaceLat*. Notice what happens in the *Render* window.
3. Change the slider position on *Contour* module. Note the change in the *Render* window.

Figure 2−27 Experimenting with Widgets

2.10. Organizing the Module Librarian

Modules and maps are organized into categories in the Module Librarian. **Figure 2−28** shows one category, called "Modules." You can reorganize the contents of these categories, displayed in scrollable columns, as described below.

Figure 2−28 Module Category

2.10.1. Creating New Categories

You can organize modules differently for each system on which you run IRIS Explorer. For example, you may create a category for each remote host, or for all the complete maps on your system.

To create new categories in the Module Librarian, select " New" from the Categories menu. When the pop−up menu appears, enter the name you want for the new category. This adds the name to the Categories menu. Select the name from the menu and an empty column of that name appears in the Librarian.

To hide categories in the Librarian, click on the category title with the right mouse button and select "Hide" when it appears.

To redisplay the category, select the category name from the Categories menu.

To update a category, select "Update" from the Display menu after you have added new modules or maps.

Note: You cannot delete categories, only hide them.

When you choose "Update" from the Display menu, new categories that have not been saved already are deleted. To save new Librarian categories, see **"Saving the Librarian Configuration"**.

2.10.2. Moving Modules Around

You can move maps or modules from an existing category to a new one by dragging and dropping them in place. Items in a category can be ordered by user preference, which is not necessarily alphabetical. You may want to put frequently launched modules at the top of the list and leave the others lower down.

Plan the order of your modules before you move them. New modules are added at the end of the list, and once they are in place, they cannot be moved again within the category.

2.10.3. Storing Modules on the Shelf

The Librarian shelf (see **Figure 2−2**) is a storage area for modules. It is like a bookshelf where you place books to which you often refer. For example, you might stack modules that you use a lot, such as *Render*, *DisplayImg*, or modules of your own creation, on the shelf. Shelved modules are still available from the Module Librarian as well.

To place a module on the shelf, use the "drag and drop" technique. Once a module is on the shelf, you can launch it immediately or leave it there to be launched later. You can launch it repeatedly from the shelf.

2.10.4. Removing Modules from the Librarian

Use the "Delete" option from the Module Launch menu (see **Figure 2−3**) to remove a module or map from a category or from the Librarian shelf.

You can restore a module you accidentally removed by selecting "Update" from the Display menu. It restores the original setting, so you also lose any unsaved changes you have made to the Librarian.

2.10.5. Saving the Librarian Configuration

Once you have arranged the IRIS Explorer modules and maps in categories and on the Librarian shelf to your satisfaction, you can save the new organization for future operations.

1. Select "Save Librarian Config" from the Module Librarian File menu.
2. When the "Save Configuration" file browser appears, save the configuration with a new filename.
3. Use a text editor to incorporate the configuration changes in the new file into the *.explorerrc* file in your home directory. If you do not currently have one, copy /usr/explorer/Explorer.config to this file. The *Category* command in this file governs the layout of categories.

   However, the *.explorerrc* file contains more information than IRIS Explorer provides when you save a new configuration, so you should edit the file rather than overwrite it. Add lines of the form

   ```
   category name module...
   ```

   for each new category or

   ```
   category shelf module..
   ```

   for each module on the Librarian shelf.

To set the module path, that specifies the directories to search for modules, and for further configuration commands see **"The .explorerrc Configuration File"** in Appendix A.

2.11. Common Problems

When all the launched modules have been connected and your map is operational, the modules fire and an object becomes visible in the *Render* or *DisplayImg* window—in theory, perhaps. In practice, this does not always happen. Here are a few common problems and ways to fix them.

Note: If a module port is "required," then it must be receiving valid data from an upstream module in order to work properly. All ports are required unless marked as "Opt(ional)."

2.11.1. You Need a Larger Memory Arena than Your System Can Give You

A message to this effect may appear when IRIS Explorer starts up. It means that you do not have enough disk space to run IRIS Explorer properly. You can:

1. Clean off your disk to make more disk space for the shared memory arena.
2. Create a temporary directory (*tempdir*) on another disk on your machine if you have one (for details, see **"The .explorerrc Configuration File"** in Appendix A).
3. Reduce the size of your arena, which is the amount of disk space necessary to hold all the IRIS Explorer files, before you run IRIS Explorer (see the reference in step 2).

2.11.2. No Object Is Displayed in the Render Window

If nothing appears in your Render window, try the following procedure:

1. Make sure all the required input ports on each module are connected to upstream modules that can send them valid data.
2. *Render* might be looking in the wrong direction. Click the right mouse button in the *Render* window (which brings up a pop−up menu) and select "Functions" then "View All."

   This brings an object that may be out of the viewing area into the center of the window.
3. The module might not be reading in the data from disk. Make sure that modules that need data from files, such as *ReadImg*, have been given access to those files.
4. The module might be disabled. Enable the module by bringing up the pop−up menu from the control panel and selecting "Enable." You can also select all the modules by using the Edit menu in the Map Editor, then choose "Enable."
5. The widgets might not be set to values which can produce data. Check the settings: turn the dial or move the slider if you have an active threshold parameter.

2.11.3. No Change in Object with Change in Parameters

If you have changed object parameters, but the changes do not appear in the object, try the following procedure:

1. Enable the module by bringing up the pop−up menu from the control panel and selecting "Enable." You can also select all the modules by using the Edit menu in the Map Editor, and choose "Enable" from the Edit menu.
2. Make sure all the required ports have data by connecting them to valid ports in other modules.
3. Some modules, such as *GenerateColormap*, have "Build/Run" toggle buttons. If the module is in "Build" mode, it will not fire and send data downstream. Change the mode to "Run."

2.11.4. Viewing Lattice Data in the Render Module

If the lattice is 1D or 2D, such as an image:

1. Connect the input port of a *LatToGeom* module to the "Output − − Lattice" port of the module that is reading in or generating lattice data.
2. Connect the *LatToGeom* module’s "Output − − Geometry" port to *Render.*

If the lattice is 3D, such as a volumetric dataset:

1. Select a module that will provide the visual representation you want, for example, *IsosurfaceLat, Contour, Slice, OrthoSlice, WireFrame,* or *VolumeToGeom.*
2. All these modules except *Slice* and *OrthoSlice* produce geometry. They can be connected to *Render* by wiring their "Output −− Geometry" ports to *Render.* The two slicers produce lattices, which must be passed through a *LatToGeom* module first.

   See **"Slicing Volumetric Data"** in Chapter 3 for more information about using slicers.

2.11.5. The Module Librarian Does Not Display a Saved Map

If you want a map to be displayed in the Module Librarian after you use "Update," the map must be in a directory that the module Librarian recognizes.

To ensure this:

1. Check that the directory in which you saved the map is in the module path. You can check this and/or add subdirectories to the module path by editing the *.explorerrc* file. See **Appendix A** for details.

   The default directory is */usr/explorer/maps/*.*
2. Check that you used the correct pathname when you saved the map. It might be in another directory that is not in the module path.

2.11.6. A Saved Map is Empty When Reopened

You need to save the map again. To prevent this from happening a second time:

1. Make sure you use "Save All" from the Module Librarian File menu to save modules that are not highlighted.
2. If you save certain modules only with "Save Selected," the title bars of those modules must be highlighted, otherwise the map is saved without any modules in it.

2.11.7. When You Try to Destroy a Module, Nothing Happens

When you select "Destroy" from the pop−up menu, IRIS Explorer sends the module a message to exit. If the module is busy computing, it may not get the message for a while. If you want it to quit immediately, select "Destroy" again from the module pop−up menu. This sends a "kill" message to the module’s controller, which then shuts down the module without waiting for a response from the module itself.

2.11.8. You Cannot Launch a Module Librarian Remotely

If you get messages from the Local Controller (LC) or Global Controller (GC) when you are trying to launch a Module Librarian for a remote system, there is probably a command generating output from your *.cshrc* file. You will see the output from the command in the error message. For example, some people print a "message of the day" or a witty saying from the *.cshrc* file.

This output interferes with the communication between the Local Controller, which is needed by the remote Module Librarian, and the Global Controller, which starts up the Local Controller on the remote system.

The offending command and its output must be removed before you can run the Module Librarian on that remote system.

Note: The best place for such output is the *.login* file. Other programs besides IRIS Explorer get confused and do not work properly when there is output from the *.cshrc* file.

2.11.9. You Get a Shared Library Error When You Launch Remote Module

Error messages that mention a shared library indicate that some part of IRIS Explorer is not properly installed on the remote system. For more information, see **"Remote Execution of Modules"**.

2.11.10. You Cannot Connect Two Modules, or a Module Produces Unexpected Data

You can check this in two ways:

1. The module may be connected to an incompatible port and is therefore receiving the wrong data type. Bring up the input and output port pads and check the port connections.
2. Verify that the module is producing the desired data type by connecting it to *PrintLat* or *PrintPyr* and printing the output. IRIS Explorer produces an ASCII file showing the structure and contents of the data, which you can look at to track down any errors.

Chapter 3

Using Modules

This chapter describes in detail some of the modules you are likely to use often, as well as a few of the more complex modules. It explains how to:

- create a colormap
- generate various types of lattices
- slice volumetric data
- create a histogram from your data
- display and manipulate images
- use *Render* to visualize your data
- create loops in maps
- create a single executable process from a group of ImageVision Library (IL) modules

3.1. Overview

Once you have data in IRIS Explorer, you can perform many operations on it to make the visualizations more meaningful and easier to interpret. For example, you can emphasize areas of common density or intensity by redefining the color palette, or examine more closely a specific subset of the data by cutting into, or slicing, the larger dataset. You can also examine the relative density of data points within a dataset by drawing graphs or histograms to show their distribution.

When you have a geometric object, or scene containing several objects, displayed in the *Render* window, you can move, highlight, and otherwise manipulate the objects in many ways to provide you with new insights into the nature of your data. You can also animate an object and record its movements for later display.

Loop controller modules provide you with a means of constructing single and nested loops in your maps. For example, you may need a loop to capture successive frames in an animation sequence.

There are many operators for processing images, embodied in the ImageVision Library modules listed in **Table 3−1** under "Process Images." You can display and record these processed images in *DisplayImg*.

The modules supplied with IRIS Explorer vary widely in function and complexity, providing you with the means to manipulate and process data in many different ways. More information about specific modules is available in these places:

- **Table 3−1** lists some of the modules grouped according to function.
- **Table 2−1** through **Table 2−3** in Chapter 2 list some of the the modules according to the kind of data they accept and deliver; for example, lattice or geometry data.
- The *IRIS Explorer Reference Pages* list all modules alphabetically and sets out the specifications of each one.

Not all the modules listed are available on all systems. The *IRIS Explorer Reference Pages* indicates the modules available for general distribution.

A selection of modules is listed by function in Table 3−1.

Footnote: Note that the multimedia modules may not be available on all platforms as they were originally designed for Silicon Graphics systems.

Module Function	Module Names
Read data in or create it from scratch	ComposePyr, GenerateColormap, GenLat, MopacView, Read2Da, ReadAudio, ReadAVS, ReadAVSfld, ReadAVSimage, ReadAVSucd, ReadAVSvol, ReadGeom, ReadHDF, ReadImages, ReadImg, ReadLat, ReadMovieBYU, ReadNTF, ReadPDB, ReadPhoenics, ReadPyr, WaveFormColormap
Process images	AbsImg, AddImg, AndImg, BlendAlphaImg, BlendImg, BlurImg,

	CompassAngleImg, CompassDirImg, CompassImg, ComplementImg, DivImg, ExpImg, ForwardFFTImg, FourierConjgImg, FourierCrossCorrImg, FourierDivImg, FourierExpFltImg, FourierGaussFltImg, FourierMagnitudeImg, FourierMergeImg, FourierMultImg, FourierPhaseImg, FourierPwrImg, GaussBlurImg, GrayScaleImg, HistEqualImg, HistNormImg, HistScaleImg, InverseFFTImg, LaplaceEdgeImg, LogImg, MaxFltImg, MaxImg, MedianFltImg, MinFltImg, MinImg, MultImg, NegImg, OrImg, PowerImg, PseudoColorImg, RankFltImg, RobertsEdgeImg, RotZoomImg, SGIPaletteImg, SharpenImg, SobelEdgeImg, SqRootImg, SquareImg, SubtractImg, ThreshImg, XorImg
Generate data from other data	AtomicSurf, ChannelSelect, ClipPyr, CropPyr, DiffLat, DisplaceLat, Gradient, Gradient3D, Interpolate, LatFunction, LatToPyr, MagnitudeLat, MinMax, Mixer, MultiChannelSelect, MultiSlice, OrthoSlice, OutLine, ProbeLat, PyrToLat, Ruler, SampleCrop, SamplePyr, ShrinkPyr, ScaleLatNode, Shell, Slice, SubSample, TriangulateDelauney, Triangulate3D
Develop geometric representations	Annotation, Ball, BallStick, BoundBox, BoundBoxPyr, Contour, GnuPlot, Graph3D, IsosurfaceLat, IsosurfacePyr, LatToGeom, Legend, MoleculeBuilder, MultiSlice, NAGAdvectSimple, QueryLat, PyrToGeom, SliceLat, SlicePyr, Streakline, VectorGen, Vectors, VolumeToGeom, WireFrame
Create images	DisplayImg, Histogram, Graph, RealityRender, Render, RenderRemote, TransformGen
Multimedia (audio and video)	AnimateCamera, AudioIn, AudioOut, ReadAudio, VideoControl, VideoDevice, VideoLabInWin, VideoStarterIn, VideoStarterInWin, VideoStarterOut, WriteAnimation, WriteAudio
Control looping sequences	AnimateCamera, For, Repeat, Render, Trigger, While
Miscellaneous	CompressPyr, DiffLat, ExpandPyr, NAGContour, NAGGraph, SwitchGeom, Timer, Trigger
Write data to disk	PrintLat, PrintPick, PrintPyr, WriteGeom, WriteImg, WriteLat, WritePyr

Table 3−1 Modules Listed by Function

3.2. Creating Colormaps

When you are looking at large amounts of data, it helps to be able to distinguish significant features easily, and a convenient way to differentiate a range of values is by using color. The *GenerateColormap* module control panel (see **Figure 3−1**) lets you select a color space, manipulate the color bands, and send the results to *Render*, where the visualized data object is colored according to its data values.

Figure 3−1 The *GenerateColormap* Module

GenerateColormap has two input ports, both of which accept the lattice data type, and several input parameters that are connected to widgets on the module control panel. The first input port accepts a colormap, which means you can read a colormap data file that contains preset parameters into the map. The *volume* map in Chapter 1 provides an example.

The second input port accepts data in the form of a lattice, which means you can feed a lattice into the colormap to set the minimum and maximum limits of the domain. Both these ports are optional.

The output port produces a colormap in the form of a 1D lattice. You can connect *GenerateColormap* to any module that has a "Colormap" input port. These include *BallStick*, *Contour*, *LatToGeom*, *PyrToGeom*, and *VolumeToGeom*.

Note: You can have several *GenerateColormap* modules in a map if your X server contains enough entries in its colormap table. See **"Running the X Server"** in Appendix A for more information.

3.2.1. Resetting the Color Bands

You change the slope, and hence the value, of a color band in order to adjust the color spectrum in the colormap. In this fashion, you can emphasize certain subsets of your data and block out others. This is easiest to do with the full−scale *GenerateColormap* control panel (see **Figure 3−1**).

The control panel has three windows:

- The palette window shows the spectrum of colors currently associated with the data range. The colormap has 256 entries which are spaced uniformly through the range.
- The editing window shows the relationship between the color bands. You can use the RGB (Red/Green/Blue) or HSV (Hue/ Saturation/Value) color systems to create a colormap.
- The scale window shows the range of data values for the map. You can change the minimum and maximum values for the domain by typing new values into the text slots on the dials.

Selecting an Output Mode

Use the *Output?* option menu to select a mode in which to work. The *Build* option disables the module temporarily while you edit the color bands. This is useful if you want to experiment without the module constantly firing and sending data to the modules downstream.

The *Run* option re−enables the module. All changes are reflected in the color palette as you make them, and are simultaneously sent to the next module downstream.

Setting the Domain Limits

Each map processes data with values over a specified range. You can set the limits of the range affected by *GenerateColormap* by turning the *Min Domain* and *Max Domain* dials. Each dial has two text slots that you can use to define the range for minimum values and maximum values.

To change a domain value (min, max, or current), click on the text slot to highlight it and type in the new value, then press **<`Enter`>**. The new value appears in the text slot and in the scale window.

Selecting a Color Band

You can edit only one color band at a time. To select a color band:

1. From the Color Space option menu, select *RGB* or *HSV*, depending on which system you want to use.
2. From the Color Band option menu, select the color band you want to adjust.
   - In the RGB color space, your options are *Red*, *Green*, *Blue*, and *Opacity*.
   - In the HSV color space, your options are *Hue*, *Saturation*, *Value*, and *Opacity*.

All four bands appear in the editing window. In the HSV color space, the bands might be superimposed on one another, therefore more difficult to see. The active band is named on the option menu button and stands out in the editing window because the control points, shown as square boxes along the band, are enlarged.

Adding and Removing Control Points

You edit color bands by adding control points at intervals along their length and then dragging the points around in the editing window. The points appear as filled boxes (see **Figure 3−2**).

Figure 3−2 Adding Control Points to a Color Band

Use the *Edit Mode* option menu to add and remove control points. The *Insert* and *Delete* options default to *Move* after every insertion or deletion of a point.

To add a new control point to a color band, select *Insert* from the *Edit Mode* option menu and click on the color band at the place where you want the new point. The point is centered between the existing points on that segment of the color band. Move it by dragging on it.

The closer together the points are, the more precise the changes that you can make to the shape of the curve. However, you cannot move a point past the one to its left or right.

To remove an existing control point from a color band, select *Delete* from the *Edit Mode* option menu and click on that control point.

Note: You can delete any point except the endpoints.

Using Slope Control Points

Each control point (except the end−points) has two empty squares associated with it, which control the slope of the curve at a given point. They are the derivatives of the curve at that point.

When you select a control point by clicking on it, the slope control points appear on either side (see **Figure 3−3**). They disappear when you select another point.

Figure 3−3 Slope Control Points

To alter the slope of the curve in the vicinity of a point, click and drag on the slope control points.

If you want to sharpen the curve toward the lower boundary of the window and the position of the slope control point blocks you, simply add another point to the line near the first point and use its derivatives to alter the slope.

3.2.2. Using Custom Colormaps

GenerateColormap creates a rainbow−like colormap when you first launch it. You can use a different colormap as your starting point if you wish.

To read in a custom colormap, first launch a *ReadLat* module and connect it to the input port of *GenerateColormap*. Then use the file browser in *ReadLat* to read in the colormap file. The *GenerateColormap* module will approximate the input colormap as closely as it can. You can use control points to edit the color curves that it creates.

IRIS Explorer has another module, *WaveFormColormap*, which uses ramp waves of variable frequency, rather than direct manipulation, to establish a relationship between data values and colors. For more information, refer to *WaveFormColormap* in the *IRIS Explorer Reference Pages.*

Creating a Custom Colormap

A colormap is a lattice with a specific format: it is a 1D, 4 vector, float uniform lattice with 2 to 256 components. You can create such a lattice by saving a map produced by *GenerateColormap* or *WaveFormColormap*, or by any other module or program that can produce a lattice in this format.

Saving a Colormap

You can save a colormap to disk by connecting a *WriteLat* module to *GenerateColormap*'s output port. Then use the file browser to give the saved colormap a filename.

Example 3−1 Manipulating a Colormap

This example illustrates how to generate a colormap for a lattice and alter the relationship between the data values and the color spectrum.

1. Launch these modules from the Module Librarian:

   GenLat, *LatToGeom*, *Render*, and *GenerateColormap*

2. Wire them up as follows:

GenLat to *LatToGeom* (Output − − Lattice to Input)

GenLat to *GenerateColormap* (Output − − Lattice to Data In)

GenerateColormap to *LatToGeom* (Colormap − − Lattice to Colormap)

LatToGeom to *Render* (Output − − Geometry to Input)

3. Change the *GenLat* settings to create a 2D Curv Full Cyl curvilinear lattice with only 1 data variable. You should have a map that resembles the one in **Figure 3−4**. To make it more similar, increase the size of the lattice in *GenLat*. *GenerateColormap* starts in HSV mode.

Figure 3−4 A Map using *GenerateColormap*

Click on the *Output?* option menu and select *Run*. Then do the following and look at the results:

1. Add two points to the red color curve (Hue), three points to the blue (Saturation), and two to the green (Value).
2. Move the points around as far as they will go. See what happens to the objects in *Render.*
3. Switch from *HSV* to *RGB*. Observe how the position of the color curves has changed, even though the color spectrum is still virtually the same.
4. Edit a color band in *Build* mode, then in *Run* mode.

3.3. Generating a Lattice

The *GenLat* module (see **Figure 3−5**) enables you to generate synthetic lattice data for test maps. You can create a large variety of lattices of different sizes, shapes, contents, and representations in order to drive other modules that accept lattices. For example, you can easily verify that a given module performs properly on any type of input lattice. Simply connect *GenLat* to the module, and then start generating different types and styles of lattices.

GenLat has no input ports and only one output port for the generated lattice. You can connect it to any module that accepts the type of lattice you want to generate.

Figure 3−5 The *GenLat* Module

3.3.1. Setting Lattice Parameters

GenLat creates a single lattice each time it fires. You can control the shape and contents of this lattice by manipulating the widgets on the control panel.

Controlling Lattice Size

Two of the most important parameters of a lattice are its size and dimensionality. If you increase these values, *GenLat* creates larger lattices, a process which takes commensurately more time. *GenLat* always creates lattices with the same number of elements on each side, for example, 8x8x8 (3D) or 25x25 (2D).

Note: If you need test lattices that do not have the same number of elements on each side, you may achieve this by combining *GenLat* with a module such as *SampleCrop*.

To alter these values, move the *Dimensions* and *Size* sliders at the bottom of the control panel. You can also type new values into the slider value areas to get an exact value.

Lattice Coordinates

All lattices created by *GenLat* have coordinates, which can be represented as uniform, perimeter, or curvilinear. The data values of the coordinates govern the visual shape of the lattice. For example, the "torus" shape produces a lattice whose coordinates are polar in the first two dimensions and linear in the others.

To set the visual shape of a lattice, click on the *Coord Type* option menu and select an option.

To set the coordinate representation for a lattice, click on the *Coord Representation* option menu and select an option.

You can generate a curvilinear lattice with coordinates that are uniform in computational space. This option is useful for testing modules which require such lattices on their input ports.

Certain combinations of coordinate representation and coordinate style do not make sense, though, because only curvilinear representation can be used to store the position information for the torus and related coordinates.

Data Values

The data values in lattices can be any of the primitive data types (see **"Data Structures"** in Chapter 7). These include byte, short, long, float, and double.

To set a data type, click on the *Data Type* option menu and select an option.

Data values are scaled either in a fixed range of zero to 255 or their maximum possible range for the integer types (char, short, or long). However, a range of −1.0 to 1.0 is used for float and double.

To set the data range, click on the *Coord Range* option menu and select an option.

Note: Using the maximum possible range for the long primitive type can generate very large numbers.

The values of the data can be generated by one of several functions. The default is "sines," which fills the data space with the product of sine functions evaluated over a certain number of cycles in each dimension. The other functions are:

- Random data
- Ramps (modulo function)
- Simulated gravity wells

DataVec Length indicates the number of components at each node of the lattice.

3.3.2. Displaying a Lattice

To display lattice data in IRIS Explorer, you can:

- cut a slice using any of the three slicers (see **"Slicing Volumetric Data"**)
- create an isosurface with *IsosurfaceLat*(see **Figure 4−15** in Chapter 4)
- create contour lines using *Contour*(see **Figure 4−15**)
- create vectors using *VectorGen*

Since *Render* accepts only Geometry data, you must wire modules that produce lattice or pyramid data to *LatToGeom*, *VolumeToGeom*, or *PyrToGeom*, which in turn can be connected to *Render.* You can also send a lattice directly to *DisplayImg.* **Figure 3−4** shows a lattice passed directly to *LatToGeom*.

3.4. Slicing Volumetric Data

A useful way to examine data is to make a 2D cut through a 3D volume, that is, to slice the volume and look at the data in that slice.

There are several ways to do this, depending on the type of data you start with and the results you want to produce. **Table 3−2** shows which slicers you can use for each combination of lattice types.

	Input		
Output			
	Uniform	Perimeter	Curvilinear
Uniform	OrthoSlice, Slice	−	−

Perimeter	−	OrthoSlice	−
Curvilinear	OrthoSlice	OrthoSlice	OrthoSlice
Pyramid	−	−	ProbeLat

Table 3−2 Slicing Options

You can also use the DataScribe to create a single module which combines all these functions. See **Chapter 7** for more information.

3.4.1. Using OrthoSlice

The simplest way to slice data is to use *OrthoSlice* (see **Figure 3−6**). This module extracts a 2D lattice from a 3D uniform, perimeter, or curvilinear lattice along one of the orthogonal axial planes. It produces the same type of lattice as it receives, except that the lattice is 2D. For example, if you feed in a 3D curvilinear lattice, *OrthoSlice* produces a 2D curvilinear lattice.

Footnote: For more information on lattices, see Chapter 3, "Using the Lattice Data Type" in the *IRIS Explorer Module Writer's Guide*

OrthoSlice cuts along the *i, j,* and *k* axes in computational space. For a uniform or perimeter lattice, these correspond to the *x, y,* and *z* physical axes. Curvilinear lattices have a more complicated relationship between computational and physical space.

Figure 3−6 The OrthoSlice Module

To select an axis for cutting a slice, click on the *I Slice, J Slice* or *K Slice* radio button (see **Figure 3−6**).

The slice number corresponds to the node number. The slice is taken along the axial plane at that particular node. You can select a number up to one less than the number of nodes.

Figure 3−7 shows a map that uses *OrthoSlice* to display part of a lattice. You can use any lattice data. For example, you can replace *GenLat* with *ReadLat* and read in an external lattice data file if you wish. A good choice is */usr/explorer/data/lattice/testVol.lat.* You can also color the output with a colormap by wiring *GenerateColormap* to *OrthoSlice.*

The modules in **Figure 3−7** are *GenLat, OrthoSlice, LatToGeom,* and *Render.* The slice is taken along the *i* axis of a half−cylinder.

Figure 3−7 Slicing a Volume with *OrthoSlice*

3.4.2. Using Slice

The *Slice* module (see **Figure 3−8**) extracts a 2D uniform slice out of a 3D uniform lattice.

Figure 3−8 The Slice Module

The slice axis need not be orthogonal to the axes of the lattice. The *Slice* module has an input port that accepts transformation coordinates from *TransformGen.* These coordinates define the plane of the slice. You can:

- Use the widgets on *TransformGen* to angle the slice in any direction you choose.
- Use the *Offset* dial on the *Slice* control panel to position the slice plane perpendicular to its axis in the volume. The plane is offset from the center of the volume.
- Use the *Resolution* slider on the control panel to set the scale for the slice lattice. It determines the number of nodes.

The size of the slice lattice can be larger than the original lattice, depending on the slice plane. At certain angles, the slice may extend beyond the bounds of the original lattice (see **Figure 3−9**), but the lattice is clipped to keep the number of samples outside the bounds to a minimum.

Figure 3−9 Slice Extended Beyond Lattice Bounds

Slice has two outputs: a 2D slice lattice and a transform to *LatToGeom*. The transform sets the orientation of the slice lattice with respect to the parent in *Render*. It must be connected to *LatToGeom*, which correctly positions the slice. **Figure 3−10** shows a map using *Slice*.

Figure 3−10 Slicing a Volume with *Slice*

3.4.3. Using ProbeLat

The *ProbeLat* module works only with curvilinear lattices, and its action is not restricted to plane surfaces (see **Figure 3−11**).

Figure 3−11 The ProbeLat Module

It takes slices from a curvilinear lattice in the form of:

- a single planar slice
- a trihedral, which is three mutually perpendicular planes (see **Figure 3−12**)
- a paddlewheel, which is three planes radiating from a center at 120^{o} to one another (see **Figure 3−12**)

The module accepts a transform from *TransformGen* which sets the orientation of the slice. The slice is output in the form of a pyramid.

You can select the form the probe takes (Probe Type) and the area of the slicing plane (Clip Size) from the *ProbeLat* control panel.

Note: For pyramids you can use *SlicePyr* and *ClipPyr*.

Figure 3−12 Probe Forms

Figure 3−13 shows a paddlewheel probe generated by ProbeLat.

Figure 3−13 Slicing a Volume with *ProbeLat*

3.5. Generating a Histogram

You can examine data distribution in a lattice by connecting any module that has a lattice output into the *Histogram* module and generating one or more histograms. The histogram (see **Figure 3−14**) shows how many times a data value occurs in the input lattice.

Figure 3−14 The *Histogram* Module

The *Histogram* module has one lattice input port and one lattice output port. The input port accepts any type of input lattice, and the output port produces a count of the input data values in the form of a lattice.

3.5.1. Displaying Histograms

The *Histogram* control panel has a display window that shows the histogram itself and scale windows that show the domain (horizontal, or *x*, axis) and range (vertical, or *y*, axis) of the data values.

The domain changes automatically according to the channel selected. You can override the default limits by entering new values in the *Min Domain* and *Max Domain* text slots on the control panel.

3.5.2. Sorting the Data

The *Histogram* module divides the range of data values in the lattice into a number of equal subranges called *buckets*. It counts the number of nodes in the lattice whose values lie in each of the buckets and produces a histogram of the counts.

To set the number of subdivisions in the data domain, you can move the *Num of Buckets* slider, or type in a new value and press `<Enter>`.

If you define a domain with a narrower spread of values than the actual dataset in a channel, then the out−of−range values are omitted from the histogram if *Outliers* is set to *Omit*; all the data points below the minimum value on the scale are put into the first bucket, and all the data points above the maximum are put into the last bucket if *Add* is specified.

The range of data counts is indicated by the height of each bucket.

3.5.3. Selecting Channels

The option menus let you choose whether you want to generate a histogram for only one data channel of the input lattice, or for all channels. Each channel shows the values for one variable in the input lattice.

To select a data channel, move the *Channel* slider (this slider is displayed only if more than one channel may be selected from the input lattice). You can also type a number into the Current Value slot at the top of the slider.

To choose between displaying one channel or all channels, click the relevant radio button.

If you select *One Channel*, a single histogram showing the data distribution in the selected channel is displayed.

If you select *All Channels*, you see as many separate but overlapping histograms as you have channels. The active histogram is white. It shows the currently selected channel, which is the channel corresponding to the number on the slider. You can change the active histogram by using the slider to select another channel (while in the *One Channel* mode).

When you select *One Channel* or *All Channels*, *Histogram* produces a 1D output lattice.

When you select *Multi−Dimensional*, you do not get a display, and the output lattice has a different format. The number of data variables in the input lattice defines the number of dimensions of the output lattice. For example, if the input lattice has three data variables, the output lattice will be 3D.

Each side of the output lattice is divided evenly into the number of buckets defined for the input lattice. The data buckets are formed by dividing the 3D lattice into smaller cubes, each of which contains the data points in that section of the total volume.

Example 3−2 Generating Histograms

This example illustrates how you can generate a histogram for each data channel of *GenerateColormap* using *Histogram*.

1. Launch a *GenerateColormap* module and a *Histogram* module from the Module Librarian.
2. Connect the *GenerateColormap* module's "Colormap − − Lattice" output port to *Histogram*'s data input port.

   When *GenerateColormap* passes data to *Histogram*, you can create a histogram for the data coming in from a single channel, or for the data coming in from all four of *GenerateColormap*'s channels.
3. Use the slider to change the selected channel and see what happens.

3.6. Displaying Images

At times, you may find it useful to display an image, as opposed to a geometric object, on the screen, perhaps to see what the original image data looks like before it is processed through a map, or so that you can compare two images, or create a composite image from two separate images.

The *DisplayImg* module (see **Figure 3−15**) accepts image data in the form of a 2D lattice, either black and white or in color.

You can display several images in the *DisplayImg* window at any given time. Simply wire as many modules as you require into the input port of *DisplayImg.*

Figure 3−15 An Image in the *DisplayImg* Module

As each module upstream from *DisplayImg* fires, an image appears in the *DisplayImg* window. The image is placed at the bottom left−hand corner of the window, and each subsequent image is stacked on top of it.

- To see the images underneath, you can change the stacking position of the images with the **`<PgUp>`** and **`<PgDn>`** keys on the numerical keypad (not the **`<Page Up>`** and **`<Page Down>`** keys to the left of the numbers). Place the cursor over the image of interest.
- To move the top image to another area of the window, click on it with the left mouse button and drag it to a new position. You can move all images around in the window by using this method.
- To return an image to its original position, place the cursor on the image and press the **`<Home>`** key on the numerical keypad. If the window contains several images and the cursor is not on any one, they all return to the lower left corner.
- To increase or decrease the size of an image by using the *zoom* widget. Click on the image you want to zoom. The slider shows the current zoom factor for that image. As you adjust the slider, the zoom factor changes with the size of the image.

You can "photograph" the image in the window and send it on to another module to be processed by clicking on the *Snap?* button. You can arrange several images in the window and *Snap?* will produce a single composite image that is the same size and shape as the original.

You can also remap the image.

Note: The *DisplayImg* window should not be obscured by any other windows, or the image will not be properly displayed.

Because the product of *DisplayImg* is also a lattice, you can send the image to any module that accepts a 2D lattice, through the lattice output port on *DisplayImg.*

Example 3−3 Comparing Two Images

This example illustrates the use of *DisplayImg* to compare an image in its original form with the same image after it has been modified.

1. Launch a *ReadImg*, a *BlurImg*, and a *DisplayImg* module in the Map Editor.
2. Connect the output port on *ReadImg* to the input port on *DisplayImg.*
3. Wire the output port on *ReadImg* to the input port on *BlurImg* by clicking on "Connect" in the cascade menu.
4. Wire the output port on *BlurImg* to the input port on *DisplayImg.*
5. Open the file */usr/explorer/data/image/flowers2.rgb* in *ReadImg*.

   When the module fires, you should have two copies of the image in the *DisplayImg* window. Use the mouse to position them in the full−scale module window.
6. Turn the dials on the *BlurImg* module and compare the top image with the bottom one (see **Figure 3−16**).
7. Replace *BlurImg* with another image−processing module and see what effect it has on the image.

Figure 3−16 Example Showing Two Images in *DisplayImg*

3.7. Visualizing Data

Since IRIS Explorer visualizes numerical data after it has been processed in various ways, the *Render* module (see **Figure 3−17**) is naturally a very important part of the process. It is *Render* that:

- displays the geometric object, which it renders from 3D geometry sent to it by compatible modules, in its window
- provides a wide selection of visual enhancement techniques for studying the object further

Figure 3−17 The Render Module

The *Render* module can accept multiple inputs, and it manages each one separately. Its viewing and editing functions are available only in the full−scale control panel. When the full−scale window is open, the Diminutif window is blanked.

Render has several viewing modes and an edit mode, in which an object can be selected and its attributes altered. The functions and attributes of the *Render* module are described fully in the *IRIS Explorer Reference Pages.*

3.7.1. Ports on Render

Before you can display an object in the *Render* window, you must wire *Render* into a map that generates the object. *Render* has six input ports: four of them accept geometry inputs and two accept lattices.

The most frequently used input port is the "Input − − Geometry" port, which accepts the geometry necessary to create an object or scene from any module with a geometry output port.

The "Annotation − − Geometry" port accepts text labels for parts of the geometry in the *Render* window. Its use is illustrated in the "pick" map in the Module Librarian.

Render has three output ports. The "Pick − − Pick" port outputs pick data to a picking module. The other two ports let you take snapshots of the object in the *Render* window and export them to other modules. This is useful for making animations.

3.7.2. Modes of Operation

Render has four mutually exclusive viewing modes. The viewing mode is a way of considering how the camera in the scene is moving. The viewing modes are selected from the *Edit* menu on *Render*'s menu bar. Each one lets you view the scene in the *Render* window in a slightly different way. You can *examine*, *fly* by, or *walk* by the object, or you can change the viewing *plane.*

You can zoom in and out, as if looking at the object through a telephoto or wide−angle lens. The object appears closer when you zoom in, although it actually remains the same distance from the camera.

To zoom in or out, hold down the left and middle mouse buttons and move the mouse towards or away from the screen.

3.7.3. Selecting an Object

Render has two picking modes, viewing or picking/selecting an object. In the *Pick/Edit* or view mode, you can change the camera parameters, that is, the position, direction, and field of view of the camera in relation to the object.

In the *User Pick Mode*, you use the mouse and buttons to select objects, attach manipulators, and query objects through picking.

To select an object, click the left mouse button on it. The object appears surrounded by a red box (see **Figure 3−18**).

Figure 3−18 A Selected Object

3.7.4. Moving an Object

You can view an object in the *Render* window from a different position by manipulating it using the left or middle mouse button in *Examiner* viewing mode. These actions change the camera position from which you are looking at the object.

To roll the camera, that is, to rotate it on an axis perpendicular, or normal, to the screen, hold the left mouse button down and drag the object, either clockwise or counterclockwise. The object spins in the direction in which you dragged it. The object rotates faster according to the speed with which you push or drag it. It continues to rotate once you release the mouse button. To stop it, click on the mouse button outside the object.

To move the camera view around the center point, that is, to move it in the plane of the screen, hold the middle mouse button down and drag the object.

3.7.5. Using the Pop−up Menu

The *Render* pop−up menu (see **Figure 3−19**) appears when you click on the background of the *Render* window. If you zoom or truck in too far and lose sight of the object, click on the window background with the right mouse button and select *View All* from the *Functions* submenu of the pop−up menu.

Figure 3−19 Render Pop−up Menu

The *View All* option resets the camera position, enabling you to see all the objects in the scene.

- To center an object in the window, choose *View Selection* from the *Viewing* menu (this item is selectable only if more than one object is currently displayed in *Render*'s window). You need to select an object in the *Pick/Edit* mode first.
- To restore the original camera angle after you have moved the object around, select *Home* from the *Functions* submenu of the pop−up menu.

You can also select the *Draw Style* (for example *wireframe*), switch *Headlight* and *Decoration* on and off and set some *Preferences* for display. The *single buffer*, *double buffer*, and *interactive buffer* options on the *Draw Style* submenu of the pop−up menu refer to the type of buffering you are using.

3.7.6. Using Graphical Shortcuts

The *Render* module provides several graphical shortcuts for manipulating objects in the Examiner, Navigator, and edit modes. To use the shortcuts, select *Decoration* from the pop−up menu. A frame containing several widgets appears around the *Render* window (see **Figure 3−20**).

The widgets include:

- thumbwheels that control the camera rotation or tilt and trucking motion. They also work in Pick/Edit mode.
- a slider that controls the zoom action.
- eight buttons that correspond to *View*, *Pick/Edit*, *Help*, *Home*, *Set Home*, *View All*, *Seek*, and *Camera Type*.

Figure 3−20 The *Render* Decorations

3.7.7. Wireframe Rendering

When you have an opaque object in the *Render* window, it may be easier to examine or move around in wireframe form. You can draw the object as a wireframe structure (see **Figure 3−21**) by selecting *Draw Style* from the *Render* pop−up menu in either viewing mode, then clicking on *wireframe*.

Note: This command acts on all the objects in the scene, not just on a selected object.

Figure 3−21 Wireframe Object in *Render*

If you are viewing a complex scene with several objects, it may take a long time to render each frame while moving it. Wireframe rendering is much faster than rendering opaque objects, so you may want the objects to be rendered in wireframe while they are moving and opaque when still.

To draw the objects in wireframe as the camera moves, use the *move wireframe* option. When you release the mouse button, the objects are again drawn in their original mode.

3.7.8. Editing Object Properties

Objects displayed in *Render* have a variety of attributes. They include positional properties, such as translation, rotation, and scale, and surface properties, such as color, specularity, and transparency. You can change or edit all these attributes using the *Render* editors in *User Pick mode*.

You edit object attributes by picking the object and then selecting an editor from the *Editors* menu.

Each editor provides access to various attributes of the object (see **Table 3−3**).

Editors	Purpose
Material Editor	Sets the material properties of ambient, diffuse, and specular color, shininess, and transparency.
Color Editor	Brings up a graphical color editor that lets you set the diffuse color on an object.
Transform Sliders	Sets the exact placement, scale, and orientation of the object in the window.

Table 3−3 Object Editors

Using the Material Editor

The Material Editor (see **Figure 3−22**) lets you change the color and texture of an object in a scene. You can set:

- Ambient intensity: the level of ambient color
- Diffuse intensity: the level of diffuse color
- Specular intensity: the level of specular color
- Emissive intensity: the amount of light the object emits
- Shininess: the amount of surface shininess on the object
- Transparency: the degree of transparency or opacity

Figure 3−22 The Material Editor

Use the sliders to set the brightness, or intensity, of each color.

To change the material color itself, click on one of the radio buttons on the Editor. The Color Picker appears and you can edit the ambient, diffuse (usually the same as ambient), specular, and emissive colors in turn. By clicking on the square buttons to the right of the radio buttons, you can select to edit one or more of these colors simultaneously. In this case, a tick mark appears to indicate which colors have been selected for editing. Naturally, in this situation they will all have the same color assigned to them.

For details on how the Color Picker works, refer to **"Using the Color Editor"** below.

Using the Color Editor

You can change the color properties of the object material using the Color Editor (see **Figure 3−23**). Use the *Sliders* menu to select *RGB*, *HSV*, or both. Choose colors from the color wheel. Use the *Edit* menu to stipulate whether or not the change is immediately reflected in the object. By selecting *Manual*, you can prevent any color change taking effect on the object until you are ready.

Use the two color squares to test new colors and store the previous one. By clicking on the three arrowed pads beneath the squares, you can switch back and forth between colors.

Figure 3−23 The Color Editor

The new color is always on the left and the previous color on the right. You can send the new color to the right square, or the old color to the left square.

The Color Pickers in the other editors work in the same way as the Color Editor, although they affect the color of different items.

Using the Transform Sliders

The Transform sliders (see **Figure 3−24**) let you rotate, translate, and resize any object that is part of the scene in the *Render* window. The changes affect only the selected object, not all the objects in the scene. The slider widgets are:

- *TRANSLATIONS*: changes the position of the object in the scene
- *SCALES*: changes the relative size of the object in the direction of the three axes (conveniently called x, y, and z). The direction of the axes is defined by the *SCALE ORIENTATION* widget.
- *ROTATIONS*: changes the orientation of the object in the scene
- *SCALE ORIENTATION*: changes the direction of the axes along which the object is scaled
- *CENTER*: changes the center about which rotations take place

When you click on a the title of a slider, for example, *TRANSLATIONS*, the slider opens up and shows three text slots, one each for the *x*, *y*, and *z* coordinates. Enter the new value and see how the position of the object changes.

Figure 3−24 The Transform Sliders

Next to the slider name in the open slider is a *Style* button. It offers three different layout styles for the widgets in the slider. To see the different widget layouts, click on *Style*.

3.7.9. Displaying Transparent Objects

If you want to make an object transparent, use the *Transp* slider in the *Material Editor* (see **"Using the Material Editor"**). To display the transparent object, you can select *Screen Door Transparancy* or any of the other sorted transparency widgets on the *Viewing* menu.

Note: Some workstations do not support sorted transparency. They can only display transparent objects in *Screen Door* format.

As its name implies, Screen Door Transparency renders objects as meshes that you can see through.

Sorted transparency, also called blended transparency, gives a more accurate rendering of the transparent object, but the scene is drawn more slowly than in Screen Door format.

3.7.10. Changing the Lighting

The *Render* module has a headlight that is situated in the camera position, which is defined as being just over your left shoulder. The headlight moves with the camera. You can also create point, spot, and directional lights, which remain with the objects in the scene. Thus, if the camera moves left, the lights appear to move to the right. You can have as many as you like on at any given time.

You can change the direction, color, and intensity of each light in the scene by using the *Lights* menu to toggle lights on and off and to edit their attributes. You can also edit ambient light levels.

Note: If you cannot see the *Lights* menu on the *Render* menu bar, just enlarge the *Render* window to the right.

Changes to the lights affect all the objects in the scene, unlike changes to the transform and material properties, which apply only to the selected object.

Editing a Light

To edit the properties of a light, for example, the headlight, select *Headlight* from the *Lights* menu and click on *Edit*. The Edit window appears (see **Figure 3−25**), displaying the light you have selected.

Figure 3−25 Light Editor

You can change the intensity of the light emanating from the source by moving the widget on the *Inten* slider to increase or decrease the light intensity. The change is reflected in the image of the light.

You can also adjust the angle at which the light shines on the object by clicking on the directional arrow and changing its position in the window.

The *Edit* menu provides access to the Color Picker window for changing the color of the light. The Color Picker works in the same way as the Color Editor.

3.7.11. Manipulating Objects

The *Manips* menu options let you rotate and translate objects in a scene by attaching one of several manipulators to them. You can use only one type of manipulator at a time, but you can attach it to each of several objects at the same time. You can use these manipulators to reposition the object in the scene. The movement is not as precise as that achieved by using the sliders in the *Transform Editor*, but it is a more direct manipulation.

To remove a manipulator from an object, click on *None* or select another manipulator from the *Manips* menu.

Example 3−4 Using the Trackball

The *Trackball* manipulator (see **Figure 3−26**) consists of opaque stripes surrounding the object. These stripes are aligned with the local axes of the object.

- To rotate the trackball, place the cursor on the stripes and drag the mouse in the desired direction, with the left mouse button held down.

  When you release the mouse, the ball continues to spin until you click the left mouse button.
- To rotate the trackball in the direction of a single axis, place the mouse on one of the stripes, then drag it in the direction of the stripe.

Figure 3−26 The Trackball Manipulator

If you start the trackball spinning and then move the mouse outside the sphere, the ball continues to spin within the plane of the screen.

You can use the *Transform Editor* to monitor the object's degree of rotation, as reflected in the rotation slider values.

3.8. Constructing Loops

You may sometimes need to go beyond the linear dataflow model of existing IRIS Explorer maps to build maps containing dataflow loops. They are needed, for example, for data interrogation through picking in *Render*, iteration through data sets using *For* or a multiple−data set reader, and convergence of a simulation using *While*.

For example, **Figure 3−27** illustrates a loop from *Render* to *QueryLat* and back to *Render*.

Figure 3−27 A Loop in a Map

The user picks on an object in *Render*, which fires and sends the pick coordinates to *QueryLat*. *QueryLat* turns the coordinates into text and sends it to *Render*. When the data arrives back at *Render*, the loop is terminated. To start the loop again, the user has to make another pick.

The common goal in all loops is to feed data from one phase of an IRIS Explorer computation back around to the next phase of the computation. As in most programming languages, loops in IRIS Explorer are a basic and necessary semantic element. It is also true that looped maps are more complex than linear, or unlooped, maps. This section describes how you can conveniently and correctly employ looped maps in your IRIS Explorer data visualizations.

3.8.1. Defining a Loop

You can create two kinds of loops, While loops and Repeat loops, like the similarly named loops in C. In While loops, a specific condition evaluates to `true` and then an iteration takes place. This sequence is repeated until the condition evaluates to `false`.

In Repeat loops, an iteration takes place and a condition is then evaluated. Similarly, the sequence is reiterated until a stopping condition is reached.

3.8.2. Loop Controllers

Each loop must contain a loop controller module which controls and terminates loop iteration. In While loops, the loop controller checks the condition at the start of the loop, and in Repeat loops, at the end of the loop. The While or Repeat sense of the loop is determined by the While or Repeat capability of the module controlling the loop. Loop controllers have either While or Repeat capability, but not both.

Loop controllers are identified in the Map Editor by a small square icon, the controller icon, in the left−hand corner of the title bar in the mini control panel (see **Figure 3−28**).

Figure 3−28 Loop Controller Modules

The controller icon is Off when a loop controller module is wired into a map but is not part of a loop, or when it is in a loop controlled by another module. When you wire a loop controller module into a loop, the icon turns green. For example, in **Figure 3−28**, AnimateCamera is the loop controller, even though both modules have controller capability (see **How Loops Work**).

You can identify a loop controller module only after it has been launched in the Map Editor; until then, it is indistinguishable from other modules in the Module Librarian.

You can also look at the *.mres* file in the Module Builder; the Build Options window indicates whether the module is a While or Repeat loop controller, or neither. For example, **Figure 3−29** shows the controller status for *AnimateCamera*, which is a Repeat loop controller module.

Figure 3−29 Build Options for Loop Controller

You confer loop controller status on a module in the Build Options window of the Module Builder (see **Figure 3−29**). Selection of *While*, *Repeat*, or *None*, which is the default, prompts the MCW to provide the correct interface for the module.

The distinguishing characteristic of a loop controller module is its ability to break its loop. IRIS Explorer has *While* and *Repeat* modules whose sole function is to control While and Repeat loops, but other modules can also function as loop controllers. **Table 3−4** lists the IRIS Explorer loop controllers

While Loops	Repeat Loops
For	AnimateCamera
Trigger	Render
While	Repeat

Table 3−4 Loop Controller Modules

3.8.3. How Loops Work

There can be only one active loop controller in a loop. If the loop contains more than one module with loop controller capability, IRIS Explorer selects the active loop controller, based on information from the loop connections. You can change the active controller by clicking on the loop icon. A message appears asking for confirmation.

Note: Even though IRIS Explorer allows you to change controllers, the change may not make sense from a programmatic point of view. It is up to you to make sure that the looped map makes computational sense and that it will terminate.

A loop fires until the loop controller detects an end−of−loop condition and stops the loop. To fire a loop, or to start up a loop that has been terminated, you can fire the loop controller module. The controller module can always cause its loop to iterate if you change one of its widgets.

In addition, any module outside the loop or upstream of the loop controller can start the loop. If you have a Repeat loop, where the controller is at the end of the loop, this includes modules in the loop as well upstream of the loop itself. In a While loop, upstream modules are all outside the loop.

If a frame results from a parameter change in a module inside a loop, the While controller will not restart the loop, whereas the Repeat controller will, because the frame came from upstream. This is important when, for example, *Streakline* is in a loop with *Render*. If a parameter changes in *Streakline*, *Render* can keep the loop going.

Controller modules such as *While* and *Repeat* pass their data through like a transistor, turning off the flow by sending a sync frame when their Condition port parameter becomes zero.

Other controller modules, like *Render*, do not really control iteration, but need to be control modules in order to support their normal wiring paradigms, such as picking in the *Render−−QueryLat* map (see **Figure 3−27**). *Render* needs to be a Repeat controller so that any module in the loop can start the loop.

3.8.4. Wiring a Loop

When you wire a loop in a map, you must include a loop controller module in the loop. If you try to close a loop without a loop controller, you will get a message saying the connection has failed because there is no controller present.

You can have:

- nested loops:
- multiple loops with a single controller
- multiple controller−capable modules in a single loop (with only one of them selected as the active controller)
- and (hopefully) anything else that makes programming sense

You cannot have intertwined loops, where two control modules are each in the other’s loop, or where a single module is controlled by more than one controller.

Each loop controller module has an additional synchronisation port *Loop Ended*. When the module is the active controller you can wire this port to the *Fire* port of another module outside of the loop, to cause that module to fire when the loop completes: when the loop controller has detected the end−of−loop condition and stops the loop. This can be useful to control nested loops, as demonstrated in the example map */usr/explorer/maps/AnimateIso.map*.

Figure 3−30 shows the wiring of the nested loops within this map. *AnimateCamera* is the loop controller for the inner loop and sends cameras to *Render*. *For* controls the outer loop. The *Loop Ended* port of *AnimateCamera* is wired to the *Fire* port of *For*, to start the next iteration of this outer loop.

Figure 3−30 Loop Wiring in AnimateIso Map

Control Arcs

The special coloring of loop control arcs (see **Figure 3−31**) can help you understand loops. Each loop has a single control arc, which is adjacent to the loop controller that governs it. The control arc is **upstream** of a While controller and **downstream** of a Repeat controller. A control arc is conceptually treated as "broken" by IRIS Explorer when it is computing which modules lie upstream of one another.

Figure 3−31 Looping Control Arc

The special control arc wiring helps to clarify otherwise ambiguous loops, such as the one in **Figure 3−32**, which shows a symmetrical map with separate While loop controllers for each nested loop. Either *While* module can control loop 3. You will be able to deduce which module controls loop 3 by reading the controller arcs. You can change which module controls loop 3 by selecting arcs and controller buttons carefully.

Figure 3−32 Ambiguous Looping Scheme

Saving Loops

If you save a map or part of a map that contains a loop, IRIS Explorer saves the identity of the active controller and will restore the map with this configuration.

In general, you would group most loops that perform a complex iteration. However, controller capability cannot be promoted from the controller module in the loop to the group module. If you wire a group module into a loop, the loop must include its own separate controller. In this case, the grouped loop constitutes a nested loop in the map.

3.8.5. Halting a Loop Iteration

An iterating loop may be terminated by the controller module, or it may be interrupted by the Map Editor. This is called "breaking the loop," which is different from "disconnecting the loop," where inter−module connections are removed.

In the Map Editor, you can also use the "Break Loop" option, which appears only on the module pop−up menu of active loop controllers (see **Figure 3−33**). It forcibly terminates a looping cycle.

Figure 3−33 Module Menu on Active Loop Controller

If your looping map seems to be hogging resources with execution highlighting on, try switching it off before you initiate a loop.

3.9. The IL Controller Module

This module is used to create a single executable process from a group of ImageVision Library (IL) modules. The module takes map file information about the constituent modules and their ports and connections, and creates an IL operator for each IL module in the map. It then chains these operators to execute the modules, in effect creating a single executable process from all the alternate module executables.

The IL Controller module sits between the ImageVision Library and IRIS Explorer. It is similar to the dataScribe module that uses a scribe file, except that the IL Controller module uses a map file. There must be both a*modulename.map* file and an *.mres* file for the module to function.

To use the IL Controller module, this is the procedure you follow:

1. Create a map from several IL modules.
2. Group the modules in the Group Editor and save the map as a group.
3. Save the grouped map file by using the *Save Selected* item on the Librarian File menu. You will be prompted to confirm that the map is saved as an imaging process. If you click on *OK*, then an *.mres* file is created with the map filename.
4. Open the *.mres* file in the Map Editor. The map comes in as a regular module and uses the IL Controller module as its alternate executable.

The main reasons for using the IL Controller module to execute a map are:

- the increase in processing speed
- the reduction in memory use

This is because the IL execution module is a "pull" model and requires much less information to compute a large image than IRIS Explorer, which uses a "push" execution model.

Chapter 4

Editing Control Panels and Functions

This chapter describes how to use two of the IRIS Explorer editors. They are:

- the Control Panel Editor, which you can use to create and modify module control panels.
- the Parameter Function Editor, which you can use to create relationships between parameters in linked modules.

These two editors allow users to modify modules and increase their power without using the Module Builder.

4.1. Overview

IRIS Explorer provides the user with the capacity to edit work in progress at many different stages in the creative process. In the Map Editor, the user edits the sequence of modules in a map. In the Module Builder, the user edits the module resources. IRIS Explorer also provides editors for modifying aspects of the modules themselves.

The Control Panel Editor enables you to create new control panels for modules, groups, and applications, and modify the appearance of existing control panels. You can define new menus on the menu bar and add or rearrange widgets on the control panel.

The Parameter Function Editor lets you establish relationships between parameters in adjacent modules in a map. The parameter value in the downstream module is then expressed as a function of the upstream parameter values.

This chapter describes how to use both these editors.

4.2. Using the Control Panel Editor

The Control Panel Editor (see **Figure 4−1**) lets you modify a module control panel or create a new one by editing the parameter widgets and menu items. You may change a widget's type, position and resize them, and change their limits and default values.

There are three places the Control Panel Editor can be used from:

- To invoke the Control Panel Editor in the Map Editor, you must be creating a group and the Group Editor must be open. Select "Control Panel" from the Edit menu in the Group Editor.
- To create a module control panel in the Module Builder, click on the "Control Panels" push−button in the Module Builder main window.
- In the DataScribe, select "Control Panel" from the View menu when you are creating a DataScribe module.

The Control Panel Editor consists of an Editor window and a Preview control panel window. You stipulate the attributes of the panel in the Editor window, and the result is displayed in the Preview window.

Figure 4−1 The Control Panel Editor

When you create a module, using *mbuilder*, or a group via the Map Editor, you can use the Control Panel Editor to change or create a new widget for each of the input parameters. If you are editing a group, you can only change previously defined widgets, i.e. you can not add new widgets to input parameters which do not already have a controlling widget.

When you edit a control panel, you can:

- Select a widget for each parameter.
- Arrange the widgets as you wish in the Preview window.
- Set the initial value and the range of each widget.
- Use the Menu Bar Editor to design the menus.
- Set a widget attribute for each menu bar item.

To save the menu and control panel configurations, click the *OK* or *Apply* buttons at the bottom of each window.

4.2.1. Choosing Parameter Widgets

The widgets on the control panel allow you to control the value of the module parameters directly.

Parameter Types

An IRIS Explorer parameter may be one of three types of scalar quantity, each of which can be represented as a widget or a menu item on the module, group, or application control panel. A parameter can be:

- a long integer (32−bit)
- a double precision floating point (64−bit)
- a character string

For more information on parameters, see **"Understanding IRIS Explorer Data Types"** in Chapter 2.

Widget Types

The widget types are listed on the Type option menu (see **Figure 4−2**) and described in detail in **"Setting Module Parameters"** in Chapter 2. All widgets must have a valid type selected, including those you place on a menu. The widget attributes are listed in the rest of the Widget Attributes pane.

The properties of the selected widget are shown in the widget properties pane on the right. To display the widget properties, double−click on the widget in the preview control panel.

To select a widget from the Control Panel Editor:

1. Click on the name of the parameter in the "Parameters" pane.
2. Select a widget from the Widget Type option menu in the Widget Attributes pane (see **Figure 4−2**). The widget appears in the Preview window.

Figure 4−2 Selecting a Widget Type

In **Figure 4−2**, for example, the parameter *scale* is highlighted in the Parameters pane. The widget type selected for it on the Type menu is a "Horz Slider" (horizontal slider). The widget is highlighted in the preview window. Its properties are listed to the right under "Slider Properties".

Figure 4−3 shows a selection of widgets, as well as the control panel "decorations." Decorations are vertical and horizontal lines, frames and labels, available from the Decorations menu (see **Figure 4−4**). You can size them and move them around in the same way as you manipulate widgets.

Figure 4−3 A Selection of Widgets

In general:

- dials are used for double precision parameters and sliders are used for long integer parameters. Text type−ins can be substituted for dials and sliders, with less functionality.
- Radio buttons and option menus can be interchanged.
- You can use text type−in slots where file browsers are used, although with more limited scope. File browsers are specifically for locating files, whereas you can use text type−in widgets for any parameter type.
- The scrolled list lets you make a selection from a list of items. Depending on how the list is defined, the user may be able to select only one, or several objects at a time.

Widget similarities are shown in **Figure 4−1**.

Widget	**Similar to**
Dials	Sliders, Text type−in slots
Radio buttons	Option menus

Text type−in slots	File browsers

Table 4−1 Similar Widgets

You can change the type of widget associated with any parameter by clicking on the parameter name and selecting a new widget type.

Hiding Widgets

The *Initially Hidden* button (see **Figure 4−2**) lets you hide widgets under certain circumstances. These are:

- When you have allocated a parameter associated with a file browser or text type−in slot to a menu in an application.

  This process is explained in detail in **"Creating Menu Items."**
- When you are building a module in the Module Builder and you want a widget to be visible or hidden according to the context the module is in.

  For example, you may have placed a channel selector on the module control panel. If the module is receiving data on one channel only, you may wish to hide the channel selector. If there is more data, you may want the channel selector displayed.

  For more information, see *cxInWdgtHide* in the *IRIS Explorer Module Reference Pages* or the on−line *man* pages.

Widget Decorations

Use the *Decorations* menu to select frames and dividers for your control panel. Once you have selected a decoration, it is listed in the Parameters pane (see **Figure 4−4**).

Figure 4−4 The Decorations Menu

Editing a Scrolled List

Four selection modes are available for the scrolled list widget (see **Figure 4−5**). You can choose which mode suits the kind of parameter you have.

The options are:

Single Select	Only one item at a time can be selected from the list, but the module user need not select any.
Browse Select	The user must select one item from the list.
Multiple Select	Several adjacent items can be selected from the list at one time.
Extended Select	Several items, not necessarily adjacent, can be selected from the list at one time.

Figure 4−5 Scrolled List Properties

4.2.2. Moving and Resizing Widgets

Click on the widget to activate it in both the Preview window and the Editor window. An active widget is enclosed by a "handlebox" of eight black squares (see **Figure 4−6**).

Figure 4−6 Widget Handlebox

- To change the location of the widget in the Preview window, click on the widget and drag it to the desired location. A solid box appears around the widget as you move it.

The Widget Attributes pane in the Editor contains the absolute location of the active widget. The origin of the control panel is in the upper left corner. As you move the widget, the X and Y starting locations change. You can type values directly into these text slots if you wish.

To resize a widget, you can:

- Type values into the width and height slots in the Editor window. This is useful if you have an exact dimension or location for the widget in mind.

A typed−in entry takes effect in the control panel when you press <**`Enter`**>.

- To change the size of the widget interactively, drag on any of the "handlebox" squares.

For really precise adjustments, use the Grid menu to activate the grid. You can size and position widgets exactly by snapping them to a grid.

The Grid menu contains a list of pixel resolutions that govern the location and size of widgets. You can select any one of these options. The default snaps to a grid size of 5 pixels.

When the "snap to grid" option is in effect, all subsequent widget moves and resizes occur only in multiples of the grid size. If you resize a widget in this mode, it becomes a factor of the grid's width or height, depending on the direction in which you resize it.

4.2.3. Setting Widget Properties

The Properties pane contains information specific to the widget, including the scalar type for dials and sliders and label information for option menus and radio buttons. A plain button has the port name as its label.

The primitive type is important for the proper operation of the module and should not be changed from its default value, which is what the underlying module is expecting. However, you can change it under special circumstances.

To change the widget primitive type, select an alternative from the Properties pane of the Control Panel Editor.

4.2.4. Setting Widget Values and Limits

To set the value of a widget, you must first select it. To select a widget for editing, double−click on it in the Preview window with the left mouse or click on it in the Parameters pane.

To select all the widgets in the control panel at once, use "Select All" on the Edit menu of the Control Panel Editor. To de−select them, click on the background of the control panel.

You can change the value of dial and slider widgets by typing a new value into the upper text slot or moving the widget pointer. You can also set minimum and maximum values by typing them into the range text slots. You must press **`<Enter>`** for settings to take effect.

You can also change the number of options in the Properties pane for radio buttons and option menus. This might not have the desired effect, however, if you add labels beyond those expected in the module itself.

A good example of this is the *Contour* module. It has three sets of radio buttons that control contouring in the I, J, or K directions. Each radio button has two options: On or Off. If you add a third option, such as Half On, to one of these radio buttons, the module will not process it.

Caution: You can seriously confuse the Map Editor if you switch a module widget type to one that is incompatible with what the module expects.

To dismiss the Control Panel Editor click the *OK* button.

The *Apply* button saves the layout information without closing the Editor window.

The *Cancel* button dismisses the Editor without implementing your changes.

4.3. Creating Menu Items

You can link a parameter with a menu item on the module menu bar instead of giving it a widget on the control panel. For example, instead of having two file browser widgets on the control panel for opening and saving files, you can set up a File menu with the menu items "Open" and "Close," each linked to a file selector.

The Preview control panel has a default menu bar with an empty "Help" menu already in place (see **Figure 4−3**).

Note: For each parameter linked to the menu bar, you need to specify a widget attribute in the Control Panel Editor as well. Use the "Initially Hidden" button to keep them from appearing on the module control panel.

4.3.1. Setting up the Menu Bar

To display the Menu Bar Editor, select "Menu Bar" from the Edit menu on the Control Panel Editor.

The Menu Bar Editor appears (see **Figure 4−7**). You can create up to eight different menus, with up to sixteen items on each menu, depending on the parameters you want to associate with a menu item.

Note: It will help the users of your modules if you follow the conventions for menu placement; that is, File menu on the far left, then Edit menu, then your other menus.

Figure 4−7 The Menu Bar Editor

Set up your menus by typing the name of each menu in the text slots in the Menu Bar Editor, for example, "File." To save them, click on *Apply.*

You can set up menu "stubs" for parameters that currently do not exist and add the items to the menu at a later date if you wish.

4.3.2. Setting up Menu Items

To create the menu options for each item on the menu bar, click on the "Menu" push−button under the menu name. The Menu Editor is displayed (see **Figure 4−8**).

You can create entries for the current menu. For example, reader modules or groups that have reader functions will usually have "New" and "Open" entries in their File menus.

Figure 4−8 The Menu Editor

From the option menu next to the item name, you can set the action to be taken when the user selects this item from the menu bar of the completed module control panel.

The action items are:

None	No action to be taken (useful as a placeholder).
Set param	Sets the value of a widget by passing a specific value to a parameter in the module or group.
File selector	Brings up a Motif file selector. It comes up in the current directory by default.
Script command	Carries out a Skm scripting command.
System command	Carries out a UNIX command and displays the result (if any) in the shell window where IRIS Explorer is running.
Quit	Destroys the module or quits IRIS Explorer when IRIS Explorer is running in application mode.

If you choose *File selector*, a button appears in the next pane, allowing you to define the file selector parameter.

If you choose *Set param*, the button appears again, with a text slot for entering the name of the parameter.

If you choose *Script command*, or *System command*, a text slot appears for entering the name of the command.

4.4. Menus in Group Control Panels

When you create a group or application control panel, you may wish to promote a menu item from an individual module to the group control panel. Unlike other widgets, which are promoted to the group panel along with their associated parameters, the actual menu entry is not promoted. Only the parameter passes to the group level. You must, therefore, create a new menu for the group control panel and redefine the parameter as a menu item at the group level.

4.5. Using the Parameter Function Editor

When a number of connections fan into the same parameter input port, you may wish to express the value on that port, as a function of some or all of the fan in connections. For example you may wish to take the average value of all the connections into the port.

The Parameter Function(P−Func) Editor (see **Figure 4−9**) lets you set up relationships between the fan in values on an input parameter port, so that the value of the parameter is expressed as a function of one or more parameter values from the upstream modules.

To do this, you define each parameter in the P−Func Editor and then create a mathematical expression that defines their relationship. Any change in the value of the upstream parameter (or parameters) is then reflected in the value of the input parameter.

Figure 4−9 The Parameter Function Editor

4.5.1. Running the Parameter Function Editor

You run the P−Func Editor from the module containing the parameter you want to control. Before you can write a parameter function for an input parameter, you must have one or more connections, from modules upstream, wired to its port.

Figure 4−10 Module Pop−up Menu

To display the P−Func Editor window:

1. Select a module that has at least one input parameter connected to an upstream module.
2. Click on the title bar of the module control panel and select "P−Func Editor" from the pop−up menu (see**Figure 4−10**).

The title bar of the P−Func Editor (see **Figure 4−9**) displays the name of the module for which you are creating the parameter function. The window also has these components:

- The Input Parameter option menu displays all the input parameters to the module. If the module has more than one parameter, you can select a different one to work with from the option menu.
- The Connections box lists all the connections to this input parameter from other modules in the network.
- The Variables box shows a short, unique alias for each incoming connection. The alias simplifies the task of entering the parameter function into the P−Func Editor.
- The Parameter Function box shows the expression defining the relationship between the upstream parameters and the selected parameter.
- The Message box displays a message after you click on *Check* or *Apply*. It will tell you whether you have entered a parameter function with incorrect syntax.

4.5.2. Defining a Parameter Expression

The P−Func Editor provides a simple set of expressions for creating compound relationships among two or more input parameters in a module. You can use all, or any subset of, the variables listed in the Variables box to construct your function. For example, if you have two connections, with variables A and B, you can type `A + 2*B`, or `A/B`, or `(A + B)/2` (the average of two inputs).

Table 4−2 lists the expressions accepted by IRIS Explorer.

Type	Expressions
Arithmetic	*, /, % (mod), +, −

Functions	min(A, B), max(A, B), exp(A), log(A), log10(A), pow(A, B), abs(A), ceil(A), floor(A), rint(A), cos(A), sin(A), tan(A), acos(A), asin(A), atan(A), atan2(A, B), cosh(A), sinh(A), tanh(A), length (returns the string length in characters)
Typecast	(long), (double), (string)
Relational	==, !=, >, >=, <, <=
Boolean	&& or AND, || or OR, ! or NOT
Ternary	A?B:C (`if A then B else C`) Multiple *if then* clauses followed by a terminal *else*

Table 4−2 Parameter Expressions

The expressions are identical to the C language functions. For more information, refer to the C language manual pages for each function.

Typecasting Parameters

IRIS Explorer accepts incoming parameters and turns them into floats by default. If you want the parameter to have a different type, you can use one of the typecast expressions (see **Table 4−2**).

For example, if you have parameters A and B coming in as doubles and you require them in integer form, you can cast them to type (long) as follows:

```
(long)A
(long)B
```

Using Strings

You can feed string parameter values to the P−Func Editor. Strings are groups of alphanumeric characters The only operator you can use on strings is +. The expression

```
(string)A + (string)B
```

indicates concatenation of the two terms. You can also create an expression of the form:

```
(string)(long)(3.3 + 8.9)
```

in which case, IRIS Explorer returns the a string of the value of "12".

You can add like terms only. For example, the expression:

```
A + (string)B
```

will produce an error.

If a string is not a number, it goes to zero. If it is a number, it returns a value. For example,

```
(double) (7+3)
```

returns a double.

Combining Functions

You can create more complex expressions using several functions. Here are two examples:

```
min(sin(A * 3.14159/180.0)
cos(B*3.14159/180.0))
```

and

```
If sin(A) < 0 then this
If sin(A) > 0 then that
else the other
```

Linking Independent Parameters

The upstream connections, supplying the values for the P−Func variables, need not be related, i.e. they can come from unrelated modules or the same module, it does not matter. If they come form different modules then there is no relationship between A and B; the parameter function relates A and B to the input parameter independently of each other.

4.5.3. Creating a Parameter Function

In this example (see below), a relationship is created between parameters from two different modules. The upstream module is *IsosurfaceLat*. It has a "Threshold" parameter (shown as a dial), which will be wired to the minimum and maximum level inputs of *Contour*, the downstream module (see **Figure 4−11**).

The parameter function is applied only when a upstream parameter (one corresponding to a variable in the P−Func expression) is changed, "Threshold" in this example. It is not applied if you change the subject parameter, Contour's "Min Level" or "Max Level" parameters, in this example. The change must be in the upstream parameter.

Example 4−1 Creating Simple Functions

This example shows how to link parameters from different modules, and then define the relationships between them. Launch the map named *simple*, consisting of *ReadLat* connected to *IsosurfaceLat* connected to *Render.*

To make a parameter connection (see **Figure 4−11**):

1. Launch a *Contour* module. Connect up *Contour*'s "Input − − Lattice" input port to *ReadLat*'s "Output − −Lattice" port.
2. Connect up *Contour*'s "Contours − −Geometry" port to *Render*'s "Input − −Geometry (Opt)" port.
3. Create a parameter connection by wiring *IsosurfaceLat*'s "Threshold" output port to *Contour*'s "Min Level" input port.
4. Now turn the dial on the *IsosurfaceLat* module. You'll see both *IsosurfaceLat* and *Contour* fire.

Figure 4−11 Connections for a Parameter Function

To create a parameter function:

1. Connect *IsosurfaceLat*'s "Threshold" output to *Contour*'s "Max Level" input port.
2. Bring up the P−Func Editor for *Contour* (see **Figure 4−12**). Select the "Min Level" parameter from the Input Parameter list.
3. A connection for *IsosurfaceLat* Threshold appears in the Connections box. This connection is defined as A in the Variables box.
4. Select the Parameter Function box by clicking in it, then type:

   ```
   A * 1.3
   ```

   Click *Apply* at the bottom of the Editor window to implement your modification.
5. Similarly, select the "Max Level" parameter from the option menu and type:

   ```
   A * 1.8
   ```

   Click *Apply* again.

Figure 4−12 Parameter Functions for a Module

Close the P−Func Editor and save the modified map under a new name.

Now when you move the dial on *IsosurfaceLat*, *Contour* fires as well. Set the number of levels in *Contour* to 2 or 3 and see what happens. The image should resemble that in **Figure 4−13**.

The values of the contour lines, therefore, are always 30% to 80% higher than the isosurface threshold.

Figure 4−13 Simple Example with a Parameter Function

If you click on the *Check* button, the P−func Editor tests the function you have entered to make sure that it is valid and prints a message in the Message pane. The Editor window stays open. For example, the second expression in the Parameter Function pane in **Figure 4−14** has been entered incorrectly.

Figure 4−14 Messages in the P−Func Editor

4.5.4. Summarizing the Process

This is a summary of the steps you take to create a parameter function in the P−Func Editor:

1. Select the input parameter for which you want to establish a parameter function from the Input Parameter option menu. It must have at least one connection to an upstream module, as shown in the Connections and Variables boxes.
2. If you wish to change the default variable names (`A, B, C, ...`), click on the variable name you wish to change, in the Variables box, see **Figure 4−12**.
3. Type a new name for the variable and press **`<Enter>`** to have the new name accepted. The default names are `A, B, C,` and so on, but you can type in any name you like to replace them.

   Note: The P−Func Editor is case−insensitive.

4. In the Parameter Function box, enter an expression that defines the relationship you want between the variables in the Variable box, if any.

Saving the Parameter Functions

Use the four buttons at the bottom of the Editor to save or reject modifications to the input parameter. The buttons function as follows:

OK	Accepts the changes you have made, implements them, and closes the Editor window.
Apply	Accepts the changes you have made, implements them, and keeps the Editor window open.
Check	Checks the validity of the function and sends a message to the Message pane.
Cancel	Closes the Editor window without accepting or implementing any changes on the screen.

Example 4−2 Creating a Complex Function

This example illustrates how to create a more complex parameter function by establishing a relationship between three parameters. The parameters are *Contour*'s Min Level and Max Level and *IsosurfaceLat*'s Threshold.

First, create a map by launching a *GenLat* module, a *Contour* module, an *IsosurfaceLat* module, and a *Render* module and wiring them together as shown in **Figure 4−15**.

To create an object in *Render* like the one in **Figure 4−15**, open *GenLat*'s full−scale control panel and set these widgets as follows:

- *Coord Type*: curv torus
- *Coord Representation*: curvilinear
- *Function*: sines
- *Data Type*: float

You want the threshold value of the isosurface to lie directly in the midrange of the contour levels generated by the *Contour* module. To do this, you connect two output ports from *Contour* to an input port on *IsosurfaceLat*, and set up a relationship between the two output parameters and the input parameter.

Figure 4−15 A Parameter Function Using Two Variables

To make these connections:

1. Connect *Contour*'s "Min Level" output parameter into *IsosurfaceLat*'s "Threshold" input parameter.
2. Connect *Contour*'s "Max Level" output parameter into *IsosurfaceLat*'s "Threshold" input parameter.
3. Bring up the P−Func Editor from *IsosurfaceLat*. The parameter of interest on *IsosurfaceLat*, "Threshold," is highlighted in the Input Parameters pane. The Connections area has two listings, *Contour.Min Level* and *Contour.Max Level* aliased as `A` and `B`.
4. In the Parameter Function box, type in the expression: **`(A + B) / 2.0`** This averages the two inputs that are fanned into "Threshold".
5. Click on the *Apply* button to establish the relationship.
6. Set *Contour*'s Min Level dial to 0.0 and the Max Level dial to 1.0. Set the number of contour levels to 4 or 5.

From now on, when you turn *Contour*'s Min and Max Level dials, the Map Editor sends these values to IRIS Explorer's parameter interpreter. It evaluates the expression and feeds the result to *IsosurfaceLat*'s "Threshold" parameter.

You can see the calculated value reflected in the position of *IsosurfaceLat*'s "Threshold" slider. There are now contour levels outside the isosurface.

You can change the transparency of the isosurface in *Render* so that you can see the contour values inside the surface as well. For more information, see **"Visualizing Data"** in Chapter 3, "Using Modules."

Chapter 5

Creating Groups and Applications

This chapter describes how to extend map construction into more complex realms. It explains how to streamline maps by combining the functions of several modules into one, thus creating a group of functions. It covers:

- types of groups
- selecting modules for a group
- using the Group Editor
- editing input and output ports
- creating a group control panel
- creating an application

5.1. Overview

In a remarkably short time, maps tend to overflow the available space, even with micro control panels. To reduce the clutter and simplify the map, you can cluster related modules together under a single control panel, called a*group*. A group is an encapsulated collection of modules with its own module control panel.

When you have an elaborate map containing several opened control panels, it is possible to misplace some of them: they become hidden behind one another or behind other windows on the screen. To minimize the visual complexity of a map, you can group several related modules under one control panel, creating a more manageable network of modules. Even with a smaller number of modules, you will probably have more accessible controls than you really need for a specific map.

Frequently, the creator of a map is not the end user but is producing the map or application for a group of co−workers or customers. Such a map is created for a specific purpose, and many of the module parameters available to the map developer will not be needed by the end user. The developer can group modules in the Group Editor to reveal only those parameters relevant to the application at hand. This restricts the functionality of the map, but also streamlines its user interface.

You can use the Group Editor to promote specific widgets and their ports from individual modules to the group control panel. When you have placed the widgets on the group control panel, you can edit them using the Control Panel Editor, which runs from inside the Group Editor. Once you have established the group, those parameters from the original set of modules that are not connected to the group control panel cannot be changed without opening the group.

5.2. Module Groups

A group is a collection of modules that serve a common purpose, sharing a single control panel and having some of the parameters of the constituent modules raised to the group level. From the viewpoint of the operating system, the constituent modules are still separate processes: from the user’s point of view, however, they behave as a single entity.

You create groups of modules that are frequently used in combination, or that together form a unit of computation. For example, you might want to create a map that processes an image using four different parameters, each of which resides in a different module. When you wire all the modules into the map, you get the parameters you want as well as several others you don’t need.

In this case, you can simplify the map by creating a group for the four image−processing modules. This reduces the apparent number of modules, and thus wires, on screen, as well as reducing the number of control panels and widgets.

5.2.1. Types of Groups

Groups have two forms, *open* and *closed*:

- An open group retains much of the character of the constituent modules. All the modules and their wiring connections are visible, and their controls are operational. The constituent modules are outlined in red.
- A closed group takes on a new character. It has a single control panel that contains selected widgets from the constituent modules, and only the widgets on this control panel work. None of the individual modules or the internal wiring is visible.

A group can contain several modules, and it can also contain other groups. You can therefore create a nested hierarchy of modules under a single control panel.

5.2.2. Selecting Modules

You must select each module before you can include it in a group.

To select several modules easily at one time, use the "lasso" technique: click the left mouse in the Map Editor background and sweep it around the modules; otherwise, hold down the < `Shift`> key select individual modules one by one.

The selected modules are saved in the group with the exact relative placement they had when you selected them. It is a good idea to arrange modules in as compact a structure as possible before you group them. This eases the task of sorting them out if you ungroup them again.

Once you have grouped modules by selecting "Group" from the Group menu, the modules behave as a single entity and have a single control panel. If you select another module, then reselect one module in an open group, all the modules in the group are selected. Likewise, when you drag any module in the group, they all move together.

5.2.3. Creating a Group

To create a new group or work with an existing group, follow these steps.

1. Select the modules you want to group together, and click on "Group" from the Group menu. A single control panel appears for the new group, which is closed.
2. Use the Group Editor (select "Edit" from the Group menu) to promote the ports and parameters you want visible in the group control panel. The Group Editor is described fully in **Editing a Group**.
3. Use the Control Panel Editor to provide widgets for parameters in the group control panel.

The name of the new group module defaults to *Group*. You can rename it anything you like in the Group Editor. Group control panels are colored differently from individual modules.

Example 5−1 Creating a Group in a Map

This example illustrates how to select three modules in a map and group them.

Figure 5−1 shows the original map with all the modules ungrouped. It is a typical map: it reads two images from disk using two *ReadImg* modules and filters both images. One image is sharpened and the other is edge−enhanced with a Sobel filter. The resulting images are blended together and sent to *DisplayImg*.

The center three modules, *SobelEdgeImg*, *SharpenImg*, and *BlendImg*, can be collapsed into a single group to simplify the map.

Figure 5−1 An Image Processing Map

1. Select the modules and then select "Group" from the Group menu. The default group control panel is shown in **Figure 5−2**.

Figure 5−2 The Closed Group

1. To open the group, select the group and then select "Open" from the Group menu. The original modules appear

with a red border, indicating that they are part of a group (see **Figure 5−2**).

Figure 5−3 The Open Group

5.3. Editing a Group

A group usually consists of several modules. Each module can have several input and output ports and numerous widgets. In the context of a group module, not all these ports and widgets make sense or are usable.

The Group Editor (see **Figure 5−4**) allows you to:

- select the ports and widgets you want in the group
- create a group control panel using the Control Panel Editor (described in **Chapter 4**)

Figure 5−4 The Group Editor

5.3.1. The Group Control Panel

The group control panel has ports, widgets, and a module pop−up menu, just like the module control panel. It is, however, a different color. The widgets on the control panel allow you to control the value of the module parameters directly.

You can reorder the module sequence in the Group Editor by grabbing the module title bar and dragging it to the desired location. This changes the sequence in which ports are shown on the module control panel’s input and output port lists.

If you are creating a group or application control panel, you get a default widget setting. The Control Panel Editor allocates to each parameter in the new control panel the same widget type and properties as the parameter had in its module control panel.

For example, the *SobelEdgeImg* module has a slider for its Bias parameter. If you create a group containing *SobelEdgeImg* and promote the Bias parameter to the group control panel, the slider appears on the group control panel by default.

You can change this default to whatever suits your needs at the group level.

5.3.2. Promoting Ports

Selecting a port for the group control panel is called "promoting" the port. In deciding which ports and widgets to promote, you need to determine how you want to reuse this group and how your users (if any) will be using the group.

To select a port, click on the port button in the Group Editor.

Ports that are not promoted to the group level are not visible in the group control panel. The ports still exist, however, and widget−based parameter input ports retain the value they had when you first created the group. You can adjust the values of hidden parameters by opening the group, using "Open" on the Group menu.

5.3.3. Required Input Ports

Groups have two kinds of required input ports: those that are required at the module level and hence at the group level also, and those that are optional at the module level and have been promoted to the group.

Some module input ports must have data on them before the module will fire. These ports are also required at the group level, if there is no connection to them within the group, otherwise, the group will not fire. In the Group Editor, required ports are automatically promoted. They appear in the Group Editor as selected and you cannot deselect them.

For example, *SharpenImg* and *SobelEdgeImg* in **Figure 5−4** both have required module input ports that also appear as required group input ports. Neither module will fire unless:

- it has a connection to its "Img In− − Lattice" port
- it has data on its "Img In − − Lattice" port

These ports will be promoted to the group level and appear on the input port menu of the group control panel. The group control panel will not fire unless both ports have data on them.

5.3.4. Renaming Ports

Port names are shown on the input and output port lists in the group control panel and on the parameter widgets, so it is important that each port have a unique name. However, group modules often contain two modules whose ports have identical names. For example, there are two "Img In" ports in **Figure 5−4**.

Sometimes you may want to rename a port even if the name does not conflict with another port in the group. The focus of the group and the intent of an individual module are different, and a new name may make more sense in the context of the group.

The Group Editor provides an "alias" text slot for each port so you can set up a unique name for it.

To change the name of a port, type a new name in the text type−in slot. **Figure 5−5** shows the Group Editor after both the "Img In" ports have been renamed.

5.3.5. Selecting Parameters

To complete the new group control panel, select these parameters in the Group Editor:

- "Sharpness" from *SharpenImg*
- "Bias" from *SobelEdgeImg*
- "Blend" from *BlendImg*

You can change the parameter names if you wish.

Figure 5−5 Renaming a Port

For example, in the context of the group, "Bias" needs some clarification so it can be renamed "Sobel Bias."

You can rename ports and parameters at any stage of the grouping process by entering the new name in the Group Editor.

Click on *Apply* or *OK* to apply the changes and set up the new control panel.

5.3.6. Grouping Optional Ports

Some ports may not be required by an individual module, but must be promoted once the module is part of a group, because there are connections to other modules in the map outside the group at the time of creating the group.

Suppose you select *ReadLat*, *Contour*, and *WireFrame* to make a group. The group has one outside connection, a hookup from *GenerateColormap* to *Contour*. The "Colormap − − Lattice" input port on *Contour* is not intrinsically a required port, but in the context of the new group, it becomes required because of its link to *GenerateColormap*. Hence, it is automatically promoted to a group−level port.

This principle also holds true for output ports. If a connection exists downstream of the group, the port with that connection is automatically promoted and indicated as selected in the Group Editor.

5.3.7. Grouping Modules in a Loop

You cannot promote loop controller capability from the loop controller to a group module. If you want to link a grouped loop into another loop, you must include a separate loop controller module in the new loop.

5.4. Creating a Group Control Panel

You can create a group control panel in two ways, by default and by conscious design.

5.4.1. The Default Control Panel

Use the Group Editor to create a default layout.

To do this, click on the *OK* button at the bottom of the Group Editor window after you have selected the ports and parameters you want. This dismisses the Group Editor, applies the changes, and updates the Map Editor. In the process, it creates a default control panel for the group.

This is useful if the group does not have any parameters for which widgets are required, or if you can use all the default settings and layouts for selected parameters.

5.4.2. The Designer Control Panel

Use the Control Panel Editor, accessible from the Edit menu of the Group Editor, to lay out the control panel of the new group and customize it (see **Chapter 4, "Editing Control Panels and Functions"**).

Figure 5−6 shows the control panel for the new group in the Map Editor, along with its maximized form. It is easily recognizable as a group module because it is a different color.

If you select the input port access pad of the new group, you see only those ports you selected in the Group Editor, with the names you chose for them.

Figure 5−6 The Group Control Panel

Example 5−2 Using a Drawing Area

Figure 5−7 shows a group control panel with a drawing area. It uses the collection of modules and maps from the previous examples, but adds some of the *DisplayImg* module's widgets, including its drawing area, promoted by selecting "Window − − Parameter" in the Group Editor.

Notice the difference in size between this map and the one in **Figure 5−7**. A considerable savings has been obtained by combining four of the original six modules into a single group.

Figure 5−7 The New Group Control Panel

5.5. Opening a Group

To display the full−scale control panel, click on the *Maximize* button, just as you would an ordinary module.

To open a group again, select "Open" from the Group menu. The original modules reappear with each module outlined in red.

While a group is opened you can interact with any of the widgets of the constituent modules to change their parameter settings.

If you have a nested hierarchy of groups, you may open all groups in the hierarchy. However you can use "Edit Group" only on a top level group, you cannot edit a group within a group. Therefore you should have finalised selecting promoted ports before including it within another group.

Click on the background to deselect the open group. To select one or more modules in the group hold down the **`<Shift>`** key and click on each module to toggle selection on and off. To select one module in the group, deselecting all other modules currently selected, hold down the **`<Alt>`** key and click on the module.

To close the group and display the group control panel again, select "Close" from the Group menu.

5.5.1. Modifying Connections

You can create and destroy connections within the opened group, those between modules belonging to the group. Use the port menus to create and destroy a connection as usual. When creating a connection you will see that compatible ports are highlighted only on modules within the group.

If you use the Disconnect and Reconnect options on these modules, only those connections between modules within the group are broken with Disconnect and recreated with Reconnect. Using Disconnect and Reconnect on a closed group breaks and recreates the connections from the promoted ports of the group, but not the connections internal to the group.

You cannot create a connection from inside the group to outside. You should promote the inside port to the group, using "Edit Group" on the closed group, and then connect selecting from the port menu of the closed group.

You will see when disconnecting to ports of modules within a group that the name of the module is preceeded by the name of the group to further identify the connection.

5.5.2. Redefining a Group

The modules within a group can be reselected, retaining the attributes of the edited group, using the "Regroup" option on the Group menu. Select modules from the open group, leaving as unselected those to be omitted from the redefined group, and selecting modules not currently within a group to be added to the group. Then select "Regroup".

The control panel of the original group is retained, except that widgets corresponding to parameters from omitted modules are removed.

Settings for promoted ports and alias names are also retained. There may be additional required ports that are promoted from the new modules within the group. Note that it is possible for a port that was "required" for the original group to no longer be so. For example, by adding a module into the group that has a connection to an originally required port, removes the need for that port to be promoted to the group. This port will still be marked as promoted in the Group Editor, but can now be deselected.

You can "ungroup" a closed or open group by selecting "UnGroup" from the Group menu. You are left with the original collection of separate modules. If the group is closed and only the group control panel is showing, it opens before it is ungrouped.

5.6. Defining an Application

When you have created a group with its own control panel, you can run it as an application. An application is a module or map that runs in a special IRIS Explorer mode outside the Map Editor. You can engage the application mode only when you start IRIS Explorer from the command line. You cannot change an ordinary IRIS Explorer session into an application−mode session.

Application mode operates on ordinary IRIS Explorer maps, which consist of one or more modules or groups. Usually, a map contains several modules, collected into one or more groups.

5.6.1. Creating an Application

There is no special procedure for creating an application, since an application is simply a map run in a special mode.

- For details on creating a group, refer to **"Module Groups"**.
- For details on creating group control panels, refer to **Chapter 4, "Editing Control Panels and Functions"**.

In most applications, you will want a drawing area in order to visualize the results of the process that the application is performing on the incoming data.

Setting Parameters in an Application

When you are building an application, special attention should be given to the values of parameters in the application map.

When a map is run as an application, the parameters in the map will have whatever values they were set to when the map was saved. This matters particularly for parameter widgets associated with input and output filenames.

For example, suppose you are building an image processing application from a map that contains *ReadImg*. While testing the map, you typed the filename *fish.rgb* into *ReadImg* and then saved the map.

When you run the map as an application, *ReadImg* will be given the specific input filename *fish.rgb* and, therefore, will attempt to read in *fish.rgb* each time it starts up. In an image processing application for general use, this may not be the best behavior. However, for some parameters, setting the values in the map to default initial values may be the correct thing to do.

When you save a map that will be used as an application, be careful to set parameters to the start−up values that the application user expects.

5.6.2. Saving the Application

You must make sure that any map saved for later use as an application has at least one open control panel, that is, maximized to its full size; otherwise, nothing will be visible when you run the application. In application mode, the mini control panels are not visible and there is no way to maximize a module control panel. You can save a map with more than one control panel maximized if you wish.

To save the map, select "Save Selected" from the File menu on the Module Librarian. Save each map with a name and the suffix *.map*.

5.6.3. Running an Application

You can run an application using the command **`−app`**.

To open and use a standard application, at the shell prompt, type:

```
explorer −app applicationname.map
```

When you run an application, only the application control panel is displayed on the screen. The IRIS Explorer window, the Map Editor window, and the Librarian are not displayed.

5.6.4. Quitting an Application

Applications generally have a main window, which is a control panel that must be visible for the application to work properly. The Motif window "Quit" command is issued to a window when the user double−clicks in the upper left hand corner of the Motif window frame.

When IRIS Explorer is running normally (not in application mode), the double−click merely dismisses maximized control panels. The user can still access the corresponding mini control panel in the Map Editor and can re−maximize the control panel from there.

In application mode, the default behavior is the same, that is, double clicking dismisses maximized control panels. This is fine for secondary windows that may have been exposed under the control of the application itself.

However, if you dismiss an application main window in this manner, you cause the application to cease operating. This is because there is no visible Map Editor and thus no way for a user to recover the maximized control panel.

It is a good idea to add "Quit" to the menu bar on the main group control panel in your application map, otherwise, the only way to quit an application is through the Motif window menu.

You can also use the script command:

```
(set−app−main−win my−module)
```

to change the default behavior of the main window so that double−clicking on the control panel for "my−module" quits the application. The user will be asked to confirm the quit operation.

5.6.5. Application Busy Indicators

In application mode, the Map Editor is not visible and so the user does not observe the usual signals that a map is executing, such as yellow highlighting and the "Busy" cursor.

Instead, IRIS Explorer displays a "Busy" cursor over all control panels when any module in an application is executing.

Note: If you turn execution highlighting off, you disable the display of the "Busy" indicator.

Debugging an Application

To run an application so you can check its development and do debugging tasks, at the shell prompt, type:

```
explorer −map applicationname.map
```

When you run IRIS Explorer with this command, the application is displayed, along with the IRIS Explorer window. The IRIS Explorer window gives you access to the Map Editor and the Module Librarian if you require it.

Chapter 6

Scripting

This chapter describes how to use the scripting language, Skm, to take full advantage of the command interface in IRIS Explorer. It covers these topics:

- running IRIS Explorer with a script
- examples of its use
- the structure and syntax of Skm commands
- the Skm Editor window

6.1. Overview

IRIS Explorer provides two user interface mechanisms: the graphical user interface (GUI) which is described in the previous chapters, and a command interface. The command interface allows you to run IRIS Explorer using text−based commands. You can issue these commands directly from the keyboard, or you can write a script that IRIS Explorer runs, using the IRIS Explorer scripting language called Skm (pronounced "scheme").

The scripting language is useful for three reasons:

- You can set up a series of commands for testing new modules, maps, or applications.
- You can run IRIS Explorer automatically by piping information into it, which is handy for running batch jobs and test suites.
- You can run IRIS Explorer remotely from another system, such as a personal computer, and you can write the results to a file.

To create a script, you use the IRIS Explorer scripting commands. The commands are described in **"Command Syntax."**

6.1.1. Running IRIS Explorer by Script

To run a script when IRIS Explorer starts up, at the shell prompt (%), type:

```
explorer −script scriptfilename
```

A map file and a script file can be specified on the command line. For example:

```
explorer −map mymap −script scriptfile.skm
```

To set up IRIS Explorer for interactive scripting, at the shell prompt (%), type either:

```
explorer −script −
```

or

```
explorer −script %
```

Both of these puts you into the Skm interpreter, which displays this start up message and prompt:

```
IRIS Explorer Version 3.0

skm>
```

If you use the "explorer −script −" option this prompt appears in the shell window you invoked explorer from. If you use the "explorer −script %" option then this message and subsequent scheme output appears in a dedicated scheme window (see later).

At this prompt, you can type Skm commands as you go and have them executed in IRIS Explorer immediately.

You can pipe commands to IRIS Explorer using "explorer −script −" with a UNIX pipe. For example:

```
cat myFile.skm | explorer −script −
```

If you do use a pipe, although Skm output will appear as normal, subsequent commands cannot be entered from the keyboard as IRIS Explorer will still be receiving its input from the pipe. A pipe sends the output of one command into a second command. For more information, refer to the UNIX documentation.

6.1.2. Creating Simple Scripts

This section gives examples of how to use the Skm scripting commands. If you are entering commands interactively, you type each command on a separate line after the Skm prompt (**`skm>`**). If you are creating a script file, simply type the commands, each on a separate line.

The *start* command is used to start modules. For example,

```
skm>(start "ReadImg")
```

launches the *ReadImg* module in the Map Editor. The name of the module is enclosed in double quotes. The command:

```
skm>(start "SharpenImg" "at" 200 300)
```

launches the *SharpenImg* module in the Map Editor at the specified location. The coordinates specify the position of the top left−hand corner in pixels. Both the name of the module and the argument "at" are enclosed, separately, in double quotes.

Modules can be destroyed using the *destroy* command. For example,

```
skm>(destroy "ReadImg")
```

destroys the *ReadImg* module that was previously launched.

Maps are started by using the *start−map* command. For example,

```
skm>(start−map "cfd")
```

launches the *cfd* map.

The *connect* command specifies two ports to be connected. The command lists the output module name, the output port name, the input module name, and the input port name. For example,

```
skm>(connect "ReadImg" "Output" "SharpenImg" "Img In")
```

connects the "Output" port of *ReadImg* to the "Img In" port of *SharpenImg*.

Defining Command Variables

You can use the *define* command to define a variable for use in other commands. For example, you can use this command:

```
skm>(define mod1 (start "ReadImg"))
```

to start the module *ReadImg* and create a reference to the module in a variable called *mod1*. Then you can destroy *ReadImg* by giving the command:

```
skm>(destroy mod1)
```

You can substitute the variable *mod1* for `"ReadImg"` whenever you choose. This is important when you launch more than one copy of the same module (see **"Creating More Complex Scripts"**).

Similarly, you can save information about a *connect* command in the variable *c1*:

```
skm>(define c1 (connect "ReadImg" "Output" "SharpenImg" "Img In"))
```

This particular connection can be referenced later as `c1`. For example, you can break the connection using either of these commands:

```
skm>(disconnect "ReadImg" "Output" "SharpenImg" "Img In")
```

or

```
skm>(disconnect c1)
```

Variables containing module and connection information remain even after the IRIS Explorer objects to which they refer are gone, so you can reconnect the modules just disconnected with the command:

```
skm>(connect c1)
```

Example 6−1 An Example Script

Here is an example of a script that demonstrates the use of *define* in simplifing module connections. These modules do not currently exist, but you can use the script as a template and substitute your own module and port names in the appropriate places.

```
(define m1 (start "GenLat"))
(define m2 (start "testLattice"))
(define m3 (start "testLatticeFortran"))
(define m4 (start "testLatticePrint"))
(define m5 (start "ReadtstLattice"))
(define m6 (start "WritetstLattice"))
(set−param m6 "Filename" "foo.foo")
(define c1 (connect m1 "Output" m2 "Input"))
(define c2 (connect m2 "new" m6 "SuperLattice"))
(define c3 (connect m2 "new" m4 "SuperLattice"))
(define c4 (connect m1 "Output" m3 "Input"))
(define c5 (connect m3 "new" m6 "SuperLattice"))
(define c6 (connect m3 "new" m4 "SuperLattice"))
(define c7 (connect m5 "SuperLattice" m4 "SuperLattice"))
(define c8 (connect m6 "Filename" m5 "Filename"))
(disconnect c8)
(fire m1)
(fire m5)
```

Saving a Script

To use the script, save it with the *.skm* suffix, for example, *Testlat.skm*, then start IRIS Explorer with the command:

```
explorer −script Testlat.skm
```

6.1.3. Advanced Uses

Skm is a comprehensive language that provides the capability for creating much more sophisticated scripts than are described in this chapter. For more information on the Skm language syntax, read **Appendix B, "The Skm Language"**.

6.2. Using the Skm Commands

You can use the Skm commands to manipulate modules and maps in the Map Editor. Some of the commands let you emulate options available from the Map Editor and module pop−up menus. Using a Skm script, you can:

- launch a module and start maps and applications
- fire a module
- enable and disable selected modules or complete maps
- switch execution highlighting on and off
- destroy selected modules or complete maps
- make and break connections between modules
- edit a map, including cutting, pasting and duplicating selected modules
- manipulate module parameters
- save maps
- quit IRIS Explorer

Table B−1 in Appendix B lists, in alphabetical order, the Skm commands you can use to perform functions in IRIS Explorer.

The structure of these commands, plus any arguments they may take is listed in **Table B−2**, of Appendix B.

6.2.1. Command Syntax

In the Skm scripting language, all commands precede arguments and operands, and all expressions are enclosed in parentheses. If you want to say "do something," the Skm expression is:

```
(do something)
```

For example, to add two numbers, the addition operator precedes the arguments. Numeric constants do not require quotes:

```
skm> (+ 1 2)
3
```

White space separates operands and operators, as well as commands and arguments. You can use the space character, a tab, or a "newline" character. Non−numeric constants without spaces are preceded by a single quote, for example:

```
'all
```

Character sequences that include spaces should be enclosed by double quotes.

6.2.2. Interactive Help

To obtain details on the syntax of a command from within Skm itself, type **`(help)`** at the Skm prompt (**`skm>`**). You can also type **`(help−command "prefix")`** and you will get a list of all commands beginning with the given prefix.

Example 6−2 Using Skm Commands

You can use the commands in several different ways (see **Table 6−3**).

Command	**Action**
(copy "Contour")	Copies one module
(copy '("GenLat" "Contour"))	Copies a symbol
(copy (all−mods))	Copies the list of all modules

Table 6−3 Using Commands

6.3. Creating More Complex Scripts

This section describes the Skm command syntax in more detail to help you create flexible scripts for performing activities in IRIS Explorer.

6.3.1. Specifying Symbols

Symbols are identifiers that are preceded by a single quote. For example:

```
(maxi "modulename" ’at X Y ’size W H)
```

You can then define `var` as a symbol:

```
(define var ’size)
```

Setting Off Comments

The semicolon is the comment character in Skm. It comments out the remainder of the line on which it is located, for example:

```
; launch the Pick map and list the active modules
```

Using the Define Command

You assign values to a variable using the *define* command. For example, to define a variable called `myVar` and assign it the value `simple`, you write:

```
(define myVar simple)
```

To display the value of a variable, type the name of the variable at the Skm prompt without using parentheses, for example:

```
skm> myVar
simple
```

The assignment of a module to a variable is important because the actual name of the module depends on whether or not there are other copies of the module running.

If, for example, you use the command *start* to launch a copy of *ReadImg*, but there is already another active copy of *ReadImg* in the Map Editor, your module will be named *ReadImg<2>*. If you then use a *connect* command that explicitly refers to *ReadImg*, you will be referencing the wrong module.

The solution is to avoid referring to modules by the name used in the *start* command. Instead, assign the result of the *start* command to a variable and use the variable for subsequent references to the module.

For example,

```
(define mod1 (start "ReadImg"))
(connect mod1 "Output" "SharpenImg" "Img In")
```

6.3.2. Testing Variable Types

Skm provides the commands *module?* and *connection?* to test the type of a variable. They return the value `()` for false or `t` for true. For example:

```
skm> (module? m1)
t
```

means that the variable `m1` contains information about a module started some time in the past. It does not, however, necessarily mean that the module still exists.

Similarly, when *connection?* returns a value of `true`, it indicates that a particular connection once existed and information about that connection persists, but the connection itself may possibly have been broken.

Creating a Procedure

You can create a procedure in Skm with the *define* and *procedure* commands. This example demonstrates how to define a procedure that takes a single argument, declared formally as *a*, and adds 1 to it. The value returned by the procedure, if any, is the last value of the procedure.

```
(procedure (a) (+ a 1))
```

This procedure does not have a name yet (procedures can be "anonymous" in Skm). To give the procedure the name *plus1*, type:

```
(define plus1 (procedure (a) (+ a 1)))
```

You can invoke this procedure as follows:

```
skm> (plus1 3)
4
```

Using a Procedure

This example shows how to carry out the action of connecting image processing modules in IRIS Explorer. Image processing modules that operate on a single input typically have an input port called "Img In" and an output port named "Img Out."

First you define a procedure that will connect specified ports in two different modules. This procedure simplifies the connection of the modules by supplying the port names automatically:

```
(define image−conn
(procedure ( mod1 mod2 )
(connect mod1 "Img Out" mod2 "Img In")))
```

Then you invoke the procedure and provide the names of the modules to be connected:

```
(image−conn "SharpenImg" "DisplayImg")
```

6.4. Output from Skm

You can use the *print* command to print the output from a Skm command or script to the "IRIS Explorer console" (either the shell from which IRIS Explorer was launched or the Skm Editor Window, depending if IRIS Explorer was started with "−script −" or "−script %"). *print* takes one or more arguments; here are two examples of its use:

```
skm> (define v1 "abc")
skm> (define v2 123)
skm> (define m1 (start "ReadImg"))
skm> (print v1 v2)
abc123
```

and

```
skm> (print "My module is " m1)
My module is #< IRIS Explorer Module Ref: ReadImg >
```

6.5. The Skm Editor Window

If you started Explorer with the "explorer −script %" option, Explorer will open a dedicated window in which Skm interactions can take place, see **Figure 6−1**. The Skm Editor Window can be opened and closed by toggling the "Skm

Window" option from the Map Editor’s Layout menu.

Figure 6−1 The Skm Editor Window

This main window is divided into four key areas described in the following subsections.

6.5.1. The Skm Output Viewer

All output from Skm is directed here, including the commands you type and the Skm cursor. One of the main advantages of using IRIS Explorer in this mode is that Skm output is separated from general IRIS Explorer standard output, for instance from modules. This makes it easier to keep track of what you have typed in Skm.

6.5.2. The Skm Command History Window

This window stores a list of commands previously entered in the Skm Command Input window. You can use the arrow keys to scroll up and down the list (you can also use the arrow keys, with the same effect, in the Skm Command Input window). When using the Command History Window, the currently selected item in the history list will be replicated in the Skm Command Input window. To re−run this command you can either "double click" on the return/enter key here or alternatively a single return/enter in the Skm Command Input window will execute the command.

6.5.3. The Skm Cursor

This tells you what state Skm is currently in, the possibilities are:

- **Skm>**
  This means Skm is ready to accept input. Type away!
- **Skm (**<integer>**) >**
  This means you have already typed something into Skm put have not finished entering the Skm command. A Skm command is complete when the number of left parenthesis match the number of right parenthesis. The integer in the bracket represents the number of unclosed right parenthesis.
- **Skm ...**
  This means Skm is busy. Further input will not be accepted until the interpreter is ready.

6.5.4. The Skm Command Input

This is where you type Skm commands. Like other text input areas you can "cut and paste" text out of and into the Skm Command Input. Press "return" or "enter" to pass the command input to the interpreter. Each command line will be stored in the Skm Command History window (see above). A single Skm command can be split over more than one command line.

Chapter 7

Using the DataScribe

This chapter explains the DataScribe environment and describes how you can use it to create data conversion modules. You can use these modules in the Map Editor to read external array data into the IRIS Explorer and also to write data out to external files. In the DataScribe, you create scripts and control panels for the conversion modules. The chapter covers:

- the components of the DataScribe
- introducing foreign data into IRIS Explorer
- using data elements in a template
- working with binary files
- working with arrays and lattice structures
- creating conversion scripts from templates
- creating a module control panel for the DataScribe module
- tracking errors

7.1. Overview

The DataScribe is a utility for converting users' ASCII and binary data into and out of the IRIS Explorer lattice data type. ASCII data must be in the form of *scalars* and/or *arrays*. A scalar is a single data item, and an array is a structured matrix of points forming a lattice. You can convert data into and out of IRIS Explorer lattice format by building a data conversion *script* in the DataScribe.

The script is a file consisting of *input* and *output templates* and their wiring. The templates contain information, in the form of *glyphs*, about the data types that you want to convert. Glyphs are icons that represent each data type and its components in visual form. The wiring connections map an input data type to its output data type. For example, you may have arrays in an ASCII file that you want to bring into IRIS Explorer. The glyphs in the input template represent the type of array you have (such as 2D or 3D), and those in the output template indicate how you want your array data to appear in IRIS Explorer. For example, you can select a small, interesting section from a large and otherwise uninteresting array by connecting input and output glyphs so that only the data you want is converted into the IRIS Explorer lattice.

Once the script file has been set up, you can create a control panel for it and save it as a module. You then set up a map that includes your newly created module, and type the name of the file that contains the data you want to convert into the module's text slot. When the map fires, your conversion module transforms the data according to the specifications in the conversion script. You can use this new module as often as you like.

The DataScribe has three main functions:

- to convert data from an external source in ASCII or binary format, such as an application or disk file, into IRIS Explorer lattices
- to convert to and from different data types within IRIS Explorer itself. For example, converting a single−channel dataset (`nDataVar=1`) for an image to a three−channel dataset (`nDataVar=3`)
- to convert data from one file format to another. For example, from ASCII to binary or vice versa

7.2. The DataScribe

The DataScribe window is the work area in which you assemble templates to create a data conversion script. The templates may contain data glyphs, parameters, or constants; the *overview* pane of the DataScribe window shows the wiring scheme that defines the rules for data conversion in each script.

You can use the Control Panel Editor to edit the control panel of a completed module script if the module has parameters. If not, the DataScribe generates a default control panel. For more details, see **Chapter 4, "Editing Control Panels and Functions."**

7.2.1. Opening the DataScribe

To bring up the DataScribe, type `dscribe` at the prompt (%) in the shell window.

The DataScribe window and the Data Type palette appear.

7.2.2. Quitting the DataScribe

To exit from the DataScribe, select *Quit* from the File menu. The Data Type palette closes automatically along with the DataScribe.

7.2.3. Scope of the DataScribe

The DataScribe can only act on data arranged in ordered matrices or arrays, including the IRIS Explorer lattice data type. The modules that it creates can have parameter inputs (see **"Making a Parameters Template"** whose values can be controlled from the module control panel. However, it does not handle:

- the IRIS Explorer geometry, pyramid, or pick data types
- conditional expressions for selecting data or traversing data in files

7.3. Creating Scripts and Templates

In each DataScribe session, you create or modify a script. A script consists of a number of templates along with rules that stipulate how data items are associated with templates. A complete script has one or more input templates and one or more output templates, with their wiring connections. Each template describes the structure of a single file in terms of data types, shown as glyphs. The wiring connections show how each input data type will be converted into an output type.

When you add a control panel to a script, you have a complete module.

You can save a completed script, which consists of a *.scribe* file and an *.mres* file. The *.scribe* file contains the wiring specifications for the templates and the *.mres* file contains widget specifications (if any) for the control panel.

To create a script, you must first create the templates. **Figure 7−1** shows how three input templates and two output templates can be combined to form a script as part of an IRIS Explorer module.

Figure 7−1 Templates and Scripts

7.3.1. Setting Up Templates

You must set up a separate template for each data file you plan to read data from or write data to, but you can have several input and output templates in a script. Special templates exist for describing user−defined constants and parameters associated with widgets (see **"Defining Parameters and Constants."**)

Note: Template, glyph, script, and component names in the DataScribe should not contain blanks.

Opening a New Template

You create input and output templates in the same way. They are distinguished by the arrow icon in the top left−hand corner of the template (see **Figure 7−2**).

Figure 7−2 Template Icons

To create a new template:

1. Open the Template menu and select *Template*. The New Template dialog box appears (see **Figure 7−3**).

   You use it to set the general properties of the template, such as its name, whether it is an input or output template, and the format of the data to be converted (ASCII, binary, or IRIS Explorer).

2. Type a name for the template and click on the radio toggle buttons to select the other attributes.

   A small icon (see **Figure 7−2**) to the left of the maximize button (top right corner of the template) indicates the file format you selected. See **Figure 7−4** for an example which includes an input ascii template and an output IRIS Explorer template.

Figure 7−3 New Template Dialog Box

   If you select an input filetype of *Binary*, you can specify the record structure of the file. The default, *Maybe*, checks for a Fortran record−structured file and converts the data accordingly. If you know for sure whether the file is Fortran record−structured or not, you can select *True* or *False*.

3. Click on the *OK*. A template opens in the DataScribe, and its wiring overview appears in the Overview pane (see **Figure 7−4**). You can open as many input and output templates as you need.

   The input template appears on the left of the DataScribe window and the output template on the right. You can distinguish them by the arrow icon next to the template name.

   The DataScribe View menu can be used to hide and redisplay the various panes in the main window. Thus, for example, you can toggle the appearance of the "Outputs" pane, if required. If one pane is switched off, the dividing line down the center of the DataScribe disappears, and the displayed template appears on the left.

To select a template for editing, click on the arrow icon. A selected template is outlined in black (see **Figure 7−4**)

Figure 7−4 Input and Overview Templates

To change the data type for a template once it has been defined, or to make notes about a template, select the template by clicking on its arrow glyph. Then select *Properties* from the Edit menu and use the Glyph Property Sheet (see **"Using the Glyph Property Sheet"**) to make changes.

Using the Overview Pane

Input and output templates are wired together in the same way that modules are wired together in the Map Editor. The wiring overview in the Overview pane displays the actual wiring connections between input and output templates. You can connect input and output template ports in the DataScribe window or use the ports on the overview templates in the Overview pane, but you will see the wires only in the Overview pane (see **Figure 7−4**).

You can list all of a template’s ports from its wiring overview by clicking on the port access pad with the right mouse button. For more information on creating ports, see **"Glyphs."**

You can hide the Overview pane by toggling "Overview" on the View menu.

7.4. The Data Type Palette

The DataScribe Data Type palette (see **Figure 7−5**) displays the data types available for data transforms in IRIS Explorer. Data types are arranged in the palette in order of complexity and are identified by color−coded icons, called *glyphs*, which are organized into scalars, arrays, lattices, and sets.

The palette is displayed when the DataScribe starts up. Selecting "Palette" on the View menu restores the window if it has been hidden or deleted.

To move a glyph from the palette to a template, select it from the pallete using the cursor, then press the left mouse button, drag it onto the template and release the button. You can also select the template, then hold down **`<Alt>`** and click on the glyph. You can reorder glyphs in the template by dragging them around and dropping them into the new position.

Figure 7−5 DataScribe Data Type Palette

7.4.1. Glyphs

Glyphs are visual representations of data items that, when collected together in various combinations, define the contents of each input and output template. Each data category is distinguished by the color and design of its glyph (see **Figure 7−5**).

Glyphs in the Data Type palette are *closed*. When you drop a closed glyph into the template window, it opens into the *terse* form (see **Figure 7−6**).

Note: Each time you place a glyph in a template, ports are created for each part of the data item that can be wired to another template (see **"Connecting Templates."**)

Each glyph in a script should have a unique name. DataScribe assigns a default unique name (of the form "Array1", "Array2", etc) to each glyph as it is added to a template. You can change this by selecting the name of the glyph using the left mouse button, then typing the new name.

Scalar glyphs have two open forms, depending on the template in which the scalar appears. Array, lattice, and set glyphs have two open forms, terse and *verbose*. The terse form opens into the verbose glyph, which shows the group structure of the glyph, in the form of more glyphs (see **Figure 7−6**).

Click on the *Open/Close* button at the right end of the slider to toggle between the terse and verbose form of the glyph in a template.

Figure 7−6 Glyph Forms

The Shape Icon

The arrow in the shape icon indicates the direction in which the axes are ordered (see **Figure 7−7**). The fastest−varying axis is at N1, that is, the tail of the arrow, and the slowest−varying axis is at N2 (2D) or N3 (3D), that is, the head of the arrow.

Data is stored in the direction the arrow is headed, so it is stored first at the arrow's base. You can change the name of the axis, but not the direction in which the data varies.

You can set the starting and ending values of any vector and array shape icons to either constant or variable values. To define the shape of an array, you enter a starting value that is used for all dimensions of the shape and an ending value for each separate dimension.

Figure 7−7 The Shape and Node Type Icons

The Node Type Icon

The default node type in the array glyphs is the integer data type. You can change the data type by dropping a different glyph onto the node type icon. For example, **Figure 7−8** shows the *4D Array* glyph from **Figure 7−7**, but with the array replaced by a *Set* glyph.

Figure 7−8 A New Array Structure

Note: Most of the glyphs within the lattice glyph are *locked* so that their node type cannot be changed (to maintain equivalence with the definition of the lattice data type). The exception is the lattice data glyph whose node type *can* be edited to produce a lattice of floats, or of integers, etc. See **"Lattice Glyphs"** for more information.

To replace a node type in an array or set glyph, select the glyph you want from the Data Types palette and drop it exactly on top of the node type slot. This inserts the new glyph into the node type slot.

Note: Be careful when you position the new glyph over the node type slot. If the new glyph is dropped over the title bar, it replaces the complete array glyph, not just the node type.

You can replace the node type with a scalar, array, or set glyph. For example, each array in **Figure 7−7** has a different node type. If you drop an array or set glyph onto the node type icon, you can construct a data hierarchy. For example, if you open the vector glyph in the *4−D Array* glyph in **Figure 7−7**, you see the nested vector structure (see **Figure 7−9**), which must in turn be defined. The new data structure consists of a 4D array, where a vector exists at each node. For more information, see **"Hierarchy in Arrays."**

Figure 7−9 A Hierarchical Array

Component Menu Button

The Component Menu button (see **Figure 7−6**) lets you list an isolated segment of an array as a separate item, which can then be connected to other templates. For example, you can list all the *x*−coordinate values of a 3D dataset, or all the pressure values in a pressure/temperature dataset. For information on isolating the fragments, see **"Selecting Array Components."**

For more information on glyphs, see **"Data Structures."**

7.4.2. Building a Script

This example illustrates how to set up a script that will read a 2D array of ASCII into an IRIS Explorer lattice. You require a data file, an input template, and an output template.

- The Data File

  The data file contains two integers (7, 5) that define the size of the subsequent 2D array (7 by 5), followed by the actual data:

```
 7  5
90 85 73 57 40 25 14
81 77 66 51 36 23 13
60 57 49 38 27 17  9
36 34 29 23 16 10  6
18 17 14 11  8  5  3
```

  Create this file in your */usr/tmp/* directory and save it as *dsExample1.data*. You now have a 7 by 5 matrix of data in a file. To get the data into an IRIS Explorer lattice, you need to create a module to read data in this format.
- The Input Template

1. Start the DataScribe and select *Template* from the Template menu. When the New Template Dialog box appears, type in a new name such as *ArrayFile*, and select *OK*.

   A new template pane appears on the input side of the DataScribe window, with the wiring overview in the Overview pane.
2. Select a `vector` glyph from the Arrays list on the Data Types palette and place it in the *ArrayFile* template. This glyph will be used to read in the dimensions of the 2D array, as specified in the first line of the input file.

Figure 7−10 An Open Vector Glyph

3. Open the vector glyph by clicking on the *Open/Close* button at its right edge. The glyph has these active areas:

   − a name ("Array1"),

− a primitive type (`integer`)

− a starting value (`1`)

− an ending value (`N`)

4. To set the vector length to 2, select `N` and replace it by typing **`2`** in the text slot. Now you have defined a file structure with two integers in it.
5. To change the name of this vector, select the text "Array1" and type a new name, for example, *res*, in its place. The name *res*, short for "Resolution," reminds us that this variable contains the dimensions of the 2D dataset.
6. Drag a 2D Array glyph from the palette and drop it into the *ArrayFile* template (see **Figure 7−11**). Open the glyph by clicking on the button.

Figure 7−11 A Simple DataScribe Script

7. Set the size of the array by clicking on the ending value for each dimension of the shape. The size of the array is defined by the values stored in the *res* vector, so you must:

   − Select `N1` in the 2D Array shape and replace it with the first value of *res*, *res*[1].

   − Replace `N2` with *res*[2].

   In the case of our *dsExample1.data* file, *res*[1] equals 7 and *res*[2] equals 5.

This completes the description of the sample file contents, and you have finished constructing the *ArrayFile* template.

- The Output Template

1. Open a new template from the Data menu, but this time, make sure that:

   − the first radio button is set to *Output* instead of *Input*

   − the second radio button is set to *Explorer*, not *ASCII*

   You can leave the name of the template unchanged from the default which DataScribe selects for it.
2. Select a 2D Unif (two−dimensional uniform) lattice glyph from the palette and drop it into the output template. Rename the lattice glyph from *Lat1* to *OutLat.*
3. Open the glyph to see the internal structure of this lattice data type. You need to specify only one thing, the number of data variables per node (*nDataVar1*). So,

   type **`1`** into the text slot to the right of *nDataVar1*

Wiring Input and Output Templates

You associate items in the input template (*ArrayFile*) with their counterparts in the output template (*Lattice*) by wiring them together, just as in the Map Editor. You can use the ports on the templates themselves, or on the overview templates. The wires appear only on the overview templates (see **Figure 7−11**).

Figure 7−12 Overview Output Port

To make the connections:

1. Click the right mouse button on the output pad of the *ArrayFile* template or overview. The port menu pops up (see **Figure 7−12**).
2. To set the size of the lattice, select *res* from the port menu then click on the input pad of *LatticeFile*, and select *dims1.* A blue wire appears, indicating that a connection has been made.
3. To specify the data, connect the *Array2* output of the *ArrayFile* port to the *data1* input of the *LatticeFile* port in the same way.
4. To save the script, select *Save As* from the File menu and type a filename, such as *dsExample1,* into the file browser. The DataScribe saves the module in the default module directory with a default control panel.

You have now built a complete DataScribe module, which you can use in the Map Editor. When you type the name of your ASCII data file into the datafile slot, the DataScribe module fires and converts the data into a 2D IRIS Explorer lattice (see **Figure 7−13**).

Running IRIS Explorer

1. Start up IRIS Explorer by typing **`explorer`** at the % prompt.
2. Launch the DataScribe module, *dsExample1*, from the Module Librarian. The *Script File* slot displays the name of the script, *dsExample1.scribe*.
3. In the empty *ArrayFile* text slot on the control panel, type the filename */usr/tmp/dsExample1.data*, which you created at the start of this exercise.
4. Launch the *NAGContour* module from the Module Librarian. Connect the *OutLat* output from *dsExample1* to the *InLat* port of the *NAGContour* module.

You should see a set of contour lines in *NAGContour*'s display window (see **Figure 7−13**). You can modify the number of contour lines by adjusting the *Contours* slider on the *NAGContour* module control panel. By default, the module selects the heights for the contour lines automatically, based on the range of the data and the number of lines requested. Try changing the heights by setting the *Mode* parameter to **`Number of Contours`** or **`Contour Interval`** and setting the levels using the dial widgets.

Figure 7−13 The DataScribe Module in the Map Editor

You might like to use your map to look at another example file which is in the *dsExample1* format. You can find the file in */usr/explorer/data/scribe/dsExample1.data*. Finally, you may like to look at the shipped version of the *dsExample1* module, which is in */usr/explorer/scribe.* You can view the script in DataScribe using the command:

```
dscribe /usr/explorer/scribe/dsExample1
```

7.5. Designing Templates for Your Data

A template describes a file structure in terms of generalized data types. In the DataScribe, you can use the same template to define different components of a data type at different times, which means that:

- You can use the same template for all files of the same structure, not just the one for which it was written.
- Your data transform may contain several input templates that are connected to a single output template. For example, when you convert a PLOT3D dataset, which consists of two files, into an IRIS Explorer lattice, you set up a separate input template for each file, but only one lattice output template.
- Similarly, you may have a single input template, but several output templates. For example, a file may have five components, each of which you want to visualize as a lattice. You can define a separate output template for each of the components, and select the one you want when you fire the map.
- You need not convert or even specify all the data in a particular dataset. You can select the parts that you want to visualize using the Component Dialog window (see **"Component Menu Button"**) and convert only those. For example, if you have a vast amount of data about a weather front, you can extract the pressure and velocity data, but ignore the temperature, humidity, and turbulence data.

7.6. Reading in Data

You can read data files in ASCII, binary, and IRIS Explorer formats into a DataScribe template, which is created using the New Template Dialog box (see **Figure 7−3**). If you select the binary file option, the DataScribe can check to see whether the file is a Fortran−generated file with record markers. It then converts the data accordingly.

7.6.1. Checking EOF

The DataScribe can read to the end of an ASCII or binary data file (EOF) and calculate its dimensions internally without your having to stipulate the file length, provided that you have no more than one undefined dimension variable per template. The dimension variable can appear only in array glyphs or sets, not in lattices (all dimensions must be defined in an IRIS Explorer lattice).

The undefined dimension variable must be:

- in a top−level array glyph. For example, you cannot have an undefined dimension variable in an array within a lattice or in a nested array.
- in the final item listed in the template. That is, an array glyph that contains an undefined dimension variable cannot be followed in the template by another glyph.
- the slowest−varying variable in the array.

It will be assigned a value that you can use in your output.

Example 7−1 Modules that check EOF

Three examples of DataScribe modules which feature EOF checking are in */usr/explorer/scribe/EofSimple.**, */usr/explorer/scribe/EofSelectData.** and */usr/explorer/scribe/EofAllData.**.

The *EofSimple* script reads in an ASCII data file, an example of which is in */usr/explorer/data/scribe/EofSimple.data*. This contains a 3 by N 2D array, where N is unspecified explicitly in the file. (In the case of the example file provided, N = 10.) The other two files can read the file */usr/explorer/data/scribe/Eof.data*, which contains a 1.19 Angstrom resolution density map of crystalline intestinal fatty−acid binding protein

Footnote: Crystallographic data by James C Sacchettini, Giovanna Scapin, Peter Reiner. Albert Einstein College of Medicine.

. Here, the file contains an 18 by 29 by N 3D array (or N slices, each of size 18 by 29), where N is to be determined by the reader. (Again, in the specific case of the example file provided, it turns out that N = 27).

You can run the *EofSimple* and *EofSelectData* modules in the Map Editor and wire them to *PrintLat*, which will print out the results. The *EofSelectData* module uses a parameter, *SelectData*, to extract a slice from the 3D dataset; see **"Defining Parameters and Constants"** for more information on this.

To illustrate the action of *EofAllData*, wire this map:

EofAllData to *IsosurfaceLat* to *Render*

Set the Threshold dial on *IsosurfaceLat* to 1.0 and look at the results.

7.6.2. Reading in Lattices

To read in an IRIS Explorer lattice that has been saved as an ASCII or binary file, open an ASCII or binary template file and place an IRIS Explorer lattice glyph in it. However, you cannot add any other glyphs, either lattices or other data types, to an ASCII or binary template that already contains an IRIS Explorer lattice glyph. DataScribe will issue a warning message if you try.

To convert two or more lattices in ASCII or binary format, you must place each lattice in a separate template file.

You can place any number of lattices in an IRIS Explorer template for conversion to or from the IRIS Explorer lattice format, but IRIS Explorer templates accept no other glyph type.

Each lattice glyph that is placed inside an IRIS Explorer template appears as a port in the module. The port will be on the input or output side, depending on which side of the script the template appears. The name of the lattice is used as the name of the port.

To see an example of a module that manipulates lattices, look at the *Swap* module in */usr/explorer/scribe*, which accepts a 2D image lattice that consists of a sequence of RGB bytes, swaps the contents of the red and blue channels and outputs the modified lattice.

7.6.3. Reading in Binary Files

The */usr/explorer/scribe* directory contains two example modules that handle binary files.

LatBinary reads in an ASCII file containing a 2D array in and outputs a 2D uniform lattice as well as the data converted to binary form. The data is in */usr/explorer/data/LatBinary.data.*

ReadBinary reads in a 2D array from a binary file and outputs the data as a 2D uniform lattice. You can use the data file generated by *LatBinary* as a sample input data file.

7.7. Data Structures

The DataScribe processes arrays, including complex 1D, 2D, 3D, or 4D arrays, which may also be nested or hierarchical. For example, the DataScribe can handle a matrix (a 2D array) of 3D arrays. Each node of the matrix will contain a volumetric set of data. To set up a template, therefore, it is necessary to understand how arrays work.

An array is a regular, structured matrix of points. In a 1D array, or *vector*, each node in the array has two neighbors (except the end points, which each have only one). In two dimensions, each internal node has four neighbors (see Figure 3−5 in the *IRIS Explorer Module Writer's Guide*). An internal node in an nD array has 2^n neighbors. This structure is the computational space of the array. To locate a particular node in the array, use array indices (see**"Using Array Indices in Expressions."**)

The DataScribe Data Types palette (see **Figure 7−5**) lists the data items you can use to define elements of your data files in the DataScribe templates. The items are organized according to their complexity into scalars, arrays, lattices, and sets and visually represented as color−coded icons or *glyphs*. Scalars are always single data items. Arrays, lattices, and sets are made up of a number of different data elements packed into a single structure.

Each data type and its visual glyph are fully described in the next few sections. **Table 7−1** lists the data elements in each category in the DataScribe.

Category	Elements
Scalars	Char
	Short (signed and unsigned)
	Integer (signed and unsigned)
	Long (signed and unsigned)
	Float
	Double
Arrays	Vector
	2D Array
	3D Array
	4D Array
	Patterns
Lattices	Uniform lattice (Unif Lat): 1D, 2D and 3D
	Perimeter lattice (Perim Lat): 1D, 2D and 3D
	Curvilinear lattice (Curvi Lat): 1D, 2D and 3D
	Generic input lattice: 1D, 2D and 3D
Sets	Any collection of scalars, arrays, and/or sets

Table 7−1 DataScribe Elements

7.7.1. Scalars

Scalars are primitive data elements such as single integers, floating point numbers, single numbers, bytes, and characters. They are used to represent single items in a template or to set the primitive type of an array. Their bit length is dependent on the machine on which the script is run, and corresponds to the C data type as below. **Table 7−1** lists the scalars and their representation.

Category	Scalar

Signed integers	Integer: int type
	Long: long int type
	Short: short int type
Unsigned integers	Uint: unsigned int type
	Ulong: unsigned long int type
	Ushort: unsigned short int type
Floating point numbers	Float: float type
	Double: double type
Characters	Char: a single ASCII character (8 bits)

Table 7−2 Scalar Values

7.7.2. Scalar Glyphs

Figure 7−14 A Scalar Glyph

Scalars are unit data items. Their glyphs have a text type−in slot (see **Figure 7−14**), where you can enter a name for each particular manifestation of the glyph. For example, you can name an integer glyph *Scalar_Integer.* The glyph structure is the same for all scalars not used in Constants templates.

In a Constants template, the Scalar glyph has a third field, scalar value. You must set this value when you use a scalar in a Constants template. You cannot set it in other types of template.

Note: Glyph names should not contain spaces.

7.7.3. Arrays

Arrays are homogeneous collections of scalars, other arrays, or sets. All values in a single array are of the same scalar type, for example, all integer or all floating point. They range from 1D arrays (vectors) to 4D arrays. The starting indices and dimensionality of the array can be symbolic expressions. The array data types include:

- Vector: a 1D homogeneous array
- 2D Array
- 3D Array
- 4D Array
- Pattern: a text−based pattern (see **"Using a Pattern Glyph"** for more information on its use)

7.7.4. Array Glyphs

Array glyphs have complex structures with at least one sublevel (see **Figure 7−15**) in the node type. A node type may be a scalar, array, or set, but not a lattice. You can edit all the elements of a verbose array glyph.

Figure 7−15 An Array Glyph

Hierarchy in Arrays

You may want to create nested data structures in your arrays, or extract portions of very large datasets. You can drop any of the scalar or array data types into the node type slot. This means that a 2D array can be set to `integer` by dropping an Integer glyph onto the node type slot, or it can be set to `vector` by placing a Vector glyph in its node type slot.

When more than one layer exists in an array, it becomes hierarchical (an array of arrays). For example, a 2D image (red, green, blue, opacity) can be represented as a 2D array with a nested vector (of length 4) at each node. Similarly, a 3D velocity field is a 3D array with a vector at each node. The vector is of length 3 for V_x, V_y, and V_z components. A pressure tensor in a 3D field is a 3D array with a 2D array of size 3 by 3 containing the nine components P_{xx}, P_{xy}, P xz, ... P_{zz}

Footnote: J.P.R.B. Walton, D.J. Tildesley and J.S. Rowlinson, "The pressure tensor at the planar surface of a liquid", Mol. Phys., 1983, **48**, 1357

at each node (see **Figure 7−16**).

Figure 7−16 A Complex 3D Array

The glyph for the nested array is similar to a closed array glyph. It has a node type icon, a label, and an *Open/Close* button. If you click on the button, the glyph opens to reveal the internal structure of the 2D array. This array is offset from its parent to show containership. The nested 2D array in the example of **Figure 7−16** contains stress information at each node of the 3D array. You can have as many levels as you need.

You can also use array indices to help streamline your efforts.

Using Array Indices in Expressions

Arrays are stored using a column−major layout (the same convention as used in the Fortran programming language), in which the *i* direction of an (*i,j,k*) array varies fastest. However, the DataScribe uses the notation of the C programming language for array indexing. So, the *i*th element of an array is `Array[i]`, but for a 2D array the (*i,j*)th element is `Array[j][i]`. Similarly, the node located at (*i,j,k*) is `Array[k][j][i]`.

The (*i,j,k*) axes correspond to the (*x,y,z*) physical axes. You can use zero− or one−based addressing in the arrays if you wish; one−based is the default.

Since *i* varies fastest, for a 2D uniform lattice the Xmax element of the bounding box is bBox[1][2] and the Ymin is bBox[2][1]. These are laid out in memory as follows:

```
bBox[1][1]   (Xmin)
bBox[1][2]   (Xmax)
bBox[2][1]   (Ymin)
bBox[2][2]   (Ymax)
```

This is important when you use a set to define a collection of scalars that you later connect to an array.

7.7.5. Lattices

IRIS Explorer lattices exist in three forms (1D, 2D, or 3D) with uniform, perimeter or curvilinear coordinates. The lattice data type is described in detail in Chapter 3, "Using the Lattice Data Type" in the *IRIS Explorer Module Writer's Guide.*

7.7.6. Lattice Glyphs

Lattice glyphs are pre−defined sets that contain array and scalar glyphs. Each type of lattice glyph has the correct data elements for that particular IRIS Explorer lattice. For examples, see **"Lattice Types."**

You can replace only the node type icon in the data array. If you try to replace another node type icon, the DataScribe informs you that the glyph is locked and cannot be altered.

Lattice Components

A lattice has two main parts: *data*, in the form of variables stored at the nodes, and *coordinates*, which specify the node positions in physical space. It also has a *dimension variable* (*nDim*) and a vector (*dims*) defining the number of nodes in each dimension. *nDim* and *dims* are the same for both the data and coordinate arrays of the lattice.

The data in a particular lattice is always of the same scalar type; for example, if one node is in floating point format, all the nodes are. Each node may have several variables. For example, a color image typically has four variables per node: red, green, blue, and degree of opacity. A scalar lattice has only one variable per node.

All lattice data is stored in the Fortran convention, using a column−major layout, in which the *I* direction of the array varies fastest. When you are preparing a data file, make sure that your external data is arranged so that it will be read in the correct order.

Coordinate values are always stored as floats. Their order depends on the lattice type (see **"Lattice Types"**). For *curvilinear* lattices, you can specify the number of coordinate values per node. This describes the physical space of the lattice (as opposed to the computational space, which is the topology of the matrix containing the nodes and is determined by *nDim* and *dims*). See Figure 3−13 in the *IRIS Explorer Module Writer's Guide* for various examples of curvilinear lattices.

Table 7−1 lists the components of an IRIS Explorer lattice. When you set up a template to bring external data into an IRIS Explorer lattice, these are the groupings into which your data should be fitted.

Component	Description
nDim	the number of computational dimensions
dims	a vector indicating the number of nodes in each dimension
nDataVar	the number of data variables per node
primType	the scalar type of the variables (byte, long, short, float, or double)
coordType	the type of physical mapping: − uniform: no explicit coordinates except bounding box coordinates − perimeter: enough coordinates to specify a non−uniformly spaced rectangular dataset − curvilinear: coordinates stored interleaved at each node
nCoordVar (curvilinear lattices only)	the number of coordinate variables per node

Table 7−3 Lattice Components

Lattice Types

There are three IRIS Explorer lattice types, defined according to the way in which the lattice is physically mapped. The DataScribe, however, also offers a fourth, generic DataScribe lattice, which is an input lattice only. You cannot use it in an output template.

Each lattice you include in a script appears as a port in the final DataScribe module. If the lattice is in an input template, it will appear as an input port, and if it is in an output template, it will form an output port.

The first IRIS Explorer type, the *uniform* lattice, is a grid with constant spacing. The distance between each node is identical within a coordinate direction. For a 2D or 3D uniform lattice, the spacing in one direction may differ from that in the other direction(s). A 2D image is a classic example of this lattice type.

Coordinate values in a uniform lattice are stored as floats, in the C convention using row−major format.

Figure 7−17shows how a uniform lattice is represented in the DataScribe. The minimum and maximum values in each direction, contained in the 2D Array called *bBox1*, define the coordinate mapping. The data type in the nested *data* vector, which is a float in **Figure 7−11**, is important because it determines the data type of the data values in the lattice.

Figure 7−17 A 3D Uniform Lattice

The second type, the *perimeter* lattice, is an unevenly spaced cartesian grid. Finite difference simulations frequently produce data in this form.

A perimeter lattice is similar to a uniform lattice in layout; it differs only in the arrangement of its coordinates (see **Figure 7−18**). You must define a vector of coordinate values for each coordinate direction. If the lattice is 2D with three nodes in the X direction and five nodes in the Y direction, the lattice will have eight coordinates. The number of coordinates is the sum of the lattice dimensions array, in this case, `dims[1] + dims [2] + dims [3]`.

Coordinate values in a perimeter lattice are stored as floats in the C convention using row−major format.

Figure 7−18 A 3D Perimeter Lattice

The third type, the *curvilinear* lattice, is a non−regularly spaced grid or body−mapped coordinate system. Aerodynamic simulations commonly use it. Each node in the grid has *nCoordVar* values that specify the coordinates. For example, the coordinates for a 3D curvilinear lattice can be a set of 3D arrays, one for each coordinate direction. The length of the coordinate array is determined by the number of coordinates per node.

The curvilinear lattice glyph has a vector for both data and coordinate values. The data type in the vector determines the data type of the data and coordinate values, respectively, in the lattice. Coordinate values for curvilinear lattices are stored interlaced at the node level in the Fortran convention, the reverse of the uniform and perimeter formats.

Figure 7−19 A 3D Curvilinear Lattices

The fourth type, the *generic* lattice (see **Figure 7−20**), is a convenient boilerplate lattice for use in input templates only. It determines the primitive data type, the number of data variables, and the number of coordinate values from the actual lattice data. The node types are always scalars; you cannot use any other data type.

The generic lattice provides a level of abstraction one step above the closely defined uniform, perimeter, and curvilinear lattices. The advantage of using a generic lattice is that you can be less specific about the data and coordinates, and IRIS Explorer does more of the work. The disadvantage of using it is that you get only a long vector of data with lengths for coordinate information

For a more detailed description of the lattice data type, refer to the *IRIS Explorer Module Writer's Guide*.

Figure 7−20 A 3D Generic Lattice

7.7.7. Sets

The set glyph lets you collect any combination of data items you want into one composite item. You can include integers, floats, vectors, arrays, and other sets in a set. You can use a set to:

- design a new data item containing elements of your choice
- consolidate a small collection of items that you use frequently

To create a set, open a set glyph in the template, then drop glyphs for the data items you want into the open set. Once your set is saved, you can include it in a template instead of dealing with each item individually.

Saving a Set

If you have a custom set that you plan to use often, you can save it into a script and then read it into other scripts using "Append" from the File menu. You can use the Edit menu options to cut and paste the set into various templates.

For example, you can collect a group of scalars into a set and wire the set into an array in another template. This is useful for setting the bounding box of a lattice. A 2D uniform lattice requires an array that specifies the `[xmin, xmax]`, `[ymin, ymax]` bounding box of the data. You can wire a set consisting of four floats into the lattice to satisfy this constraint (see **Figure 7−21**).

Figure 7−21 A Set for a Bounding Box

Using Nested Sets

You can nest sets within other sets and within arrays. To nest a set in an array, drop the set glyph into the data slot in the array and then create the set by dropping glyphs into the set. For example, **Figure 7−22** shows an open set glyph that contains a 3D Array glyph, which in turn has a set glyph in the data type slot.

Note: You must promote any component selections from a set up to array level in order to wire them into another template. See **"Selecting Array Components"** for more details.

Figure 7−22 Nested Sets

The ASCII formatting feature, described in **"Converting Formatted ASCII Files,"** lets you retain or recreate the layout and spacing of ASCII data files in DataScribe input and output templates. The layout is set with the ruler in the Template Property Sheet. However, you cannot format data in a set that is nested within an array. If you want to use formatting in a set, the set must be a top−level set and may not contain any other sets itself.

7.8. Using a Pattern Glyph

Figure 7−23 A Pattern Glyph

Pattern glyphs (see **Figure 7−23**) are a special kind of array glyph that can be used only in input templates. You can use them to traverse character strings in files and search for patterns in them. The verbose pattern glyph looks similar to a vector but has three additional features:

- a comparison or termination string, which the DataScribe uses to determine the length of the pattern.
- a wildcard toggle button. When the button is on, you can do simple searches using the two UNIX wildcard characters: ***** (substitute with any string) and **?** (substitute with any single character)

  You can also use UNIX search expressions, such as *grep* or *egrep*, to search for a specific string. To activate this capability, open the Glyph Property Sheet for the pattern glyph and click on the "Reg Exprs" button (see **Using the Glyph Property Sheet**). Clicking on "Exact" or "Wildcards" on the Property Sheet is an alternative to toggling the wildcard button, the choice is shown on this button when "Apply" is selected.
- a *Case Sensitive* toggle button to activate case sensitivity. When it is on, the DataScribe matches the upper/lowercase pattern in the termination string. When the button is off it ignores case, for example, "Explorer" matches "explorer."

Example 7−2 Creating a Template with Pattern Glyphs

This example illustrates how to use the pattern glyph in an input template.

A file is usually composed of a header followed by several arrays. The header consists of a number of character strings that provide sizing information and other clues to the file's contents. **Figure 7−24** shows an example of such a file.

Figure 7−25 shows an input template that describes this file structure using pattern glyphs. To read the file into IRIS Explorer, see **Example 7−3**.

Figure 7−24 File Structure for Pattern Template

The sections shown in **Figure 7−24**, up to the 2D floating point array, correspond to the glyphs in the *FirstFile* template shown in **Figure 7−25**. The text is converted by means of pattern glyphs. The word in the termination slot on each glyph (`size:`, `size:`, and `Here:`) indicate the point at which each pattern ends.

Figure 7−25 An Input Template Using Patterns

7.8.1. Using the Glyph Property Sheet

Once you have created a template and defined the characteristics of each glyph in it, you may want to change a property, such as the name or data type, of the template or of a glyph in it. You can also annotate a complex script by adding comments on the templates, glyphs, and glyph components in the script. These notes can save you time in the future.

To edit or annotate a template or glyph, highlight it by clicking on its icon, then select *Properties* from the Edit menu. The Glyph Property Sheet is displayed. The fields on the Sheet differ for each glph type. **Figure 7−26** shows the Glyph Property Sheet for a glyph *VolumeData* in a template *ArrayFile.*

Figure 7−26 The Glyph Property Sheet

Glyphs have limited space for names, limits and values. A property sheet shows more detailed information about them, as well as providing a field for notes and comments such as:

- the data format, the data source, any anomalous behavior to watch out for, or why this data is important
- formatting information for ASCII files

For more details about using the ruler in the Property Sheet, see **"Converting Formatted ASCII Files."**

Every time you make a change on the Glyph Property Sheet, click on *Apply* to update the glyph or template. Changes to the Property Sheet update the glyph automatically when you click on *Apply*, but changes to the glyph do not update the Property Sheet unless you close it and open it again.

Note: The DataScribe does not limit the amount of information you can enter in a text slot on the Property Sheet. Use the arrow keys to scroll up and down the lines of text and the down arrow when entering a new line.

7.9. Defining Parameters and Constants

The DataScribe provides two special input templates for setting up parameters and constants. All other templates are bound either to a file (one template per file) or to a lattice (one template per data type), but they can also use the Parameters and Constants templates.

7.9.1. Making a Parameters Template

Parameters are those scalars that you want to manipulate when your DataScribe module fires in the Map Editor. For example, you may want to:

- select a particular dataset in a file
- cut a 2D slice from a 3D dataset
- change the name of a file

Figure 7−27 shows the title bar of the Parameters template. It can be identified by the icon on the left, a small circle containing an arrow pointing upward.

Figure 7−27 Parameters Template Title Bar

To create a Parameters template, select *Parameter* from the Template menu. A new Parameters template appears in the DataScribe window.

You can drop any number of `long` (long integer) or `double` (double precision) scalar glyphs into this template. IRIS Explorer does not accept any other data types for parameter values. Each must have a unique name.

You must use the Control Panel Editor to assign a widget to each parameter in a Parameters template so that they can be manipulated in the Map Editor.

7.9.2. Making a Constants Template

Occasionally you will want to have a collection of scalars whose values can be set and used by other templates. You can achieve this by creating a Constants template.

Figure 7−28 shows the title bar of the Constants template. It can be identified by the icon on the left, a small circle containing a square.

Figure 7−28 Constants Template Title Bar

To open a new Constants template, select *Constants* from the Template menu. A new Constants template appears in the DataScribe window.

You can drop any number of scalars or sets of scalars into the Constants template. You can use any of the scalar data types, but no other data types from the DataScribe palette.

You can also enter values into the value slot of each data item. These values can be in the form of symbolic expressions.

7.10. Connecting Templates

Once you have created input and output templates, you need to associate the data items in the input templates with their counterparts in the output templates. You do this by wiring the output ports of the input template to the input ports of the output template, just as in the Map Editor.

There are two types of associations that you have to consider:

- ordering variables within a script
- connecting dissimilar data items between templates

7.10.1. Ordering Variables in a Script

Symbols are defined sequentially within a script, and one variable can reference another only if the referenced variable is declared first. The DataScribe traverses files, including data files, from top to bottom and left to right.

Hence, if you want to use a variable to set the limits of an array or as a selection index for a sub−array in a template, you must define the variable before you use it. You can define the variable:

- in a Parameters or Constants template, in which case this template must precede the file template in which the variable is used (for example, see **Figure 7−30**).
- earlier in the file template itself (or example, see **Figure 7−45**).

You can use any variable in any template if it has been previously defined.

7.10.2. Connecting Data Items Between Templates

There are some basic rules for wiring together data items between input and output templates. You can legally wire:

- a scalar to a scalar
- a set to a set
- a set to an array
- an array to an array of same or differing dimension

This works if there are the same number of values in each data item. Since a set has no shape, however, an ordering mismatch can occur even if the count is correct.

If you wire a scalar value into a non−scalar data item, such as an array, then you will associate the scalar value with the first item of the array.

If you have some scalars, for example, integers, in an input template and you want to connect them to a vector in an output template, you can create a set that contains the integers in the input template. You can then wire the set in the input template to the vector in the output template.

If you connect incompatible data items, the DataScribe displays a warning message, and the script may fail if you try to run it as a module.

Some wiring combinations may produce misleading results, and the DataScribe may display a warning when the script is parsed. In particular, you should be careful when:

- The data source is larger than the destination.

  For example, if you connect a five−node set to a vector that is expecting three values, the first three nodes of the set will be transmitted and the fourth and fifth ignored.The source data is truncated to conform to the destination size.
- The data destination is larger than the source.

  For example, if you wire a five−node vector to a 5x5 node 2D array, only five values will be filled in the 2D array, and the rest will be set to zero. The destination is only partially filled with the source data.

Example 7−3 A Wiring Exercise

This example expands on the previous one, **Example 7−2** to create a script that reads in the file with patterns (see **Figure 7−30**).

A copy of this script, saved as a module called *dsPatEx.mres*, resides in */usr/explorer/scribe*. The data file is located in */usr/explorer/data/scribe*.

To build the input and output templates:

1. Place a pattern, an integer, a pattern, another integer, a final pattern, and a 2D array in the first input template.
2. Set the names of these glyphs as follows:
- the termination string of the first pattern (call it *Header*) to "size:"
- the name of the first integer to *xres*
- the name of the second pattern to *Ypattern* and its termination string to "size:"
- the second integer to *yres*

3. Give the last pattern *PresHeader* a termination string of "Here:."
4. Name the 2D array *matrix* and set its shape to *xres* by *yres*.
5. Create an output template of type IRIS Explorer and place a 2D uniform lattice in it.
6. Set the *nDataVar* entry to 1.

You need to fill in two fields in this output template, the *dims* vector and the data itself. In the input template, the *xres* and *yres* contain the dimension information, but you cannot wire scalars directly to a vector. However, you can build a Constants template that organizes the two scalar values into a set, which can be wired to the *dims* vector.

To build the Constants template:

1. Create a new Constants template and place a set glyph in it.
2. Name this set *matrixSize* and place two integers into the set.
3. Name the first integer *ires* and set its value to `xres`.
4. Name the second integer to be *jres* with a value of `yres`.

You can wire this set into the *dims* vector of the output lattice.

To wire up the templates:

1. Bring up the port pop−up menu from the Constants template and select *matrixSize*.
2. Now bring up the output template's pop−up menu and select *dims*. The blue connection wire appears in the Overview pane (see **Figure 7−29**).

Figure 7−29 Making the First Connection

3. Assign the data by bringing up the pop−up menu on the file template and selecting *matrix*.
4. From the output template pop−up menu, select *data*.

The lattice output template now has all the information it needs to convert the data file into IRIS Explorer.

To test the script, select *Parse* from the File menu. The DataScribe traverses the script and tests for errors such as undefined symbols and illegal links. If any error messages appear in the Messages pane, refer to **"Tracking Down Errors."**

Save the script and then create the data file described in **Figure 7−24**. Enter the name of the data file into your new DataScribe module in the Map Editor. You can try it out by connecting its output to *Contour*.

Figure 7−30 A Script Using Patterns

7.11. Selecting Array Components

Frequently, a file will contain a dataset that is too big to be read completely into memory, or perhaps only part of the data is relevant, such as one slice of a 3D array, or every other node in the dataset. You can separate out the relevant parts of a dataset and turn them into independent data items that can be wired into other data items in a template. These parts are called array components, or fragments, and you isolate them using the Array Component Dialog window.

To display the Array Component Dialog window, click the right mouse button on the component menu button of any array glyph. Initially, the component menu contains only one entry, <*New*>. When you click on <*New*>, the Component Dialog window appears (see **Figure 7−31**).

You use this window to select the subset of the original array in which you are interested. Use the Name slot to call the component anything you like; for example, *xValues* for the x values of a 2D array.

You use the matrix of text type−in slots to specify the region of the array that you have selected. The From and To slots contain the starting and ending values for the data subset.

Figure 7−31 Array Component Dialog Window

The Stride slot indicates the step size. For example, if the stride is 1, each consecutive value in the specified sequence is taken. If the stride is 2, every second value is taken.

There are as many rows of slots as there are dimensions in the array (two in **Figure 7−31**), arranged in order with the slowest−varying variable first. The default specifies all areas of the array, but you can change the information in any of the slots (see the following examples). You cannot change the variable order, however.

Note: If the data component is nested within the structure of the item, you must promote the fragment to the top level of the glyph. This technique is illustrated in the examples below.

Example 7−4 Using the Component Dialog Window

This example illustrates how to extract data fragments from an array by using the Component Dialog window. The sample script files are in */usr/explorer/scribe*, called *ReadXData.mres* through *ReadXYZData.mres*. The data files are located in */usr/explorer/data/scribe/**. The example goes through the technique in detail for the first file. You can open the other files and see how they build on the first one.

This is the first ASCII data file, *ReadXData.data*:

```
10                <−− Dimension Variable
10.0 1000.0       <−− Column 1: X Co−ordinate values
20.0 2000.0             Column 2: Data values
30.0 3000.0
40.0 4000.0
50.0 5000.0
60.0 6000.0
70.0 7000.0
80.0 8000.0
90.0 9000.0
100.0 10000.0
```

The following script, called *ReadXData.mres*, extracts the X values from this and reads them into a 1D curvilinear lattice.

Open this file in the DataScribe and look at it (see **Figure 7−32**). The input template is in ASCII format and has two glyphs, one for the array dimensions and one for the data. The *res* vector holds the dimension variable. Its shape is defined as 1 (only one data variable) and its value is `long.` It must be defined first because the 2D array for the dataset uses this value.

The *dataSet* 2D array glyph has its *i* value defined as 2 (two columns in the data file) and its *j* value defined as the first value of *res* (the number of data entries, which is the number of *x* values).

The output template is of IRIS Explorer type and contains a 1D curvilinear lattice called *DataLattice*. The data variable (*nDataVar*) and the coordinate variable (*nCoordVar*) are both defined as 1.

Figure 7−32 Selecting the X Value of a 2D Array

To select the *x* values only, you need to create separate data components for the *x* and data input values in the 2D array *dataSet* glyph and a component for the x output values in the *coord2* Vector in the curvilinear lattice. The data values are connected straight into the data vector.

To isolate the *x* values in the input template, you would:

1. Select *New* from the Component menu and fill out the text slots as shown in **Figure 7−33**:

   The fragment is called *xPos*.

   The *i* variable goes from 1 to 1, that is, only the first value in the row is taken.

   The *j* variable goes from 1 to the last value (*res[1]*) in the column of *x* values.

2. Click on the *OK* to apply your changes.

Figure 7−33 Component Dialog Window for X Values

To isolate the data values in the input template, you would:

1. Select *New* from the Component menu and fill out the text slots as shown in **Figure 7−34**.

   The fragment is called *data*.

   The *i* variable goes from 2 to 2, that is, only the second value in the row is taken.

   The *j* variable goes from 1 to the last value (*res[1]*) in the column of data values.

2. Click on the *OK* to apply your changes.

Figure 7−34 Component Dialog Window for Data Values

The *dataSet* 2D array has no nested data items, and the *x* and data components were created at the top level. When you click on the component menu, both items appear on it.

They also appear on the port menus of the input template and its overview, indented under the name of the glyph that contains them (see **Figure 7−35**).

Figure 7−35 New Menus for Input Template

To isolate the *x* values in the output template, you would:

1. Open up the *coord2* vector glyph. Its data type is another vector, nested inside the top−level vector (see**Figure 7−36**). Open this vector.

   The shape of the *coord2* vector takes its value from the *dims2* vector just above it. The shape of the data type vector takes its value from the *nCoordVar* scalar (see **Figure 7−32**).

Figure 7−36 Output Lattice Coordinate Vector

2. Select *New* from the Component menu and fill out the text slots as shown in **Figure 7−37**. The fragment is called *xCoord*.

Because this is a 1D lattice, there is only an *i* variable, which goes from 1 to 1, that is, the first value in the row.

3. Click on the *OK* to apply your changes.

Figure 7−37 Output Coordinate Values (xCoord)

The *xCoord* fragment has been defined as part of a vector within another vector, and hence will not appear on the port menu. Only top−level array fragments can be connected to one another. To promote the fragment to the level of the *coord2* vector, you must define another component at the top level that contains the lower−level data fragment.

To promote the *xCoord* fragment to a port menu, you would:

1. Close the data type vector.
2. Select *New* from the Component menu of the *coord2* vector and fill out the text slots as shown in **Figure 7−38**.

   The fragment is called *allX*. In the data type vector, you defined one single item, *xCoord*. In this fragment, you define the set of all *xCoord*s. The DataScribe will thus convert all instances of *xCoord* as defined in *allX* when this item is wired up.

   The *i* variable goes from 1 to the last value in the dimensions vector (*dims[1]*), that is, all the values in the column.
3. The Selection element menu now contains two items: the vector, which is the data type, and *xCoord*, which defines a subset of that data type. Select *xCoord*.
4. Click on the *OK* to apply your changes.
5. Click on the port menu and you will see *allX* displayed as an option (see **Figure 7−33**).

Figure 7−38 Output Coordinate Values (allX)

To wire up the input and output templates, make these connections:

- *res* to *dims2*
- *dataSet: xpos* to *coord2:allX*
- *dataSet: data* to *data2*

The above example demonstrated promoting lower level fragments to the top−level, so they appear on the port menu. You could have omitted creating the *xCoord* and *allX* components and simply wired *dataSet: xpos* to *coord2* as *nCoordVar2* is set to 1, therefore requiring only x co−ordinates.

Example 7−5 Other Examples

The file *ReadXDataData.mres* has two data values for each x position. In the script, the input *dataSet* array has two data components for the data, *fx* and *gx*, instead of data in the previous example.

The output template has two 1D curvilinear lattices; *fx* is connected to the data vector in the first one and *gx* to the data vector in the second one. *res* is connected to the *dims* vector and *xPos* to the *coord* vector in both lattices.

The files *ReadXYData.mres* and *ReadXYZData.mres* have two and three sets of coordinate values, respectively, for each set of data values. You can extract an *xPos* and a *Ypos* component in the input *dataSet* array, and create an *xCoord* and *yCoord* fragment, which are in turn promoted to *allX* and *allY* in the coordinate vector in the output template. In the second script, you can also extract the *zCoord* fragment.

The file *ReadXYZ3DData.mres* shows how to extract the *x*, *y* and *z* coordinate values separately, as well as the data. This script has only one set of data values. A nested vector appears in the input 3D array, so the coordinate components must be promoted in the input template as well as in the output template this time.

7.12. Converting Formatted ASCII Files

Languages such as Fortran support a large number of formatting options, and parsing the output from a program can take time and effort. The DataScribe provides a ruler for formatting ASCII data that makes reading and writing formatted files easier.

7.12.1. Using the Property Sheet Ruler

You can use rulers only in:

- templates for ASCII data files
- top−level arrays and sets; an array may not contain another array, a vector for example, and a set may not contain any other sets

To display the ruler for an array, select the array in the ASCII template and select "Properties.Properties..." on the Edit menu to display the Glyph Property Sheet.

The ruler for ASCII input arrays contains a scalar glyph icon. You use this icon to delineate which fields of a line in an ASCII file contain data. When the module fires, it looks for a numerical value for this scalar in the given field of the line. The data type of the number is given by the scalar type of the array. For example, if the array in the input template is `double`, the values in the field are assumed to be double precision.

The ruler for output arrays has glyphs for scalars, ASCII strings, and tabs (see **Figure 7−39**). Rulers for input arrays are simpler than those for output arrays.

Figure 7−39 The Output Template Ruler

To place an icon in the ruler, drag and drop it in position. You can drag on the Scalar and String icon "handles" to extend them horizontally in the ruler. The Tab icon takes up only one space. The ruler can accept up to 132 characters, but there are two restrictions on the way the space can be used:

- If you use the Scalar glyph to position floats or doubles in the ruler for output templates, you must provide a minimum of nine spaces for each float or double value. The DataScribe needs one space for the sign, at least three for the number, one for E and four for the magnitude (sign plus three).

  For example, it formats 1.1 as **`+1.1E+100`**. For every extra digit in the number, add one more space. If you have formatting problems, check your spacing.
- The DataScribe does not support hexadecimal or octal numbering, for example, **`0x186`** (hex) or **`0768`** (oct).
- It also does not support commas or D format, for example, **`1.0d−05`** in Fortran.

Example 7−6 Reading the Formatted File

This example describes how to use the ruler in the Glyph Properties Sheet to set the format of an ASCII file in the DataScribe.

Here is an example of a formatted ASCII file:

```
Sample Formatted Data
Resolution is: 8 10 12
Data Follows:
xcoord: 0.000 ycoord: 0.000 zcoord: 0.000 temp: 0.375 pres: −0.375
xcoord: 0.143 ycoord: 0.000 zcoord: 0.000 temp: 0.432 pres: −0.432
xcoord: 0.286 ycoord: 0.000 zcoord: 0.000 temp: 0.403 pres: −0.403
xcoord: 0.429 ycoord: 0.000 zcoord: 0.000 temp: 0.295 pres: −0.295
...
```

The script shown in **Figure 7−40** reads the five arrays in this file: three coordinate arrays and two data arrays. To read this file into IRIS Explorer, you must extract the resolution vector, define the rest of the header with a pattern glyph, and read the five arrays, separating the character filler from the data.

You use a 2D array in the input template for reading in the data with five components defined for the x,y and z coordinates and the two data values. You need to set the formatting shown in the input template in the 2D array Property Sheet.

Bring up the Property Sheet by selecting *Properties* from the Edit menu. The Glyph Property Sheet contains information about the array, including its name, bounds, and any annotations. It also contains a ruler for setting the format.

In this case, you use the ruler in the Property Sheet to strip off the leading ASCII strings, which are *xcoord, ycoord, zcoord, temp,* and *pres.*

Figure 7−40 Reading a Formatted ASCII File

Figure 7−41 shows the ruler for the 2D Array glyph in the FormattedFile template. The arrangement of text in each line of the ASCII file is reflected in the ruler. You set up each text field by dragging the Scalar icon from below the ruler and dropping it into the active area, where it appears as a small box with handles. Spaces between the boxes represent white space matching the white space in the file.

Once you have positioned the icon in the ruler, you can:
- Resize it by dragging on the handles at each end.
- Move it around by clicking on the box and dragging it to the desired location.

Figure 7−41 Ruler for Reading an ASCII File

Example 7−7 Writing the Formatted File

The DataScribe offers more options for creating an ASCII format for an output file. These include:
- setting the textual annotation (the *xcoord: ycoord: zcoord:* text in the previous example, see **Figure 7−6**)
- setting special delimiters, such as tabs

The next example demonstrates their use.

This example illustrates how to write the file similar to the one that the DataScribe module in the previous example is designed to read in see **Example 7−6**. **Figure 7−42** shows the script with its input and output templates. Scripts may become very complicated as you strive to find ways of dealing with complex data.

Figure 7−42 Writing a Formatted ASCII File

Figure 7−43 shows the ruler for the output array, with its three icons.
- The Scalar icon defines a portion of the output line to be filled in with a numerical value contained in the array. Note that this produces E formatted float and double numbers, so a length of at least 9 must be specified.
- The String icon delimits a character field.
- The Tab icon indicates where tabs should go into the output line.

String icons can be resized and positioned just as Scalars can. To set the value of the string, you can type directly into the icon once it is resized.

The scrollbar at the bottom of the ruler allows you to traverse the length of the line (up to 132 fields). Only a small portion of the output line is shown in the default Property Sheet at any given time. However, you can resize the entire window to reveal the entire line.

Figure 7−43 An Output Array Ruler

Example 7−8 Building a Curvilinear Lattice Reader

The module for this example, *dsCurvEx.mres*, resides in the directory */usr/explorer/scribe* on your system. The data is located in */usr/explorer/data/scribe/dsCurvEx.data.* The example shows a file of data points with *x, y,* and *z* coordinates and a value for each data point, organized as a 3D array of data. The data file looks like this:

```
Output from XYZ Simulation
Date: 11 Sept 1991
Resolution 5 10 15
   x−coord      y−coord     z−coord     function
 0.10000E+01 0.00000E+00 0.00000E+00 0.50000E+00
 0.18000E+01 0.00000E+00 0.00000E+00 0.50000E+00
 0.26000E+01 0.00000E+00 0.00000E+00 0.50000E+00
 0.34000E+01 0.00000E+00 0.00000E+00 0.50000E+00
 0.42000E+01 0.00000E+00 0.00000E+00 0.50000E+00
 0.50000E+01 0.00000E+00 0.00000E+00 0.50000E+00
 0.80902E+00 0.58779E+00 0.00000E+00 0.50000E+00
 0.14562E+01 0.10580E+01 0.00000E+00 0.50000E+00
 0.21034E+01 0.15282E+01 0.00000E+00 0.50000E+00
 0.27507E+01 0.19985E+01 0.00000E+00 0.50000E+00
...
```

"Resolution" gives the dimensions of this dataset of points. The dataset is set up as a 3D array of 5 x 10 x 15 points.

The script for this file contains:

- patterns to skip over the header
- a vector to get the data resolution
- a pattern to skip the text line beginning "x−coord."
- a 3D array of vectors for the coordinates and data.

The coordinates and data are interlaced so you need to unravel them by putting each set of coordinates in the 3D array into a separate output array. This means collecting *x*, *y*, and *z* coordinates into three separate arrays using component selection.

To do this, you create a component at the vector level that contains the first node (*xCoord*), and create another component at the 3D array level that contains everything at that level, but only *xCoord* at the lower level.

This will strip off all the *x* coordinates and deliver them into a 3D array; the process is similar for the *y* and *z* coordinates, as well as the functional values. If you want to sample the dataset in each direction prior to reading it in, you can add a Parameters template to sample the volume in each direction independently. **Figure 7−44** shows the completed script.

Try it out in the Map Editor by connecting *WireFrame* to its output port.

Figure 7−44 A Curvilinear 3D Lattice Reader

Example 7−9 Creating a PLOT3D Reader

PLOT3D is a commonly used application in the field of computational fluid dynamics. It has several specific file formats; this example summarizes the process for building a module that can read the most commonly used variant of PLOT3D.

The data to be extracted is a 3D curvilinear lattice. It resides in two input files, one that contains the coordinate information, and another contains the nodal values: the real data. The structure of both files is quite straightforward. The coordinate file contains a short dimensioning vector followed by three arrays containing the coordinate data. The *x*, *y*, and *z* coordinates are grouped separately.

The data file contains the same dimensioning vector followed by four scalars that describe some of the flow regime. After these scalars is the data array. It is a 5 vector of 3D arrays. This array can be viewed as either a 3+1D array or a 4D array. This example takes the 4D approach.

You can put all five of the 3D arrays into the output lattice, but you probably want to look at only one at a time. To do this, you create a Parameters template that contains an integer to choose the component of the data vector. You then assign a slider to that integer in the Control Panel Editor.

The completed control panel contains a widget for the component as well as text slots for the script, coordinate, and data files.

Two example input files for *ReadPlot3D* can be found in the directory */usr/explorer/data/plot3d*; here, *wingx.bin* is the coordinate (XYZ) file, and *wingq.bin* is the data (Q) file. Finally, the script file for the module (see **Figure 7−45**) can be found at */usr/explorer/scribe/ReadPlot3D.scribe*.

Figure 7−45 The PLOT3D Script

7.12.2. More Complex Expressions

Complex expressions such as array bounds, indices of selected fragments, and constant values can be set to symbolic expressions in the DataScribe using a C−like language.

You can use the same language syntax and operators as are used in the Parameter Function (P−Func) Editor to evaluate these expressions. For more information, see **"Using the Parameter Function Editor"** in Chapter 4 .

7.13. Saving Scripts and Modules

A data transform script consists of at least one input template, at least one output template, and the wiring that connects them. Use the File menu to save your work and clear the work area.

To save a script or module, select *Save* or "Save As" from the File menu.

If you create a new script from scratch, you save it as a script. The DataScribe automatically appends the extension *.scribe* to the filename you give it.

If your script has no parameters, the DataScribe provides a default control panel and creates another file, the module resource file, when you save it. The module resource file, *moduleName.mres*, describes the module attributes, including its control panel, the executable name, and all the ports. The script file, *moduleName.scribe*, includes all the templates and their connections.

Saving Modules with Control Panels

You need only create a control panel if you have parameters defined in a Parameters template in your script. Otherwise, the DataScribe provides a default control panel. If you save a script containing a Parameters template and have not yet created a control panel, the DataScribe prompts you to do so.

Select *Control Panel* from the View menu.

The Control Panel Editor appears, allowing you to assign widgets to each of the glyphs in the Parameters template, position each widget, and set their current values and ranges. For information on using the Control Panel Editor, see **Chapter 4, "Editing Control Panels and Functions."**

Checking for Errors

Before you save your script, you can check it for errors. To do so, select *Parse* from the File menu. For more information, see **"Finding and Correcting Errors."**

Creating a Diagnostic Template

You can use a diagnostic template to check your script if you are not getting the results you want, or think you should be getting.

To do this, create an ASCII output template that will write the data it receives to an ASCII file. This is the diagnostic template. You can then wire up the troublesome input template to the diagnostic output template and create a diagnostic script. Create a control panel for it, fire the module in a map, and read the results from the ASCII file.

The ASCII file shows the values of entities such as array coordinates, and you can examine them for errors. For example, you might find that you have the coordinates *x* `=0,` *y* `=0,` where *x* and *y* should be 8. Hence you have an empty area in the output lattice. Once you have found the error, you can change the input template to eliminate the error.

7.14. Using a DataScribe Module with the Map Editor

Once you have created a module from a script, you can use it in the Map Editor to transform data. You can launch the module from the Module Librarian, just as you would any other module.

The default module control panel has two text slots, one for the data filename and one for the script filename. If you have created widgets for controlling module parameters, those will appear as well.

If you have already built a DataScribe module and merely want to update its script, you can save the new script as a script and then type its name into, or select it from, the module's file browser.

You should be careful when doing this, however. If you change any control panel attributes such as the widget layout or widget settings, or add new parameters to the script, then the new script will be inconsistent with the old module. You will have to create and save a new module in order to run the new script. It is better practice to create a new module control panel for each script.

Figure 7−13 shows a DataScribe module wired into a map in the Map Editor.

7.15. Finding and Correcting Errors

When you have constructed all the templates in a script, you can use the DataScribe parser to test the script for inconsistencies and assignment errors.

To do this, select *Parse* from the File menu.

7.15.1. Tracking Down Errors

The DataScribe produces messages for two types of discrepancies that it may find when a template is parsed. These are:

- Warning conditions

  Warnings indicate that something is amiss, but the problem is not severe enough to invalidate the script. You can still save the script and use it in a module, although the results might not be what you desired.
- Errors

  Errors encountered in the script render it invalid. You need to correct each error before the module will work.

The Messages pane at the bottom of the DataScribe window displays a list of all the discrepancies encountered during parsing.

If you do not want to display the messages, you can hide the window by selecting *Messages* from the View menu.

7.15.2. Interpreting Parsing Messages

Here is an abbreviated list of possible warning conditions and errors with suggested solutions.

Uninitialized variables:

[1] WARNING Constants::NumPixels −> Output data object does not have a value.

Please make a wire into this object or enter a value expression

Failure to wire to an output that needs data:

[0] WARNING Lattice::nCoordVar −> Output data object does not have a value.

Please make a wire into this object or enter a value expression

An unset scalar in a Constants template:

[0] WARNING Constants::Glyph −> Constant data object does not have a value.

Please enter a value expression for this constant

An undefined variable used in a component expression:

[0] ERROR foo has not been defined

Bad end index 1, foo, for fragment SingleComponent

Failure to assign a termination string to a pattern:

[0] ERROR DataFile::Pattern_1 −> No regular expression in pattern

Setting a scalar to a non−scalar value (array or lattice):

[3] ERROR Non−scalar Plot3DLattice used in expression

Lattice::Long_3 −> Bad value expression

Syntax error in an expression:

[0] ERROR illegal statement

Constants::NumPixels −> Bad value expression

Bad initial value for a scalar:

[1] ERROR Non−scalar res used in expression

Constants::NumPixels −> Bad value expression

Setting the value in a file−based input template:

[0] WARNING DataFile::Alpha −> A data object in a input template may not have a value expression.

The value expression will be ignored

Cannot use arrays, sets or lattices in expressions:

[1] ERROR Non−scalar SingleComponent used in expression

Lattice::nCoordVar −> Bad value expression

Wiring mismatch:

[0] WARNING :: −> Source of wire is a scalar, but its destination is a non−scalar

[0] WARNING :: −> Source of wire is an array, but its destination is not an array

[0] WARNING :: −> Source of wire is a lattice, but its destination is not a lattice

Excessive wiring:

[0] WARNING Lattice::nDataVar −> Data object has both an input wire and a value expression.

The value expression will be ignored

Note: The DataScribe module does not produce diagnostic error messages beyond those typically found in an IRIS Explorer module.

Appendix A

Configuring Your IRIS Explorer Environment

This appendix explains how to configure various aspects of your system in order to use the IRIS Explorer more effectively. It includes:

- modifying your *.explorerrc* file
- setting environment variable(s)
- configuration commands
- remote execution and passwords
- running the X server
- creating a larger kernel

A.1. The *.explorerrc* Configuration File

IRIS Explorer reads configuration information from files on start−up. This information tells IRIS Explorer, among other things:

- where to look for modules
- how to configure the Module Librarian
- where and how to run remote modules
- what size the memory arena should be

This information can come from two separate files, one global to the host machine and one private to the user. The global configuration file is in */usr/explorer/Explorer.config* and the private copy is in the user's home directory in the file called *.explorerrc.*

A.1.1. Remote Execution of Modules

The general idea of IRIS Explorer is that you initially run it on your own workstation, but then you can use it to run some modules on remote workstations.

Software Requirements

Each machine that you use to execute modules remotely must have most of IRIS Explorer installed on it. In general, the version installed remotely must match the version installed on the local machine.

This sense of remote execution is different from that offered by the X Window System, which lets you remotely log into a workstation on which IRIS Explorer is installed, reset the DISPLAY environment variable to reference your own screen, and then run IRIS Explorer. In this case, all of IRIS Explorer will initially be running on the remote machine, but the user interface will appear on the local machine.

Defining Hosts for Remote Modules

To execute IRIS Explorer modules remotely, you must first add any host names used to your *.explorerrc* file, in this format:

```
host hostname
```

For example, suppose your local machine, called *billiejo*e, is networked to another machine named *bob*. If you can remotely login into *bob*, and IRIS Explorer is installed on it, then you can very simply set up the two machines for remote execution.

Add this line to the *.explorerrc* file:

```
host bob
```

Do this for every host you want to run modules on. When you execute IRIS Explorer again, these host names will be listed on the Hosts menu in the Module Librarian window.

IRIS Explorer uses the *rsh* command to create a computation server on machines used for execution of remote modules. Hence, you must be able to access the remote hosts you use with the *rsh* command. IRIS Explorer also provides an alternative to *rsh*, which is *cxrexec*. It is capable of prompting for a password.

See **"Remote Execution and Passwords"** for more information.

A.1.2. If IRIS Explorer Is Not Installed in */usr/explorer*

If IRIS Explorer is not installed in the directory */usr/explorer*, then you must either:

- create a symbolic link from */usr/explorer* to the actual installation directory.

  If your system uses shared libraries, as do UNIX systems, you must install the symbolic link.
- define an environment variable, *EXPLORERHOME*, to contain the name of the directory where it is installed. See **"IRIS Explorer Environment Variables"**.

For example, if IRIS Explorer on the machine *billiejoe* is located at */usr/local/explorer*, then you must set *EXPLORERHOME* to be */usr/local/explorer* in the *.cshrc* file (see *csh(1)*).

Suppose that on the remote machine *bob*, IRIS Explorer is located at */usr/people/jim/explorer*. Then the *.cshrc* file in your home directory on host *bob* must contain this line:

```
setenv EXPLORERHOME /usr/people/jim/explorer
```

The easiest thing to do is to install IRIS Explorer in the same location on all machines, which by default is */usr/explorer*.

Using the Switch Statement in a Shared .cshrc

If you have a single *.cshrc* file on several workstations, each of which may have IRIS Explorer installed in a different location, you can use the *csh* "switch" statement to set *EXPLORERHOME* correctly according to the current machine.

For example:

```
switch (‘hostname‘)
case billiejoe:
      setenv EXPLORERHOME /usr/local/explorer
      breaksw
case bob:
      setenv EXPLORERHOME /usr/people/jim/explorer
      breaksw
endsw
```

Note: The character (‘) surrounding `hostname` in the previous example is an accent grave, not an apostrophe.

A.1.3. The *.explorerrc* File

Both the *IRIS Explorer.config* and *.explorerrc* files can contain commands that configure IRIS Explorer. The syntax is line−oriented; a backslash at the end of a line indicates that the next line is a continuation.

Exclamation points (!) at the beginning of a line designate comments (but the # sign will also work). The line syntax is very similar to that of shell scripts, namely:

```
command [arguments] [modifiers]
```

The first word designates the command, and subsequent fields designate "arguments" or "modifiers" to the command.

Either single or double quotation marks may be used to surround arguments that contain embedded blank characters.

Most arguments to commands may encode UNIX environment variables using the shell dollar−sign notation. For example, *$EXPLORERHOME* will expand to the value of the environment variable *EXPLORERHOME*, as will *${EXPLORERHOME}*. The C−shell "tilde" (~) notation for designating home directories is recognized as well. However, strings surrounded by single quotation marks (apostrophes) are not expanded in this manner.

If the file permissions are set to be executable, the file will be executed and the output read as the configuration file. If IRIS Explorer recognizes the #!CPP notation, the file is run through the C preprocessor and all environment variables are defined to the C preprocessor as C preprocessor symbols. This means you can use conditional statements for values such as hostname, which makes the file more flexible.

Note: If you use the #!CPP notation in your .explorerrc file, then you must use ! rather than # at the beginning of a comment line.

This is an example of a typical *.explorerrc* file. Comments start with an exclamation point (!) in column 1.

```
!
! Define some remote hosts
host willard
host louis

! Also look for modules in my local directory
modulepath −append ~/modules

! Set the directory where the arena and pipes go
set tempdir /usr/tmp/explorer

! Add some frequently used modules into the shelf
category shelf Render DisplayImg IsosurfaceLat ReadImg\ ReadLat Contour\

! Put the Geometric modules into their own category
category Geometry BoundBox Contour IsosurfaceLat \
   IsosurfacePyr LatToGeom PyrToGeom \
   PickLat ReadGeom VolumeToGeom

! Increase the maximum arena size for larger data sets
set arenasize 24mb
```

A.1.4. Configuration Commands

The commands that are accepted in an IRIS Explorer configuration file are described below. Optional portions are enclosed in square brackets. Ellipses denote that the previous argument may be repeated any number of times. Commands (reserved words) are shown in bold.

Setting the Module Path

The module path is a list of directories in which IRIS Explorer looks for modules and maps. The default list contains only *$EXPLORERHOME/modules*, but the *Explorer.config* file shipped with IRIS Explorer extends it. You can reset module paths from scratch, or you can extend them using the "−append" or "−prepend" modifiers.

The modulepath command states where to look for modules.

```
modulepath [−prepend] path ...
modulepath [−append] path ...
```

This command defines a list of directories to be used. By default, the existing module path is extended by the list for directories (this is equivalent to using the "−append" modifier). The default path value is *$EXPLORERHOME/modules.* Used alone, the command clears the module path to the default path value. If the "−prepend" modifier is used, the *path* is added at the head of the current directory list.

set

You can use the *set* command to set the temporary directory and the size of the arena.

set tempdir

IRIS Explorer uses a number of temporary files during its execution. While almost all of these are small, the shared memory arena file can get quite large.

```
set tempdir path
```

The *set tempdir* command states where to place any temporary files that may be needed during the course of execution, such as named pipes and shared memory arena files, if applicable. Typically, such files are placed in a uniquely named subdirectory of the directory. The default value for the base directory is */usr/tmp.*

Because shared memory arenas can become large, it is often advantageous to place the *tempdir* somewhere other than below */usr.*

set arenasize

On some workstations, modules use shared memory for data communications. The shared memory is mapped into a file that resides on disk as determined by the *set tempdir* command above.

```
set arenasize string
```

For a shared memory system, make the maximum size of the shared memory arena the designated size.

The "string" should be a number optionally followed by "mb" or "kb" for megabytes and kilobytes, respectively. This option may also be set from the command line. The default size is 16Mb.

category

Defines a new Module Librarian category with the name "name" and containing the list of modules given.

```
category name [−append] [−filter] [arg] module ...
```

The "−append" modifier appends a list of modules to a specified category. The list of modules can contain module names, such as *GenLat*, or directory names. In the latter case, the directory is added to the module path, and all modules and maps in the directory are added to the category. For example,

```
category myModules ~/explorer/modules
```

adds all modules in directory *~/explorer/modules* to the category myModules. You can use the "−filter" modifier with an **arg** to filter the list of modules against the argument pattern. The **arg** accepts shell wildcards such as *, ? and [. For example,

```
category Readers −filter Read* Modules
```

puts all the *Read** modules in directory *Modules* in a category called *Readers.*

There is a designated category in the Module Librarian called "shelf." Modules in the shelf category are displayed in a small panel at the bottom of the Librarian’s window.

host

Modules are organized by categories and by hosts in the Module Librarian. When you add a host, modules can be executed on that machine.

```
host name [−command string]
```

The modifier "−command" may be used to define an alternate way of starting a local communications server (LC) on remote hosts. The single argument after "−command" (which must be enclosed in quotation marks if it contains blanks) specifies the alternate command.

The UNIX command IRIS Explorer uses to start the LC on a remote host is the given command (*rsh* by default) followed by the hostname, followed by a shell command that will be invoked remotely to start the LC. For example, suppose this appears in your *.explorerrc* file:

```
host bob −command "myrsh −x"
```

IRIS Explorer will invoke

```
myrsh −x bob <shell command to start the LC>
```

If the word HOST appears in the alternate command, the hostname is substituted in its place when executing the command. Therefore, you can use the command:

```
host bob −command "rsh HOST −l joe"
```

to start a remote LC under the account "joe" on *bob*.

A.2. IRIS Explorer Environment Variables

You can set this environment variable:

EXPLORERHOME

By default, the IRIS Explorer system is installed into the directory */usr/explorer*. If you choose to move the directory elsewhere, for example, because you need the disk space, then you will need to define a UNIX environment variable named *EXPLORERHOME*. Normally, this is not necessary. A more reliable way to keep IRIS Explorer elsewhere is to place a symbolic link from */usr/explorer* to the correct location.

There are also several environment variables that pertain to the building and installation of new modules. They are described in Chapter 2, "Understanding the Module Builder" in the *IRIS Explorer Module Writer's Guide*.

A.3. Remote Execution and Passwords

In this release, IRIS Explorer uses the *rsh(1)* command to start local communications servers (LC) on remote machines. This takes place only the first time you create a librarian or launch a module on a remote machine. In order for IRIS Explorer to be able to start LCs, *rsh* to those machines must work. For example, if you want to launch modules on a host named "labserver," verify that you have access via *rsh* to that machine by typing the following command:

```
/usr/bsd/rsh labserver pwd
```

If it responds with the name of your remote home directory, all is well. If it responds with "Permission denied" you need to enable *rsh*.

You can enable access to that machine by creating a file named *.rhost* in your remote home directory. See the *rsh(1)* manual page for details.

The "−command" modifier on the host line allows the redefinition of the command to use to try to get a module communications server running on a remote machine. By default, the *rsh* command is used. This may not be appropriate for all installations, because *rsh* does not allow for password prompting and the associated protections.

IRIS Explorer provides an alternate command similar to *rsh* that can prompt for a password. This command, *cxrexec*, uses the *rexec* library call and prompts for a password in a small pop−up window. For example, if access to a host named "securehost" requires a password, the following line in *.explorerrc* will result in the password prompt being issued:

```
host securehost −command cxrexec
```

The command *cxrexec* has the same command line options as *rsh*. Either command can be modified in *.explorerrc* to add options. However, when additional *rsh* or *cxrexec* options are listed, the entire command must be enclosed in quotation marks:

```
host superhost −command "rsh HOST −l guest"
host superhost −command "cxrexec HOST −l guest"
```

These examples demonstrate another feature of the alternate command option: wherever the word "HOST" appears, the name of the remote host will replace it.

If the word "HOST" does not appear in the command, it is appended to the end. Hence, the following two lines are functionally equivalent:

```
host superhost −command "rsh HOST"
host superhost −command "rsh"
```

Other UNIX programs may be used instead of *rsh* and *rexec*, depending on local conventions.

A.4. Running the X Server

The type of visual you run on your X server determines the number of colors available for modification, and this affects how many *GenerateColormap* modules you can run at one time.

A.4.1. Selecting a Visual

The X server supports several classes of color visuals. Each visual class uses a different method to display color. The *GenerateColormap* module is designed to function properly with the *PseudoColor* and *TrueColor* visual classes. The PseudoColor visual uses the hardware colormap to display a small number of colors from a much larger palette of possible colors. The TrueColor visual class writes colors directly to the graphics frame buffer.

The visual classes have a *depth* that specifies the number of bit planes it uses for display. A PseudoColor visual with a depth of 8 displays 256 colors at one time; one with a depth of 12 displays 4096 colors. The TrueColor visual supports depths of 24 and 12 bits. The 24−bit TrueColor visual can display 8 bits for red, green and blue, or over 16 million possible colors at one time. The 12−bit TrueColor visual displays 4096 colors at one time, with 4 bits for red, green and blue.

Choosing the best visual depends on your workstations hardware and the other applications you run. Your workstation must have at least as many bit planes as the depth of the visual that you choose. You generally want to run the 24−bit TrueColor visual if you can; otherwise, try the 12−bit TrueColor or PseudoColor visuals.

High−quality applications should run properly with any of these visuals, but other applications may not. Some applications, such as those that do color table animation, may not run with a TrueColor visual. Some applications may make assumptions about the depth of a PseudoColor visual, for example, that the color index will fit into 8 bits. You can start with the highest quality visual supported by your workstation, try your other applications, and then adjust the visual to fit your system’s capabilities.

Note: See your local documentation on how to reconfigure the X server.

Appendix B
The Skm Language

This appendix describes the syntax and use of the Skm language to create scripts for the IRIS Explorer command interface.

B.1. Skm Syntax

The Skm scripting language uses prefix notation. All expressions are enclosed in parentheses and all commands precede arguments and operands. If you want to say "do something," the Skm expression is:

```
skm>(do something)
```

To add two numbers, the addition operator precedes the arguments:

```
skm>(+ 1 2)
3
```

To launch a module:

```
skm>(start "ReadImg")
```

White space separates operands and operators. White space is generally just the space character but may also be the tab or newline character. Non−numeric constants should be surrounded by double quotes. Numeric constants do not require quotes.

Footnote: Non−numeric constants without embedded spaces may actually be preceded by a single quote. Only strings containing embedded spaces need be surrounded by double quotes.

Assignment is performed with the *define* command. To define a variable called *var* and assign it the value 3, write:

```
skm> (define var 3)
```

To change its value, write:

```
skm> (define var 4)
```

To display the value of a variable, type the name of the variable at the prompt (without parentheses).

```
skm> var
4
```

B.1.1. The Skm Data Types

The basic Skm data types include:

- numbers
- symbols
- strings
- lists
- procedures

There is no distinction between real and integer numbers in Skm—all numbers have floating point precision. Symbols are identifiers that are preceded by a single quote. For example, *var* is a symbol below:

```
(define var ’foo)
```

Strings are character sequences surrounded by double quotes. String operators are discussed in a later section. Lists are sequences of identifiers surrounded by parentheses. Procedures are discussed in a subsequent section.

Any variable in Skm can be used to contain any data type. Thus, you can store both a number and a string into the same variable.

```
skm> (define var 23)
23
skm> (define var "hello there")
"hello there"
```

The Comment Character

The semicolon is the comment character in Skm. It comments out the remainder of the line on which it is located. As seen above, the define operator returns a value that is printed after the Skm statement is evaluated. In the discussion that follows, the printing of a return value (when present) is sometimes omitted for the sake of simplifying the presentation.

B.1.2. IRIS Explorer Operators

This section introduces the Skm commands used to perform functions in IRIS Explorer. Those commands that accept one or more module names as arguments can handle zero or more module names listed in sequence, or a Scheme list of module names. The semantics of the commands when given no arguments depends on the particular command, but in general it means to apply the command to all selected modules. **Table B−1** lists the available commands.

Command	Purpose
all−mods	List all modules
all−groups	List all groups
break−loop	Break a loop by halting a named module
connect	Establish a connection between two modules
connection?	Test a variable for connection information
connection−list	List all modules connections
copy	Copy selected modules and groups
cut	Cut selected modules and groups
destroy	Destroy selected modules and groups
disable	Disable selected modules and groups
disconnect	Remove a connection between two modules
dup	Duplicate selected modules and groups
enable	Enable selected modules and groups
exec−hilite−off	Turn off execution highlighting
exec−hilite−on	Turn on execution highlighting
fire	Fire selected modules or groups
get−param	Get the value of a module’s parameter
get−pfunc	Get the value of a module’s P−Func
help	Provide information about command syntax
help−mod	Show module’s help page
hide−widget	Hide a module’s widget
interrupt	Interrupt a running module
log	Show module’s log window
loop−ctlr?	Test for the active loop controller module
loop−ctlr−capable?	Test for a loop controller−capable module

loop−ctlr−off	Turn loop controller module off
loop−ctlr−on	Turn loop controller module on
main−log−off	Toggle the Log window off
main−log−on	Toggle the Log window on
maxi−at	Create a maximized module control panel
maxi	Maximize the control panels of selected module
micro	"Micro−ize" the control panel of a module
mini	Minimize the control panel of a module
module?	Test a variable for module start information
module−to−filename	Print the file path of a module's resource file (*.mres* file)
module−to−host	Print the name of the host a module is running on
param−list	Print a list of a module's parameter value(s)
paste	Paste the contents of the editing buffer
pause	Cause the script execution to pause
perf−report−off	Switch off performance reporting for selected modules
perf−report−on	Switch on performance reporting for selected modules
pfunc−list	Print a list of a module's P−Func value(s)
save	Save selected modules to a named map file
select	Select named modules
selected−mods	List the selected modules
set−app−main−win	Designate a maximized module control panel as the main window for an application. If this window is quit while IRIS Explorer is in application mode, the application itself quits.
set−param	Set a parameter value in a module
set−param−minmax	Set min and max values for a range parameter
set−pfunc	Set a module's P−Func value
show−widget	Expose a module's hidden widget (see hide−widget)
start−map	Start a map
start	Start a module
quit−explorer	Exit IRIS Explorer
unmaxi	Remove module's maximized control panel
unmicro	"Unmicro−ize" the control panel of a module
unselect	Unselect named modules
unselected−mods	List the modules not selected

Table B−1 Skm Commands

B.1.3. Command Syntax

Table B−2 summarizes the command syntax. Square brackets enclose optional arguments.

This information is available from within Skm by typing **`(help)`**. You can also type **`(help−command "prefix")`** and you will get a list of all commands beginning with the given prefix.

Command
(all−mods)
(all−groups)
(break−loop "modulename")
(connect `"modulename"` "portname" "modulename" "portname")
(connection? variable)

```
(connection−list "modulename")
(copy "modulename" ...)
(cut "modulename" ...)
(destroy "modulename" ...)
(disable "modulename"...)
(disconnect "modulename" "portname" "modulename" "portname")
(dup "modulename" ...)
(enable "modulename"...)
(exec−hilite−off "modulename"...)
(exec−hilite−on "modulename"...)
(fire "modulename" ...)
(get−param "modulename" "portname")
(get−pfunc "modulename" "portname")
(help [prefix])
(help−mod"modulename" ...)
(hide−widget "modulename" "widgetname")
(interrupt "modulename"...)
(log "modulename"...)
(loop−ctlr? variable)
(loop−ctlr−capable? variable)
(loop−ctlr−off "modulename"...)
(loop−ctlr−on "modulename"...)
(main−log−off)
(main−log−on)
(maxi−at "modulename" [  [ ‘at X Y]  [ ‘size W H] ] )
(maxi "modulename" ...)
(micro "modulename"...)
(mini "modulename"...)
(module? variable)
(module−to−filename "modulename")
(module−to−host "modulename")
(param−list "modulename")
(paste)
(pause seconds)
(perf−report−off  "modulename" ...)
(perf−report−on  "modulename" ...)
(pfunc−list "modulename")
(save  "filename" "modulename" ...)
(select "modulename" ...)
(selected−mods)
(set−app−main−win "modulename")
(set−param "modulename" "portname" value)
(set−param−minmax "modulename" "portname" minval maxval)
(set−pfunc "modulename" "portname" value)
(show−widget "modulename" "widgetname")
(start−map "map−name" [ ‘at X Y] )
```

(start "modulename" [‘at X Y])
(quit−explorer)
(unmaxi "modulename")
(unmicro "modulename")
(unselect "modulename" ...)
(unselected−mods)

Table B−2 Skm Command Syntax

Some commands have an alternate syntax. Any command that takes a double quoted string specifying a module name can also take a variable of type <*Explorer Module*>. Such variables are returned by the *start* command when it executes successfully. Similarly, a shortcut way of specifying a connection is to use a variable of type<*Explorer Connection*>. These variables are returned by the *connect* command. Use of these special variables is explained in greater detail later.

B.1.4. IRIS Explorer−related Command Descriptions

This section discusses the most commonly used IRIS Explorer−related Skm commands, along with those whose operation is not obvious.

start

The *start* command is used to start modules. Here are a few examples:

```
(start "ReadImg")
(start "SharpenImg" "at" 200 300)
```

The *start* command returns a value of type <*ExplorerModule*> which contains the name of the started module; therefore, you can assign the return value in the following manner:

```
(define mod (start "ReadImg"))
```

The variable *mod* can now stand in for the string "ReadImg" whenever needed.

destroy

Modules can be destroyed using the *destroy* command. So to destroy the module *ReadImg*, you can type:

```
(destroy "ReadImg")
```

or if the value returned by *start* was saved in a variable named *mod*, you can use:

```
(destroy mod)
```

Note that assigning the value returned by *start* is more than a convenience; it is required under some circumstances. There, you requested that the module *ReadImg* be launched. It is possible, though, that a *ReadImg* module already exists. If so, then the module started as a result of this code will be named *ReadImg<2>*, not *ReadImg*. Subsequent connect commands that explicitly refer to the string *ReadImg* will reference the pre−existing *ReadImg* module and will fail to have the desired effect.

The solution is to avoid referring to modules by the name used in the *start* command. Instead, assign the value returned by *start* to a variable and use the variable for subsequent references.

start−map

Maps are started using the *start−map* command:

```
(start−map "cfd")
```

connect

The *connect* command takes an output module and port and an input module and port:

```
(connect "ReadImage" "Output" "SharpenImg" "Img In")
```

As with *start*, a value is returned by *connect* that may be saved in a variable:

```
(define c1 (connect "ReadImg" "Output" "SharpenImg" "Img In"))
```

You can delete the connection with:

```
(disconnect "ReadImg" "Output" "SharpenImg" "Img In")
```

or

```
(disconnect c1)
```

Variables containing module and connection information persist even after the IRIS Explorer objects to which they refer are gone. Thus, you can reconnect the modules just disconnected with:

```
(connect c1)
```

Testing Variable Types

Skm provides the predicates *module?* and *connection?* to test the type of a variable.

```
(module? c1)
()
(connection? c1)
t
```

A predicate returns either `()` (false) or `t` (true). The following:

```
(module? m1)
t
```

means that the variable *m1* contains information about a module started some time in the past. The `true` return value does not necessarily mean the module still exists.

B.2. Skm Language Features

The sections that follow describe the programming language features of Skm.

B.2.1. Comparison Operators

Skm provides two basic comparison operators: *eq?* and *eqv?*. The first, *eq?*, determines if two symbols evaluate the same. The latter, *eqv?*, is more general. It determines the equality of both symbols and numbers.

eq?

Below are examples using *eq?* to test the equality of symbols.

```
(eq? ’test ’test)
t
(eq? ’test ’hat)
()
```

But, you can’t use *eq?* for numbers:

```
(eq? 3 3)
()
```

eqv?

Here, *eqv?* is shown for numbers and symbols:

```
(eqv? 3 3)
t
(eqv? 3 4)
()
(eqv? ’test ’test)
t
```

Numbers can also be compared with the following operators:

```
=, !=, <, <=, >, >=
```

You can use these operators only to compare numbers.

B.2.2. Conditional Operators

A conditional test can be performed using the *if* operator, which takes the form:

```
(if test−expr then−expr else−expr)
```

For example:

```
(if (eq? expr−1 expr−2)
  (do−something)
  (do−something−else))
```

If more than one statement belongs in either the *then* or the *else* portion of the *if* construct, use the *begin* operator. It is analogous to curly braces in C and to *begin−end* in Pascal−like languages.

```
(if (eq? expr−1 expr−2)
  (begin
     (do−this)
     (do−that))
     (do−something−else))
```

B.2.3. String Operators

Skm provides the set of string operators listed below:

string?	Returns true if its argument is a string.
string−length	Returns length of string argument.
string−ref	Returns a character at indicated position.
string−set!	Sets a character at indicated position.
string=?	Compares two strings. Returns true if equal.
substring	Returns substring of given string between start and stop index.

string?

string? is a predicate that tests an argument to see if it is a string. It takes a single argument. *string?* returns `t` or `()`, depending on the type of its argument.

```
(string? "some−string")
t
(string? 23)
()
```

string−length

*string−length*returns the length of a string argument.

```
(string−length "some−string")
11
```

string−ref

*string−ref*returns a specific character from a given string. The first argument is the string; the second is the index of the character to be extracted. Indexing is origin zero (that is, the index of the first character in the string is zero). The returned character is still a string and may be used in all string commands.

```
(string−ref "hello" 1)
 "e"
```

string−set!

*string−set!*sets a character in a string. The first argument is the target string; the second argument is the index into the target string; the third argument is the character that replaces the indexed character in the target string. The target string is changed as a side effect. *string−set!* returns `t` if it succeeded and `()` otherwise. The index is origin 0.

```
(string−set! "hello" 0 "J")
t
;; string is changed to "Jello"
```

string=?

The *string=?* function compares two strings for equality. If equal, `t` is returned.

```
(string=? "skm" "skm")
t
(string=? "skm" "scheme")
()
```

substring

The *substring* function returns a substring of a string using the supplied begin and end indices.

```
(substring "abcde" 0 3)
"abc"
```

string−append

The *string−append* function appends two strings.

```
(string−append "abc" "def")
"abcdef"
```

B.2.4. Loading Script Files

To read in a script file from within script code, use the *load* command:

```
(load "myfile.skm")
```

B.3. Procedures in Skm

Creating a procedure in Skm is done with the *define* and *procedure* constructs. The following defines a procedure that takes a single argument, declared formally as ’a,’ and adds one to it. The value returned by the procedure, if any, is the last value of the procedure.

```
(procedure (a)
 (+ a 1)
)
```

This procedure doesn’t have a name yet (procedures can be "anonymous" in Skm). The following gives the procedure the name plus1.

```
(define plus1 (procedure (a) (+ a 1)) )
```

You would invoke this procedure with:

```
(plus1 3)
4
```

Here is the definition and use of a procedure called "add−pair," which takes two numbers and adds them together:

```
(define add−pair
 (procedure ( arg1 arg2 )
 (+ arg1 arg2)
 )
)

(add−pair 2 2)
4
```

B.3.1. Using a Skm Procedure

Connecting image processing modules in IRIS Explorer is a common occurrence. Image processing modules that operate on a single input typically have an input port named "Img In" and an output port named "Img Out".

This procedure simplifies connecting such modules by supplying these port names automatically.

```
(define image−conn
 (procedure ( mod1 mod2 )
 (connect mod1 "Img Out" mod2 "Img In")))
```

You can use this procedure as follows:

```
(image−conn "SharpenImg" "DisplayImg")
```

B.3.2. Skm Output

Skm output is handled by the *print* command. It takes one or more arguments and generates output to the IRIS Explorer console (the shell from which IRIS Explorer was launched).

```
(define v1 "abc")
(define v2 123)
```

```
(define m1 (start "ReadImg"))
(print v1 v2)
abc123
(print "My module is " m1)
My module is #< IRIS Explorer Module Ref: ReadImg >
```

B.3.3. Looping

In Skm, the procedure *for−loop* is used to implement a style of iteration similar to that used by C. The format of the *for−loop* call is:

```
(for−loop start op stop step loop−body−proc)
```

Like the typical C `for loop` structure, the Skm *for−loop* takes a start count, a comparison operator, a stop count, and an increment. Instead of having a loop body as does the C construct, *for−loop* takes as its final argument a loop−body construct (explained below) or a procedure of one input argument. The input argument to this loop−body procedure is the loop count.

To create a loop that prints all numbers between zero and nine (inclusive), you write:

```
(for−loop 0 < 10 1
   (loop−body(i)
  (print i)
)
0
1
2
3
4
5
6
7
8
9
```

The loop−body construct is used as the final argument to *for−loop*. The loop−body construct takes no arguments or a single input argument. In the latter case, the argument is a variable that is scoped locally to the loop body and that serves as the loop counter.

If the loop counter is not needed, loop−body is used as follows:

```
(for−loop 0 < 3 1 ;; do something 3 times
  (loop−body()
  (do−something)
  )
)
```

Since the built−in *print* procedure can be called with a single input argument, the procedure can be used directly as the last argument to *for−loop* (without having to use the loop−body construct). Here is an alternate implementation of the print example above:

```
(for−loop 0 < 10 1 print)
```

The next example presents a doubly nested loop. The outer loop variable is `i` and the inner loop variable is `j`.

```
(for−loop 0 < 5 1
  (loop−body(i)
  "outer loop: i = " i )
  (for−loop 1 < 4 1
  (for-loop 1 < 4 1
     (loop-body(j)
       (print " inner loop: j = " j )
     )
  )
)
)

outer loop: i = 0
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 1
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 2
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 3
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
outer loop: i = 4
 inner loop: j = 1
 inner loop: j = 2
 inner loop: j = 3
()
```

B.3.4. Lexical Scoping of Variables

Variables created with *define* are globally scoped regardless of the context in which *define* occurs. Thus, the following uses of *define* both introduce variables into the global environment.

```
(define var1 1)
1
(begin
(define var2 2))
2
(print var1 " " var2)
1 2
```

The `(begin ... )` block in the second example does not behave as does the creation of a local variable in C where the variable goes out of scope at the end of the block:

```
{
int var2 = 2;
}
```

Arguments passed to a procedure in Skm are scoped locally to that procedure. Thus, in the following example *define* introduces neither a nor b into the global environment:

```
(define test
   (procedure ( a b )
      (define a 1)
      (define b 2)))
```

It seems at first that the example above contradicts the earlier statement that *define* always introduces variables into the global environment. However, *define* here is not being used to create a variable (the variables a and b are created at the entry to the procedure). Rather, *define* is being used to change the value of these variables. You can also use the assigned operator `set!` here.

The let Construct

Skm provides a special construct called *let*, which sets up a locally scoped environment. The format of *let* is:

```
(let     ((var1 init-val)
   (var2 init-val)
   ....
   (varN init-val))

let-body)
```

Any number of locally scoped variables can be created and initialized in the preamble of the *let* statement. The variables thus created and initialized are visible only within the code present in *let*, as seen in the following example:

```
(let ((a 1)       (b 2))
   (print "inside let, a = " a " b = " b))
inside let, a = 1 b = 2
```

B.3.5. List Processing

A list can be created using *list.*

```
(list ’a ’b ’c ’d)
(a b c d)
```

As indicated above, when Skm prints out a list, it surrounds the list with parentheses.

A Skm command that returns a list is *all−mods*:

```
(all−mods)
("SharpenImg" "ReadImg")
```

The first element of a list can be extracted using *first.* A list with the first element removed is obtained through the use of the *rest* command.

Footnote: *first* is also known as *car* and *rest* is also known as *cdr.*

```
(first ’(a b c d))
a
(rest ’(a b c d))
(b c d)
```

The list `(a b c d)` is quoted so that it will not be evaluated by Skm before being passed to *first* and *rest*.

B.4. Module Interface to Scripting

Modules in C or C++ may issue scripting commands through this API routine:

```
int cxScriptCommand( char * string );
```

This call sends a string containing the Skm code to the Map Editor for evaluation. A valid example is:

```
cxScriptCommand
( "(start \"ReadImg\" ’at 100 100 )" );
```

Embedded double quotes must be escaped in accordance with C syntax for string constants. The input string must contain a valid and complete Skm expression. You may not start an expression in one call and finish it in a subsequent call, as, for instance, in this example:

```
cxScriptCommand ( "(start \"ReadImg\" " );
cxScriptCommand ( " ’at 100 100 ) ");
```

However, multiple independent or nested expressions can be sent in a single call:

```
char * string = "(define a 1) (define b 2)"
cxScriptCommand ( string );
```

Multiple calls can be made as long as each one contains independent expressions:

```
cxScriptCommand ( "(define a 1)" );
cxScriptCommand ( "(define b 2)" );
```

Calling *cxScriptCommand* results in a message transmission and several context switches. Where it is possible to bundle expressions in a single string, performance will be improved.

Some validity checking is done before the script is sent to IRIS Explorer. If *cxScriptCommand()* returns a `−1`, then the validity test failed and no script code was sent. Currently, the validity test will fail if the string contains no parentheses or an uneven set of left and right parentheses.

Normal interpreter output for commands issued by modules is suppressed. Only those error messages generated as a result of module script commands appear on IRIS Explorer’s `stderr`. Thus, input that is valid when the user types it at the > prompt, is not valid when generated by a module; in particular, querying the value of a variable if the query is not in parentheses.

Given this definition:

```
(define var 100)
100
```

the following is valid when typed by a user, but is not accepted from a module, since the printing of evaluated expressions is suppressed for module input:

```
var
100
```

CxScriptCommand() performs asynchronously with respect to module execution, and IRIS Explorer does not acknowledge Skm code execution.

Footnote: *cxScriptCommand* returns −1 if the Skm code was not sent to IRIS Explorer at all.

Consequently, a module cannot know for sure whether the script code had the desired effect. For debugging purposes, however, certain types of errors are noted on IRIS Explorer’s console output (`stderr`).

As with input typed in directly by the user, script commands perform synchronously with respect to one another when required. Thus in the following sequence, *ReadImg* and *DisplayImg* will exist when the *connect* command is issued, assuming no failure during launch.

```
char * string1 = "(define m1 (start ’ReadImg))"
char * string2 = "(define m2 (start ’DisplayImg))";
char * string3 = "(connect m1 ’Output m2 \"Img In\")";

char * string [LONG_LEN];

strcpy ( string, string1 );
strcpy ( string, string1 );
strcat ( string, string2 );
strcat ( string, string3 );

cxScriptCommand ( string );
```

Remember to not refer to modules by the name used in the *start* command (since the actual module launched may have an instance number appended to its name). Instead, use the name returned by the *start* command.

Global Namespace

There is a single global namespace in the script interpreter shared by the user typing at the IRIS Explorer user interface console and by all modules issuing scripting commands. The global namespace can be exploited by certain applications, but it also represents a hazard to the unwary.

For example, suppose you have written a module, *MyMod*, that under certain conditions will issue a scripting command to start another module. You have assigned the value returned by the *start* command to a variable, called *mod*, used for subsequent manipulations of the launched module.

A problem arises, though, if a user launches more than one instance of *MyMod*. All instances of *MyMod* will assign the result of the *start* command to the global interpreter variable *mod1*. The result is that *mod1* will contain a reference to the last started module and earlier references will be overwritten.

The solution to this problem is to name variables distinctively so that other modules avoid touching them. A good way to get distinctive names is to prepend the module name to the variable. Module names are unique within IRIS Explorer, so the variable namespaces should not overlap.

In some cases, this does not matter. For instance, in an IRIS Explorer application in which there is only one module that issues scripting commands, the module can name variables without restriction. There can even be more than one module issuing script commands in an application, as long as the module writer has coordinated them with respect to interpreter variable naming.

To be perfectly safe, an application map must be run as an application and not as a simple map. When run as an application, the application controls which modules are started. When run as a map, the application is not in total control since the user has access to the Module Librarian and the Map Editor and can thus always launch unrelated maps and modules that may themselves issue script commands.

B.4.2. A Complex Example

This is an example of advanced Skm programming. Given the built−in function *mods−to−list*, we will construct a procedure that determines if a particular module exists. The procedure is named *mod−exists?* and is specified as follows:

```
(define mod−exists?
 (procedure(mod)
 (mod−in−list? mod (all−mods))))
```

mod−exists? is fairly simple. It passes its single argument *mod* and the output of *all−mods* to a helper procedure, *mod−in−list,*which is defined below. As *mod−in−list?* is the last statement in the procedure, its return value will be returned by *mod−exists?*.

Here is the definition of *mod−in−list?*:

```
(define mod−in−list?
 (procedure ( mod m−list )
    (if (null? mod) ()
    (if (null? m−list) ()
     (let ((m (first m−list)))
     (if (null? m) ()
       (if (string=? mod m) t
      (mod-in-list? mod (rest m-list))))))))
```

mod−in−list? takes two arguments: a module name string and a list of module name strings. First, *mod−in−list?* validates its arguments by checking both the target module name and the list of modules to make sure they are not NULL. The procedure assigns the first element of the list to the temporary variable *m* using a *let* statement and compares it with the target module name. If equal, the procedure returns `t` (true).

If not equal, the procedure continues the search by recursively calling itself with the target string and the module list is reduced by having the first element removed.

Here is an improved version of *mod−exists?* that performs some error checking on its arguments.

```
(define mod-exists?
 (procedure(modname)
    (if (null? modname)
       ; then
       (print "mod-exists?: null arg")
       ; else
       (if (not (string? modname))
         ; then
         (print "mod-exists?: supplied with non-string arg")
      ; else
       (mod-in-list? modname (all-mods))))))
```

As defined, *mod−exists?* works only when given a module name as a string.

B.4.3. Known Bugs

Launching a module on a remote machine with the *on* argument to the start command will not work the first time if no Module Librarian has been created for that machine and if no prior module has been launched on that machine in the current IRIS Explorer session. When such a module fails to launch, it will (as a side effect) cause a Module Librarian to be created for that remote machine. After a few seconds have passed and the Librarian is present, subsequent attempts to launch modules on that machine should succeed.

Parentheses within double−quoted strings may confuse the Skm parser.

Note: When setting the value of a parameter using *set−param*, you may need to know something about the type of widget associated with the parameter.

B.4.4. References

Skm is similar to the Scheme programming language. A good Scheme language text may be of some value to those seeking advanced understanding of Skm.

A thorough, well−written text is *Scheme and the Art of Programming* by Springer and Friedman. A quick and amusing introduction is *The Little Lisper* by Friedman and Felleisen.

Additional information may be found on the network, for pointers see **`http://www.nag.co.uk/0/visual/IE/iecbb/Info.html`**.

B.4.5. Copyright Notice

Skm is based on the SIOD interpreter by George Carrette. Note that features of SIOD not explicitly documented herein are not supported. The following copyright notice accompanies SIOD as distributed on *altdorf.ai.mit.edu*.