SC38-0486-03

IBM Visualization Data Explorer

User’s Reference

Version 3 Release 1 Modification 4

IBM

IBM Visualization Data Explorer

SC38-0486-03

User’s Reference

Version 3 Release 1 Modification 4

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xi.

Fourth Edition (May 1997)

This edition applies to IBM Visualization Data Explorer Version 3.1.4, to IBM Visualization Data Explorer SMP Version 3.1.4, and to all subsequent releases and modifications thereof until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers’ comments appears at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation
Thomas J. Watson Research Center/Hawthorne
Data Explorer Development
P.O. Box 704
Yorktown Heights, NY 10598-0704
USA

If you send information to IBM, you grant IBM a nonexclusive right to use or distribute that information, in any way it believes appropriate, without incurring any obligation to you.

Contents

Figures

Tables

Notices

Products, Programs, and Services

References in this publication to IBM* products, programs, or services do not imply that IBM intends to make these available in all countries in which it operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give the user any license to those patents. License inquiries should be sent, in writing, to:

International Business Machines Corporation
IBM Director of Licensing
500 Columbus Avenue
Thornwood, New York 10594
USA

Trademarks and Service Marks

The following terms, marked by an asterisk (*) at their first occurrence in this publication, are trademarks or registered trademarks of the IBM Corporation in the United States and/or other countries.

AIX
IBM
IBM Power Visualization System
RISC System/6000
Visualization Data Explorer

The following terms, marked by a double asterisk (**) at their first occurrence in this publication, are trademarks of other companies.

AViiON	Data General Corporation
DEC	Digital Equipment Corporation
DGC	Data General Corporation
Graphics Interchange Format (GIF)	CompuServe, Inc.
Hewlett-Packard	Hewlett-Packard Company
HP	Hewlett-Packard Company
iFOR/LS	Apollo Computer, Inc.
Motif	Open Software Foundation
NetLS	Apollo Computer, Inc.
Network Licensing Software	Apollo Computer, Inc.
OpenWindows	Sun Microsystems, Inc.
OSF	Open Software Foundation, Inc.
PostScript	Adobe Systems, Inc.
X Window System	Massachusetts Institute of Technology

Copyright notices

About This Reference

This manual contains detailed descriptions of IBM Visualization Data Explorer* tools for transforming, realizing, and rendering data. Each description includes the script-language syntax for invoking the tool; input and output specifications; details of module function; and names of sample visual programs that demonstrate the module's function.

It is strongly recommended that first-time user's of Data Explorer consult the *IBM Visualization Data Explorer QuickStart Guide* before proceeding (see "Related Publications and Sources").

Users creating visual programs or scripts should consult this reference in conjunction with *IBM Visualization Data Explorer User's Guide*, which contains information about the development and modification of visual programs and the syntax of the Data Explorer scripting language.

Typographic Conventions

Boldface	Identifies commands, keywords, files, directories, messages from the system, and other items whose names are defined by the system.
Italic	Identifies parameters with names or values to be supplied by the user.
`Monospace`	Identifies examples of specific data values and text similar to what you might see displayed or might type at a keyboard or that you might write in a program.

Related Publications and Sources

IBM Publications

- *IBM Visualization Data Explorer User's Guide*, SC38-0496

  Details the main features of Data Explorer, including the data model, data import, the user interface, the Image window, and the visual program editor. and the scripting language. Of particular interest to programmers: chapters on the data model and the scripting language.

- *IBM Visualization Data Explorer User's Reference*, SC38-0486

  Contains detailed descriptions of Data Explorer's tools.

  Note: Consult this reference if you are creating visual programs or scripts.

- *IBM Visualization Data Explorer Programmer's Reference*, SC38-0497

  Contains detailed descriptions of the Data Explorer library routines.

  Note: Consult this reference if you are writing your own modules for Data Explorer.

Non-IBM Publications

The following treat various aspects of computer graphics and visualization:

Adobe Systems Incorporated, *PostScript Language Reference Manual*, 2nd Ed., Addison-Wesley Publishing Company, Massachusetts, 1990.

Aldus Corporation and Microsoft Corporation, *Tag Image File Format Specification, Revision 5.0*, Aldus Corporation, Washington, 1988.

Arvo, Jim, ed., *Graphics Gems II*, Academic Press, Inc., Boston, Massachusetts, 1991.

Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F., *Computer Graphics: Principles and Practice*, Addison-Wesley Publishing Company; Massachusetts, 1990.

Friedhoff, Richard M., and Benzon, William, *Visualization: The Second Computer Revolution*, New York, Harry N. Abrams, Inc., 1989.

Glassner, Andrew, ed., *Graphics Gems*, Academic Press, Inc., Boston, Massachusetts, 1990.

Hill, F.S., Jr., *Computer Graphics*. Macmillan Publishing Company, New York, 1990.

Kirk, David, ed., *Graphics Gems III*, Academic Press, Inc., Boston, Massachusetts, 1992.

Robin, Harry, *The Scientific Image: from cave to computer*, Harry N. Abrams, Inc., New York, 1992.

Rogers, David F., *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, New York, 1985.

Rogers, David F. and Adams, J.Alan, *Mathematical Elements for Computer Graphics*, 2nd Ed., New York, McGraw-Hill Book Company, 1990.

SIGGRAPH Conference Proceedings, Association for Computing Machinery, Inc.: A Publication of ACM SIGGRAPH, New York, various years.

Tufte, Edward, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, 1983.

Other sources of information

For additional ideas, consult the “DX Repository,” available through anonymous FTP (`ftp.tc.cornell.edu.` in directory `pub/Data.Explorer`), and gopher (`ftp.tc.cornell.edu.` port 70). This public software resource includes information and visual programs contributed by Data Explorer users from around the world. We encourage you to contribute your innovations and ideas to the Repository, in the form of new modules, macros, visual programs, and tips and tricks you discover as you learn and master Data Explorer.

On the Internet, the newsgroup `comp.graphics.apps.data-explorer` is used by customers around the word to share information and ask questions. This newsgroup is also followed by Data Explorer developers.

If you have access to the World Wide Web, you can find the Data Explorer home page at `http://www.almaden.ibm.com/dx/`.

Chapter 1. Data Explorer Tools

Data Explorer tools (i.e., tool modules) perform a variety of functions to generate visual images. In the VPE window, their icons can be used as "building blocks" to create visual program *networks*, from which images are generated in the Image window. They can also be invoked from script-language programs or from the command line (when Data Explorer is in "script mode"). And they can be grouped together in macros for greater efficiency.

Many modules are accessible through both the graphical user interface and the scripting language. Some, however, are available only through the user interface: the Interactors, the Color Editor, Image, Pick, Probe, Receiver, and Transmitter. Most have default values for parameter inputs, values that are (as much as possible) appropriate for the data involved. These defaults enable the new user to begin visualizing data with only a few connections and to customize a visualization program one step at a time.

Tools are divided into the categories listed in the top palette of the VPE window:

- Annotation
- DXLink
- Debugging
- Flow Control
- Import and Export
- Interactor
- Interface Control
- Realization
- Rendering
- Special
- Structuring
- Transformation
- Windows
- {ALL} (an alphabetic listing of all the tools).

The directory **`/usr/lpp/dx/samples/programs`** contains visual program examples that use these tools, along with a subdirectory (...**`/SIMPLE`**) of simpler examples. Use the **`Open`** file-selection dialog box to call up either directory (adding "/*.net" to the directory name when you enter it in the **`Filter`** text field). For a description of a visual program, either highlight the program name in the **`Files`** list of the dialog box and click on the **`Comments`** pushbutton or select the **`Application Comment`** option of the **`Help`** pull-down menu (after the visual program is loaded).

The directory **`/usr/lpp/dx/samples/scripts`** contains script examples for all of the modules. Descriptions are included as comments in the script files themselves.

The rest of this chapter summarizes the tool categories and the tools they contain. The order in which they appear is the same as that displayed in the palettes of the VPE window. For detailed descriptions of individual tools, see Chapter 2, "Functional Modules" on page 15.

Note: Two modules that appear in Chapter 2 are available only in script mode and so are not listed in any of the Visual Program Editor categories:

Executive	Executes an executive command (see "Executive" on page 126).
KeyIn	Waits for input from the system prompt in the window from which you invoked Data Explorer (see "KeyIn" on page 193).

1.1 Tool Categories

Annotation

These tools add various kinds of information to a visualization.

AutoAxes	Creates an axes box for a specified data set, using a specified camera. The camera is required so that the AutoAxes module can determine how to position the axes box and labels.
AutoGlyph	Produces glyphs (a representational figure, such as an arrow) for every data value in an input Field. The size and type of glyph are based on the data it receives.
Caption	Creates a caption for an image. The caption position specified in pixel- or viewport-relative coordinates. Captions always remain aligned to the screen.
ColorBar	Creates a color bar to be displayed on the screen. It accepts a color map as input.
Format	Creates a string from numeric values. For example, you can create the string "`Isosurface value = 23.4`", where 23.4 is the output of an Interactor.
Glyph	Produces glyphs for every data value in an input Field. This module allows more precise control of glyph size than AutoGlyph does.
Legend	Produces a legend which associates a string with a color.
Parse	Separates a string into its component parts (e.g., the string "data = 3.7" into "data", "=", and the floating point value "3.7").
Plot	Creates a 2-dimensional plot from a line. You can specify: customized labels for the axes; axes limits; y-axes on the left and right sides of the plot; and logarithmic or linear axes.
Ribbon	Produces a ribbon of specified width from an input line. If a "normals" component is present on the line (e.g., if the **`curl`** option was used to create a streamline or streakline), the twist of the ribbon will correspond to the direction of the "normals."
Text	Produces text that appears in the "space" occupied by the image. That is, unlike Caption, this module specifies the text position and size in the same coordinates as the data.
Tube	Produces a tube of specified diameter and number of sides from an input line. If a "normals" component is present on the line (e.g., if the **`curl`** option was used to create a streamline or streakline), the twist of the tube will correspond to the direction of the "normals."

DXLink

These tools are used to control Data Explorer from a separate program.

Note: According to UNIX alphabetical convention, capital letters precede lower case. Thus DXLink precedes Debugging in the category pallet of the VPE, and the DXLink routines precede Direction in the tools pallet and in the routine descriptions in Chapter 2, “Functional Modules” on page 15.

DXLInput	Enables a remote DXLink application to set a parameter value in a visual program.
DXLInputNamed	Enables a remote DXLink application to set a parameter value in a visual program, but also allows the name of the variable to be set via a wire into the DXLInputNamed tool.
DXLOutput	Sends a value to a remote application.

Debugging

These tools facilitate the analysis of a program’s execution.

Describe	Presents information about any Data Explorer object in the Message window.
Echo	Echoes a message. It can also print simple Array Objects (e.g., output of the Statistics module). In the user interface this output appears in the Message window.
Message	Displays a message to the user, either in the Message window or in a pop-up dialog box.
Print	Prints information about an Object (e.g. a Field or a Camera). You can specify the level of detail. In the user interface the output appears in the Message window.
System	Allows you to execute arbitrary operating-system commands.
Trace	Enables the tracing of time spent in a module, tracing of memory, or the use of debugging flags or user-written modules.
Usage	Prints the amount of memory currently being used by Data Explorer.
Verify	Checks an Object for internal consistency.
VisualObject	Creates a renderable object representing an object’s hierarchy.

Flow Control

These tools control the flow of execution in a visual program.

For additional information, see Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

Done	Specifies whether an executing loop should terminate.
Execute	Allows the user to change the execution state of a visual program without using the **Execute** pull-down menu in the user interface.
First	Indicates whether the current iteration of the loop is the first.

ForEachMember	Initiates a loop for each member of a group or item in an array.
ForEachN	Iterates through a specified set of integers.
GetGlobal	Retrieves an object from the cache. Maintains state between executions.
GetLocal	Retrieves an object from the cache.
Route	Routes an Object through selector-specified output paths.
SetGlobal	Places an object in the cache. Maintains state between executions.
SetLocal	Places an object in the cache.
Switch	Switches the output between a list of inputs.

Import and Export

The first two tools listed, along with ReadImage and WriteImage, are concerned with the flow of data into and out of a visual program. The others typically process data immediately after it has been imported.

Export	Exports Objects created in Data Explorer to an external data file (in Data Explorer file format).
Import	Brings data into Data Explorer from a specified file. If the file contains more than one variable or contains multiple frames of data, portions of the data can be specified for importation. Supported formats are native Data Explorer format, CDF, netCDF, HDF, and General Array format.
ImportSpreadsheet	Brings data into Data Explorer from spreadsheet, or tabular, data file.
Include	Includes (or excludes) points based on their data values (e.g., removing all points with data values greater than 9.3). It can also be used to remove data marked as invalid.
Partition	Subdivides data to take advantage of parallelism for Data Explorer SMP. You can control the level of subdivision.
ReadImage	Reads an image from an external file.
Reduce	Reduces the resolution of a data set, filtering and resampling the set at a lower resolution.
Refine	Increases the number of samples in a data set. This module interpolates data values or colors at the new positions from the data or color values at the original positions. It can also convert connections from quads or faces to triangles, and from cubes to tetrahedra.
Slab	Takes a positional subset of a data set with regular connections (cubes, quads, and lines). You can specify along which axis to take the subset, where the slab should begin, and how thick the slab is to be. The module performs no interpolation.

Slice	Takes a positional subset of a data set with regular connections, like the Slab module. However, unlike the Slab module, it creates output with a dimensionality 1 less than the dimensionality of the input. For example, you can import a data set as a 4-dimensional grid, with the fourth dimension representing time. You can then use the Slice module to create 3-dimensional slices of the data set.
Stack	Stacks a series of *n*-dimensional Fields to form a single (*n*+1)-dimensional Field. This module can also be used to increase the dimensionality of a single Field.
Transpose	Transposes the positions of a Field. For example, it can interchange the *x* and *y* axes.
WriteImage	Writes an image to an external file in a specified format. The same functionality is provided by the **SaveImage** option of the **File** pull-down menu of the Image window.

Interactor

These tools provide interactive control of inputs to other modules in visual programs. They can be used only with the user interface.

For additional information, see 7.1, "Using Control Panels and Interactors" on page 128 in *IBM Visualization Data Explorer User's Guide*. Interactors are named after the type of data they control:

FileSelector	Presents a standard Motif file-selector dialog box. The output is a file name.
Integer	Presents a stepper, slider, dial, or text interactor. The output is a whole number.
IntegerList	Presents a scrolled-list or text interactor. The output is a list of integers.
Reset	Presents a toggle button. The output is one value the first time it is run, and a different value on subsequent runs until the toggle is selected again.
Scalar	Presents a stepper, slider, dial, or text interactor. The output is a real number.
ScalarList	Presents a stepper, slider, dial, or text interactor. The output is list of scalar values.
Selector	Presents an option menu, a set of radio buttons, or a toggle button. The outputs are a value and a string, representing a choice from a selection.
SelectorList	Presents a selection list. The output is a list of values and a list of strings, representing a choice of one or more from many.
String	Presents a text interactor. The output is a text string.
StringList	Presents a scrolled-text list. The output is a list of strings.
Toggle	Presents a toggle button. The output is one of two values (set or unset).

Value	Presents a text interactor. The output is a value (scalar, vector, tensor, or matrix).
ValueList	Presents a scrolled list or text interactor. The output is a value list.
Vector	Presents a stepper or text interactor. The output is a vector.
VectorList	Presents a scrolled-list or text interactor. The output is a vector list.

Interface Control

These tools are used to control Data Explorer tools from within a visual program.

ManageColormapEditor	Allows colormap editors to be opened and closed from within a visual program.
ManageControlPanel	Allows control panels to be opened and closed from within a visual program.
ManageImageWindow	Allows image windows to be opened and closed from within a visual program.
ManageSequencer	Determines whether the Sequence control panel is displayed or not.

Realization

These tools create structures for rendering and display (e.g., bands, triangle connections, isosurfaces, and boundary boxes).

AutoGrid	Maps a set of scattered points onto a grid. The grid will be created automatically by Data Explorer.
Band	Divides a Field into bands based on given division values.
Connect	Creates triangle connections for a Field of scattered positions.
Construct	Constructs an arbitrary Field. You can specify the origin, the deltas, the counts in each dimension, and the data. You can also use Construct to create a Field containing a “positions” component with given position values (e.g., the output of the ProbeList tool).
Enumerate	Generates a numeric list.
Grid	Produces a set of points on a grid. You can construct rectangles, ellipses, lines, crosshairs, and bricks. You can specify the size of the object and the number of points in the grid.
Isolate	Shrinks the connections elements of a Field so that they can be individually seen.
Isosurface	Creates surfaces or lines of constant value. For volumetric inputs, it creates isosurfaces; for surface inputs, contour lines. Use the ClipPlane module to display the interior of an isosurface.

MapToPlane	Maps a 3-dimensional data Field onto an arbitrary plane. You can specify a point and a normal to define the plane.
Regrid	Maps a set of scattered points onto a specified grid.
RubberSheet	Deforms a surface field (composed of triangles, quads, or lines) by an amount proportional to the data value at each point. You can use the Normals or FaceNormals module to add shading before rendering.
Sample	Produces a set of points in an arbitrary field. For example, you can produce a set of approximately evenly spaced samples on an isosurface.
ShowBoundary	Creates a renderable object that is the boundary of a volumetric field.
ShowBox	Creates a set of renderable lines that represent the bounding box of a field.
ShowConnections	Creates a set of renderable lines that represent the connections of elements in a field.
ShowPositions	Creates a set of renderable points that represent the positions of a field.
Streakline	Computes a line that traces the path of a particle through a changing vector field. The data input to the module is a vector field series, or a single field that is a member of a vector field series. You can control the starting points of the streaklines, and can optionally provide a curl field to produce twist on any ribbons or tubes constructed from the streakline.
Streamline	Computes a line that traces the path of a particle through a constant vector field. The data input to the module is a vector field. You can control the starting points of the streamlines, and can optionally provide a curl field to produce twist on any ribbons or tubes constructed from the streamline.

Rendering

These tools create a displayable image from a renderable object or modify the visual characteristics of the object(s) being displayed. For example, Normals and FaceNormals could be used to add shading to a geometrical structure created with RubberSheet (Realization category), while Light and AmbientLight could be used to change its default lighting.

AmbientLight	Produces an ambient light. You can specify the color of the light. Ambient lights illuminate a surface equally, regardless of direction, so an object illuminated only with ambient light is not shaded. Note that it is not necessary to specify an ambient light, because a small amount of ambient light is built into the object model. However, explicitly specified lighting overrides the default lighting.
Arrange	Creates a single image from a collection of images. You can specify how many images to put in a row.

AutoCamera	This module produces an appropriate camera as input to the Render or Display module. Specifying a "look-from" direction changes this default camera. The width of the image in model units (orthographic projection) or the field of view (perspective projection), or the image size and aspect ratio can also be changed.
Camera	Produces a camera for input to the Render or Display module. You can specify the "look-to" point, the "look-from" point, the width of the image in units corresponding to those in the data, the image size, and the aspect ratio. Camera differs from AutoCamera in that it specifies a "look from" a *point* instead of a *direction*.
ClipBox	Clips an object using a box. You can specify the box explicitly, as the two corners defining a box, or it can be the bounding box of an object. The module renders only that part of the object inside the box. The Render, Display, or Image tool actually performs the clipping.
ClipPlane	Clips an object by a plane. The user specifies the plane with a point and a normal. The side of the plane the normal projects into is the side that will be clipped (i.e., not displayed). The Render, Display, or Image tool performs the clipping.
Display	Renders and/or displays an image to the screen. If a camera is not provided, the Display module expects the first input to be an image (e.g., the output of Render or Arrange).
FaceNormals	Computes normals on a surface. When you use the FaceNormals module, each face of the surface is flat-shaded.
Image	This tool renders and displays an image to the screen. It performs like the AutoCamera and Display modules combined. Using the Image tool to render an image enables many direct interactors that are not available when using AutoCamera and Display. These options are available in the **`View Control`** option of the **`Options`** pull-down menu in the Image window. For example, see "Controlling the Image: View Control..." on page 74 in *IBM Visualization Data Explorer User's Guide*. The Image tool is available only in the graphical user interface.
Light	Produces a distant point light. You can specify the direction and color of the light. Note that it is not necessary to specify a light, because there is a default light built into the object model; however, explicitly specified lighting overrides the default lighting.
Normals	Computes point or face normals for shading a surface. For example, you can use this module to produce shading on rubbersheets and boundaries. However, shading on an isosurface is smoother if you use the gradient shading options built into the Isosurface module.
Overlay	Overlays two images. The result is a new image that can be displayed using the Display module. The new image is a pixel-by-pixel sum of the two images, where 1 – **`blend`**

	attenuates the base-image pixels, and `blend` attenuates the overlay-image pixels, (`blend` is a value between 0 and 1). You can also perform chromakeying by specifying `blend` as an RGB color or as a string specifying a color.
Render	Renders an object and creates an image. The object can be any combination of volumes, surfaces, lines, and points, and you can clip the object. You need to provide a camera to the Render module to specify the viewing direction.
Reorient	Rotates or inverts an image or a group of images.
Rotate	Rotates an object around specified axes. The Render, Display, or Image tool actually performs the rotation.
Scale	Changes an object's dimensions along the *x*, *y*, and *z* axes. The scaling is actually performed in the Render, Display, or Image tool.
ScaleScreen	Scales all screen objects (typically captions and color bars) by a specified amount.
Shade	Allows you to specify object-shading parameters such as specularity.
Transform	Moves, rotates, and resizes an object. The Render, Display, or Image tool actually performs the translation.
Translate	Moves an object along the *x, y*, and *z* axes. The Render, Display, or Image tool actually performs the translation.
UpdateCamera	Makes specified alterations to the input camera.

Special

Tools in this category can be used in visual programs for a variety of purposes.

Note: For details, see the sections of the *IBM Visualization Data Explorer User's Guide* referred to in the following list.

Colormap	Presents an interactor tool for creating color maps.
Input	Defines an input to a macro. See 7.2, "Creating and Using Macros" on page 149 for information.
Output	Defines an output of a macro. See 7.2, "Creating and Using Macros" on page 149 for information.
Pick	Allows the user to pick objects in a scene using the mouse. See "Using Pick" on page 87 for information.
Probe	Allows the user to pick an xyz point in a scene using the mouse. See "Using Probes (Cursors)" on page 85 for information.
ProbeList	Allows the user to pick multiple xyz points in a scene using the mouse. See "Using Probes (Cursors)" on page 85 for information.
Receiver	Receives the output of a transmitter for "wireless" connections. See "Using Transmitters and Receivers" on page 106 for information.

Sequencer	Allows animation of a visual program. See "Using the Sequencer" on page 68 in *IBM Visualization Data Explorer User's Guide* for information.
Transmitter	Outputs an Object for "wireless" connections. See "Using Transmitters and Receivers" on page 106 for information.

Structuring

These tools manipulate Data Explorer data structures. Their functions include the creation of hierarchies, selection of elements in a hierarchy, allowing operations on components other than "data," manipulation of Field or Group components, and determining which branches of a visual program are to be executed.

Append	Adds Objects as members to an existing Group.
Attribute	Extracts an attribute from an Object.
ChangeGroupMember	Inserts, renames, or deletes a member of an existing Group.
ChangeGroupType	Changes the type of a Group.
Collect	Collects Objects into a Group. For example, you can use Collect to collect a streamline, an isosurface, and a light. You can then pass the collection to the Image tool.
CollectMultiGrid	Collects Objects into a Multigrid (a Group that will be treated as a single data Object).
CollectNamed	Collects Objects into a Group (like Collect) but allows each Object in the Group to be given a name.
CollectSeries	Collects Objects into a Series. You give each element of the series a series position (e.g., a time tag).
CopyContainer	Copies the top container Object.
Extract	Extracts a component from a Field (e.g., the "colors" component).
Inquire	Returns information about the input Object (e.g., data type, number of elements, etc.).
List	Concatenates several items into a single list.
Mark	Marks a specified component in a Field as the data component. Many modules operate only on the "data" component. Thus the Mark module allows modules to operate on components other than data (e.g., "positions" or "colors"). If a "data" component already exists, it is preserved in a "saved data" component. You can restore it by using the Unmark module.
Options	Associates attributes with an Object. For example, plotting options such as marker type can be specified by adding attributes to the line.
Remove	Removes a specified component from a Field.
Rename	Renames a component in a Field. For example, you could rename the "colors" component to "front colors" to get only front-facing colors. You should be aware that component

	names have special meanings to modules using them. See Chapter 3, “Understanding the Data Model” on page 15 in *IBM Visualization Data Explorer User’s Guide*.
Replace	Replaces a component in one Field with a component from another, or with an input Array.
Select	Selects members out of a Group or elements from a list. two isosurfaces, a streamline, and a mapped plane. You could then use the `selector` input to the module to choose which of these is the output of the module.
Unmark	Undoes the action of the Mark module. It moves the “data” component back to the component you specify, and the “saved data” component is restored to the “data” component.

Transformation

These tools generally modify or add to components of the input Field without changing its underlying positions and connections. For example, AutoColor creates a “colors” component based on the data values of a Field and Compute performs a mathematical operation on the “data” component of a Field.

AutoColor	Automatically colors a data Field. By default, the minimum data value is blue, the maximum data value is red, and data values between are blue to red through cyan, green, and yellow. You can control what subset of the data range to color and what portion of the color wheel to use. You can also control the opacity of the object and the intensity of the colors. AutoColor also chooses appropriate colors and opacities for volume rendering.
AutoGrayScale	Automatically colors a data field using a gray scale.
Categorize	Categorizes data, creating an integer data component along with a lookup table into which the integers reference.
CategoryStatistics	Performs various statistics such as minimum, maximum, number of items, etc., on categorical data.
Color	Allows you more control over the coloring of an object than is possible with AutoColor. You can specify a string representing a color (e.g., “spring green”), an RGB color, or a color map (from the Colormap Editor, for example) to be applied to a data Field. You can also use the Color module to make objects (e.g., isosurfaces) translucent.
Compute	Performs point-by-point arithmetic on a Field or Fields. For example, it can add the “data” component of one Field to the tangent of the “data” component of another Field. You can also use the Compute module to select components from a vector data Field; construct a vector “data” component from a set of scalar input data Fields; or perform conditional operations.
Compute2	Differs from Compute in allowing the expression to be passed in by means of an input tab (e.g., from a Selector interactor).

Convert	Converts between hue, saturation, and value color space and red, green, and blue color space. You can convert either a single vector, a list of vectors, or a color map.
DFT	Performs a discrete Fourier transformation on a 2- or 3-dimensional field.
Direction	Converts azimuth, elevation, and distance to Cartesian coordinates (x, y, and z), which are useful for specifying camera viewing directions and for specifying ClipPlane or MapToPlane normal directions.
DivCurl	Computes the divergence and the curl of a vector field.
Equalize	Applies histogram equalization to a field.
FFT	Performs a fast Fourier transformation on a 2- or 3-dimensional field.
Filter	Filters a Field. You can specify a name describing a filter (e.g., “laplacian”) or a filter matrix explicitly.
Gradient	Computes the gradient of a scalar Field.
Histogram	Computes a histogram and the median of an input data Field. You can then use Plot to visualize the result as a 2-dimensional graph.
Lookup	Uses one object to look up the value of another object in a field.
Map	Maps Fields onto one another. For example, you can map a data Field onto an isosurface or onto a tube that has been formed from a streamline.
Measure	Performs measurements (e.g., surface area or volume) on a data Object.
Morph	Applies a binary morphological operator, such as erode or dilate.
Post	Changes the dependency of data (and other components) between positions and connections.
QuantizeImage	Converts an image from a format containing (potentially) as many colors as pixels to a quantized image with a color map with a user-specified number of colors between 8 and 256.
Sort	Sorts a list or Field based on values in the data component.
Statistics	Computes the statistics of a Field: the mean, standard deviation, variance, minimum, and maximum. You can use these statistics as input for other modules, or print them using the Echo or Print modules.

Windows

These tools create or supervise image windows.

ReadImageWindow	reads back an image (pixels) from an Image or Display window, for both hardware and software rendered images.
SuperviseState	watches for and acts on mouse and keyboard events in a Display window, based on user-defined callbacks.

SuperviseWindow	creates a window which will be monitored by SuperviseState.

Chapter 2. Functional Modules

The entries that follow describe the Data Explorer functional modules. They are listed in alphabetical order by module name and contain the following information:

- Module name and short description (see Note 1)
- Syntax
- Inputs and default values
- Outputs
- Details of module function
- Components (see Note 2)
- Script language example(s) (in some cases)
- Example program(s) (see Note 3)
- Related modules or information or both.

Notes:

1. The name of the tool category to which a module belongs is printed opposite the module name (i.e., the name listed in the "Categories" palette of the Visual Program Editor window). The absence of a category name signifies that the module is not available in the graphical user interface.
2. This subsection is omitted if the module does not produce a Field as output.
3. These programs are contained in the directory **/usr/lpp/dx/samples/programs** or in its subdirectory ...**programs/SIMPLE**.

AmbientLight

Category

Rendering

Function

Produces an ambient light.

Syntax

```
light = AmbientLight(color);
```

Inputs

Name	Type	Default	Description
color	vector or string	[0.2 0.2 0.2]	color and intensity of light

Outputs

Name	Type	Description
light	light	the ambient light

Functional Details

The AmbientLight module produces a light that equally illuminates all surfaces, regardless of location or direction.

color specifies the color of the light either as an RGB vector or as a string. If it is a string, it must be one of the defined color-name strings (see “Color” on page 75).

If no value is specified for this parameter, Data Explorer incorporates an ambient light of color [0.2 0.2 0.2] (low-intensity white light) to the scene. The system also uses a distant light of color [1.0 1.0 1.0] (high-intensity white light). (See “Light” on page 197.)

If a value is specified, the system removes the default lights. Therefore, if you use AmbientLight and want shading, you must also add a distant light. In addition, if you desire shading, you should use only relatively small amounts of ambient light (a value less than about 0.5). Use the Collect module to incorporate the resulting light into the scene that is given to the Render, Display, or Image tool.

Ambient lights have no effect on volume-rendered objects.

Example Visual Programs

```
ThunderGlyphSheet.net
UsingLights.net
SIMPLE/Light.net
```

See Also

Collect, Convert, Light

Append

Category

Structuring

Function

Adds one or more specified Objects to an existing Group.

Syntax

```
group = Append(input, object, id,...);
```

Inputs

Name	Type	Default	Description
input	Group	(no default)	group to which an object is to be added
object	Object	(no default)	Object to be added
id	scalar or string	(no default)	series position or name of Object
...	...	...	additional object-id pair(s)

Outputs

Name	Type	Description
group	group	the group with Objects added

Functional Details

This module differs from Collect, CollectNamed, CollectMultiGrid, and CollectSeries, which create a new Group with the specified Objects as members. For example, if the input to the module is a series with four members, the output will be a series with four + *n* members, where *n* is the number of objects specified.

input	must be a Group. The type of this Group determines the type of the output Group.
object	is an Object to be added as a member to the Group **input**.
id	specifies additional information to be associated with the appended Object. For a named Group, this could be the name of the member. For a Series, it is the series position value. If **input** is a series, then **id** is a required parameter. Otherwise, it is optional.

A single Append module can specify a maximum of 21 **object-id** pairs. In the user interface, the default number of enabled parameter tabs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Components

All components are propagated to the output.

See Also

Collect, CollectMultiGrid, CollectNamed, CollectSeries, Select

Arrange

Category

Rendering

Function

Arranges images for display.

Syntax

image = Arrange(**group**, **horizontal**, **compact**, **position**, **size**);

Inputs

Name	Type	Default	Description
group	image group	none	images to be displayed
horizontal	integer	infinity	number of images in horizontal dimension
compact	vector	[0 0]	makes the image compact in x or in y or in both
position	vector	[.5 .5]	position of each image in its frame
size	vector	[0 0]	force size of each frame to this number of pixels

Outputs

Name	Type	Description
image	image	resulting image

Functional Details

This module is useful for displaying a collection of images in an orderly arrangement.

group is a group of images (e.g., the output of Collect).

horizontal specifies the number of images in the horizontal direction. If there are more images in **group** than **horizontal**, they are arranged in rows below the first, each of length **horizontal**.

The **compact**, **position**, and **size** parameters are useful when the images are of different sizes. In constructing an output image, Arrange creates in effect a regular arrangement of output blocks, in rows and columns, one original image being placed in each block. The size of the blocks and the placement of the original images in them is controlled by these three parameters:

compact specifies how the resulting image is to be "compacted."

If the first (width) component:

= 0, the width of each column is set to the width of the widest image in the input group.

= 1, the width of each column is set to the width of the widest image in that column.

If the second (height) component:

= 0, the height of each row is set to the height of the tallest image in the input group.
= 1, the height of each row is set to the height of the tallest image in that row.

Thus a setting of [0, 0] will place each image in a box of the same size, as determined by (1) the widest and tallest images present in the input group or (2) by the value of **`size`**, if it is nonzero.

`position` specifies the placement of an image in an output block if the block is larger than the image. A value of [0 0] would place it in the lower left corner; [.5 .5], in the center; [1 0], in the lower right corner; and so on.

`size` specifies, in pixels, the width or height of the block containing each input image in the resulting image. This parameter overrides the setting of the corresponding component of **`compact`**.

If **`size`** is less than the size of the *largest* image, it will default to the size of the largest. That is, **`size`** cannot shrink or crop an image. If a component of **`size`** is set to zero, the dimensions of the images in that row or column are used to set the size of the output image blocks, depending on the setting of **`compact`**.

If you want to filter or reduce an image, you must do so before arranging it together with another image.

Components

All input components are propagated to the output.

Example Visual Programs

```
PlotLine.net
UsingCompute.net
UsingFilter.net
SIMPLE/Arrange.net
IndependentlyArrange.net
```

IndependentlyArrange.net illustrates an interactive alternative to using Arrange.

See Also

Collect, Display, Overlay, Render

Attribute

Category

Structuring

Function

Extracts an attribute from an Object.

Syntax

```
object = Attribute(input, attribute);
```

Inputs

Name	Type	Default	Description
`input`	object	none	object from which the attribute is obtained
`attribute`	string	"name"	attribute name

Outputs

Name	Type	Description
`object`	object	attribute value

Functional Details

`input`	is an Object with an attribute to be extracted.
`attribute`	is the name of the attribute to be extracted.

There are various attributes which have definite meanings in Data Explorer (see "Options" on page 224). You can also add your own attributes to objects. In addition, some modules (RubberSheet, Isosurface, AutoGrid, Tube, Ribbon) add attributes specifying the value of a particular value computed and used in creating a realization.

Example Visual Program

`UsingAttributes.net`

See Also

Options, RubberSheet, Isosurface, AutoGrid, Tube, Ribbon

AutoAxes

Category

Annotation

Function

Generates an axes box to enclose an object in the Image window.

Syntax

```
axes = AutoAxes(input, camera, labels, ticks, corners, frame, adjust,
                cursor, grid, colors, annotation, labelscale, font,
                xticklocations, yticklocations, zticklocations,
                xticklabels, yticklabels, zticklabels);
```

Inputs

Name	Type	Default	Description
input	object	none	object to be enclosed
camera	camera	none	viewpoint
labels	string list	no labels	labels for axes
ticks	integer or integer list	15	the approximate number of major tick marks (0 to suppress)
corners	vector list or object	input object	bounds of axes
frame	flag	0	0: only the frame of back faces drawn 1: entire frame drawn
adjust	flag	1	0: end points not adjusted 1: end points adjusted to match tick marks
cursor	vector	no cursor	cursor position
grid	flag	0	0: grid lines not drawn 1: grid lines drawn
colors	vector list or string list	appropriate color(s)	color(s) for annotation
annotation	string list	"all"	annotation objects to be colored
labelscale	scalar	1.0	scale factor for labels
font	string	standard	font for labels
xticklocations	scalar list	appropriate	set of x tick locations
yticklocations	scalar list	appropriate	set of y tick locations
zticklocations	scalar list	appropriate	set of z tick locations
xticklabels	string list	xticklocations	x tick labels
yticklabels	string list	yticklocations	y tick labels
zticklabels	string list	zticklocations	z tick labels

Outputs

Name	Type	Description
axes	color field	the axes box plus the input

Functional Details

If you are using the Image tool in the user interface, you should use the **AutoAxes** option of the **Options** pull-down menu in the Image window, rather than the AutoAxes module.

input	is the object around which an axes box is to be generated.
camera	is the camera used to render the object (using Display or Render). The viewpoint determines which of the box edges AutoAxes creates and which way the labels face.
labels	specifies labels for the *x*, *y*, and *z* axes. If **input** has an "axis labels" attribute, it is used as the default for **labels**. This attribute can be imported with the original data or it can be added using the Options module.
ticks	can be an integer list, in which case it represents the approximate number of tick marks to be used on each of the three axes. If you specify **ticks** as a single integer, AutoAxes uses approximately that many tick marks in total for all three axes.
corners	specifies the extent of the axes. If this parameter is not specified, the module automatically sizes the box to enclose the specified input object. If it is specified as a list of two vectors, the module uses the points specified rather than the bounds of **input** to determine the extent of the axes. In that case, **corners** specifies the end points of the diagonal of the axes (e.g., [*X*min, *Y*min, *Z*min] and [*X*max, *Y*max, *Z*max]). Alternatively, you can specify **corners** as an object, and the module uses the bounds of that object to determine the size of the axes. This feature is useful, for example, to enclose varying isosurfaces based on a given data field. You can use the data field as the **corners** parameter to keep the axes box a constant size, even when the isosurface changes in size.
frame	specifies the kind of frame to be placed around the object. By default (**frame** = 0), the three back faces of the box (from the viewpoint of **camera**) are opaque, serving as background. If **frame** = 1, the module draws the outlines of the other three faces of the box as well.
adjust	adjust the end points of the axes to end exactly at the tick marks. The default is no adjustment.
cursor	specifies the position for the placement of a cursor.
grid	specifies grid lines drawn from the major tick marks.
colors	specifies the color(s) of one or more components of the axes. It can be a single color (RGB vector or name string) or a list of colors. Color names can be any of the defined color-name strings (see "Color" on page 75). Its effects are partly determined by the **annotation** parameter.

If **color** is a single string and **annotation** is unspecified or is “all,” then the specified color is used for all axes annotation. Otherwise, the number of colors in **color**s must match the number of **annotation** strings exactly and in a one-to-one correspondence.

annotation combines with **color** to specify the color of one or more components of the axes. This parameter can be a single string or a set of strings from the following list: “all,” “background,” “grid,” “labels,” and “ticks.” You may remove the background by specifying “background” and setting **color** to “clear.”

labelscale determines the size of the axes and tick-mark labels. For example, **labelscale** = 2.0 will display the labels at double their default size.

font specifies the font for axes and tick-mark labels. The default for axes is a variable-width font (“variable”), and for tick-marks a fixed-width font (“fixed”).

```
area          gothicit_t    pitman          roman_ext
cyril_d       greek_d       roman_d         script_d
fixed         greek_s       roman_dser      script_s
gothiceng_t   italic_d      roman_s         variable
gothicger_t   italic_t      roman_tser
```

For more information, see Appendix E, “Data Explorer Fonts” on page 307 in the *IBM Visualization Data Explorer User’s Guide*.

xticklocations
list of explicit locations for tick marks on the x-axis. If specified, overrides the value as determined by the **ticks** parameter.

yticklocations
list of explicit locations for tick marks on the y-axis. If specified, overrides the value as determined by the **ticks** parameter.

xticklocations
list of explicit locations for tick marks on the z-axis. If specified, overrides the value as determined by the **ticks** parameter.

xticklabels list of labels to be associated with **xticklocations**. If **xticklabels** is specified, and **xticklocations** is not specified, then **xticklocations** defaults to the integers 0 to n-1 where n is the number of items in **xticklabels**.

yticklabels list of labels to be associated with **yticklocations**. If **yticklabels** is specified, and **yticklocations** is not specified, then **yticklocations** defaults to the integers 0 to n-1 where n is the number of items in **yticklabels**.

zticklabels list of labels to be associated with **zticklocations**. If **zticklabels** is specified, and **zticklocations** is not specified, then **zticklocations** defaults to the integers 0 to n-1 where n is the number of items in **zticklabels**.

Notes:

1. If you have applied scaling to your object using the Scale module, AutoAxes disregards that scaling when labeling the axes. This allows you to display data conveniently with an aspect ratio much different from 1, and still have correct labels on the axes. AutoAxes only disregards the top-level transformation and does not disregard the transformations of the Translate or Rotate module. (The macro AutoScaleMacro in `/usr/lpp/dx/samples/macros` can be used just before Image or AutoAxes to scale the object. See also How to Avoid Stretching or Squashing Glyphs on page 41.) Because scaling is disregarded if not done at the top level, you cannot have a Collect module following the Scale module. For example, if you want to Collect a caption with an object, you should use Scale *after* the Collect of the object with the caption.
2. If you are setting the **`corners`** parameter as a vector list, and the input **`object`** is scaled, **`corners`** should be in units of the scaled object.
3. If **`corners`** is more restrictive than the given **`ticklocations`**, then the given locations outside the corners are not shown.
4. If **`corners`** is less restrictive than the given **`ticklocations`**, or if **`corners`** is not specified, then all given tick locations are shown, whether or not there is data there.
5. If **`ticklocations`** is specified, then the data range determines the extent of the axes, unless **`corners`** is specified, in which case the given corners are used.
6. You can change the point at which a change is made from fixed-format to scientific notation for tick labels by setting the `DXAXESWIDTH` environment variable (see "Other Environment Variables" on page 292 in *IBM Visualization Data Explorer User's Guide*).

Example Visual Programs

```
AutoAxesSpecifyTicks (as part of the Image tool)
GeneralImport1.net (as part of the Image tool)
SIMPLE/AutoAxes.net
```

See Also

Collect, Color, Options, Plot

AutoCamera

Category

Rendering

Function

Constructs a camera for viewing an object.

Syntax

```
camera = AutoCamera(object, direction, width, resolution,
                    aspect, up, perspective, angle, background);
```

Inputs

Name	Type	Default	Description
object	object or vector	none	object to be looked at
direction	vector, string, or object	"front"	position of camera
width	scalar or object	input dependent	width of field of view
resolution	integer	640	pixels across image
aspect	scalar	0.75	height/width
up	vector	[0 1 0]	up direction
perspective	flag	0	0: orthographic projection 1: perspective projection
angle	scalar	30.0	view angle (in degrees) (for perspective projection)
background	vector or string	"black"	image background color

Outputs

Name	Type	Description
camera	camera	resulting camera

Functional Details

This module differs from Camera in specifying a *direction* from which to view a specified object. (Camera specifies a “look-from” *point*.) It will automatically create an appropriate Camera for a given object for use with Display or Render.

object is the object for which a camera is to be created. You can specify this parameter as a 3-dimensional position in space. In that case, **width** must also be explicitly specified (see below) as a numeric value, since the module has no means of estimating object size.

direction specifies the direction from the camera eye to the center of the object and can be any of the following strings: “front,” “back,” “left,” “right,” “top,” “bottom,” and their corresponding “off” values ("off-front," "off-back," etc., which are all slightly offset from the direct positions). Hyphens and spaces are not required for the offset (i.e., "offleft" and "off left" are both valid, and Data Explorer ignores capitalization.

The direction "front" means from the positive *z* direction; "back" means from the negative *z* direction; "left" means from the negative *x* direction, "top" means from the positive *y* direction, and so on.

Note: This parameter controls only the direction of viewing; it does not allow you inside the object. For interior views, use the ClipPlane or ClipBox module, make the object translucent, or use the Camera module in perspective mode.

This parameter can also be specified as a vector to be added to the look-to point. You can use the Direction module for constructing look-from directions. But note that only the direction of the parameter is important; its magnitude does not affect the size of the object in the image.

`width`	specifies the width of the image in the units of **`object`**.
`resolution`	specifies the width of the image in pixels.
`aspect`	specifies the height-to-width ratio of the image.
`up`	specifies a vector that will be aligned with the vertical axis of the image. The default is [0 1 0].

`perspective` specifies the method of projection used in rendering **`object`**. The choices are perspective (0) and orthographic (1).

Perspective projection
: This method produces a realistic rendering of objects, but does not preserve the exact shape and measurements of the object (e.g., parallel lines usually do not project as being parallel). The camera is positioned at the vertex of the viewing angle (see **`angle`** below). The base of this angle is determined by the **`width`** parameter. Thus the actual camera position is determined by **`width`** and **`angle`** along the **`direction`** vector.

Orthographic projection
: This method produces a somewhat artificial view of an object (the distance between the front and back of an object appears small compared to the distance between the object and the camera), but it preserves exact scale measurements and parallel lines.

: The **`angle`** parameter has no effect on orthographic projection. The size of the object can be changed only by **`width`** (the default is a field of view slightly greater than the width of the object).

For more information on these projection methods, consult a text on computer graphics.

`angle` specifies the viewing angle in degrees for perspective projection.

`background`	specifies the color of the image background as either an RGB color or a color-name string. The string can be any of the defined color-name strings (see "Color" on page 75).

Example Visual Programs

```
MovingCamera.net
PlotLine.net
PlotTwoLines.net
UsingCompute.net
UsingMorph.net
```

See Also

Camera, Color, Direction, Render, Display

AutoColor

Category

Transformation

Function

Automatically colors a field.

Syntax

```
mapped, colormap = AutoColor(data, opacity, intensity, start, range,
                             saturation, min, max, delayed, outofrange);
```

Inputs

Name	Type	Default	Description
`data`	data field	none	field to be colored
`opacity`	scalar	input dependent	opacity, between 0 and 1
`intensity`	scalar	1.0	color intensity
`start`	scalar	0.6666	starting color (default = blue)
`range`	scalar	0.6666	range of color (default = blue to red)
`saturation`	scalar	1	saturation, between 0 and 1
`min`	scalar or data field	min of data	minimum of data to be colored
`max`	scalar or data field	max of data	maximum of data to be colored
`delayed`	flag	0	0: apply maps 1: delay applying color and opacity maps (byte data only)
`outofrange`	vector list or string list	[.3 .3 .3]	how to color out-of-range points

Outputs

Name	Type	Description
`mapped`	color field	color-mapped input field
`colormap`	field	RGB color map used

Functional Details

This module colors a specified field (`data`) by mapping hues to data values.

`data` is an input field with data. If the input is a vector field, the colors are based on the magnitude of the data. If the input consists of matrices, the colors are based on the determinants.

`opacity` specifies the opacity of the resulting object. Allowed values range from 0 to 1. Its default value is 1 for surfaces and 0.5 for volumes.

intensity scales the amount of color. For opaque surfaces, the parameter scales from black (0) to full color (1). For volumes, it controls the brightness of the object when viewed along its longest dimension. Values greater than 1 can be used to brighten translucent surfaces or volumes that appear too dim. See also "Coloring Objects for Volume Rendering" on page 113.

start and **start – range**
specify the colors applied to the minimum and maximum data values mapped. By default, the minimum data value is colored blue (0.6666), and the maximum is colored red (0 or 1; colors are defined cyclically from –∞ to ∞, so that a **start** of –1 = 0 =1, and so on).

saturation specifies the saturation of the colors used. This value must be between 0 and 1.

min and **max** specify the minimum and maximum data values mapped. If neither is specified, the minimum and maximum values of **data** are mapped. If **min** is scalar, the minimum data value is mapped to that value. If **min** is a data field, the minimum data value of that field is used.

The **max** parameter is interpreted in corresponding fashion. If **min** is a data field and **max** is unspecified, the module uses the minimum and maximum values of that field.

For volumes, regions with values outside the **min-max** range are invisible; for surfaces, such regions are gray by default.

delayed determines whether "delayed colors" are created. Such colors are available only for byte data and they use less memory.

When **delayed** = 1:

- The "colors" component is a pointer to the "data" component, and a "color map" component is created. (This component is a color lookup table with 256 entries, representing the appropriate color for each of the 256 possible data values.)
- If **opacity** is also specified, an opacity map is created with 256 entries, while the "opacities" component is a copy of the "data" component.
- The module adds a "direct color map" attribute to the output object. (See "Using Direct Color Maps" on page 111.)

outofrange specifies the coloring of data that fall outside the **min-max** range. This parameter applies only to surfaces; out-of-range data values for volumes are always invisible. If the parameter value is a single color (RGB vector or color-name string), it is applied to both the upper and lower out-of-range points. If it is a list of two colors, then it is applied to the lower and upper out-of-range points, respectively. Color strings can be any of the defined color-name strings (see "Color" on page 75) or either of strings "color of min" and "color of max."

Notes:

1. AutoColor adds colors to the “colors” component. For “front colors” or “back colors,” use the Rename module following AutoColor. (See Appendix E, “Data Explorer Fonts” on page 307, in *IBM Visualization Data Explorer User’s Guide*.)
2. This module also outputs the RGB color map used, in the output **`colormap`**. The “positions” component contains the data values, and the “data” component contains the corresponding RGB colors. You can use this color map as an input to the ColorBar module. For byte data, the color map always contains 256 entries for the 256 possible data values. If the input to AutoColor is a group, then a different color map will be constructed for each member of the group. In that case, the **`colormap`** output of the module is a group of color maps. Use the Select module to select the color map you want to display using ColorBar.
3. If you AutoColor a group of volumes, you may find that the resulting image is black because the renderer does not support coincident volumes.

Components

Adds a “colors” component. An “opacities” component is added if **`opacity`** is less than 1 or if the input data is a volume. If **`delayed`** = 1, the “colors” component is a copy of the “data” component and a “color map” component is created. Similarly, an “opacity map” component is created if **`opacity`** is less than 1 or the input is a volume.

Example Visual Programs

Many example visual programs use AutoColor, including:

```
AlternateVisualizations.net
ContoursAndCaption.net
InvalidData.net
MappedIso.net
RubberTube.net
ThunderGlyphSheet.net
VolumeRendering.net
SIMPLE/Autocolor.net
```

See Also

Color, ColorBar, AutoGrayScale, Display

AutoGlyph

Category

Annotation

Function

Assigns an appropriate glyph to each point in a data field.

Syntax

```
glyphs = AutoGlyph(data, type, shape, scale, ratio, min, max);
```

Inputs

Name	Type	Default	Description
data	data field	none	set of points to which glyphs will be assigned
type	scalar, string, field, or group	input dependent	glyph type
shape	scalar	1.0	factor to describe shape of glyph (must be greater than 0)
scale	scalar	1.0	scale factor for size of glyphs (must be greater than 0)
ratio	scalar	0.05 or 0	ratio in size (scalars or vectors) between smallest and largest glyphs (must be greater than or equal to 0)
min	scalar or field	min of data value or 0	data value that gets minimum-size glyph
max	scalar or field	max of data	data value that gets maximum-size glyph

Outputs

Name	Type	Description
glyphs	color field	set of glyphs

Functional Details

This module creates a glyph, or representation (e.g., an arrow), for each data value in a data field **data**. For data dependent on positions, a glyph is placed at the corresponding position. For data dependent on connections, a glyph is placed at the center of the corresponding connection element. If there is no “data” component, a glyph is placed at each position.

The “base” size (i.e. size of a glyph for the average data value) is based on AutoGlyph's estimate of an appropriate glyph size, as modified by **scale**.

For “text” glyphs, a “data” component is required.

AutoGlyph differs from the Glyph module in the interpretation of the **scale** parameter. For AutoGlyph, you specify the size of the glyphs relative to the default-size glyph chosen by AutoGlyph. In contrast, when using Glyph, you may specify a scaling factor that is multiplied by the data value to obtain the size of the glyph in world units.

data is the data field for which glyphs are to be created.

type specifies the kind of glyph to be used.

By default, scalar fields are represented by filled circles or spheres, vector fields by arrows, tensor fields by groups of arrows, and string data by text. For 2-dimensional scalar fields, the glyphs are circles, and for 2-dimensional vector fields, the arrows are flat. For 3-dimensional fields, spheres or rounded arrows (rockets) are used.

The **type** parameter will override a default glyph. If **type** is a string, it must specify one of the following: "arrow," "arrow2D," "circle," "colored text," "cube," "diamond," "needle," "needle2D," "rocket," "rocket2D," "sphere," "speedy," "spiffy," "square," "standard," or "text." The value "standard" specifies the default glyph type and is a medium-quality glyph appropriate to the data; "spiffy," a higher-quality glyph; and "speedy," a lower-quality, but more quickly rendered glyph. Lower quality glyphs also consume less memory.

You may alternatively specify **type** as a scalar value between 0 and 1, where:

- 0 is the same as "speedy"
- 1 is the same as "spiffy"
- 0.5 is the same as "standard"

There are approximately 5 different quality glyphs for each glyph type. The one closest to the value specified is used.

Specifying "text" or "colored text" for **type** puts a text representation of the data value at each point. For text glyphs, the text is by default 15 pixels high, and the **shape**, **ratio**, **min**, and **max** parameters have no effect. The **scale** parameter can be used to increase or decrease the size of the text glyphs. For example, specifying **scale** = 2 makes the text 30 pixels high. The difference between "text" and "colored text" is that text is always white when "text" is specified, but will be the color of the input if "colored text" is specified. Note that after the AutoGlyph module, Color can be used to color text glyphs to a different color if desired. The font for the text glyphs can be specified by appending "font = *fontname*" to "text" or "colored text," where *fontname* is any of the defined fonts supplied with Data Explorer:

```
area          gothicit_t      pitman             roman_ext
cyril_d       greek_d         roman_d            script_d
fixed         greek_s         roman_dser         script_s
gothiceng_t   italic_d        roman_s            variable
gothicger_t   italic_t        roman_tser
```

For more information, see Appendix E, "Data Explorer Fonts" on page 307 in *IBM Visualization Data Explorer User's Guide*. In addition, you may supply user-defined fields and groups to the **type** parameter (see 40).

The remaining five parameters are interpreted somewhat differently, depending on whether the type of data involved are (1) scalar or (2) vector or tensor.

Scalar Data

shape	is ignored for scalar data
scale	sets the scale factor for the size of the glyphs relative to the default glyph.
ratio	controls the ratio between the smallest and the largest glyphs. The default value is 0.05.
min and **max**	are the data values used to set glyph sizes. Their default values are the minimum and maximum data values respectively. Unless you change **ratio** from its default value of 0.05, the glyphs representing data with a value of **min** are 5 percent the size of the glyphs representing data with a value of **max**. You can make the size of the glyphs directly proportional to the data value by setting both **min** and **ratio** to 0.

Data values smaller than **min** are colored pale pink, unless the original positions already had colors, in which case the colors are left unchanged. These data values are represented by glyphs of a size proportional to
min – *datavalue*.

Vector and Tensor Data

shape	is used to change the thickness of the glyph relative to its length. The default value for **shape** is 1 (e.g., to make a glyph twice as thick, set this parameter to twice the default value (i.e., to 2).
scale	sets the scale factor for the size of the glyphs relative to the default glyph.
ratio	determines which data values are displayed. To display only those values larger than 0.5 * **max**, set this parameter to 0.5. The default value is 0.0.
min and max	are the data values used to set glyph sizes.

For *3x3 symmetric tensor data* three arrows are drawn, representing the eigenvectors of the matrix. For *2x2 symmetric tensor data* two arrows are drawn, representing the eigenvectors of the matrix. For *vector data*, one arrow is drawn, representing the magnitude of the data. The default value of **min** is 0 (zero), and the length of the glyphs is always directly proportional to the magnitude of the data (or the magnitude of the eigenvector). If you set **min** to a value greater than zero, all data with a magnitude smaller than **min** are represented by a dot if **type** is “needle,” and by a small sphere otherwise. This can be useful to weed out small and perhaps noisy vectors. You can also set **ratio** to a value larger than zero; in that case, all data with a magnitude smaller than **ratio** × **max** is shown by a small sphere or dot. This allows you, for example, to set **ratio** = 0.5 to see only those vectors with a magnitude greater than half of **max**.

If you have vector data and want to show all the vectors as having the same length, you can use the **norm** function in the Compute module to normalize all the vectors to unit magnitude before passing them to AutoGlyph.

You can also use a field as the input for **min** and **max**, in which case the statistics of that field are used instead. If the input **data** is changing from frame to frame, and you want to keep the glyph sizes consistent from frame to frame, you can use the entire field or series as the input for **min** and **max**.

Note that if you set **min** and **max**, and there are data values far outside that range, then it is possible to get very large glyphs. This is because **max** is used to set the size scaling for the glyphs. Data values much larger in absolute value than **max** have proportionally larger-sized glyphs.

User-supplied and Annotation Glyphs

You may also pass your own glyph in as **type** as the object to place at each data point. The dimensionality of the positions of the glyph must be either 2-D or 3-D, and the connections type must be "triangles" or "lines." This glyph can be any Data Explorer field (e.g., an imported object, an isosurface, or a constructed object).

There are two ways to use your own glyphs: as user-supplied glyphs and as annotation glyphs.

User-supplied Glyphs

If you pass in a single field as **type**, that object is used as a glyph. It is drawn large or small depending on the data value, and it inherits the color of the data point, if present. The size of the object should be approximately unity for the default sizing of AutoGlyph to work well. Glyphs for scalar fields should be centered at the origin; glyphs for vector fields should have their base at the origin and the end that you want to point in the direction of the vector field at (0, 1, 0); that is, the glyph will be stretched along its y-dimension. You can use any combination of Scale, Rotate, and Translate to scale, orient, and position your template glyph before passing it to AutoGlyph.

Annotation Glyphs

If you pass in a group of objects as **type**, AutoGlyph interprets these as "annotation glyphs." Each object in the group must be a single field. It is assumed that you want data values equal to zero to be represented by the zeroth glyph in the group, data values equal to one to be represented by the first glyph in the group, and so on.

Thus the data component of **data** in this case must be of the type integer, unsigned integer, byte, unsigned byte, short, or unsigned short. The size of the glyph in the resulting output will be the size of the glyph in the glyph group, modified by the scaling factor **scale**. Colors and other components will be maintained from the *input glyphs* to the output object, rather than from the colors of **data**. The **shape**, **ratio**, **min**, and **max** parameters are ignored for annotation glyphs. Annotation glyphs should be positioned at the origin. You can use any combination of Scale, Rotate, and Translate to scale, orient, and position your template glyph before passing it to AutoGlyph.

Components

Creates new “positions” and “connections” components. In the case of a 3-D glyph, a “normals” component is added for shading purposes. All components that match the dependency of the “data” component are propagated to the output; all others are not. If the input has “binormals” and “tangent” components, they are not propagated to the output.

Example Visual Programs

```
AnnotationGlyphs.net              ThunderGlyphSheet.net
ConnectingScatteredPoints.net     UsingGlyphs.net
PickStreamline.net                UsingTextAndTextGlyphs.net
ProbeText.net                     SIMPLE/AutoGlyph.net
```

See Also

Glyph, Sample

How to Avoid Stretching or Squashing Glyphs

Users often use the Scale module to scale a collection of objects prior to rendering, if the axes have very different scales. This can cause a problem if the visualization includes glyphs, as the glyphs will be stretched or squashed as well. You can place the Scale module before Glyph or AutoGlyph, but if you want AutoAxes to show the original (rather than the scaled) position values, this will not work. One way of solving this problem lies in the fact that Glyph and AutoGlyph can accept a user-defined Glyph:

Give either module a “user-defined” glyph that is *inversely* squashed, so that when you use Scale, the glyph ends up with the correct shape. Proceed as follows:

- Create a field with a single point at the origin, using Construct ([0 0 0], data = 0) for scalar (sphere) glyphs
- Feed this field to either module to make a single sphere or arrow.
- Now scale the glyph, using a scale factor that is the inverse of the one you want to use on the entire data set. For example, if you are going to scale your data by [1 1 .001], then scale the single glyph by [.001 .001 1].
- Feed this scaled glyph to the second parameter of the module, which puts glyphs on all the data points. The module will use the squashed glyph as its template. After the template is scaled by [1 1 .001], the glyphs won’t be squashed anymore.
- A macro, UnsquishGlyphMacro.net, which performs this operation, can be found in `/usr/lpp/dx/samples/macros`. This macro (and this technique) works only for scalar data.

AutoGrayScale

Category

Transformation

Function

Automatically colors a field using a “gray” scale.

Syntax

```
mapped, colormap = AutoGrayScale(data, opacity, hue, start, range,
                                 saturation, min, max, delayed,
                                 outofrange);
```

Inputs

Name	Type	Default	Description
data	data field	none	field to be colored
opacity	scalar	input dependent	opacity (between 0 and 1)
hue	scalar	0.0	hue
start	scalar	0.0	starting intensity
range	scalar	1.0	range of intensity
saturation	scalar	0.0	saturation (between 0 and 1)
min	scalar or data field	min of data	minimum of data to be colored
max	scalar or data field	max of data	maximum of data to be colored
delayed	flag	0	0: apply maps 1: delay applying color and opacity maps (byte data only)
outofrange	vector list or string list	{“color of min, ” “color of max”}	how to color out-of-range data values

Outputs

Name	Type	Description
mapped	color field	color-mapped input field
colormap	field	RGB color map used

Functional Details

This module maps the *intensities* of a color (**hue**) to the data values of a specified field (**data**).

data is an input field with data. If the input is a vector field, the intensities are based on the magnitude of the data. If the input consists of matrices, the intensities are based on the determinants.

opacity specifies the opacity of the resulting object. Allowed values range from 0 to 1. Its default value is 1 for surfaces and 0.5 for volumes.

hue sets the hue. Blue is 0.6666, red is 0 or 1, and colors are defined cyclically from $-\infty$ to ∞ (i.e., **hue** of –1 = **hue** of 0 = **hue** of 1, and so on).

Note: This parameter will have no effect if **saturation** is set to 0 (see below).

start and **range**
specify the intensities applied to the minimum and maximum data values mapped. The value of **range** can be any positive number. By default, the minimum data value has an intensity of 0, and the maximum an intensity of 1. See also “Coloring Objects for Volume Rendering” on page 113.

saturation specifies the saturation of the colors used. This value must be between 0 (the default) and 1.

min and **max** specify the minimum and maximum data values mapped. If neither is specified, the minimum and maximum values of **data** are mapped. If **min** is scalar, the minimum data value is mapped to that value. If **min** is a data field, the minimum data value of that field is used.

The **max** parameter is interpreted in corresponding fashion. If **min** is a data field and **max** is unspecified, the module uses the minimum and maximum values of that field.

For volumes, regions with values outside the **min-max** range are invisible; for surfaces, such regions are gray by default.

delayed determines whether “delayed colors” are created. Such colors are available only for byte data and they use less memory.

When **delayed** = 1:

- The “colors” component is a pointer to the “data” component, and a “color map” component is created. (This component is a color lookup table with 256 entries, representing the appropriate color for each of the 256 possible data values.)
- If **opacity** is also specified, an opacity map is created with 256 entries, while the “opacities” component is a copy of the “data” component.
- The module adds a “direct color map” attribute to the output object. (See “Using Direct Color Maps” on page 111.)

outofrange specifies the coloring of data that fall outside the **min-max** range. This parameter applies only to surfaces; out-of-range data values for volumes are always invisible. If the parameter value is a single color (RGB vector or color-name string), it is applied to both the upper and lower out-of-range points. If it is a list of two colors, then it is applied to the lower and upper out-of-range points, respectively. Color strings can be any of the defined color-name strings (see “Color” on page 75) or either of the strings “color of min” and “color of max.”

Notes:

1. Directly displayed grayscale images will use more distinct colors if you take advantage of direct color maps by using the **`delayed`** parameter. Use Compute to convert your data to bytes if they are not already in that form.
2. AutoGrayScale adds colors to the "colors" component. For "front colors" or "back colors," use the Rename module following AutoGrayScale.
3. This module also outputs the RGB color map used, in the output **`colormap`**. The "positions" component contains the data values, and the "data" component contains the corresponding RGB colors. You can use this color map as an input to the ColorBar module. For byte data, the color map always contains 256 entries for the 256 possible data values. If the input to AutoGrayScale is a group, then a different color map will be constructed for each member of the group. In that case, the **`colormap`** output of the module is a group of color maps. Use the Select module to select the color map you want to display using ColorBar.
4. If you AutoGrayScale a group of volumes, you may find that the resulting image is black because the renderer does not support coincident volumes.

Components

Adds a "colors" component. An "opacities" component is added if **`opacity`** is less than 1 or the input data is a volume. If **`delayed`** = 1, the "colors" component is a copy of the "data" component, and a "color map" component is created. Likewise, an "opacity map" component is created if **`opacity`** is less than 1 or the input is a volume.

Example Visual Program

`AlternateVisualizations.net`

See Also

Color, AutoColor, ColorBar

AutoGrid

Category

Realization

Function

Maps scattered points onto a grid.

Syntax

```
output = AutoGrid(input, densityfactor, nearest,
                  radius,exponent,missing);
```

Inputs

Name	Type	Default	Description
input	field or vector list	none	field with positions to regrid
densityfactor	scalar or vector	1.0	densityfactor for grid
nearest	integer or string	1	number of nearest neighbors to use
radius	scalar	appropriate	radius from grid point to consider
exponent	scalar	1.0	weighting exponent
missing	value	no default	missing value to be inserted if necessary

Outputs

Name	Type	Description
output	field	regridded field

Functional Details

This module uses a specified set of scattered points (**input**) to assign data to every position of a grid. This module differs from Regrid in that you do not supply a grid; one is constructed automatically for you.

input should be either (1) a field with a 1-, 2-, or 3-dimensional "positions" component or (2) a list of 1-, 2-, or 3-dimensional vectors. In the second case, the vectors are interpreted as positions.

densityfactor specifies how to modify the automatically generated grid. By default the automatically generated grid will have approximately as many points as there are points in **input**, and each cell will be square. If **densityfactor** is a single scalar value, then there will be **densityfactor** times as many cells in each dimension. If **densityfactor** is a vector, then you can modify the number of cells in each dimension differentially.

nearest must be an integer or the string "infinity." An integer value specifies the number of nearest points (to each grid point) to be used in computing an average data value for that grid point.

radius	By default, only those points within an automatically computed radius will be considered when assigning data values to grid positions. This automatically computed radius is twice the largest dimension of a cell in the grid. You can explicitly set the radius to a value measured in the units of the "positions" component of **input**. If you specify **radius**="infinity", then no cutoff in distance will be used; all points will contribute to the result at each grid point. AutoGrid attaches a "AutoGrid radius" attribute to **output**, with a value equal to the radius used, or the string "infinity".
exponent	The averaging method is a weighted average. The expression for this average is 1/radius$^{(\mathbf{exponent})}$. The default value is 1.0, reducing the expression to the reciprocal of the radius.
missing	is used when **radius** is set to a value other than "infinity." The parameter specifies how to treat those grid points for which no points in **input** occur within the specified radius. If **missing** *is not set*, the module creates an "invalid positions" component, and grid points with no assigned data value are invalidated. If **missing** is set, the data value is inserted for the missing data values. It must match the data component of **input** in rank, type, and shape. All components that are position-dependent are treated in the same way as the "data" component.

Components

Adds a "connections" component. The "positions" and "connections" components are those of **grid** while all components in **input** that depend on "positions" will be present in the output, modified by averaging.

Example Visual Programs

```
SIMPLE/AutoGrid.net
ConnectingScatteredPoints
```

See Also

Connect, Regrid, Include

Band

Category

Realization

Function

Divides a specified field into bands.

Syntax

```
band = Band(data, value, number, remap);
```

Inputs

Name	Type	Default	Description
data	scalar field	none	field to be divided
value	scalar or scalar list	data mean	band divisions
number	integer	no default	number of divisions
remap	string	"low"	data values applied to bands

Outputs

Name	Type	Description
band	field or group	divided input

Functional Details

The values used for dividing a field into bands are specified by **value** or **number**.

data is the field to be divided and must have 2-dimensional connections (triangles or quads).

value specifies the value(s) at which the divisions are to be created (i.e., the data values in the data field). If this parameter and **number** (see below) are both unspecified, the module, by default, constructs the division at the arithmetic mean of the data set.

number specifies the number of equal divisions to be created between the minimum (min) and maximum (max) data values of the field. The first division is created at min + delta, the second at min + 2 * delta, and the last at max − delta, where delta = (max−min)/(number+1).

Note: This parameter is ignored if **value** is specified.

remap assigns data values to the elements in each band:

"low"	specifies that all connection elements are assigned the value of the lower boundary of the band (the lowest data value if it is the first band).
"high"	specifies that all connection elements are assigned the value of the upper boundary of the band (the largest data value if it is the last band).
"original"	specifies that the data values remain unchanged.

Components

This module creates new "positions" and "connections" components, and adds a "colors" component if one is not already present. The "data" component is modified according to the value of **`remap`**. The output data are connection dependent.

Example Visual Programs

```
AlternateVisualizations.net
BandedColors.net
InvalidData.net
SIMPLE/Band.net
```

See Also

Isosurface

Camera

Category

Rendering

Function

Constructs a camera for viewing an object.

Syntax

```
camera = Camera(to, from, width, resolution, aspect, up,
                perspective, angle, background);
```

Inputs

Name	Type	Default	Description
to	vector or object	[0 0 0]	look-to point
from	vector or object	[0 0 1]	position of camera
width	scalar or object	100	width of field of view (for orthographic projection)
resolution	integer	640	horizontal resolution of image (in pixels)
aspect	scalar	0.75	height/width
up	vector	[0 1 0]	up direction
perspective	flag	0	0: orthographic projection 1: perspective projection
angle	scalar	30.0	view angle (in degrees) (for perspective projection)
background	vector or string	"black"	image background color

Outputs

Name	Type	Description
camera	camera	resulting camera

Functional Details

This module differs from AutoCamera in that it specifies a "look-from" *point* for viewing an object (AutoCamera specifies a *direction* from which to view an object).

to specifies a point in the scene that appears at the center of the image; the default is the origin of world space. This parameter can also be specified as an object, in which case the center of the object serves as the look-to point.

Note: This parameter cannot have the same value as **from** (see below).

from specifies the location of the camera; the default is [0 0 1]. This parameter can also be specified as an object, in which case its value is the center of the object's bounding box.

Note: This parameter cannot have the same value as **to** (see above).

width functions only for an orthographic projection (see **perspective** below). It specifies the width of the field of view, in world coordinates. If this parameter is specified as an object, the module uses a value that is slightly larger than the diagonal of the object's bounding box.

resolution specifies the width of the image in pixels.

aspect specifies the height-to-width ratio of the image.

up specifies a vector that will be aligned with the vertical axis of the image.

perspective specifies the method of projection used in rendering **object**: perspective (0) or orthographic (1).

Perspective projection

> This method produces a realistic rendering of objects, but does not preserve their exact shape and measurements (e.g., parallel lines usually do not project as being parallel). The camera is positioned at the vertex of the viewing angle (see **angle** below). The two end-points of that angle are the left and right sides of the image area. Thus the wider the angle, the greater the amount of object space that can be fitted into the viewing area.
>
> **Note:** The **width** parameter has no effect on perspective projection.

Orthographic projection

> This method produces a somewhat artificial view of an object (the distance between the front and back of an object appears small compared to the distance between the object and the camera), but it preserves exact scale measurements and parallel lines. The size of the object can be changed only by **width** (the default is a field of view slightly greater than the width of the object).
>
> The **angle** parameter has no effect on orthographic projection.
>
> In orthographic projection, only the direction of the **from-to** vector is important, not its magnitude. The object's distance does not affect its size in the image, which can be changed only with the **width** parameter. In perspective projection, the object's size can be changed only by changing **from** or **angle** (because **width** is ignored).

For more information on these projection methods, consult a text on computer graphics.

angle specifies the viewing angle in degrees. The vertex of this angle is **to**.

background	specifies the color of the image background as either an RGB color or a color-name string. The string can be any of the defined color-name strings (see “Color” on page 75).

Example Visual Programs

```
FlyThrough.net (uses the macro InterpolatePositionsMacro.net)
FlyThrough2.net
```

See Also

AutoCamera, AutoColor, Direction, Render

Caption

Category

Annotation

Function

Displays a caption on the screen.

Syntax

```
caption = Caption(string, position, flag, reference,
                  alignment, height, font, direction, up);
```

Inputs

Name	Type	Default	Description
string	string or string list	none	the caption to be displayed
position	vector	[0.5 0.05]	where to display the caption
flag	flag	0	0: viewport-relative coordinates 1: pixel coordinates
reference	vector	same as **position**	reference point on caption
alignment	scalar	input dependent	range: 0.0: left-justify ... 1.0: right-justify
height	integer	15	caption height (pixels)
font	string	"variable"	caption font
direction	vector	[1 0]	direction of baseline
up	vector	perpendicular to baseline	direction of vertical strokes

Outputs

Name	Type	Description
caption	color field	string object that can be rendered

Functional Details

The caption produced is aligned in parallel with the screen.

string is the caption to be displayed. This parameter can also be a list of strings, for a multiline caption. Alternatively, you can separate substrings for a multiline caption by using \n.

position specifies the position of the caption in units determined by **flag** (see below).

flag	determines the type of coordinates used in placing the caption: 0: viewport-relative 1: pixel
reference	specifies the reference point for the caption that is to be placed at **position**: [0 0] = bottom left of the caption [1 1] = top right If **flag** = 0, the default is the same as the current specification of **position**. If **flag** = 1, the default is the lower left corner of the caption.
alignment	determines the alignment of a multiline caption from left justified (0.0) to right justified (1.0). For intermediate values, justification is defined by linear interpolation (e.g., a value of 0.5 centers the lines).
height	determines the height of the caption characters in pixels. A negative value generates an inverted caption.
font	specifies the font for a displayed caption. You can specify any of the defined fonts supplied with Data Explorer. These include a variable-width font ("variable," the default) and a fixed-width font ("fixed"):

```
area          gothicit_t    pitman        roman_ext
cyril_d       greek_d       roman_d       script_d
fixed         greek_s       roman_dser    script_s
gothiceng_t   italic_d      roman_s       variable
gothicger_t   italic_t      roman_tser
```

For more information, see Appendix E, "Data Explorer Fonts" on page 307 in the *IBM Visualization Data Explorer User's Guide*.

direction	determines the orientation of the caption (i.e., the direction of its baseline).
up	determines the direction of the vertical strokes of the caption font.

Notes:

1. To change the color of a caption, use the Color module (see "Color" on page 75).
2. If you are using Render or Display to render an object which contains a caption, when you change the resolution of the camera the size of the caption in pixels will not change. If you want the caption to take up the same proportion of the image, use the ScaleScreen module to change the size of the caption.

Components

Creates "positions," "connections," and "colors" components.

Example Visual Programs

Many example visual programs use Caption, including:

```
AlternateVisualizations.net
ContoursAndCaption.net
Sealevel.net
UsingAttributes.net
UsingIsosurface.net
SIMPLE/Caption
SIMPLE/ScaleScreen.net
```

See Also

Color, Format, Text, ScaleScreen, Legend

Categorize

Category

Transformation

Function

Categorizes components of a field

Syntax

```
output = Categorize(input, name);
```

Inputs

Name	Type	Default	Description
input	field	none	field to categorize
name	string or string list	“data”	component to categorize

Outputs

Name	Type	Description
output	field	with additional lookup components

Functional Details

input	is the field containing the components to categorize
name	is the name or names of the components to categorize

The Categorize module converts a component of any type to an integer array that references a newly created “lookup” component, which is a sorted list of the unique values in the original component. This serves to

1. reduce the size of a component that contains duplicate values,
2. allow conversion of string or vector data to “categorical” data,
3. detect repeated values in a component, and
4. create a sorted list of the unique values in a component for inspection.

Each component that is categorized will yield its own lookup component named “compname lookup”, where compname is the name of the categorized component.

For example, if the component name is “state” and its values are {“MO”, “CA”, “MO”, “NH”, “AK”, “NH”} then Categorize(field, “state”) would convert component state to: {2, 1, 2, 3, 0, 3} and produce a new component, “state lookup” containing the values {“AK”, “CA”, “MO”, “NH”}.

Notes:

1. Categorize works on scalar, string, or vectors of any type, with the lookup component sorted in order of x, y, z, ... If the lookup component has fewer items than the original component, then there are duplicate values in the original component. If the lookup component has 256 or fewer items, the categorized component will be of type unsigned byte; otherwise it will be of type int.
2. Categorical data can be converted back to its original values using either the Lookup module or Map. If the lookup component is of type string, it can be input as the **`labels`** parameter of Plot, ColorBar, or AutoAxes to label the values 0, 1, .. n-1 with the corresponding strings. This helps automate the labelling of categorical plots. Data imported by ImportSpreadsheet can be categorized on import directly by specifying the components to categorize. Statistics on the categorized component, and another associated component, can be found with CategoryStatistics. Include can be used to remove data by category.

Components

Modifies the components specified by **`name`**, replacing it by a list of indices. Adds a new component with the name "**`name`** lookup" which is a lookup table for component **`name`**.

Example Visual Programs

```
Duplicates.net
Categorical.net          (Categorize is called on import by ImportSpreadsheet)
```

See Also

CategoryStatistics, ImportSpreadsheet

CategoryStatistics

Category

Transformation

Function

Calculate statistics on data associated with a categorical component

Syntax

```
statistics = CategoryStatistics(input, operation, category, data, lookup);
```

Inputs

Name	Type	Default	Description
input	field	(none)	field for which to compute statistics
operation	string	“count”	operation to perform (“count”, “mean”, “sd”, “var”, “min”, “max”)
category	string	“data”	component with categorical values
data	string	“data”	data component for statistics
lookup	integer, string, value list	“category lookup”	lookup component

Outputs

Name	Type	Description
statistics	field	field with data containing the statistics and positions for the category values

Functional Details

input	field containing the categorical and data components
operation	calculation to perform
category	component with categorical values. This component must be an integer type (int, ubyte, ...)
data	data component for statistics. This component must be scalar.
lookup	lookup component (optional)

CategoryStatistics calculates statistics on a scalar component associated with a categorical component. If the operation is “count”, the **data** component is ignored and the number of counts in each category is calculated, corresponding to a histogram of the unique values in the categorized component.

For example, if **input** is a Field with component “state” containing the entries {1,0,1,2,3}, component “state lookup” containing the entries {“CA”, “NY”, “PA”, “VA”}, and a component “sales” containing the entries {1.2,1.0,1.4,1.7,1.8}, then CategoryStatistics(input,“mean”,“state”,“sales”) will produce an output field where the “positions” component will contain the indices {0,1,2,3} and the “data” component will contain the mean value for sales for each state, that is {1.0,1.3,1.7,1.8}.

The output of CategoryStatistics is a field with a "positions" component corresponding to the categorical indices, and a "data" component corresponding to the requested statistics. The "positions" component will consist of the integers 0 to N-1, where N can be determined in a number of ways:

- If no **`lookup`** component is specified, and if a "categoryname lookup" component is not found, (where "categoryname" is the string specified by **`category`**), then the output field will simply have positions from 0 to MAX_N, where MAX_N is the maximum integer found in the **`category`** component.
- If, on the other hand, a "categoryname lookup" component is found, or **`lookup`** is specified, then the number of category bins will be the number of items in **`lookup`**. **`lookup`** can also simply be an integer specifying the number of category bins.
- If a lookup table is provided, then for convenience, a "categoryname lookup" component will be placed in the output containing the values corresponding to the categorical indices.

Components

Creates an output field with a "positions" component representing the categorical indices, and a "data" component containing the requested statistics. Creates a "categoryname lookup" component if a lookup table is specified using the **`lookup`** parameter.

Example Visual Programs

```
Duplicates.net
Zipcodes.net
```

See Also

Categorize, Statistics, Lookup

ChangeGroupMember

Category

Structuring

Function

Insert, rename, or delete a member of an existing group

Syntax

```
changed = ChangeGroupMember(data, operation, id, newmember, newtag);
```

Inputs

Name	Type	Default	Description
data	group	none	input group object
operation	string	none	how to alter the group member
id	integer	none	index or name of existing group member
newmember	object	operation dependent	new or replacement group member
newtag	scalar or string	no default	new series position or member name

Outputs

Name	Type	Description
changed	group	group with one member changed

Functional Details

This module allows you to replace or insert a member of an existing group.

data is the group to be modified

operation is the operation to perform on the specified member of **data**. If **data** is a series group, then **operation** must be one of “insert before”, “replace”, “insert after”, or “delete”. If **data** is any other kind of group, then **operation** must be one of “insert”, “replace”, “delete”. Note that for most applications, the order of members in a group is not important, and the order of members in a group is not guaranteed to be maintained by modules in Data Explorer unless the group is a Series.

id is the index (from 0 to n-1, where n is the number of members of **data**) or name of an existing member of **data**

newmember is the new or replacement object to be placed in **data**

newtag is the series position or member name for **newmember**. If **data** is a series, this parameter is required.

Components

This module does not modify any components of the input **`data`**.

Example Visual Programs

`ManipulateGroups.net`

See Also

ChangeGroupType, Collect, CollectSeries, CollectNamed, CollectMultiGrid

ChangeGroupType

Category

Structuring

Function

Changes the group type

Syntax

```
changed = ChangeGroupType(data, newtype, idlist);
```

Inputs

Name	Type	Default	Description
data	group	(none)	input group object
newtype	string	(none)	type for output group
idlist	scalar list or string list	input dependent	series positions or member names

Outputs

Name	Type	Description
name	type	description
changed	group	different type of group with same members as input

Functional Details

ChangeGroupType allows you to change the group type of an input object, for example, change a generic group to a series, or change a series to a multigrid. The output group **changed** will contain the same members as the input group **data**. You can also use this module to change the series positions of a Series group.

data is the input group to be changed.

newtype is the type for the output group. It must be one of “series”, “multigrid”, or “generic”. Note that any combination of objects can be placed in a generic group, but there are restrictions on the objects which can be placed in a series or multigrid. In these cases, all the members must have the same data type and connection type.

idlist is the list of series positions or list of member names. The number of items in this list, if given, must be the same as the number of members in **data**. This parameter is optional. If you are changing the group type to series and do not provide this parameter, the members of the series will automatically be given series positions 0, 1, ..., n-1, where n is the number of members in **data**.

Components

This module does not change any components of the input object. It changes only the group type.

Example Visual Programs

`ManipulateGroups.net`

See Also

ChangeGroupMember, Collect, CollectSeries, CollectNamed, CollectMultiGrid

ClipBox

Category

Rendering

Function

Prepares a specified object for clipping by a box.

Syntax

```
clipped = ClipBox(object, corners);
```

Inputs

Name	Type	Default	Description
object	object	none	object to be clipped
corners	vector list or object	no clipping	corners specifying clipping box

Outputs

Name	Type	Description
clipped	object	object marked for clipping

Functional Details

This module constructs an object so that it can be clipped by a box. Data Explorer renders only the portion of the object that lies in this clipping box, which is defined by **corners** (see below).

object is the object to be clipped.

corners defines the clipping box in one of two ways:

by specifying two of its corners with a vector list (of two vectors). The module interprets the two vectors as opposite corners of the clipping box.
by using the bounding box of the specified object (when **corners** itself specifies an object).

Notes:

1. The specified object must be of a kind for which a bounding box can be constructed. Otherwise an error results. In general, bounding boxes can be constructed for all geometric objects, but not for objects such as captions.
2. All translucent objects in the scene should be clipped by the same object. In addition, objects can be clipped by only one clipping box or one clipping plane.
3. The effect of ClipBox occurs during rendering, and its use does not affect the behavior of modules upstream from the renderer. For example, ShowBox will draw a box around the *unclipped* object.

Components

All input components are propagated to the output.

See Also

ClipPlane

ClipPlane

Category

Rendering

Function

Prepares an object for clipping by a plane.

Syntax

```
clipped = ClipPlane(object, point, normal);
```

Inputs

Name	Type	Default	Description
object	object	none	object to be clipped
point	vector	center of object	a point on the clipping plane
normal	vector	[0 0 1]	perpendicular to the clipping plane

Outputs

Name	Type	Description
clipped	object	object marked for clipping

Functional Details

This module constructs an object so that it can be clipped by an infinite plane. Data Explorer renders only the portion of the object that lies on the side opposite that pointed to by **normal** (see below).

object	is the object to be clipped.
point,	specifies a point in the clipping plane. This specification, together with **normal**, defines the clipping plane. The default is the center of the object to be clipped.
normal	specifies a normal to the clipping plane. This normal projects to the side of the plane that is to be clipped. All parts of the specified object that lie on the opposite side are retained.

Notes:

1. The specified object must be of a kind for which a bounding box can be constructed. Otherwise an error results. In general, bounding boxes can be constructed for all geometric objects, but not for, say, captions.
2. All translucent objects in the scene should be clipped by the same object. In addition, objects can be clipped by only one clipping plane or one clipping box.
3. The effect of ClipPlane occurs during rendering, and its use does not affect the behavior of modules upstream from the renderer. For example, ShowBox will draw a box around the *unclipped* object.

Components

All input components are propagated to the output.

Example Visual Program

`UsingClipPlane.net`

See Also

ClipBox

Collect

Category

Structuring

Function

Collects objects into a group.

Syntax

```
group = Collect(object, ...);
```

Inputs

Name	Type	Default	Description
object	object	no default	object to be collected
...			more objects to be collected

Outputs

Name	Type	Description
group	group	the group of objects

Functional Details

This module creates a group from a set of input objects.

object is an object to be placed in a group.

Note that fields placed in the generic group created by the module retain their individuality. For example, AutoColor colors each field in the group based on its own minimum and maximum data values; and Sample samples each field individually. These examples contrast with the composite fields created by Partition and the multigrid field created by CollectMultigrid.

The Collect module adds objects to the group without assigning them names. Use the Select module to select an object from the group by number, where the first object is number 0.

If Collect is called without parameters—Collect()—then the output is an empty group.

A single module can collect a maximum of 21 objects. In the user interface, the default number of enabled tabs is two. However, tabs can be added and removed with the appropriate **Input/Output Tab** options in the **Edit** pull-down menu of the VPE.

Components

All input components are propagated to the output.

Example Visual Programs

Nearly every example visual program uses Collect.

See Also

Append, CollectMultiGrid, CollectNamed, CollectSeries, Select

CollectMultiGrid

Category

Structuring

Function

Collects objects together in a multigrid group.

Syntax

```
multigrid = CollectMultiGrid(object, name, object, name, ...);
```

Inputs

Name	Type	Default	Description
object	object	(no default)	object to be collected
name	string	(no default)	name of object
...			additional object-name pair(s)

Outputs

Name	Type	Description
multigrid	group	the output multigrid

Functional Details

The CollectMultiGrid module creates a Multigrid group from specified objects. The data and connection types of each Field in the Multigrid must be the same.

object is an object to be placed in the group.

name is an optional parameter for specifying the name of an object in the multigrid group. For example, the Select module could be used to select a member by name.

Data Explorer modules treat a Multigrid group as a single entity. The positions may be disjoint or overlapping. If they overlap, the “invalid positions” and “invalid connections” components can be used to identify the valid positions in a given region (see “Invalid Positions and Invalid Connections Components” on page 23 in *IBM Visualization Data Explorer User’s Guide*.)

A single CollectMultiGrid module can collect up to 21 objects together in a group. In the user interface, the default number of enabled parameter pairs (i.e., **object** and **name**) is two. Tabs can be added to the module icon and removed with the appropriate **Input/Output Tab** options in the **Edit** pull-down menu of the VPE.

Components

All components are propagated to the output.

See Also

Append, CollectMultiGrid, CollectNamed, CollectSeries, Select

CollectNamed

Category

Structuring

Function

Collects named objects together in a group.

Syntax

```
group = CollectNamed(object, name, ...);
```

Inputs

Name	Type	Default	Description
object	object	none	object to be collected
name	string	none	name of object
...			additional object-name pair(s)

Outputs

Name	Type	Description
group	group	the group of objects

Functional Details

This module creates, from specified objects, a group of named members.

object is an object to be placed in the group.

name is the name to be associated with the preceding object. Each name must be unique.

The input parameters **object** and **name** must be paired. Objects can then be selected from the group by name or by number (see "Select" on page 291).

Note that fields placed in the generic group created by the module are treated individually (compare CollectMultiGrid). For example, AutoColor colors each field in the group according to the minimum and maximum data values of that field, and Sample samples each field individually.

A single CollectNamed module can collect up to 21 objects together in a group. The default number of enabled parameter pairs (i.e., **object**, **name**) is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Components

All input components are propagated to the output.

See Also

Append, Collect, CollectMultiGrid, CollectSeries, Select

CollectSeries

Category

Structuring

Function

Collects objects into a series.

Syntax

```
series = CollectSeries(object, position, ...);
```

Inputs

Name	Type	Default	Description
object	object	no default	object to be collected
position	scalar	no default	series position of object
...			additional object-position pair(s)

Outputs

Name	Type	Description
series	series	the output series

Functional Details

This module creates a series group from specified objects (e.g., a time series from a set of fields). The data and connections types of each field in the series must be the same.

object	is an object to be placed in the series
position	specifies the series position (e.g., time tag) of the preceding object

A single CollectSeries module can have up to 21 **object**,**position** pairs. The default number of enabled parameter pairs (i.e. **object**,**position**) is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

The input parameters **object** and **position** must be paired. Note that items placed in a series can be selected only by ordinal number, not by series position.

A single CollectSeries module can collect up to 21 objects together in a group. The default number of enabled parameter pairs (i.e., **object**, **position**) is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Components

All input components are propagated to the output.

Example Visual Programs

```
ManipulateGroups.net
UsingCompute3.net
UsingStreakline.net
```

See Also

Append, Collect, CollectMultiGrid, CollectNamed, Select, Stack

Color

Category

Transformation

Function

Colors a field.

Syntax

```
colored = Color(input, color, opacity, component, delayed);
```

Inputs

Name	Type	Default	Description
input	field	none	field to be colored
color	field or vector or string	no color added	RGB color
opacity	field or scalar	input dependent	opacity
component	string	"colors"	component to be colored
delayed	flag	0	0: apply maps 1: delay applying color and opacity maps (byte data only)

Outputs

Name	Type	Description
colored	color field	color-mapped input field

Functional Details

This module adds a specified color to a specified input object.

input is the field to be colored.

color specifies how the input field is to be colored. The specification can be the vector value of an RGB color, a string, or a color map.

If **color** is an RGB color, the value should be in the range of 0–1 (but see "Coloring Objects for Volume Rendering" on page 113). The Convert module can convert HSV (hue, saturation, and value) colors to RGB.

If **color** is a string, that string should come from a lookup table, which can be specified by setting the DXCOLORS environment variable or by using the `-colors` flag with the `dx` command. If no table is specified, Data Explorer will search (in the order shown) for one of the following:

1. the file `colors.txt` in `DXROOT/lib`
2. the same file in **/usr/lpp/dx/lib** (if this was not the setting for DXROOT). Note that the colors in this file correspond to the X

Window System** color list, except that the Data Explorer colors are squared first (see Appendix F, “Data Explorer Colors” on page 313 in *IBM Visualization Data Explorer User’s Guide*).

When entering the names of colors, note that the module:

- accepts spaces in names.
- ignores capitalization.
- accepts the spellings *gray* and *grey*.

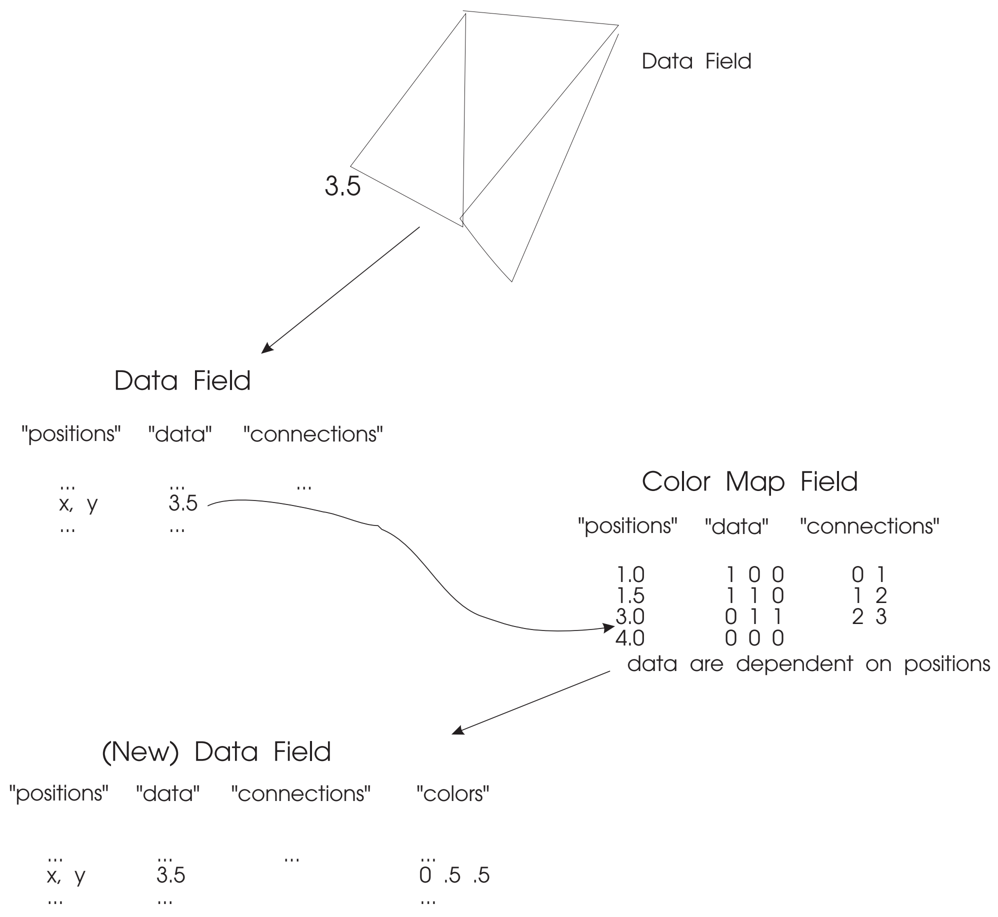
If **color** is a color map it can be the output of the Colormap Editor (the first output) or an imported color map. Note also that if this parameter is a color map (as opposed to a single color), then the **input** parameter must contain a “data” component; if the “data” component consists of vector data, the color and opacity mapping are based on the magnitude of the data. If this parameter is an imported .cm file (see “Import” on page 165), the color-map part of the color-opacity map is extracted and used.

Omitting the color specification allows you to change the opacity of an object without modifying its color (see page 76 for a description of a color map).

A well-formed color map should contain a 1-dimensional “positions” component and a 3-dimensional “data” component. As with any map in Data Explorer, the “positions” component represents the domain in which to look up values, and the “data” component represents the range, that is, the values which are associated with items in the “positions” component.

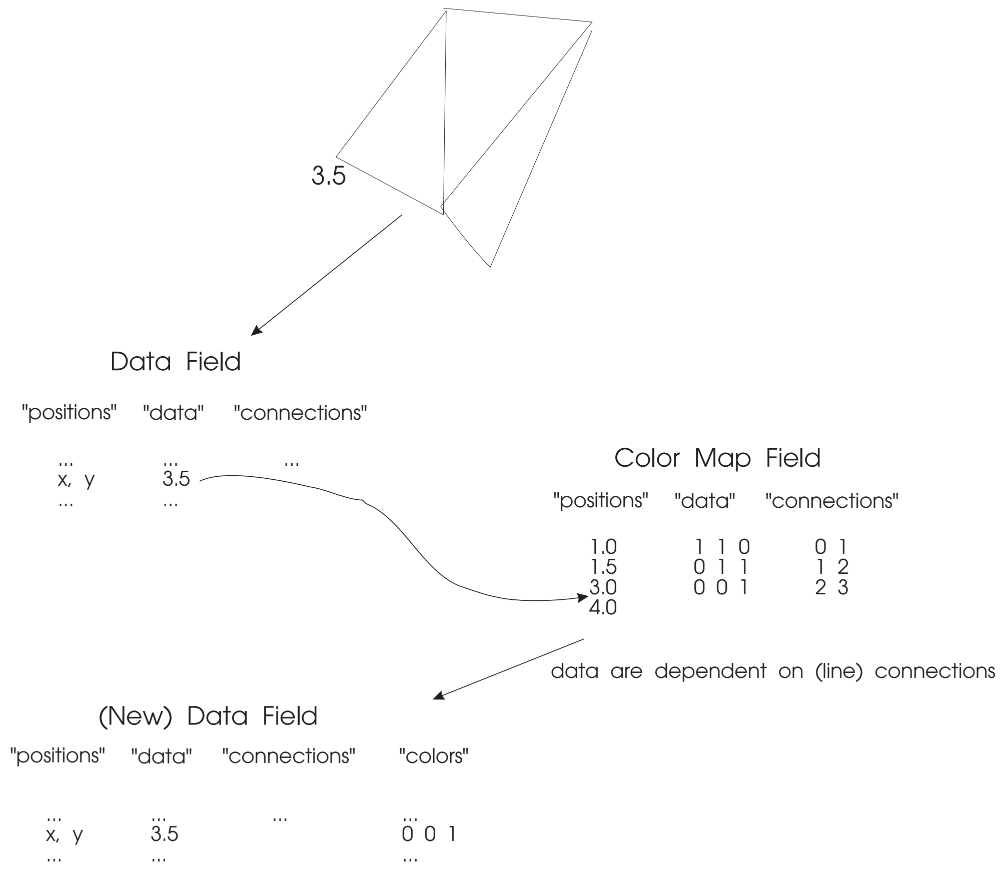
Color maps can specify either smoothly varying colors or constant colors across a set of ranges of data values. If the color map has position-dependent data, then linear interpolation will be used to derive colors for data values in the data field between those given in the “positions” component of the map (see Figure 1 on page 77).

Figure 1. Position-dependent colormaps. In this figure a diagram of a data field is shown. One of the data values, 3.5, has been indicated. The field structure is also shown, with "positions", "data", and "connections" components. When a color map is applied to this field using the Color module, the data value 3.5 is used as a lookup value into the "positions" component of the color map. The color map has a "data" component which is dependent on (in a one-to-one correspondence with) the "positions" component. The color map contains colors for the value 3 (the RGB value [0 1 1], or cyan) and for the value 4 (the RGB value [0 0 0], or black). Data Explorer interpolates between these two colors to derive the color [0 .5 .5], or dark cyan, which is then placed in the "colors" component of the data field as the color corresponding to the data value 3.5.



If the colormap has connection-dependent data, then the color for any data value in the data field between two values in the "positions" component of the map will be constant (see Figure 2 on page 78).

Figure 2. Connection-dependent colormaps. In this figure a diagram of a data field is shown. One of the data values, 3.5, has been indicated. The field structure is also shown, with "positions", "data", and "connections" components. When a color map is applied to this field using the Color module, the data value 3.5 is used as a lookup value into the "positions" component of the color map. The color map has a "data" component which is dependent on (in a one-to-one correspondence with) the "connections" component. The color map contains colors for the range 1 to 1.5 (the RGB value [1 1 0], or yellow), the range 1.5 to 3 (the RGB value [0 1 1], or cyan) and for the range 3 to 4 (the RGB value [0 0 1], or blue). Since 3.5 lies in the range 3 to 4, Data Explorer looks up the color [0 0 1] and places it in the "colors" component of the data field as the color corresponding to the data value 3.5.


3.5
Data Field
"positions" "data" "connections"
x, y 3.5
Color Map Field
"positions" "data" "connections"
1.0 1 1 0 0 1
1.5 0 1 1 1 2
3.0 0 0 1 2 3
4.0
data are dependent on (line) connections
(New) Data Field
"positions" "data" "connections" "colors"
x, y 3.5 0 0 1

In either case, the "connections" component of the map should be a set of lines connecting the positions.

The Colormap Editor puts out a well-formed colormap. The Construct module can also be used to create a color map. For example, you have 10 data values to which you want to apply particular colors. List those 10 data values as the first parameter to Construct. Then list the 10 colors (RGB vectors) as the last parameter to Construct. This will automatically create a position-dependent color map of the appropriate structure to use with the Color module. Alternatively, if you list only 9 colors (RGB vectors) as the last parameter to Construct, a connection-dependent color map will be created.

For surfaces, RGB colors in a color map should range between 0 and 1. Compute can be used to convert colors from the range of 0 to 255 to the range 0 to 1. To choose appropriate colors for volumes see "Coloring Objects for Volume Rendering" on page 113. Note that if you have an HSV (hue, saturation, and value) color map, it can be converted to an RGB (red, blue, green) color map using the Convert module.

opacity can be a scalar value or a field specifying an opacity map. This map may be the output of the Colormap Editor (the second output) or an imported opacity map. If it is an imported .cm file (see “Import” on page 165), the opacity part of the color-opacity map will be extracted and used (see below for a description of an opacity map).

The input color or opacity can also be groups of color or of opacity maps, as long as the hierarchy of the group matches that of **input**.

For surfaces, the default value of **opacity** is 1.0; the valid range is 0–1. For volumes, the default value is 0.5. If the object to be colored is a volume with an aspect ratio much different from 1, it may appear dark from certain viewing directions. In that case, use the Compute module to multiply the contents of the data component of the opacity and color maps by a scale factor greater than 1 before using them as an input to the Color module. (If you are using delayed colors, modify the “color multiplier” and “opacity multiplier” attributes. See the **delayed** parameter, described below.)

A well-formed opacity map should contain a 1-dimensional “positions” component and a 1-dimensional “data” component representing opacities. For surfaces, valid opacities range between 0 (transparent) and 1 (opaque). To choose appropriate opacities for volumes see “Coloring Objects for Volume Rendering” on page 113. Just as with a color map, the “data” component may be either position-dependent or connection-dependent. An opacity map can be created either with the Colormap Editor or using the Construct module as described for color maps.

component specifies the component to which the module adds colors. The default is the “colors” component, which applies to both the front and the back of the object, but you can specify “front colors” or “back colors” instead. Either one, if present, takes precedence over the “colors” component. If you specify **component** as “colors,” the module deletes any existing “front colors” or “back colors” components. Which faces are “front” and which faces are “back” depends on how “connections” component of the faces is defined (see “Standard Components” on page 19 in *IBM Visualization Data Explorer User’s Guide*).

delayed causes Color to create “delayed colors.” This option is valid only for byte data. When **delayed** = 1:

- the “colors” component is a copy of the “data” component, and a “color map” component is created (i.e., a color lookup table with 256 entries representing the appropriate color for each of the 256 possible data values).
- (if **opacity** has been specified) the “opacities” component is a copy of the “data” component, and an opacity map with 256 entries is created.
- the module adds a “direct color map” attribute to the output object (see “Using Direct Color Maps” on page 111).

Components

Adds a “colors” component. An “opacities” component is added if **`opacity`** is less than 1 or the input data is a volume. If **`delayed`** = 1, the “colors” component is a copy of the “data” component and a “color map” component is created. Likewise, an “opacity map” component is created if **`opacity`** is less than one or the input is a volume.

Example Visual Programs

Nearly every example visual program uses Color, including:

```
AlternateVisualizations.net
Sealevel.net
UsingColormaps.net
SIMPLE/Color
```

See Also

AutoColor, Caption, Convert, Map

ColorBar

Category

Annotation

Function

Creates a color bar.

Syntax

```
colorbar = ColorBar(colormap, position, shape, horizontal, ticks,
                    min, max, colors, annotation, labelscale, font);
```

Inputs

Name	Type	Default	Description
colormap	field	none	color map
position	vector	[0.95, 0.95]	the position of the color bar (in viewport-relative coordinates)
shape	vector	[300 25]	length and width of the color bar (in pixels)
horizontal	flag	0	0: vertical orientation 1: horizontal orientation
ticks	integer	input dependent	approximate number of tick marks along the bar
min	scalar or object	map min	minimum value on bar
max	scalar or object	map max	maximum value on bar
label	string	no defaults	label for color bar
colors	vector list or string list	appropriate	colors for annotation
annotation	string list	"all"	annotation objects to be colored
labelscale	scalar	1.0	scale factor for labels
font	string	standard	font for labels
ticklocations	scalar list	appropriate	tick locations
ticklabels	string list	ticklocations	tick labels

Outputs

Name	Type	Description
colorbar	color field	the color bar

Functional Details

The color bar generated by this module can be collected with the rest of the objects in the scene (by using a Collect module) and incorporated into an image.

`colormap` must be a color map (e.g., the second output of AutoColor or the first output of the Colormap Editor). The input can also be an imported **`.cm`** file (see “Import” on page 165), in which case the color-map part of the color-opacity map is extracted and used. (A color map has a 1-dimensional “positions” component, representing the data values, and a 3-dimensional “data” component, representing the RGB color assigned to each data value.)

`position` is a 2-dimensional vector (or a 3-dimensional vector whose *z*-component is ignored) indicating the position of the color bar in the final image. In viewport-relative coordinates, [0 0] places the bar at the lower left, and [1 1] at the upper right. These same coordinates determine the reference point that is used to position the bar relative to its placement in the image (e.g., for **`position`** = [0 0], the lower left corner of the bar is placed in the lower left corner of the image).

`shape` is a 2-vector that specifies the length and width of the color bar, in pixels. For both horizontal and vertical orientations, the first element of the vector is the length and the second is the width.

`horizontal` determines whether the orientation of the color bar is vertical (0) or horizontal (1).

`ticks` specifies the number of tick marks to be placed on the color bar (the actual number will at least approximate the specification). The default varies the number in accord with the size of the bar.

`min` and **`max`** specify the limits of the color bar. The values can be scalar. If **`min`** is an object (a data field), the minimum and maximum data values of that field are used to set the corresponding limits of the bar. If **`min`** is scalar and **`max`** is an object (a data field), the maximum is the maximum data value of that field. When neither is specified, the minimum and maximum used are those of **`colormap`**.

`label` specifies a user-supplied label for the color bar.

`colors` and **`annotation`**
set the colors of certain components of the color bar.

`colors` can be a single color (RGB vector or color-name string) or a list. The color-name string must be one of the defined color names (see “Color” on page 75).

`annotation` can be a single string or a list of strings, chosen from the following: “all.,” “frame,” “labels,” and “ticks.”

If **`annotation`** is not specified or is “all”—*and* if **`colors`** is a single string—then **`colors`** is used for all color-bar annotation. Otherwise the number of colors must match the number of annotation strings exactly. The default frame color is “clear.”

`labelscale` determines the size of the axes and tick-mark labels. For example, **`labelscale`** = 2.0 will display the labels at double their default size.

font specifies the font used for axes and tick-mark labels. You can specify any of the defined fonts supplied with Data Explorer. These include a variable-width font ("variable", the default for axes labels) and a fixed-width font ("fixed", the default for tick-marks labels).

area	gothicit_t	pitman	roman_ext
cyril_d	greek_d	roman_d	script_d
fixed	greek_s	roman_dser	script_s
gothiceng_t	italic_d	roman_s	variable
gothicger_t	italic_t	roman_tser	

For more information, see Appendix E, "Data Explorer Fonts" on page 307 in *IBM Visualization Data Explorer User's Guide*.

ticklocations specifies the explicit location for tick marks. If specified, overrides the value as determined by **ticks**.

ticklabels specifies the list of labels to be associated with the tick locations specified by **ticklocations**. If **ticklabels** is specified, and **ticklocations** is not specified, then **ticklocations** defaults to the integers 0 to n-1 where n is the number of items in **ticklabels**.

Notes:

1. **min** and **max**, if given, or the extent of the data if **min** and **max** are not given, set the extent of the color bar.
2. If you are using Render or Display to render an object which contains a color bar, when you change the resolution of the camera the size of the color bar in pixels will not change. Thus if you want the color bar to take up the same proportion of the image, use the ScaleScreen module to change the size of the color bar.

Example Visual Programs

```
BandedColors.net
Sealevel.net
UsingColormaps.net
UsingIsosurface.net
VolumeRendering.net
SIMPLE/ColorBar.net
SIMPLE/ScaleScreen.net
```

See Also

AutoColor, AutoGrayScale, Color, Map, ScaleScreen, Legend

Colormap

Category

Special

Function

Produces a color map.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
`data`	field	none	object used to derive min or max
`min`	scalar	min of data	minimum of color-map range
`max`	scalar	max of data	maximum of color-map range
`nbins`	integer	20	number of bins in the histogram
`colormap`	field or group	none	RGB color map to be used
`opacity`	field	none	opacity map to be used
`title`	string	"Colormap Editor"	title of editor

Outputs

Name	Type	Description
`rgb`	field	RGB color field
`opacity`	field	opacity field

Functional Details

This interactive module allows the user to create color maps that are applied to data. Through inputs to the module (either the outputs of other tools or values set in its configuration dialog box), the Colormap Editor can be "data driven." If it is not data driven, then the attributes (such as `min` and `max`) are taken from values set by the user in the Colormap Editor.

Note: The Colormap Editor itself is invoked by double-clicking on the module's icon in the VPE window. The module's configuration dialog box is accessed from the `Edit` pull-down menu in the same window.

`data` input can be used to derive the minimum and maximum values in the color and opacity maps. If this parameter is specified, a histogram of the "data" component becomes available in the Colormap Editor.

`min` sets the minimum value used in the color and opacity maps. If this parameter is specified, it overrides the minimum value set by `data`. If neither `min` nor `data` is specified, the value is set in the minimum field of the Colormap Editor dialog box.

max	sets the maximum value in the color and opacity maps. If this parameter is specified, it overrides the maximum value set by **data**. If neither **max** nor **data** is specified, the value is set in the maximum field of the Colormap Editor dialog box.
nbins	specifies the number of data bins in the histogram displayed by the Colormap Editor. A value of 0 indicates that the histogram should not be computed. The histogram is accessible through the **Axis Display** in the **Options** pull-down menu in the Colormap Editor.
colormap	specifies the color map used to initialize the color-map portion of the Colormap Editor. For example, the map may be an imported `.cm` file, which contains both a color and an opacity map. If this file is passed to the **colormap** parameter, both maps will be used to initialize the maps in the editor. Alternatively, a simple map field can be passed to the appropriate parameter(s). Note that only the color information of the map is used to initialize the Colormap Editor; the "positions" component of the map (representing the data range to which the colors apply) are ignored. This is to allow a colormap to be used for any input data set which may be passed to the **data** parameter; the range of the map will be taken from **data**, rather than from the colormap. If you want to use the actual "positions" of the colormap, then simply use Mark to mark the "positions" component of the colormap, and pass the output of Mark to the **data** parameter of Colormap.
opacity	specifies the opacity map used to initialize the opacity-map portion of the Colormap Editor.
title	specifies the title of the Colormap Editor window.

Example Visual Programs

```
UsingColormaps.net
VolumeRendering.net
```

An example that uses a data-driven Colormap is:

```
DataDrivenInteractors.net
```

See Also

AutoColor, Color

Compute

Category

Transformation

Function

Evaluates an expression on each data point in a specified field or value list.

Syntax

```
output = Compute(expression, input, ...);
```

Inputs

Name	Type	Default	Description
expression	string	none	expression to be computed
input	field or value list	no default	input value
...	...	...	more input values

Outputs

Name	Type	Description
output	field, value, or value list	output values

Functional Details

This module applies an expression to every data value in a field.

expression	is the mathematical expression to be applied to **input**. Table 1 on page 87 lists the operators.
input	is the field or value list to which **expression** is to be applied. If there are more than one, the input fields must be isomorphic (i.e., their hierarchies must match exactly).

A single Compute module can operate on a maximum of 21 **input** values. In the user interface, the default number of enabled input tabs is two. The default number of enabled input tabs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

In the user interface, variables have names and **expression** does not require quotation marks. If the vector variable is “sample,” the vector elements are “sample.x,” “sample.y,” ... or “sample.0,” “sample.1,” ....

In the scripting language, the parameters of an expression are indicated by $*n*, where *n* is the index of the parameter, and counting begins at 0. The first three components of a vector can be expressed by “*.x*,” “*.y*,” and “*.z*,” or by “*.0*,” “*.1*,” and “*.2*.” The syntax for creating a vector from multiple inputs is [*$0, $1, $2, ...*], where the commas separators are required.

Thus, in the user interface the expression used to add the value 3 to every data value of the input field would be “a + 3.,” while in the scripting language it would be “$0 + 3.”

Multiple expressions can be entered if they are separated by semicolons; the final expression is used as the output. For example, “temp=a.x; sin(temp)” is equivalent to “sin(a.x)”.

Compute uses C-language-style order of precedence for mathematical operations.

Operations applied to invalid elements of fields typically result in invalid elements. (For more information, see “Invalid Positions and Invalid Connections Components” on page 23 in the *IBM Visualization Data Explorer User’s Guide*.) For example, if at a particular element field *“a”* and *“c”* are valid, but field *“b”* is not, *“a+b”* is invalid, and *“a? b: c”* is valid only if *“a”* is 0; otherwise, it is invalid. An exception is the “invalid” function, which returns the integer 1 if the entry is invalid and 0 otherwise.

Operations such as multiplying a vector by a scalar have the expected effect: each element of the vector is multiplied by the scalar. Likewise, the addition of two equal-length vectors or equal-size matrices results in an element-by-element sum. The Compute module defines the multiplication (`*`) of two equal-length vectors as an element-by-element multiplication (as opposed to the dot product). Other operations, such as the sine of a vector, are defined similarly.

Constants may be specified as double precision by using scientific notation with “d” indicating the exponent (e.g., `1d0` is double-precision `1`).

Note that you can convert a “ref” type invalid component (see “Invalid Positions and Invalid Connections Components” on page 23 in *IBM Visualization Data Explorer User’s Guide*) using the expression “byte (invalid (a))”. Then use Replace to substitute this new array into the original field as the “invalid positions” or “invalid connections” component.

Table 1 (Page 1 of 4). Operators for the Compute Module

Functions	Types of Operands
Trigonometric Functions (argument in radians)	
sin(a), cos(a), tan(a), asin(a), acos(a), atan(a), atan2(a, b)	float, double
Hyperbolic Functions	
sinh(a), cosh(a), tanh(a)	float, double
Logarithmic Functions	
log(a), In(a) (natural logarithm), log10(a) (log base 10—see Note 1), exp(a)	float, double
Unary Functions	
+a, –a (negation)	any type
Binary Functions	
a+b, a–b, a*b, a/b, a%b (modulus—see Note 1), a^b or a**b (exponentiation—see Note 4)	any type
Vector Functions (see Note 1)	
a dot b or dot(a, b)	float vector
a cross b or cross(a,b)	float 3-vector
mag(a)	double, float vector
norm(a)	float vector
Miscellaneous Functions	
sqrt(a)	float, double, complex
pow(a, b) (see Note 4)	float, double, complex

Table 1 (Page 2 of 4). Operators for the Compute Module

Functions	Types of Operands
abs(a) (see Note 2)	double, float, integer, complex
arg(a)	complex only
sign(a)	all real types
min(a, b, ...), max(a, b, ...)	scalar
invalid(a) (see Note 5)	any type
random(a,seed) (see note 8)	produces random values in the range 0<=r<1 for each item in a.
Type Manipulation Functions	
int(a), float(a), byte(a), char(a), double(a), short(a), sbyte, ubyte, ushort, uint (see Note 6)	float, integer, byte, short, double
trunc(a), floor(a), ceil(a), rint(a)	float, double
complex(a,b) or complex(a),	float, integer, byte, short, double
real(a)	complex only
imag(a)	complex only
Vector Construction	
[a, b, ...]	any type
Vector Selection Functions	
a.x or a.0, a.y or a.1, and so on	vector
select(a,b) (selects bth element of a, where element is of rank r−1)	a is a vector, b is an integer
Conditional Functions	
a?b:c if a != 0, then b, else c. (b and c must be of the same type.) Expressions b and c are always evaluated, and the output value depends on the value of a.	a is an integer
Logical Operations	
binary: <, >, <=, >= (true = 1; false = 0)	any scalar type
binary: ==, !=	any type
Unary: ! (not), binary: && (and), ‖ (or)	integer
Bitwise Operations	
and(a,b), xor(a,b), or(a,b), not(a) (one's complement)	byte, int
String functions (see Note 7)	
strcmp(a,b) compares strings a and b and returns 0 if a==b, a negative integer if a<b, and a positive integer if a>b stricmp is identical to strcmp except that it ignores case	strings
strlen(a) returns the length of string a	string

Table 1 (Page 3 of 4). Operators for the Compute Module

Functions	Types of Operands
strstr(a,b) finds substring b in string a and returns 1-based offset of b in a. Thus strstr('artist','art') = 1 strstr('artist','picasso') = 0 strstr('monet','one') = 2 stristr(a,b) is identical to strstr except that it ignores case	strings

Notes:

1. The vector functions, modulus function, and log10() function do not accept complex numbers as arguments.
2. Given an integer or floating-point value, the abs() function yields the absolute value. Given a complex number, the abs() function yields a real absolute value.
3. If a string is passed as a parameter to Compute, it is treated as an array of bytes, with its values being the ASCII values of the string elements.
4. This function returns a floating-point value if inputs are floating point or integer. It returns a complex if the first input is complex.
5. Returns integer 1 if entity is marked invalid, and 0 otherwise.
6. Byte and char are same as ubyte.
7. Strings can be passed into Compute as the data component or directly from String and StringList interactors. Strings can also be specified in the Compute expression directly by enclosing them in single quotes. A single quote character is represented by a pair of single quotes. Thus if a is the string “can't”, then strcmp(a,'can''t') = 0.
8. The random number generator will be seeded by the integer **seed**. If **a** corresponds to a group then each member of **a** will be seeded by **seed**+n, where n corresponds to the member's enumerated location in **a**. This results in repeatable behavior even when a composite field is being processed in parallel on an SMP machine. To generate different random results, use a different **seed**.

To operate on a component other than “data,” use the Mark and Unmark modules together with Compute.

A single Compute module can operate on a maximum of 21 **...Input** values. The default number of enabled input tabs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Note: Other than a divide by zero, Compute operations are unchecked.

Components

Modifies the “data” and “invalid positions” or “invalid connections” components. All other components are propagated to the output.

Script Language Examples

1. The Compute module converts all the temperature data to Fahrenheit.

   ```
   tempf = Compute("$0*(9.0/5.0) + 32", tempc);
     ⋮
   ```

2. The input field is a vector field of dimension 9. The output field is scalar, consisting of the sixth component of the input field.

   ```
   new_field = Compute("$0.5",field);
   ```

3. The input field is a vector field, and the output field is also a vector field, with the *x* component multiplied by 2.

   ```
   new_field = Compute("[2*$0.x, $0.y, $0.z]", field);
   ```

or

```
new_field = Compute("[2, 1, 1]*$0", field);
```

4. Here **field1** is a vector field and **field2** and **field3** are scalar. Therefore, the output field will have a data component equal (on a point-by-point basis) to the magnitude of **field1** added to the quantity **field2** divided by 4.5 times **field3**.

```
new_field = Compute("mag($0) + $1/($2*4.5)", field1, field2, field3);
```

5. The Mark module to place the "positions" component of the 2-dimensional object **slice** in the "data" component, allowing Compute to operate on the positions. The formula string is assigned to **function** to simplify the call to Compute and is equivalent to the following formula for converting **slice** positions to 2-dimensional vectors:

$$[x, y, z] = \left[\ \sin\left(2\pi \times \frac{(x+1)}{3.9}\right), y, -\cos\left(2\pi \times \frac{(x+1)}{3.9}\right)\right]$$

When the computation is done, the Unmark module replaces the original "data" component for use in **warped**. The resulting positions are warped onto the shape of a cylinder.

```
⋮
slice = Slice(electrondensity, "z", 5);
    // Mark the positions so that they can be computed on
    // The original x positions go from -1 to 2.9
    // The original y positions go from -3 to 2.9
markedslice = Mark(slice,"positions");
    // Warp the positions onto the shape of a cylinder
pi = 3.14159;
exp = "[sin(2*$1*($0.x+1)/3.9), $0.y, -cos(2*$1*($0.x+1)/3.9)]";
warped = Compute(exp, markedslice, pi);
    // Unmark the warped positions, returning them to the positions
    // component
warped = Unmark(warped, "positions");
⋮
```

6. This example differs from the previous one in that the **exp**) function warps the positions onto a double cone shape, by implementing the following formula:

$$[x, y, z] = \left[y \times \sin\left(2\pi \times \frac{(x+1)}{3.9}\right), y, -y \times \cos\left(2\pi \times \frac{(x+1)}{3.9}\right)\right]$$

```
⋮
    // Now warp the positions onto the shape of a doubled cone
    // by multiplying the x and z positions by the original
    // y value, which goes from -3 to 2.9
exp = "[$0.y*sin(2*$1*($0.x+1)/3.9),$0.y, -$0.y*cos(2*$1*($0.x+1)/3.9)]";
warped = Compute(exp, markedslice, pi);
    // Unmark the warped positions, returning them to the positions
    // component
warped = Unmark(warped, "positions");
⋮
```

Example Visual Programs

Many of the example visual programs use Compute, including:

```
ComputeOnData.net
DataDrivenInteractors.net
PlotTwoLines.net
WarpingPositions.net
```

See Also

Mark, Unmark

Compute2

Category

Transformation

Function

Evaluates an expression on each data point in a specified field or value list.

Syntax

```
output = Compute2(expression, name, input, ...);
```

Inputs

Name	Type	Default	Description
`expression`	string	none	expression(s) to be computed
`name`	string	no default	name of input that follows (defaults to "a")
`input`	field or value list	no default	input value
`name1`	string	no default	name of input that follows (defaults to "b")
`input1`	field or value list	no default	input value
...	...	...	...

Outputs

Name	Type	Description
`output`	field, value, or value list	output values

Functional Details

The primary advantage of this alternative form of Compute is that, in the user interface, **expression** is an input that can be provided by another tool (e.g., a selector interactor). All the functions of Compute (see "Compute" on page 86) are also functions of Compute2.

expression is a mathematical expression to be evaluated for a set of **input** values. Table 1 on page 87 lists the operators.

This parameter is followed by one or more name-input pairs.

name is name of the variable in the expression to be evaluated.

input is the input to which **expression** is applied. If there are more than one, the input fields must be isomorphic (i.e., their hierarchies must match exactly).

A single Compute2 module can operate on a maximum of 21 **input** values. The default number of enabled input tabs is two. (Tabs can be added to the module icon and removed with the appropriate **`Input/Output Tabs`** options in the **`Edit`** pull-down menu of the VPE.)

Example Visual Programs

```
ComputeMultiLine.net
SIMPLE/Compute2.net
```

See Also

Compute

Connect

Category

Realization

Function

Creates triangle connections for a field of scattered positions.

Syntax

```
output = Connect(input, method, normal);
```

Inputs

Name	Type	Default	Description
input	field or vector list	none	field with positions to be connected
method	string	"triangulation"	connection method to be used
normal	vector	[0 0 1]	normal to the projection plane

Outputs

Name	Type	Description
output	field	connected field

Functional Details

The triangle connections created by this module form a surface.

input should be (1) a field with a 2- or 3-dimensional component or (2) a list of 2- or 3-dimensional vectors. In the second case, the vectors are interpreted as positions.

method specifies the method of connection. At present, Delaunay "triangulation" (involving Voronoi tesselation of a plane) is the only method supported.

normal specifies the plane to which 3-dimensional positions are projected before triangulation is computed. Note that the output positions will be the original 3-dimensional points.

Degenerate triangles (i.e., those with colinear vertices) are not included in the output object.

Components

Adds a "connections" component. All components that depend on the "positions" component are unchanged in the output.

Example Visual Program

```
ConnectingScatteredPoints.net
SIMPLE/Connect.net
```

See Also

Regrid, AutoGrid

Construct

Category

Realization

Function

Constructs a field with regular connections.

Syntax

```
output = Construct(origin, deltas, counts, data);
```

Inputs

Name	Type	Default	Description
origin	vector list	input dependent	origin for positions or a list of positions
deltas	vector list	input dependent	deltas for positions
counts	integer or vector	input dependent	number of positions in each dimension
data	value list or string list	no default	data, last index varies fastest

Outputs

Name	Type	Description
output	field	the output field

Functional Details

This module creates a field by defining its positions and connections.

origin specifies either the origin of a field with regular positions or a list of positions for a field with irregular positions.

deltas should be used only for creating a field with regular connections. It specifies one of the following:

- a vector of the same shape as **origin**, if **origin** is specified
- a list of vectors that match **origin**, the number of vectors being equal to the number of dimensions in **origin**. (This specification can be used to define a grid with non-orthogonal axes.)

If **origin** is not specified, the dimensionality of the output positions is derived from the dimensionality of **deltas**. The default value is a unit vector.

counts specifies the number of positions in each dimensions for a field with regular connections. The module interprets this parameter according to the value specified by **origin**. If **origin** specifies:

- *a single vector:* the field will have regular positions and regular connections. (If **counts** has been specified as a single number, the module creates that number of positions in each dimension.)
- *a vector list:* the field will have irregular positions and regular connections. The product of the counts must match the number of positions given by **origin**.

To create a field consisting of just the items specified by **origin** as the "positions" component, do not specify **counts**. If the list of vectors contains more than one item, the output has, in addition to the "positions" component containing the points in **origin**, a "connections" component of element type "lines".

data specifies one or more data values associated with the positions or connections of the field.

If specified, this parameter must be either a single value or a list of values. If it is a single value or a list with a length that matches the number of positions [e.g., $n_0 \times n_1 ...$, where n_0, n_1, ..., are the **counts**], then the output data component is dependent on the "positions" component. If the list has a length that matches the number of connection elements (e.g., $(n_0\text{-}1) \times (n_1\text{-}1) ...$), then the output data is dependent on the "connections" component. Any other number of items in **data** is an error.

The type of the "data" component is the type of the input **data**. If **data** is a string list, the "data" component will be TYPE_STRING.

Note: If none of the first three parameters is specified (**origin**, **deltas**, or **counts**), Construct creates an empty field.

Components

Creates "positions" and "connections" components. A "data" component is created if **data** is not null.

Example Visual Programs

```
AnnotationGlyphs.net
ConnectingScatteredPoints.net
ProbeText.net
Sealevel.net
UsingColormaps.net
UsingStreakline.net
SIMPLE/Construct.net
```

Convert

Category

Transformation

Function

Converts between RGB and HSV color spaces.

Syntax

```
Output = Convert(data, incolorspace, outcolorspace, addpoints);
```

Inputs

Name	Type	Default	Description
`data`	vector list or field	none	input colors or color map
`incolorspace`	string	"hsv"	color space of input
`outcolorspace`	string	"rgb"	color space of output
`addpoints`	integer	input dependent	add positions to ensure valid color transformations

Outputs

Name	Type	Description
output	vector list or field	output colors or color map

Functional Details

`data` specifies the value(s) to be converted. If the parameter value is a vector or list of vectors, the output is a vector or list of vectors. These must be of 3-vectors, representing hue, saturation, and value (HSV) or red, green, and blue (RGB).

If the parameter value is a field, the module converts the “data” component of that field (which must consist of 3-vectors) from one color space to the other.

The range of hues is 0 (red) to 1 (red); 0.3333 = green and 0.6666 = blue. Values outside this range simply “wrap.” That is, a hue of 0 = a hue of 1 = a hue of 2, and so on.

The range of saturation is 0 (white) to 1 (pure color).

The range of value is 0 (black) to 1 (full intensity).

`incolorspace` is the color space of the input (**`data`**) and must be “rgb” or “hsv.”

`outcolorspace` is the color space of the output and must be “rgb” or “hsv.”

`addpoints` specifies whether points are to be added to the resulting output:

If **`data`** is a color map (i.e., if it has 1-D positions and 3-D data), the default value (1) specifies that points will be added to the “positions” component of the map so that the conversion between color spaces (which is nonlinear) remains valid. A parameter value of 0 (zero) specifies that no points are to be added.

A parameter value of 1 (one) is valid *only* if the input is a color map. If the input object is *not* a color map (i.e., if it has positions of dimensionality greater than 1), the default value is 0 (zero) and the “data” component is converted as specified by **`incolorspace`** and **`outcolorspace`** in one-to-one fashion.

Components

Modifies the “data” component and may modify the “positions” component. All other input components are propagated to the output.

Example Visual Programs

```
AlternateVisualizations.net
UsingCompute.net
```

See Also

Color

CopyContainer

Category

Structuring

Function

Copies the top container object

Syntax

```
copy = Copy(input);
```

Inputs

Name	Type	Default	Description
`input`	object	none	object to copy

Outputs

Name	Type	Description
`copy`	object	copy of input

Functional Details

This module copies an object.

`input` is the object to copy. Only the “container” object (the field or group) and any attributes attached to the container object are copied; members or components of the container object are not copied.

Components

This module copies the specified information from the input to the output.

DFT

Category

Transformation

Function

Computes a discrete Fourier transform.

Syntax

output = DFT(**input, direction, center**);

Inputs

Name	Type	Default	Description
input	field	none	field to be transformed
direction	string	"forward"	direction of the transform
center	flag	0	center the result of the transform

Outputs

Name	Type	Description
output	field	transformed data

Functional Details

This module computes the discrete Fourier transform of a 2- or 3-dimensional regular data set.

input specifies the field to be transformed.

direction is one of the following: “forward,” “inverse,” or “backward” (the last two are interchangeable).

center specifies whether zero frequency should be placed at the center of the transformed field or at the origin of the positions array.

Components

All scalar components of **input** are individually Fourier-transformed and output as complex float. All other input components are propagated to the output. Thus a float 2-vector input produces a complex 2-vector output.

Example Visual Programs

`FFT.net`

See Also

FFT, Filter, Morph

DXLInput

Category

DXLink

Function

Enables a remote DXLink application to set a parameter value in a visual program.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
default	value, string, object	no default	default value (overridden by the remote application)

Outputs

Name	Type	Description
output	object	the default value or the value sent from a remote application.

Functional Details

This tool receives variable values from a remote application that uses the DXLink library of function calls (see Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*). The value received from the application is passed as output from the DXLInput tool.

The module’s label (set in the **Notation** field of the Configuration dialog box) is used to establish a global variable, which is then set by the DXLSet... functions of the DXLink development library. One of the parameters of these functions is the global variable name, which must be the same as the label of the DXLInput tool that is intended to receive the value given in the function call. Changing the text of the **Notation** field in DXLInput’s configuration dialog box changes the label displayed on the tool’s VPE stand-in, just as it does for Receivers and Transmitters.

If there is no remote application setting the default values, DXLInput will output the value specified by **default**. If a remote application sets the value, that value overrides **default**. This parameter provides a mechanism that makes it easier to debug visual programs that are intended to be used and controlled by remote DXLink applications.

Note: The **default** value must be set in the module’s configuration dialog box. Any value set by a connection to the output of another tool will *not* be overridden by a value set by a remote application.

See Also

DXLOutput, DXLInputNamed, Input, Output

Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*.

DXLInputNamed

Category

DXLink

Function

Enables a remote DXLink application to set a parameter value in a visual program, while also setting the name of the variable.

Syntax

```
output = DXLInputNamed(name, default);
```

Inputs

Name	Type	Default	Description
name	string	none	name of variable
default	value, string, object	no default	default value (overridden by the remote application)

Outputs

Name	Type	Description
output	object	the default value or the value sent from a remote application

Functional Details

This tool receives variable values from a remote application that uses the DXLink library of function calls (see Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*). The value received from the application is passed as output from the DXLInputNamed tool.

This module differs from DXLInput in that the name of the variable is set using the **name** parameter, rather than using the Notation field of the Configuration Dialog box. This enables the variable name to be passed in via a wire, rather than preset for the module. This allows you to place DXLInputNamed in a macro and have different instances of the macro use different variable names.

name	is the variable name. This variable is then set by DXLSet... functions of the DXLink development library.
default	is the default value for the output of DXLInputNamed.

If there is no remote application setting the default values, DXLInputNamed will output the value specified by **default**. If a remote application sets the value, that value overrides **default**. This parameter provides a mechanism that makes it easier to debug visual programs that are intended to be used and controlled by remote DXLink applications.

Note: The **default** value must be set in the module’s configuration dialog box. Any value set by a connection to the output of another tool will *not* be overridden by a value set by a remote application.

See Also

DXLInputDXLOutput , Input, Output

Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*.

DXLOutput

Category

DXLink

Function

Sends a value to a remote application.

Syntax

`DXLOutput(`**`label, value`**`);`

Inputs

Name	Type	Default	Description
`label`	string	none	name associated with **`value`**
`value`	value list or string list	none	value sent to a DXLink application

Functional Details

This module sends variable values from a visual program to a remote application that uses the DXLink library of function calls (see Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*). Values are sent as strings and in the same format as that used by the Echo tool.

`label` is similar in function to the label set in the DXLInput configuration dialog box in that it associates a name (i.e., a global variable name) with a value. This global variable name is used in the **`DXLSetValueHandler()`** routine of the DXLink library (see “Receiving Messages from the Server” on page 171 in *IBM Visualization Data Explorer Programmer’s Reference*).

The remote application should install a handler for the various **`label`**ed global variable names in a program (see DXLSetValueHandler in “Receiving Messages from the Server” on page 171 in *IBM Visualization Data Explorer Programmer’s Reference*). In general, **`label`** values should be unique within a program.

Note: If you are using the Data Explorer user interface, **`label`** does not appear as an input parameter but is set in the **`notation`** field of the DXLOutput configuration dialog box.

`value` is the value that is to be converted to a string and sent to a remote application. This value must be a Data Explorer value list or string (e.g., a scalar, integer list, matrix, etc.). It cannot be a Field or Group object.

See Also

DXLInput,Input , Output, DXLInputNamed

Describe

Category

Debugging

Function

Describes a Data Explorer object

Syntax

`Describe(object, options);`

Inputs

Name	Type	Default	Description
`object`	object	none	Object to be described
`options`	string	“all”	how much information to provide

Functional Details

`object` Specifies any Data Explorer object, for example, a group, field, or scalar value. This module will query the object and present information about it in the Message Window (use Open Message Window in the Windows menu).

`options` specifies how much information to provide. **`options`** must be one of “all”, “structure”, “details”, or “render”:

“all”	(the default) prints out all three types of the following information. Specifying any one of the following keywords restricts the output to that subset of the information
“structure”	describes the top level object
“details”	describes in more detail any data in the object
“render”	describes whether the object is renderable (i.e. ready for input to the Image or Display tools), and if not suggests what needs to be added or changed to make it so.

Components

Does not modify the input object.

Example Visual Programs

`SIMPLE/Describe.net`

See Also

Print, Echo, VisualObject

Direction

Category

Transformation

Function

Converts azimuth, elevation, and distance to an [x, y, z] position.

Syntax

```
point = Direction(azimuth, elevation, distance);
```

Inputs

Name	Type	Default	Description
azimuth	scalar	0	azimuth in degrees
elevation	scalar	0	elevation in degrees
distance	scalar	1	distance

Outputs

Name	Type	Description
point	vector	*x, y, z* position

Functional Details

The output of Direction can be used, for example, as input to the Camera or AutoCamera module to set the look-from direction.

azimuth	the angular distance (in degrees) from the positive *z*-axis to the positive *x*-axis.
elevation	the angular distance (in degrees) from the positive *xz* plane toward the positive *y*-axis.
distance	is measured in user units.

Example Visual Program

`MovingCamera.net`

See Also

AutoCamera, Camera, ClipPlane

Display

Category

Rendering

Function

Displays an image or renders a scene and displays an image.

Syntax

```
where = Display(object, camera, where, throttle);
```

Inputs

Name	Type	Default	Description
object	object	none	object to render or image to display
camera	camera	no default	camera if rendering is required
where	window or string	the user's terminal	host and window for display
throttle	scalar	0	minimum time between image frames (in seconds)

Outputs

Name	Type	Description
where	window	window identifier for Display window

Functional Details

object is the object to be displayed or to be rendered and displayed.

camera is the camera to be used to render **object**. If **camera** is not specified, the system assumes that **object** is an image to be displayed (e.g., the output of the Render module).

Note: A transformed camera cannot be used for this parameter.

where specifies the host and window for displaying the image. On a workstation, the format of the parameter string is:

X, *display, window*

where **X** refers to the X Window System; *display* is an X server name (e.g., host:0); and *window* is a window name (and must not begin with two #-characters). As a rule, it is not necessary to set this parameter. But when it is set, the resulting Image window is not controlled by the user interface (e.g., it has no menu options). The function of this parameter is to specify another workstation on which to display an image (e.g., by setting it to "*X,*`workstationname`*:0,* `message`"). Using the Image tool, you can display the same image to another workstation simply by connecting the module's two outputs to the two inputs of Display and setting **where**.

If you are using SuperviseState or SuperviseWindow to control user interactions in the Display window, then **`where`** should be set with the **`where`** output of SuperviseWindow.

Note: If you are using the **`where`** parameter, it is important to set its value *before* the first execution of Display.

`throttle` specifies a minimum interval between successive image displays. The default is 0 (no delay).

`where` The output can be used, for example, by ReadImageWindow to retrieve the pixels of an image after Display has run.

Notes:

1. In the user interface, you must use the Image tool rather than Display if you want to use many of the interactive image-manipulation functions provided by Data Explorer. For more information, see "Controlling the Image: View Control..." on page 74 in *IBM Visualization Data Explorer User's Guide*. However, see "SuperviseWindow" on page 336 and "SuperviseState" on page 332 for a discussion of how to create your own interaction modes when using the Display window.
2. The Display module can render surfaces, volumes, and arbitrary combinations of surfaces and volumes. (However, the current volume-rendering algorithm does not support coincident or perspective volumes.) To render an object, that object must contain a "colors" component. Many modules add a default color. In addition, volume rendering (e.g., of cubes, as opposed to lines) requires an "opacities" component. all surfaces, the lack of an "opacities" component implies an opaque surface.
3. Choosing appropriate color and opacity maps for volume rendering can be difficult. The AutoColor, AutoGrayScale, and Color modules use heuristics to generate good values; as a rule of thumb, the opacity should be 0.7/T, and the color value 1.4/T (where T is the thickness of the object in user units). See also "Coloring Objects for Volume Rendering" on page 113.

Changing the Resolution of an Image: If you are using Display without a camera to simply display an image, you can increase or decrease the resolution of the image by using Refine or Reduce, respectively, on the image before passing it to Display (see Refine and Reduce).

Pasting Images Together: The Arrange module can be used before Display to lay out images side by side, or one above the other (see Arrange).

Delayed Colors and Opacities (Color and Opacity Lookup Tables) Delayed colors are a way of compactly storing color and opacity information. By default, whenever you use one of the coloring modules (AutoColor, AutoGrayScale, Color), the colors and opacities are stored one-per-data value as a floating point RGB 3-vector or floating point value, respectively, ranging from 0 to 1. However, if you have unsigned byte data, then it is much more efficient to use "delayed colors" and "delayed opacities". When you use delayed colors or opacities, the "colors" or "opacities" component is simply a copy of (actually a reference to) the "data" component. When rendering occurs, these components are interpreted as indices with which to look up a color or opacity value in a table.

If you specify the **delayed** parameter as 1 to any of the coloring modules, they will automatically perform this “copy” of the “data” component, and will attach a “color map” or “opacity map” component which contains 256 RGB colors, or 256 opacities. If you already have a color or opacity map, either imported or created using the Colormap Editor, and wish to use delayed colors or delayed opacities, you can pass your color map or opacity map to the Color module as the **color** or **opacity** parameter, and set the **delayed** parameter to Color as 1.

The structure of a color map or opacity map is described in “Color” on page 75. The Colormap Editor produces as its two outputs well-formed color maps and opacity maps. Alternatively, if you already have a simple list of 3-vectors or list of scalar values, and want to create a color map or opacity map, you can do this using Construct. The first parameter to Construct should be [0], the second should be [1], and the third should be 256. This will create a “positions” component with positions from 0 to 255. The last parameter to Construct should be your list of 256 colors or opacities.

If you are reading a stored image using ReadImage, and the image is stored with a colormap, you can specify that the image should be stored internally in Data Explorer with delayed colors by using the **delayed** parameter to ReadImage.

You can also convert an image (or object) to a delayed colors version by using QuantizeImage.

Using Direct Color Maps: If you are using delayed colors (see “Delayed Colors and Opacities (Color and Opacity Lookup Tables)” on page 110 and “ReadImage” on page 250) and displaying images directly (i.e. you are not providing a camera), Display will use the provided color map directly instead of dithering the image. (Depending on your X server, you may need to use the mouse to select the Image or Display window in order for the correct color to appear.) If you do not want Display to use the color map directly, use the Options module to set a “direct color map” attribute with a value of 0 (zero).

Attribute Name	Value	Description
direct color map	0 or 1	whether or not to use a direct color map

Using Default Color Maps: When displaying non-delayed color images in 8-bit windows, Display assumes that it can specify 225 individual colors. If this number is not currently available in the shared color map, Display will find the best approximations available. However, this may lead to a visible degradation of image quality. Display may instead use a private color map. This decision is based on the worst-case approximation that it must use with the default color map. If this approximation exceeds a threshold, a private color map will be used. The approximation quality is measured as Euclidean distance between the desired color and the best approximation for that color, in an RGB unit cube.

An environment variable, DX8BITCMAP, sets the level at which the change to using a private color map is made. The value of DX8BITCMAP should be a number between 0 (zero) and 1 (one), and it represents the Euclidean distance in RGB color space, normalized to 1, for the maximum allowed discrepancy. If you set DX8BITCMAP to 1, then a private color map will never be used. On the other hand, if you set DX8BITCMAP to –1, then a private color map will always be used.

The default is 0.1. See also the `-8bitcmap` command line option for Data Explorer in Table 5 on page 295 in *IBM Visualization Data Explorer User's Guide*.

Gamma Correction: Displayed images generated by Display or Image are gamma corrected. Gamma correction adjusts for the fact that for many display devices a doubling of the digital value of an image's brightness does not necessarily produce a doubling of the actual screen brightness. Thus, before displaying to the screen, the pixel values are adjusted non-linearly to produce a more accurate appearance.

The environment variables DXGAMMA_8BIT, DXGAMMA_12BIT, and DXGAMMA_24BIT are used to specify values for gamma of 8-, 12-, and 24-bit windows, respectively. If the appropriate DXGAMMA_*n*BIT environment variable is not set, the value of the environment variable DXGAMMA will be used if one is defined. Otherwise, the module uses the system default, which depends on the machine architecture and window depth. This default is always 2 (two) except for 8-bit sgi windows, for which it is 1 (one). Note that the default depends on the machine on which the software renderer is running, not on the machine that displays the image.

Obtaining a WYSIWYG image of a higher resolution: If you wish to render a displayed image at a higher resolution (for example to write to an output file), you can usually simply use Render on the same object as **`object`**, with a new camera (see "AutoCamera" on page 31 or "Camera" on page 49). However, if **`object`** contains screen objects (captions and color bars), the new image will not be WYSIWYG (What You See Is What You Get), with respect to the displayed image, because the sizes of captions and color bars are specified in pixels rather than in screen-relative units. The ScaleScreen module (see "ScaleScreen" on page 289) allows you to modify the size of screen objects before rendering.

Image Caching: When given a **`camera`** input, the Display module (or Image tool) caches rendered images by default. The result is faster redisplay if the same object and camera are later passed to the module.

To turn off this automatic caching, use the Options module to attach a "cache" attribute (set to 0) to **`object`**.

It is important to remember that this caching is separate from the caching of module outputs, which is controlled by the `-cache` command-line option to `dx`.

Changing Rendering Properties: You can change the rendering properties of an object by using the Options module. The following table lists the shading attributes that can be set by the Options module for interpretation by the Display tool. (See the section on surface shading in *IBM Visualization Data Explorer Programmer's Reference* for more information.)

Attribute	**Type**	**Default**	**Description**
`"ambient"`	scalar	1	coefficient of ambient light k_a
`"diffuse"`	scalar	.7	coefficient of diffuse reflection k_d
`"specular"`	scalar	.5	coefficient of specular reflection k_s
`"shininess"`	integer	10	exponent of specular reflection *sp*

As a rule of thumb, except for purposes of special effects, k_a should be *1* and $k_d + k_s$ should be about 1. The larger k_s, the brighter the highlight, and the larger *e*, the sharper the highlight. The Shade module provides a shortcut for setting rendering properties.

The attributes listed above apply to both the front and back of an object. In addition, for each attribute "*x*" there is also a "front *x*" and a "back *x*" attribute that applies only to the front and back of the surface, respectively. So, for example, to disable specular reflections from the back surfaces of an object, use the Options module to set the "back specular" attribute of the object to 0.

The determination of which faces are "front" and which are "back" depends on the way in which the "connections" component of the faces is defined. "Front colors" applies to clockwise faces, and "back colors" applies to counterclockwise faces.

Coloring Objects for Volume Rendering: The volume renderer interprets colors and opacities as values per unit distance. Thus the amount of color and degree of attenuation seen in an image object is determined in part by the extent of the object's volume. The Color, AutoColor, and AutoGrayScale modules attach "color multiplier" and "opacity multiplier" attributes to the object so that colors and opacities will be appropriate to the volume, while maintaining "color" and "opacity" components that range from 0 to 1 (so that objects derived from the colored volume, such as glyphs and boundaries, are colored correctly). See "Rendering Model" on page 153 in *IBM Visualization Data Explorer Programmer's Reference*.

These attributes adjust the colors and opacities to values that should be "appropriate" for the object being colored. However, if the simple heuristics used by these modules to compute the attribute values are not producing the desired colors and opacities, you have two alternatives.

- One is to modify the result by changing the multiplier values of the color and opacity attributes:
  1. extract the "color multiplier" and "opacity multiplier" with the Attribute module;
  2. modify them with the Compute module; and
  3. replace them in the object with the Options module.
- A second is to multiply the values in the "color" or "opacities" component:
  1. mark the component ("colors" or "opacities") with the Mark module;
  2. modify the values with the Compute module; and
  3. "unmark" them with the Unmark module to return them to the appropriate component.

Only the first of these methods should be used for "delayed" colors.

Finally, if you color a group of volumes and the resulting image is black, the reason is that the current renderer does not support coincident volumes.

Attribute	Type	Description
color multiplier	scalar	Multiplies values in the "color" component
opacity multiplier	scalar	Multiplies values in the "opacity" component

Shading: Objects are shaded when rendered only if a “normals” component is present. Many modules (e.g. Isosurface) automatically add “normals”, but the FaceNormals, Normals, and Shade modules can also be used to add normals. Even if an object has “normals”, shading can be disabled by adding a **shade** with a value of 0 (the Shade module can do this).

Attribute Name	Values	Description
shade	0 or 1	used to specify whether or not to shade when normals are present

Object fuzz: *Object fuzz* is a method of resolving conflicts between objects at the same distance from the camera. For example, it may be desirable to define a set of lines coincident with a plane. Normally it will be unclear which object is to be displayed in front. In addition, single-pixel lines are inherently inaccurate (i.e. they deviate from the actual geometric line) by as much as one-half pixel; when displayed against a sloping surface, this *x* or *y* inaccuracy is equivalent to a *z* inaccuracy related to the slope of the surface. The “fuzz” attribute specifies a *z* value that will be added to the object before it is compared with other objects in the scene, thus resolving this problem. The fuzz value is specified in pixels. For example, a fuzz value of one pixel can compensate for the described half-pixel inaccuracy when the line is displayed against a surface with a slope of two.

Attribute	Type	Description
fuzz	scalar	object fuzz

To add fuzz to an object, pass the object through the Options module, specifying the attribute as **fuzz** and the value of the attribute as the number of pixels (typically a small integer).

Anti-aliasing and Multiple Pixel Width Lines: Hardware rendered images can be made to anti-alias lines, or draw lines as multiple pixels wide. Note that these options are *not* available in software rendering. To specify anti-aliasing of lines, use the Options module to set an attribute on the object passed to Display of **antialias** with the value of “lines”. To specify multiple pixel width lines, use the Options module to set an attribute of **line width** with a value of the number of pixels wide you want the line to be.

Attribute	Values	Description
antialias	“lines”	causes lines to be anti-aliased
line width	n	causes lines to be drawn with a width of n pixels

Rendering Approximations: Data Explorer provides access to the hardware accelerators on the workstation, in addition to the default software rendering techniques. The hardware enhancements are available only on workstations that are equipped with 3-D graphic adapters. On systems without such adapters, only the software rendering options are available. This enhancement is intended to provide increased interactivity, especially in operations that involve only the rendering process.

Data Explorer can also provide accelerated rendering by approximating the rendering using points, lines, and opaque surfaces. Such geometric elements are often sufficient to approximate the appearance of the desired image, and thus are useful for preliminary visualizations of the data.

The approximations fall into three main categories: bounding box, dots, and wireframe. Wireframe is available only as a hardware rendering technique.

If you are using the graphical user interface and the Image tool, you can access the rendering options by using the **Rendering Options** option on the **Options** pull-down menu in the Image window. This option invokes a dialog box that allows you to set the rendering approximations for continuous and one-time execution. (For more information, see "Rendering Options..." on page 91 in *IBM Visualization Data Explorer User's Guide*.)

If you are not using the Image tool, then you must use the Options module to set various attributes that control the rendering approximations. The following table lists the attributes that control rendering approximations, together with the permissible values for each attribute.

Attribute Name	Values	Description
"rendering mode"	"software" "hardware"	use software rendering use hardware rendering
"rendering approximation"	"none" "box" "dots" "wireframe"	complete rendering object bounding box only dot approximation to object wireframe approximation to object
"render every"	*n*	render every *n*th primitive render everything (default)

Note: If you do not pass a camera to Display (i.e., if **object** is already an image), Display will always use software to display the image, regardless of the setting of any rendering options using the Options tool.

Differences between Hardware and Software Rendering

1. For hardware rendering, when specifying "dots" for "rendering approximation," lines are drawn in their entirety, whereas for software rendering only the line end points are drawn. The "render every" and "wire" approximations are available only with hardware rendering. When the "box" approximation is specified, hardware rendering will show the bounding box of each field in the rendered object, while software rendering will show only the bounding box of the entire object.
2. Some graphics adapters do not support clipping. On such adapters, "ClipBox" and "ClipPlane" have no effect.
3. For some hardware platforms, surfaces specified with opacities are rendered by the hardware as screen-door surfaces (i.e., every other pixel is drawn, letting the background show through). This allows only one level of opacity and completely obscures a semi-opaque surface that is behind another semi-opaque surface. The transparency effect is hardware dependent, and can produce a completely opaque or completely transparent appearance. True transparency is supported for OpenGL platforms.

4. The image displayed by the hardware rendering can be different from the image produced by the software rendering. This is a result of several differences in rendering techniques. The hardware rendering does not provide gamma correction, causing images to be slightly darker. Normals are not reversed when viewing the “inside” of a surface, with the result that lighting effects are much dimmer on the “inside” of a surface. Attributes applied to the “inside” of a surface (e.g., “back colors”) are ignored.
5. When using hardware rendering, the `where` parameter to Display cannot specify a host other than the one on which the Display module is running. However, it can specify a different display attached to the same host.
6. The hardware renderer does not duplicate the “dense emitter” model used by the software renderer for rendering volumes. Only the data values at the boundary of the volume are rendered, producing the appearance of a transparent boundary of the volume.
7. For hardware rendering, a wireframe rendering approximation is not intended to produce the same visual results as ShowConnections.
8. Hardware rendering handles colors between 0.0 and 1.0. If colors are outside this range, each color channel is independently clamped, before lighting is applied. In software rendering, clamping is done after lighting is applied.
9. Hardware rendering does not support view angles of less than 0.001 degree.
10. Anti-aliasing and multiple pixels width lines is only available in hardware rendering.

Texture Mapping: If the machine on which Data Explorer is running supports OpenGL or GL, then texture mapping is available using hardware rendering. Texture mapping is the process of mapping an image (a field with 2-dimensional positions, quad connections, and colors) onto a geometry field with 2-dimensional connections and, typically, 3-dimensional positions (e.g., a color image mapped onto a rubbersheeted height field). The advantage of texture mapping over the use of Map, for example, is that the resulting image may have much greater resolution than the height map.

The geometry field must have 2-D connections (triangles or quads) and must also have a component, with the name “uv,” that is dependent on positions and provides the mapping between the image and the positions of the geometry field. This component consists of 2-vectors. The origin of the image will be mapped to the uv value [0 0], and the opposite corner to the uv value [1 1].

The texture map is the image to be mapped onto the geometry field. One requirement of the image (imposed by the hardware) is that the number of pixels in each dimension must be a power of 2 greater than or equal to 32. The two dimensions do not have to be the same size. The texture map should be attached to the geometry field as an attribute, with the attribute name “texture,” which can be done with the Options module. A texture-mapped image can be retrieved from the Display window using ReadImageWindow and then written to a file using WriteImage.

Attribute Name	**Value**	**Description**
texture	a texture map	specifies a texture map

Components

The **object** input must have a “colors,” “front colors,” or “back colors” component.

Script Language Examples

1. This example renders two views of the object and displays them in two separate windows, as specified by the **where** parameter.

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
isosurface = Isosurface(electrondensity, 0.3);
camera1 = AutoCamera(isosurface, "front", resolution=300);
camera2 = AutoCamera(isosurface, "top", resolution=300);
image1 = Render(isosurface, camera1);
image2 = Render(isosurface, camera2);
Display(image1,where="X, localhost:0, view from front");
Display(image2,where="X, localhost:0, view from top");
```

2. This example sets the rendering mode to “hardware” with the approximation method of “dots.”

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
isosurface = Isosurface(electrondensity, 0.3);
from = Direction(65, 5, 10);
camera = AutoCamera(isosurface, from);
isosurface=Options(isosurface, "rendering mode", "hardware",
                  "rendering approximation", "dots");
Display(isosurface,camera);
```

Example Visual Programs

```
MovingCamera.net
PlotLine.net
PlotTwoLines.net
ReadImage.net
ScaleScreen.net
TextureMapOpenGL.net
UsingCompute.net
UsingMorph.net
```

See Also

Arrange, Collect, Filter, Image, Render, Reduce, Refine, ScaleScreen, Normals, FaceNormals, SuperviseWindow, SuperviseState, ReadImageWindow, Options

DivCurl

Category

Transformation

Function

Computes the divergence and curl of a vector field.

Syntax

```
div, curl = DivCurl(data, method);
```

Inputs

Name	Type	Default	Description
`data`	vector field	none	field whose divergence and curl are to be computed.
`method`	string	"manhattan"	method to use

Outputs

Name	Type	Description
`div`	scalar field	divergence field
`curl`	vector field	curl field

Functional Details

`data`	is the vector field to be operated on. The “data” component of the output fields (`div` and `curl`) contains the divergence and curl respectively. The other components of the original field remain unchanged in the output fields.
`method`	specifies the algorithm used to make the computation. Currently, the only method supported is “manhattan.”

Components

Modifies the “data” component for the outputs. All other input components are propagated to the outputs.

Example Visual Program

`WindVorticity.net`

See Also

Gradient, Streakline, Streamline

Done

Category

Flow Control

Function

Specifies whether a loop is to be terminated.

Syntax

Done(**done**);

Inputs

Name	Type	Default	Description
done	flag	none	0: no termination 1: terminate the loop

Functional Details

This module terminates a loop. If **done** is set to:

0 The macro containing the Done module will continue to execute until **done** = 1 or until one of the other looping tools (ForEachN or ForEachMember) terminates the loop.

1 The loop is terminated upon completion of the current loop iteration, regardless of how many iterations have been, or remain to be, executed.

Typically, loops are initiated with ForEachMember or ForEachN, although they can also be implemented with a Get/Set pair and Done.

Notes:

1. If this module is used in the scripting language, the results are defined only if it is used inside a macro.
2. If **done** is set to 1 by an interactor *during* the execution of a loop, the new setting will not take effect until the loop is completed: new interactor values are considered at the end of an execution, and a complete loop is considered to be a *single* execution.
3. Simply placing a Done icon in a visual program and setting the parameter **done** to 0 (zero) will create an infinite loop.

Example Visual Programs

```
Bounce.net
SimpleGetSetLoop.net
```

See Also

First, ForEachMember, ForEachN, GetGlobal, GetLocal, SetGlobal, SetLocal

Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

Echo

Category

Debugging

Function

Echoes a message.

Syntax

```
Echo(string, ...);
```

Inputs

Name	Type	Default	Description
string	value list or string list	no default	message to be printed
...			additional messages to print

Functional Details

This module prints its arguments as a string.

string	is the value to be echoed.

A single Echo module can echo a maximum of 21 strings. In the user interface, the default number of enabled **string** tabs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

The module can print:

- Integer, float, and string values
- Integer, float, and string lists
- 2-D and 3-D vectors.

If the input is of another type, the module will echo the object class (e.g, “Field”).

In the user interface, the output of Echo appears in the Message window.

See Also

Print, Describe

Enumerate

Category

Realization

Function

Generates a numeric list.

Syntax

```
list = Enumerate(start, end, count, delta, method);
```

Inputs

Name	Type	Default	Description
start	scalar or vector	input dependent	first value in list
end	scalar or vector	input dependent	last value in list
count	integer	input dependent	number of entries in list
delta	scalar or vector	input dependent	numeric spacing
method	string	"linear"	method

Outputs

Name	Type	Description
list	value list	the numeric list

Functional Details

This module creates a list of numeric values.

start	specifies the first value in the list.
end	specifies the last value in the list.
count	specifies the number of items in the list.
delta	specifies the numerical spacing between successive items in the list. If **start** and **end** are vector values and this parameter is specified, it must be specified as a matching vector.
method	specifies the type of list to be created. At present, the specification must be “linear.” The list created is a linear sequence of numbers from **start** to **end**, containing **count** items or containing items separated by a spacing interval of **delta**.

Note: Only three of the first four parameters are required. If all four are specified, **delta** is ignored.

Example Visual Program

`ContoursAndCaption.net`

Equalize

Category

Transformation

Function

Applies histogram equalization to a field.

Syntax

equalized = Equalize(**data, bins, min, max, ihist, ohist**);

Inputs

Name	Type	Default	Description
data	scalar field	none	data to be equalized
bins	integer	input dependent	number of equalization bins
min	scalar or field	min of data	lower bound of equalization
max	scalar or field	max of data	upper bound of equalization
ihist	field	histogram of data	input distribution
ohist	field	uniform distribution	output distribution

Outputs

Name	Type	Description
equalized	scalar field	histogram-equalized data

Functional Details

This module equalizes an input data field so that the histogram of the output approximates a specified distribution (**ohist**), which by default is a uniform distribution.

If **data** specifies a series, the histogram of the entire series is used to construct the probability distribution, which is then applied to each field in the series.

data	is the scalar data field to be equalized.
bins	is the number of bins to be used in creating the equalization histogram. The default value is 100, unless **data** consists of byte values. In that case, the default is: **max** – **min** + 1

min and **max** specify the range of the data values to be equalized. Values outside that range remain unchanged.

Notes:

1. If neither parameter is specified, the values used are the minimum and maximum of the input field.
2. If **min** is a scalar value, it is the minimum value equalized.
3. If **min** is a data field, the minimum data value of that field is used.
4. **max** is similarly interpreted.
5. If **min** is a data field and **max** is unspecified, the module uses the minimum and maximum values of that field.

ihist is the histogram used to determine the equalization function required, and by default it is the histogram of **data**. If **ihist** is specified explicitly and the number of bins specified does not match the number in the histogram, the module resamples the histogram so that it contains the specified number.

ohist is the type of output histogram desired. By default, the module equalizes to a uniform distribution. You can equalize to some other distribution by specifying an appropriate histogram for this parameter.

Note: A well-formed histogram (for **ihist** and **ohist**) consists of a field with a "positions" component that defines the bins of the histogram; a "connections" component that connects the positions; and a "data" component that contains the number of items in each bin. The data should be connection dependent ("dep" "connections").

Components

Equalize modifies the data component. All other components are unmodified.

Example Visual Programs

`UsingEqualize.net`

See Also

Filter, Morph, Histogram

Execute

Category

Flow Control

Function

Allows the user to change the execution state of a visual program without using the **Execute** menu.

Syntax

Available only in the user interface.

Inputs

Name	Type	Default	Description
action	string	"end"	operation to be executed

Functional Details

The **action** parameter takes any one of three arguments: “once,” “on change,” or “end.” Note that “end” does not turn off the sequencer.

See Also

ManageColormapEditor, ManageControlPanel, ManageImageWindow, ManageSequencer

Executive

Category

Interface Control

Function

Executes an executive command.

Syntax

```
Executive(command, value);
```

Inputs

Name	Type	Default	Description
command	string	none	command to be executed
value	object	no default	command-dependent value

Functional Details

This module tells the Data Explorer executive to run an internal system command. In the user interface, it is available only through the **Execute Script Command** option of the **Options** pull-down menu of the Message window. And with the exception of the "print" command, it should be invoked only through the scripting language.

command	is the executive command to be executed.
value	is a **command**-dependent value.

The types of commands executed by the Executive module fall into four categories:

- printing system information
- distributed processing
- module definition
- system functions.

Printing System Information: The following list summarizes the kinds of system information available with the "print" command.

- Version information for Data Explorer:

  ```
  Executive("print version");
  ```

- Information about the system environment:

  ```
  Executive("print env");
  ```

- A module definition:

  ```
  Executive("print mdf","modulename");
  ```

- A list of the execution groups:

  ```
  Executive("print groups");
  ```

- A list of the attached hosts:

  ```
  Executive("print hosts");
  ```

Distributed Processing in Script Language: Use the following commands to control distributed processing in the script language. (In the user interface, distributed processing should be controlled through the **Execution Group Assignment...** option of the **Connections** pull-down menu in the VPE window. See 9.1, "Using Distributed Computation" on page 178 in *IBM Visualization Data Explorer User's Guide*.)

- To attach an execution group to a host requires the name of the host machine and any options to be used when Data Explorer is started on that host. Data Explorer establishes communication with the named host via a TCP/IP socket connection. The syntax for attachment is:

  ```
  Executive("group attach",  "group1:hostname -dxopt1 -dxopt2 ...");
  $sync
  ```

  or

  ```
  Executive("group attach", { "group1:hostname -dxopt1 -dxopt2 ...",
                              "group2:hostname -dxopt1 -dxopt2 ...",
                              "group3:hostname -dxopt1 -dxopt2 ...", ...});
  $sync
  ```

- Detaching an execution group from a host requires only the group's name in the command:

  ```
  Executive("group detach", "group1");
  $sync
  ```

  or

  ```
  Executive("group detach", { "group1", "group2", ... });
  $sync
  ```

- The "host disconnect" command terminates the connection to a host:

  ```
  Executive("host disconnect", "hostname1");
  $sync
  ```

  or

  ```
  Executive("host disconnect", { "hostname1", "hostname2", ... });
  $sync
  ```

Note: These Executive calls must be followed by `$sync`, as shown.

Module Definition: To load the module definition file (mdf) of an outboard module after Data Explorer has been started, use the script syntax shown here. (In the user interface, use the **Load Module Description(s)...** option of the **File** pull-down menu in the VPE or Image window.)

```
Executive("mdf file", "mdf_filename");
$sync
```

Note: The call to load the mdf file must be followed by `$sync`, as shown.

UserInteractor Definition To load a UserInteractor after Data Explorer has been started, use the following script syntax:

```
Executive("loadinteractors", "userinteractor_filename");
$sync
```

Note: In the User Interface this can be done using the **Execute Script Command** option in the **Commands** menu of the Message Window.

System Functions: The following commands execute system functions:

- To flush all variables in the internal cache:
  ```
  Executive("flush cache");
  ```
- To flush all variables in the system dictionary:
  ```
  Executive("flush dictionary");
  ```
- To flush all macro definitions from the system:
  ```
  Executive("flush macros");
  ```

Example

In this example, the execution groups “mine” and “yours” are attached to hosts “ours” and “theirs” respectively. Also, Data Explorer is started with a memory allocation of 50MB on the host “theirs.”

```
Executive("group attach",{"mine:ours","yours:theirs -memory 50"});
```

Export

Category

Import and Export

Function

Writes an external data file.

Syntax

```
Export(object, name, format);
```

Inputs

Name	Type	Default	Description
object	object	none	object to write
name	string	none	file name to write to
format	string	"dx"	format in which to write the file

Functional Details

The Export module writes **object** to the file **name** in the specified **format**.

You can specify **format** as `dx`, `array`, or `vrml`.

If **format** = `dx`, this specification can be followed by one or more of the following modifiers:

byteorder Can be “msb” or “lsb” for most significant byte first or least significant byte first, respectively.

dxformat Can be “ieee,” or “text” for IEEE floating point or ASCII format, respectively. You can also specify “ascii” as a synonym for “text”, and “binary” as a synonym for “ieee”.

where Can be one of the following three keywords:

“follows” The data for each object follows the header for that object.

“1” The data for each object is contained in a separate data section in the file.

“2” The data for each object is contained in a separate file.

For “follows” and “1,” the module writes a single file with the file name (*name*.`dx`). For “2,” the module writes files. The object headers are written to *name*.`dx`; the data are written to *name*.`bin` (binary data) or *name*.`data` (text data). For more information, see Appendix B, “Importing Data: File Formats” on page 241 in *IBM Visualization Data Explorer User’s Guide*.

If **format** = **“**`array`**,”** this specification can be followed by one or both of the following modifiers:

quotes Puts quotation marks around string data.

headers	Puts a header of descriptive information at the top of the file.

The data are written out in ASCII spreadsheet format (in columns). Only position-dependent data are supported. All position-dependent data are written to the output file. The first column will contain the positions themselves.

If **object** contains a group of fields, these are written out one after the other.

If **format** = “vrml”, or the extension of the filename is .wrl, then a VRML 2.0 format file will be written. Because VRML files are written out in ASCII they can be very large, so you should reduce the number of polygons before exporting if possible, for example by using SimplifySurface. The object exported must be a field or group with two- or three-dimensional positions. Data Explorer objects are exported using VRML geometry nodes IndexFaceSet, IndexLineSet, PointSet, or ElevationGrid depending on the connections element type and regularity of positions. Colors and normals are exported with each object, if they exist.

Example Visual Program

```
UsingSwitchAndRoute.net
SimplifySurface.net
```

See Also

Import, SimplifySurface

Extract

Category

Structuring

Function

Extracts a component from a field.

Syntax

```
output = Extract(input, name);
```

Inputs

Name	Type	Default	Description
input	field	none	the field from which a component is to be extracted
name	string	"data"	the component to extract

Outputs

Name	Type	Description
output	object	the named component

Functional Details

This module creates an **output** object containing only the **name** component from the **input** field. If **input** is a group, **output** is a group of array objects.

Components

Only the components extracted are propagated to the output.

Example Visual Programs

```
MultipleDataSets.net
PlotTwoLines.net
UsingTextAndTextGlyphs.net
```

See Also

Mark, Remove, Rename, Replace, Unmark

FFT

Category

Transformation

Function

Computes a fast Fourier transform.

Syntax

```
output = FFT(input, direction, center);
```

Inputs

Name	Type	Default	Description
`input`	field	none	field to be transformed
`direction`	string	"forward"	direction of the transform
`center`	flag	0	center the result of the transform

Outputs

Name	Type	Description
`output`	field	transformed data

Functional Details

This module computes the fast Fourier transform of a 2- or 3-dimensional regular data set.

`input`	specifies the field to be transformed.
`direction`	is one of the following: “forward,” “inverse,” or “backward” (the last two are interchangeable).
`center`	specifies whether zero frequency should be placed at the center of the transformed field or at the origin of the positions array.

Note: This module requires that the number of data items in each dimension be a power of 2. If it is not, use the DFT module.

Components

All scalar components of **`input`** are individually Fourier-transformed and output as complex float. All other input components are propagated to the output. Thus a float 2-vector input produces a complex 2-vector output.

Example Visual Programs

`FFT.net`

See Also

DFT, Filter, Morph

FaceNormals

Category

Rendering

Function

Computes face normals for flat shading.

Syntax

```
normals = FaceNormals(surface);
```

Inputs

Name	Type	Default	Description
`surface`	geometry field	none	surface on which to compute face normals

Outputs

Name	Type	Description
`normals`	field	the surface with face normals

Functional Details

This module computes face normals for a surface. A “normals” component is necessary to produce shading when an object is rendered. In contrast to the output of the Normals module, the normals produced by this module are always connection dependent rather than position dependent. Therefore, when they are rendered, the result is faceted shading.

When this module is used, any colors that reside at vertices are averaged for each face at the time of rendering, resulting in a single color for each face.

The FaceNormals module assumes that the triangles or quads have consistent point orderings (i.e., so that the directions of the face normals will be consistent throughout the mesh).

The Shade module can also be used to compute normals for shading.

Components

Creates a “normals” component that is dependent on connections. All other components are propagated to the output.

Example Visual Program

```
Thunder_cellcentered.net
SIMPLE/FaceNormals.net
```

See Also

Isosurface, Normals, Shade

FileSelector

Category

Interactor

Function

Produces file names as outputs.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`filename`	string	full path name of selected file
`basename`	string	name of file (not including directory name)

Functional Details

This module can be used whenever a file name is needed (e.g., as input to the Import or ReadImage tools). It generates two output strings, both of which are file names:

The first (**`filename`**) is the complete path to the file.

The second (**`basename`**) is the file name, without the path. (For more information, see 7.1, "Using Control Panels and Interactors" on page 128 in *IBM Visualization Data Explorer User's Guide*.)

Notes:

1. **`FileSelector`** cannot be data driven.
2. The **`FileSelector`** dialog box can access only those files that are accessible to the user interface.

Example Visual Program

`MultipleDataSets.net`

See Also

Integer, IntegerList, Scalar, ScalarList, Selector, String, StringList, Value, ValueList, Vector, VectorList

Filter

Category

Transformation

Function

Filters a field.

Syntax

```
output = Filter(input, filter, component, mask);
```

Inputs

Name	Type	Default	Description
input	field	none	data to be filtered
filter	value or string	"gaussian"	filter to be used
component	string	"data"	component to be operated on
mask	value or string	"box"	rank-value filter mask

Outputs

Name	Type	Description
output	field	filtered data

Functional Details

The specified filter can be convolution or rank-value type.

Convolution filters, as the name implies, perform a convolution of the filter coefficients with the input data (i.e., the output value at a given point is the sum of the product of the filter coefficients and the corresponding values neighboring the point in question).

Rank-value filters sort all of the elements under a mask surrounding the point in question and either select or interpolate the value specified by the given rank. For “min,” “max,” and “median” (special cases of rank-value filters), the rank values are respectively *1*, *n*, and (*n* + *1*)/2, where *n* is the number of nonzero elements in the mask.

Convolution filters are useful for neighborhood smoothing, edge detection, and other gradient-based operations. Rank-value filters are useful for random-noise removal and morphological operations.

input is the object to be filtered. Each field containing the component to be filtered must also contain both a “positions” and a “connections” component. The “connections” component must be regular.

filter specifies, by name, the filter to be used on **input**. See Table 2 on page 139 for valid names.

component	specifies the field component to be filtered. By default, Filter operates on “data.” To filter an image, this parameter should specify “colors.”
	If the component to be filtered contains several channels (e.g., red, green, and blue in an image), each channel is filtered independently.

Table 2 on page 139 defines the filters and masks that can be specified by name. While most of these are 2-dimensional, some are also available in 1- and 3-dimensional versions (as indicated by “1d” or “3d” after their names). If possible, the module selects a filter that matches the dimensionality of **input**. However, if a filter of lower dimensionality is available, it can (and must) be specified by appending “1d” or “2d” to the name. That is, to specify a filter of lower dimensionality, the specification must be explicit.

Table 2. Filter Names. The names listed here are all valid specifications for the `filter` *parameter. The specification is not case sensitive.*

Filter Name	Description
4-connected	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} / 5$
8-connected	$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} / 9 =$ *box*:2*d*
6-connected	3-D analog of 4-connected
26-connected	3-D analog of 8-connected
box	Box filter
box:1d	
box:2d	
box:3d	
compass:e	
compass:n	
compass:ne	
compass:nw	
compass:s	
compass:se	
compass:sw	
compass:w	
gaussian	Same as gaussian:3x3
gaussian:2d	Same as gaussian:3x3; can be used to force a 2-D gaussian to be applied to 3-D data.
gaussian:3x3	3 × 3 Gaussian, $\sigma = 1.0$
gaussian:5x5	5 × 5 Gaussian, $\sigma = 1.0$
gaussian:7x7	7 × 7 Gaussian, $\sigma = 1.0$
isotropic	
kirsch	
laplacian	Same as laplacian:2d
laplacian:1d	
laplacian:2d	
laplacian:3d	
line:e-w	
line:n-s	
line:ne-sw	
line:nw-se	
prewitt	
roberts	
smoothed	Same as prewitt
sobel	

Notes:

1. When a lower-dimensional filter is applied to higher-dimensional input, the input is separated into lower-dimensional units that are filtered and then reassembled into a higher-dimensional structure. For example, a 2-dimensional filter applied to a 3-dimensional field will result in individual slices of the input being filtered and then restacked.
2. Arbitrary filter kernels and masks may also be specified as matrices.
3. If the value specified by `filter` is a matrix, the module performs convolution filtering and uses the values given as the filter coefficients. Filters must have odd dimensions (e.g., 5 × 5), since the active point is defined as the central point in the filter.

4. If the value specified in **filter** is a scalar or one of the strings “min,” “median,” or “max,” the module performs rank-value filtering and uses the value of **mask** for sorting the elements.

   Masks, like filters, must have odd dimensions (e.g., 5 × 5). When specifying a mask, remember that nonzero elements in the mask matrix signify inclusion; zeros signify exclusion. In this way, the built-in matrices (e.g., “box”) can be used interchangeably as filters or masks.

5. Data along the boundary are replicated to fill the overlap region for the filter.

Components

Modifies the component specified by **component**. All other input components are propagated to the output.

Example Visual Program

`UsingFilter.net`

See Also

Compute, Morph

First

Category

Flow Control

Function

Indicates whether the current iteration is the first iteration of a loop.

Syntax

This module has no inputs.

Outputs

Name	Type	Description
`first`	flag	Is this the first iteration of the loop?

Functional Details

The output on the first iteration of a loop is 1 (one), and on all subsequent iterations 0 (zero).

Typically, loops are initiated with ForEachMember or ForEachN, although they can also be implemented with a Get/Set pair and Done. First is often useful for resetting the GetGlobal tool at the beginning of a loop. Note that if GetLocal is used in a loop, First is unnecessary.

Note: If this module is used in the scripting language, the results are defined only if they are used inside a macro.

See Also

Done, ForEachMember, ForEachN, GetGlobal, GetLocal, SetGlobal, SetLocal

Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

ForEachMember

Category

Flow Control

Function

Iterates through the members of a group or the items of an array.

Syntax

```
member, index, last = ForEachMember(object);
```

Inputs

Name	Type	Default	Description
`object`	group, value list, or string list	(none)	object to be iterated through

Outputs

Name	Type	Description
`member`	object	current member
`index`	integer	index number of member
`last`	flag	status of loop

Functional Details

This module initiates an iteration for each member of a group or item in an array unless execution is terminated earlier by Done. In the user interface, this module would usually be part of a macro, and *only* the contents of the macro would be executed for each iteration of the loop. If this module is placed in the top level visual program, the entire program will be executed during the loop.

Typically, loops are initiated with ForEachMember or ForEachN, although they can also be implemented with a Get/Set pair and Done.

Note: If this module is used in the scripting language, the results are defined only if they are used inside a macro.

`object`	is the object to be iterated over: either a group or a list.

This module has three outputs:

`member`	identifies the member currently being iterated.
`index`	is the index number of the member being iterated (in a zero-based counting system).
`last`	is a flag indicating whether or not this is the last iteration through the loop.

Example Visual Program

```
Accumulate.net
```

See Also

Done, First, ForEachN, GetGlobal, GetLocal, SetGlobal, SetLocal

Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

ForEachN

Category

Flow Control

Function

Iterates through the specified set of integers.

Syntax

```
current, last = ForEachN(start, end, delta);
```

Inputs

Name	Type	Default	Description
start	integer	none	value of the first integer through the loop
end	integer	none	value of the last integer through the loop
delta	integer	1	numerical interval between successive integers output by the loop

Outputs

Name	Type	Description
current	integer	the current integer
last	flag	status of loop

Functional Details

This module initiates an iteration for each integer in the set unless execution is terminated earlier by Done. Its function is similar to that of a "for" in standard programming languages. In the user interface, this module would usually be part of a macro, and *only* the contents of the macro would be executed for each iteration of the loop. If this module is placed in the top level visual program, the entire program will be executed during the loop.

Typically, loops are initiated with ForEachMember or ForEachN, although they can also be implemented with a Get/Set pair and Done.

Note: If this module is used in the scripting language, the results are defined only if they are used inside a macro.

start	is the value of the first integer of the loop
end	is the value of the last integer of the loop
delta	is the interval between successive integers of the loop

This module has two outputs:

current	is the value of the integer currently being iterated.

`last`	is a flag indicating whether or not this is the last iteration through the loop.

Example Macro and Program

```
FactorialMacro.net
Factorial.net
```

See Also

Done, First, ForEachMember, GetGlobal, GetLocal, SetGlobal, SetLocal

Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

Format

Category

Annotation

Function

Formats a string.

Syntax

string = Format(**template, value, ...**);

Inputs

Name	Type	Default	Description
template	string	none	format control string
value	value list or string	no default	value to format
...			more values to format

Outputs

Name	Type	Description
string	string	formatted string object

Functional Details

This module uses a format-control string and input values to create a formatted output string. Each insertion is matched to a format-control specification, which always begins with a % symbol. Strings, scalars, integers, and vectors can be formed into output strings.

template is the format-control string used in creating the formatted output. It resembles a C-language `printf` format string.

Data Explorer copies all characters other than format specifications to the output (see next parameter).

value is the value to be placed in the output string. The character following the % controls the type of conversion:

- **c** single characters
- **d** integers
- **f** floating point (fixed number of digits following the decimal point)
- **g** general (scientific notation if appropriate)
- **s** strings.

Note: **f** and **g** will also format 2- and 3-vectors and lists.

To output a %, use %%. You can insert numbers between the % and the conversion character to control the width of the field and the number of significant digits formatted. A minus sign (–) left-justifies the output in the field; the default is right-justification. A format control string of

```
"%-10.4f"
```

indicates left justification of a floating-point number, minimum of 10 columns total, with 4 places to the right of the decimal point.

A single call to the Format module can format a maximum of 21 values. In the user interface, the default number of enabled **value** tabs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Script Language Examples

1. In this example, *value1* is set to 32.567799

   ```
   value1 = 32.567799;
   ```

2. This outputs the string “number = 32.567799.”

   ```
   output = Format("number = %f", value1);
   Echo(output);
   ```

3. This example outputs the string “number = 32.57.”

   ```
   output = Format("number = %3.2f", value1);
   Echo(output);
   ```

4. This example outputs the string “number = 32.57.”

   ```
   output = Format("number = %11.2f", value1);
   Echo(output);
   ```

5. In this example, *value2* is set to 134569888 and *value3* is set to “New York.”

   ```
   value2 = 134569888;
   value3 = "New York";
   ```

6. This example outputs the string
   “number = 134569888.000000, state = New York.”

   ```
   output = Format("number = %f, state = %s", value2, value3);
   Echo(output);
   ```

7. This example outputs the string “number = 134569888, state = New York.”

   ```
   output = Format("number = %10.0f, state = %s", value2, value3);
   Echo(output);
   ```

8. This example outputs the string “number = 1.3457e+08, state = New York.”

   ```
   output = Format("number = %g, state = %s", value2, value3);
   Echo(output);
   ```

9. This example outputs the string
   “number = 1.3457e+08, state = New York.”

   ```
   output = Format("number = %g, state = %15s", value2, value3);
   Echo(output);
   ```

Example Visual Programs

```
ContoursAndCaption.net
GeneralImport1.net
FormatListMacro.net
InvalidData.net
PlotTwoLines.net
SalesOnStates.net
Sealevel.net
```

See Also

Caption, Echo, Text

GetGlobal

Category

Flow Control

Function

Retrieves an object from the cache. State is maintained between executions of any macros containing GetGlobal.

Syntax

object, link = GetGlobal(**object, reset**);

Inputs

Name	Type	Default	Description
object	object	none	object to be output if nothing has been set
reset	flag	0	0: the object cached by SetGlobal (if there is one) is passed to output 1: **object** (not the cached object) is passed to output.

Outputs

Name	Type	Description
object	object	retrieved object
link	string	link to corresponding SetGlobal module

Functional Details

GetGlobal works with SetGlobal to place objects in and retrieve them from the cache. GetGlobal is equivalent to (and replaces) Get in previous versions of Data Explorer. GetGlobal differs from GetLocal in that GetLocal and SetLocal are used when the state maintained by the Get/Set pair should be reset when a macro containing these modules is reexecuted. In contrast, GetGlobal and SetGlobal will maintain state when the macro is reexecuted. Note that for a single execution of a macro (for example, throughout the execution of an entire loop), state is of course maintained by both GetLocal and GetGlobal.

object	specifies the object to be output by GetGlobal if nothing has been placed in the cache (e.g., as on the first execution of a visual program) or if **reset** = 1.
reset	causes the module to output **object**. If this parameter is set to 0 (zero), GetGlobal retrieves the last object placed in the cache by SetGlobal (if there is one). Otherwise, the module passes **object** to the output.

The **link** output is to be connected to the **link** input of the corresponding SetGlobal module. GetGlobal must be used with SetGlobal, rather than with

SetLocal. SetGlobal must be executed on the same machine as GetGlobal (i.e., it cannot be distributed to a different machine).

Notes:

1. The Reset interactor can be used to provide `reset`. However, if you are using GetGlobal and SetGlobal in a loop, you should not use the Reset interactor to provide this parameter, because the Reset interactor will output the reset value for one execution, which is an *entire* execution of the loop. In general, there are performance advantages to using GetLocal, rather than GetGlobal with the First module supplying the `reset` parameter. While the result will be the same, using GetLocal will ensure that all previous results of the macro will be cached and ready for reuse. If you use GetGlobal, only the last result of the macro is cached.
2. Whenever there is a GetGlobal inside a macro, all outputs of the macro will be stored in the cache with the cache attribute “cache last”. In other words, whenever any input to the macro changes, the old results of the macro will be deleted from the cache and the new results will be stored in the cache. There is no way for the user to turn off caching for this macro. The results are also locked in the cache, and cannot be flushed. This is because GetGlobal maintains state for the macro that may not be reproduced given the inputs to the macro. Inconsistent behavior might result if results were not cached in this way.
3. Previously created visual programs which use Get and Set will be run using GetGlobal and SetGlobal. You can explicitly change your Gets and Sets to either GetGlobals and SetGlobals or GetLocals and SetLocals using the Edit menu (see Assign Get/Set Scope on page 158 in *IBM Visualization Data Explorer User’s Guide*). There are performance advantages to using GetLocal and SetLocal whenever you do not need to maintain state between executions of your macro which uses Gets and Sets. A rule of thumb is that if you are using the First module to supply the reset parameter of Get, you should use GetLocal instead of GetGlobal (and the use of First is then unnecessary).

A detailed description of the behavior and use of the GetLocal, GetGlobal, SetLocal, and SetGlobal modules can be found in 4.6, “Preserving Explicit State” on page 45 in *IBM Visualization Data Explorer User’s Guide*.

Example Visual Programs

```
SIMPLE/GetSet.net
```

See Also

Done, First, SetGlobal, GetLocal, SetLocal, Reset

Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

GetLocal

Category

Flow Control

Function

Retrieves an object from the cache. State is not maintained between executions of any macros containing GetLocal.

Syntax

`object, link` = GetLocal(**`object, reset`**);

Inputs

Name	Type	Default	Description
`object`	object	none	object to be output if nothing has been set
`reset`	flag	0	0: the object cached by SetLocal (if there is one) is passed to output 1: **`object`** (not the cached object) is passed to output.

Outputs

Name	Type	Description
`object`	object	retrieved object
`link`	string	link to corresponding SetLocal module

Functional Details

GetLocal works with SetLocal to place objects in and retrieve them from the cache. GetLocal differs from GetGlobal in that GetLocal and SetLocal are used when the state maintained by the Get/Set pair should be reset when a macro containing these modules is reexecuted. In contrast, GetGlobal and SetGlobal will maintain state when the macro is reexecuted. Note that for a single execution of a macro (for example, throughout the execution of an entire loop), state is of course maintained by both GetLocal and GetGlobal.

`object` specifies the object to be output by GetLocal if nothing has been placed in the cache (e.g., as on the first execution of a visual program) or if **`reset`** = 1.

`reset` causes the module to output **`object`**. If this parameter is set to 0 (zero), GetLocal retrieves the last object placed in the cache by SetLocal (if there is one). Otherwise, the module passes **`object`** to the output.

The **`link`** output is to be connected to the **`link`** input of the corresponding SetLocal module. GetLocal should always be used with SetLocal, not with SetGlobal. SetLocal must be executed on the same machine as GetLocal (i.e., it cannot be distributed to a different machine).

It is not necessary to use the First tool to provide input to the **`reset`** parameter of GetLocal inside a macro, since this is done automatically whenever the macro is reexecuted.

A detailed description of the behavior and use of the GetLocal, GetGlobal, SetLocal, and SetGlobal modules can be found in 4.6, "Preserving Explicit State" on page 45 in *IBM Visualization Data Explorer User's Guide*.

Example Visual Programs

```
Accumulate.net
Bounce.net
SimpleGetSetLoop.net
```

See Also

Done, First, SetLocal, GetGlobal, SetGlobal, Reset

Chapter 4, "Data Explorer Execution Model" on page 37 in *IBM Visualization Data Explorer User's Guide*.

Glyph

Category

Annotation

Function

Creates a glyph (visual representation) for each data value in a data field.

Syntax

```
glyphs = Glyph(data, type, shape, scale, ratio, min, max);
```

Inputs

Name	Type	Default	Description
data	field	none	the set of points to which glyphs will be assigned
type	scalar, string, field, or group	input dependent	glyph type
shape	scalar	1.0	factor to describe the shape of the glyph (must be greater than 0)
scale	scalar	input dependent	scale factor for size of glyphs (must be greater than 0)
ratio	scalar	0.05 or 0	ratio in size (scalars or vectors) between smallest and largest glyphs
min	scalar or field	min of data or 0	data value that gets the minimum-size glyph
max	scalar or field	max of data	data value that gets the maximum-size glyph

Outputs

Name	Type	Description
glyphs	color field	set of glyphs

Functional Details

This module creates a glyph, or representation, for each data value in a data field (**data**). For data that are dependent on positions, a glyph is placed at the corresponding position. For data that are dependent on connections, a glyph is placed at the center of the corresponding connection element.

The Glyph module differs from AutoGlyph in its interpretation of the **scale** parameter. With Glyph, you can specify a multiplication factor that is applied to each data value to obtain the size of the glyph in world units. With AutoGlyph, the scaling is relative to the default glyph size.

To create sphere glyphs with a radius equal to the data value, set **`scale`** to 1 and both **`ratio`** and **`min`** to 0. To create arrow glyphs with length equal to the magnitude of the data value, set **`scale`** to 1.

For descriptions of the parameters, see "AutoGlyph" on page 37.

Components

Creates new "positions" and "connections" components. In the case of a 3-D glyph, a "normals" component is added for shading purposes. All components that match the dependency of the "data" component are propagated to the output, all others are not. If the input has "binormals" and "tangent" components, they are not propagated to the output.

Example Visual Programs

```
GeneralImport1.net
Imide_potential.net
PlotTwoLines.net
ProbeText.net
```

See Also

AutoGlyph, Sample

Gradient

Category

Transformation

Function

Computes the gradient of a scalar field.

Syntax

```
gradient = Gradient(data, method);
```

Inputs

Name	Type	Default	Description
`data`	scalar field	none	field whose gradient is to be computed
`method`	string	"manhattan"	method to use

Outputs

Name	Type	Description
`gradient`	vector field	gradient field

Functional Details

`data`	is the scalar field.
`method`	specifies how the gradient will be computed. At present, the only method supported is "manhattan."

Components

Modifies the "data" component. All other input components are propagated to the output.

Example Visual Programs

```
ComputeOnData.net
Imide_potential.net
InvalidData.net
UsingCompute2.net
UsingIsosurface.net
SIMPLE/Gradient.net
```

See Also

DivCurl, Normals

Grid

Category

Realization

Function

Creates an output geometry.

Syntax

```
grid = Grid(point, structure, shape, density);
```

Inputs

Name	Type	Default	Description
`point`	vector	none	starting point from which the grid is constructed
`structure`	string	"brick"	type of grid
`shape`	vector list	structure dependent	size and shape of structure
`density`	integer list	{3, 3, ...}	number of points to be put on each grid element

Outputs

Name	Type	Description
`grid`	geometry field	output grid

Functional Details

`point` is the starting point from which the grid is constructed.

`structure` specifies the type of grid geometry and must be one of the following: "brick " (the default), "crosshair," "ellipse," "line," "point," or "rectangle."

For a structure specified as "point," it is neither necessary nor useful to specify **`shape`** or **`density`**. For the other valid structures, the output object extends symmetrically from **`point`**.

`shape` is a list of vectors that characterize the grid structure, specifying its size and orientation: A line requires one vector; rectangles and ellipses, two vectors; and bricks and crosshairs, three vectors. For "brick," the default **`shape`** specification is {[1 0 0] [0 1 0] [0 0 1]}; for other structures, the default specification is the appropriate corresponding vector list.

`density` determines the number of points in the output. For crosshairs and bricks, the specification is a list of length 3; for rectangles, a list of length 2; and for the other structures, a single integer. In each case, **`density`** is the number of points along the edge in question. Bricks and rectangles are filled.

Note: Specifying **`point`** and **`shape`** as 2-dimensional vectors can produce a 2-dimensional output **`grid`** for the structures “ellipse,” “line,” “point,” and “rectangle.”

Components

Creates “positions” and “connections” components.

Example Visual Programs

```
InvalidData.net
MakeLineMacro.net
PlotLine.net
PlotTwoLines.net
ThunderStreamlines.net
SIMPLE/Grid.net
```

See Also

Construct, Glyph, Map, Streakline, Streamline

Histogram

Category

Transformation

Function

Constructs a histogram from input data and computes the median.

Syntax

histogram, median = Histogram(**data, bins, min, max, out);**

Inputs

Name	Type	Default	Description
data	scalar list or vector list or scalar field or vector field or series	none	field to be operated on
bins	integer or vector	256 for bytes, 100 otherwise	number of bins in histogram
min	scalar or vector or field	min. of data	minimum value to operate on
max	scalar or vector or field	max. of data	maximum value to operate on
out	flag or vector	0	0: exclude out-of-range values 1: include out-of-range values

Outputs

Name	Type	Description
histogram	field or series	the histogram
median	scalar	median of the input data

Functional Details

The median is determined from an interpolation performed in the bin containing the median element (or elements, if there are an even number of samples).

data is the data whose frequency distribution is to be computed and then represented in a histogram. **data** can consist of scalars, 2-vectors, or 3-vectors, with resulting one-dimensional, two-dimensional, or three-dimensional histograms, respectively.

bins specifies the number of bins into which the range from **min** to **max** will be divided. If **data** consists of vectors, then **bins** can be a vector of the same length specifying the number of bins in each dimension.

min and **max** specify the range of values for which the histogram is computed.

- If neither parameter is specified, the values used are the minimum and maximum of the input data values.

- If **min** is a scalar value, it is the value used.
- If **min** is a data field, the minimum data value of that field is used.
- **max** is similarly interpreted.
- If **min** is a data field and **max** is unspecified, the module uses the minimum and maximum values of that field.
- If **data** consists of vectors, then **min** and **max** may also consist of vectors of the same length to specify the range in each dimension. Otherwise, the values given for **min** and **max** will apply to each dimension.

out specifies whether the module ignores data values outside the range of **min** and **max** or includes them in the first and last bins respectively. If **data** consists of vectors, then **out** may be a vector of the same length specifying how to treat each dimension.

Notes:

1. For the output, a "positions" component is constructed that consists of **bins** +1 points, corresponding to the boundaries between bins.
2. A regular "connections" component is constructed that consists of a set of line segments connecting the end points.
3. The "data" component, which is connection dependent, contains the counts for the corresponding bin.
4. The interval for each bin is closed on the **min** side and open on the **max** side. To include the maximum data value:
   - set **out** to include outlying values

     or
   - set **max** to a value slightly larger than the maximum data value.

To see the results of Histogram for scalar data, pass its output to the Plot module. For 2-vector data, pass its output to the RubberSheet module with a scale factor of 1. For 3-vector data, use either the Isolate module or use Include followed by ShowBoundary.

Components

Creates new "positions," "connections," and "data" components. The data, representing the number of items in each bin, is connections dependent.

Example Visual Program

```
UsingEqualize.net
SIMPLE/Histogram.net
```

See Also

Equalize, Plot, Scale

Image

Category

Rendering

Function

Renders an object and displays it as an image.

Syntax

This module is accessible only through the user interface.

Inputs

Name	Type	Default	Description
object	object	none	object to be rendered and displayed
renderMode	flag	0	software=0, hardware=1
defaultCamera	camera	no default	the reset camera
resetCamera	camera	no default	reset the camera
bkgndColor	vector or string	"black"	image background color
throttle	scalar	0	minimum time between image frames (in seconds)
recordEnable	flag	0	enable frame recording
recordFile	string	"image"	file name for frame recording
recordFormat	string	"rgb"	file format for frame recording
recordRes	integer	no default	Image resolution for recording
recordAspect	scalar	no default	Image aspect ratio for recording
axesEnabled	flag	0	produce axes
axesLabels	string list	no labels	labels for axes
axesTicks	integer or integer list	15	number of major tick marks (0 to suppress)
axesCorners	vector list or object	input object	bounds of axes
axesFrame	flag	0	flag for axes frame type
axesAdjust	flag	1	whether to adjust the end points to match tick marks
axesCursor	vector	no cursor	cursor position
axesGrid	flag	0	show grid lines on background
axesColors	vector list or string list	appropriate color(s)	color(s) for annotation
axesAnnotate	string list	"all"	annotation objects to be colored
axesLabelScale	scalar	1.0	scale factor for labels
axesFont	string	"standard"	font for labels
intrctnMode	string	"none"	sets interaction mode of window

Name	Type	Default	Description
`title`	string	"Image"	Image title
`axesXTickLocs`	scalar list	no default	locations for x-axis ticks
`axesYTickLocs`	scalar list	no default	locations for y-axis ticks
`axesZTickLocs`	scalar list	no default	locations for z-axis ticks
`axesXTickLabels`	string list	no default	labels for x-axis ticks
`axesYTickLabels`	string list	no default	labels for y-axis ticks
`axesZTickLabels`	string list	no default	labels for z-axis ticks

Outputs

Name	Type	Description
`renderable`	object	object, ready for rendering
`camera`	camera	camera used
`where`	window	window identifier for the Image window

Functional Details

This module functions much like a combination of AutoCamera and Display. However, it activates the Data Explorer direct-interaction features of the Image window, which are unavailable when you use the combination of AutoCamera and Display. These features include resizing of the image, pan/zoom, 3-D cursors, mouse-driven rotation, navigation in the image, mouse control of the look-to and look-from points, and direct user control of Image-window size. The **`View Control...`** dialog box (accessed from the **`Options`** pull-down menu in the Image window) permits explicit specification of these features.

Note: Since Image both renders and displays its input, you would not normally pass it an already existing image (because it would be interpreted as a large number of quadrilaterals). If you do have an image (e.g., previously imported or rendered by one of the appropriate modules), you should use the Display tool, without specifying a camera.

While the Image tool is only available while using the User Interface, you can implement your own interactions with an image using SuperviseWindow, SuperviseState, and Display. Thus direct interaction can be implemented from a stand-alone program using DXCallModule (see 12.10, “Module Access” on page 127 in *IBM Visualization Data Explorer Programmer’s Reference*), from a program interacting with Data Explorer using DXLink (see Chapter 16, “DXLink Developer's Toolkit” on page 157 in *IBM Visualization Data Explorer Programmer’s Reference*), or even from a script (see Chapter 10, “Data Explorer Scripting Language” on page 187 in *IBM Visualization Data Explorer User’s Guide*). See “SuperviseWindow” on page 336 for a discussion of these tools.

If it is installed, hardware graphics acceleration is available as one of the **`Rendering Options...`** in the **`Options`** pull-down menu of the Image window.

Usually you will use only the first parameter, **`object`**. The other parameters, which are hidden by default, control aspects of the image which can also be modified by various pulldowns in the menu of the Image window.

To open the Configuration dialog box for Image, you select the Image tool and use the **Edit** menu **Configuration** option.

object specifies the object to be rendered.

renderMode specifies whether the rendering should be software (renderMode=0) or hardware (renderMode=1).

defaultCamera is the camera to be used when the **Reset** button in the **View Control** dialog box or the **resetCamera** parameter (see below) is set to 1 (one).

resetCamera (when set to 1) resets the camera to

- **defaultCamera** if that parameter is provided, *or*
- the standard "front view" if **defaultCamera** is not provided.

bkgndColor specifies the color of the image background as either an RGB color or a color-name string.

throttle specifies a minimum interval between successive image displays. The default is 0 (no delay).

recordEnable specifies that the created images are to be saved to a file.

recordFile is the name of the file to which images are saved.

recordFormat specifies the format in which the images are to be written. See "WriteImage" on page 374 for more information.

recordRes specifies the horizontal resolution (in pixels) of the recorded image.

recordAspect specifies the aspect ratio (i.e., the ratio of vertical to horizontal size) of the recorded image.

axesEnabled specifies that a set of axes is to be drawn around **object**.

axesLabels...axesFont
are (except for "axes" in the name) the same as the corresponding parameters described for "AutoAxes" on page 27.

intrctnMode specifies one of the following as the interaction mode of the Image window:

```
"camera"      "none"       "roam"
"cursors"     "panzoom"    "rotate"
"navigate"    "pick"       "zoom"
```

title is the title of the Image window.

axesXTickLocs specifies the list of x-axis tick locations (see "AutoAxes" on page 27)

axesYTickLocs specifies the list of y-axis tick locations (see "AutoAxes" on page 27)

axesZTickLocs specifies the list of z-axis tick locations (see "AutoAxes" on page 27)

axesXTickLabels
specifies the list of x-axis tick labels (see "AutoAxes" on page 27)

axesYTickLabels
specifies the list of y-axis tick labels (see "AutoAxes" on page 27)

axesZTickLabels
specifies the list of z-axis tick labels (see "AutoAxes" on page 27)

The first output of the Image tool (**renderable**) is the object just before rendering (including, for example, any axes attached).

The second output (**camera**) is the camera used to render **object** (including, for example, any zooms or rotations applied by direct interactors).

The third output **where** is the window identifier for the Image window. This can be used, for example, by ReadImageWindow to read back the pixels from the image.

Recording images displayed in the Image window

1. select **Save Image...** or **Print Image...** in the **File** pull-down menu, *or*
2. use the **record...** parameters (see above).

Caching of Objects Internally in the Image Tool: The Image tool is implemented as a macro comprising a number of modules. You can control the caching of intermediate results between these modules using the special “Internal Caching” option in the module’s configuration dialog box (see *IBM Visualization Data Explorer User’s Guide*, “Using Data Explorer Effectively”.

How the Image Window Centers on an Object: The first time the Image tool executes after being placed on the canvas, it “centers” itself on **object**. From then on, it maintains that viewpoint unless the user explicitly changes it (e.g., by using rotate, zoom, roam, etc.). Thus, if the object is moving in space, you may at times see only part of it or even none of it. You can “reset” the camera at any time with **ResetCamera** in the **View Control...** dialog box or by using the Image parameter **resetCamera**.

Getting the Output Image: If you want to extract the rendered image so that, for example, you can filter it or arrange it with other images (see “Filter” on page 137), then simply pass the first and second outputs of Image to the first and second inputs of Render. Alternatively, pass the **where** output to the ReadImageWindow module. Using the ReadImageWindow module allows you to capture the pixels from a hardware-rendered image. Note however, that for some platforms, if the image window is obscured (by another window for example) the obscured pixels may not be present in the captured image.

Tracking the Image window with another window: If you want to have two images visible, and want to alternatively control the viewpoint, resolution, etc. of one of the windows while having the second window display a different object from the same viewpoint, simply use the **camera** output of the Image tool as input to the **camera** input of a Display tool. The first input of Display should be the other object you wish to view. For an example, see WindVorticity.net.

For more information on using the Image tool in the user interface, see 6.1, “Using the Image Window” on page 74 in *IBM Visualization Data Explorer User’s Guide*. Also see Display for information under the following headings:

“Differences between Hardware and Software Rendering” on page 115,
“Rendering Approximations” on page 114,
“Using Default Color Maps” on page 111,
“Using Direct Color Maps” on page 111,
“Changing Rendering Properties” on page 112,
“Gamma Correction” on page 112,

"Image Caching" on page 112,
"Texture Mapping" on page 116

See also Color for information on "Coloring Objects for Volume Rendering" on page 113.

Example Visual Programs

Nearly every example visual program uses the Image tool. `ImageTool.net` uses the hidden-by-default parameters. `Image_wo_UI.net` demonstrates a substitute Image macro using SuperviseWindow, SuperviseState, and Display, which can be used independent of the User Interface.

See Also

Arrange, AutoCamera, Display, Render, AutoAxes, SuperviseWindow, SuperviseState

Import

Category

Import and Export

Function

Reads an external data file.

Syntax

```
data = Import(name, variable, format, start, end, delta);
```

Inputs

Name	Type	Default	Description
`name`	string	none	name of file containing data to be read, or “!command”
`variable`	string or string list	format dependent	variable to be read
`format`	string	file extension or content	"dx," “general,” “netcdf,” “CDF,” “hdf,” “cm”
`start`	integer	first frame	first data frame to be imported
`end`	integer	last frame	last data frame to be imported
`delta`	integer	1	increment between frames

Outputs

Name	Type	Description
`data`	object	object containing requested variables

Functional Details

From an external data file this modules creates Data Explorer objects that can be processed by other modules.

`name` is the name of the data file being imported. If the parameter specifies an absolute path name, the system attempts to open the file. Otherwise, it first searches the current directory (i.e., the directory from which Data Explorer was invoked) and then, if necessary, the directories specified by the environment variable DXDATA (see C.1, “Environment Variables” on page 292 in *IBM Visualization Data Explorer User’s Guide*).

Note: This parameter can also specify an external filter (see External filters on page 168).

If **`name`** contains a series, the parameters **`start`**, **`end`**, and **`delta`** can be used to import a portion of the data (see parameter descriptions below).

`variable` specifies the variable(s) to be imported.

format specifies the format of the data to be imported. Valid format names are: "dx," "general," "netcdf," "CDF," "hdf," and "cm." These keywords can also be used as extensions on file names.

start and **end**
specify the first and last data frame to be imported from a data file containing a series.

delta specifies the increment in counting the data frames in the range from **start** to **end**. For example, if the first and last frames are 10 and 20 respectively, and **delta** = 2, the output **data** is a series group with six members (frames 10, 12, 14,...).

For Future Reference

If the data set being imported is changed (e.g., by editing) during a Data Explorer session, and if the cache is enabled (the default condition), it may be necessary to reinitialize the Data Explorer executive to access the new data. To do so, select **Reset Server** in the **Connections** pull-down menu of the VPE window.

Resetting the server flushes the executive cache. The next time the visual program is invoked, the entire network executes (not just the portions affected by changes) and Import will reaccess the data set.

Specifying that the module's output not be cached has the same effect. Select the appropriate option in:

- the "Cache" option menu of the module's configuration dialog box, *or*
- the "Set Output Cacheability" option menu in the **Edit** pull-down menu.

  Note that it may be necessary to apply the same restriction to any module downstream from Import.

  To specify that *no* outputs are to be cached, use the **-cache off** option when starting Data Explorer.

Data Explorer format files. A Data Explorer data file consists of one or more header and data sections that describe the structure and values of user data. The header section is a text description of one or more Data Explorer objects, and the data section is either a text or binary representation of the data values.

If **variable** specifies more than one object, the module creates a group and each object is added to the group by name. If **variable** is not specified, the default object is imported. This object can be specified with the **default** keyword in the Data Explorer file format (see B.2, "Data Explorer Native Files" on page 244 in *IBM Visualization Data Explorer User's Guide*). If it is not specified, the default object is the last object defined in the data file.

Any Data Explorer object in a Data Explorer data file can be specified for import, including Lights, Cameras, and Transforms, as well as more common objects such as Series, Groups, and Fields. The data can be in a separate file from the header, and header and data sections can be interspersed. And the data can be specified in a variety of formats (see see B.2, "Data Explorer Native Files" on page 244 in *IBM Visualization Data Explorer User's Guide*).

General array importer files. You can use the general format described in 5.1, “General Array Importer” on page 63 in *IBM Visualization Data Explorer QuickStart Guide* to import data from various file formats and convert the data to objects. This format allows you to describe the structure of your data so that Data Explorer can create Data Explorer objects from it. If you do not specify a variable, then all variables are imported.

Normally, the **name** parameter in this case is the general array header file. However, the **name** parameter can be the data file if the extended form of the **format** parameter includes the header file as a template. The **format** parameter can also include any set of keyword-value pairs as a comma-separated list. The specified values are used instead of those in the header file. This format is useful for data files with similar header files where only the size of the data changes. An example for the **grid** keyword is:

format =“general, template=headerfile, grid=num_x x num_y x num_z ...”

An example parameter for the **points** keyword is:

format = "general, template=headerfile, points=n"

You may also omit `template=headerfile` if all the necessary information is specified by the keyword-value pairs.

netCDF files. When the netCDF file is opened, variables matching the **variable** parameter are read in as field objects. If you give no field name, all fields are read in and placed as separate fields in a group. Each group member is named using the name of the field in the netCDF file. If more than one variable has the same field name, a composite field is created.

You can import both regular and irregular data. If the data are regular, nonzero origins and non-unit spacing can be handled. You can also import scalar, vector, and tensor data. For irregular data, “positions” and “connections” are determined from information in the netCDF variable attributes associated with the field. Additional components can also be read in and added to the field, based on netCDF attribute information.

For a detailed description of the attributes required in a netCDF file, and an example of the correct format, see B.4, “netCDF Files” on page 281 in *IBM Visualization Data Explorer User’s Guide*.

CDF files. When the CDF is opened, variables matching the “variable” parameter are read in as fields. If “variable” is not specified, then all variables are imported and placed as fields in a group. Each group member is named using the name of the field (CDF variable) in the CDF. If the CDF contains records, then variable(s) are imported as a series. Some CDF variables become the “positions” component of the field, while others become the “data” component of the field. For a series, the values of the record-varying variable become the “series positions” attribute(s). Variable and global attributes present in the CDF are imported as object attributes. Only CDF r-variables are supported. See *IBM Visualization Data Explorer User’s Guide* for more information on importing data from a CDF.

HDF files. Scientific DataSets are read in as fields. If there are more than one DataSet in the HDF file, you can specify the variable as a number corresponding to the position of the data set (0 corresponds to the first file). If no variable is specified, all fields are read in and placed as separate fields in a group. Each group member is named using the label (if it exists) from the HDF file.

If scales are present, they are interpreted as "positions" with regular "connections." Otherwise, the positions are a regular grid with regular connections. For more information on HDF, see B.6, "HDF Files" on page 288 in *IBM Visualization Data Explorer User's Guide*.

CM files. Import will import saved color-map files. (To save a color map explicitly as a separate **.cm** file, choose **Save As...** in the **File** menu of the ColorMap Editor.)

The imported file will be a group containing the color map as the first field and the opacity map as the second field. (Alternatively, you can import just one of these maps by specifying the **variable** parameter to Import as "colormap" or "opacity" respectively.)

The color map is a field with a 1-dimensional "positions" component (the data values) and a 3-dimensional "data" component (the colors). Similarly the opacity map is a field with a 1-dimensional "positions" component (the data values) and a 1-dimensional "data" component (the opacities).

You can pass the imported color and opacity maps to (1) the **color** and **opacity** tabs of the Color module or (2) the **color-map** and **opacity** parameters of the Colormap tool.

When a .cm file is imported, the result is not only information describing the color and opacity maps themselves, but also information specifically intended for the Colormap Editor regarding control points. Users are not expected to create their own .cm files (other than by writing them using the **Save As** command in the Colormap Editor), as the content of this file is not documented. However users can import any field which has the appropriate color or opacity map structure and use it as input to either the Color or the Colormap tools. For more information on the structure of color and opacity maps, see "Color" on page 75.

External filters. If the first character of the **name** parameter is "!" (e.g., "!ext2dx mydata.ext mydata.dx"), the rest of the string following the exclamation point is interpreted as a shell command to be executed. The command should be the name of an external filter program with any required arguments. The filter program can be any *user-supplied* program that reads data from other file formats or generates data, but it must output "dx" or "general array" format as standard output. The Import module waits for the program to execute, reads the output of the program, and imports the objects with the same options as if reading directly from a file.

Example Visual Programs

Nearly every example visual program uses the Import module. Most import Data Explorer format files. Two example programs that import general array format files are:

```
GeneralImport1.net
GeneralImport2.net
```

An example program that uses the external filter option is:

`ImportExternalFilter.net`

An example program that uses the extended form of the **`format`** parameter is:

`MRI_2.net`

See Also

Export, Partition, ReadImage, ImportSpreadsheet

ImportSpreadsheet

Category

Import and Export

Function

Import spreadsheet format data

Syntax

```
field, labellist = ImportSpreadsheet(filename, delimiter,
                                     columnname, format, categorize,
                                     start, end, delta,
                                     headerlines, labelline);
```

Inputs

Name	Type	Default	Description
filename	string	(none)	name of file to import
delimiter	string	" "	one-character delimiter (what separates the columns)
columnname	string list	(all)	names of columns to import
format	string	1-d	import as 1-d or 2-d field
categorize	string list	""	list of columns to categorize during import
start	integer	(first record)	record (row) to begin importing
end	integer	(last record)	record (row) to end importing
delta	integer	1	increment of rows to import
headerlines	integer	0	number of lines to skip before start of data/column labels
labelline	integer	no default	line number labels are on

Outputs

Name	Type	Description
field	field	a field with each of the columns as a component, with the name of the column as the component name
labellist	string list	a list of the imported column names

Functional Details

ImportSpreadsheet imports spreadsheet (i.e. tabular) ASCII data. Each column in the file is imported as a separate component in the resulting output field. The name of the component is taken from a name at the top of the column in the file, if present. If no name is found, a default name of “column#” is used instead.

If any column entry is NULL or consists of just white space this entry is treated as invalid data in Data Explorer. If it is a column of type

- string
  A place holder of “ ” is placed at that entry,
- float or int
  A -999 is placed at that entry.

In addition, an “componentname missingvalues” component is created which references those invalid entries. Also an “invalid positions” component is created which is the union of all the “componentname missingvalues” components. For more information on invalid positions, see “Invalid Positions and Invalid Connections Components” on page 23 in *IBM Visualization Data Explorer User’s Guide*.

filename is the file to import.

delimiter specifies a one-character delimiter which defines the columns. If you do not specify **delimiter**, white space is assumed to delimit the columns.

Note: The tab delimiter is specified as “\t”.

columnname is a list of the names of the columns you wish to import.

format must be either “1-d” or “2-d”. If you specify “1-d”, then positions of the output field will simply be the indices (row numbers) from 0 to number-of-rows. The field will have as many components as there are imported columns, with each component named by the column name. If you specify “2-d”, then the output field will be a c x r grid, where r is the number of imported rows, and c is the number of imported columns. It will have a single data component which contains all the values in the imported rows and columns. If you specify “2-d”, then the columns imported can not mix string data with numerical data.

categorize specifies columns to be categorized, using the Categorize module. If “allstring” is specified, all columns with a data type of “string” are categorized. For additional information, see “Categorize” on page 55.

headerlines specifies the number of lines to skip before the start of the data/column labels, for skipping comments at the top of the file. Note that this would typically be necessary only when the data being imported is all strings, or if you have comments at the top of the file that could be misinterpreted as labels or data.

labelline specifies the line number labels are on. Note that this would only be necessary when the data being imported is all strings.

start, end, and **delta**
specify the records (rows) you wish to import.

Example Visual Programs

```
Categorical.net
Duplicates.net
Zipcodes.net
```

See Also

Categorize, Import

Include

Category

Import and Export

Function

Includes data points in (or excludes them from) a data set.

Syntax

```
output = Include(data, min, max, exclude, cull, pointwise);
```

Inputs

Name	Type	Default	Description
data	field	none	the field from which to select points
min	scalar or vector	min of data	minimum value to include
max	scalar or vector	max of data	maximum value to include
exclude	flag	0	0: include selected range 1: exclude selected range
cull	flag	1	0: culled points marked invalid 1: culled points removed
pointwise	flag	0	if 1, ignore connections when selecting points

Outputs

Name	Type	Description
output	field	the field with selected points

Functional Details

This module determines which points in a data set are to be treated as valid by other modules. It does so by removing or invalidating data values that fall within (or outside) a specified range, thereby *including* the remaining points in the set. If there are connections (or faces or polylines) in the field, and **pointwise**=0, then Include also removes all invalid connections (connections containing at least one invalid position) and all unreferenced positions (positions not referred to by any valid connection, face, or polyline element).

data is the data field that the module operates on.

min and **max** specify a range of data values whose function is determined by the **exclude** flag (see below).

exclude specifies whether the values to be excluded from the data set lie within or outside the range defined by **min** and **max**.

- 0: Removes or invalidates all elements or positions whose data values lie *outside* the specified range.

- 1: Removes or invalidates all elements or positions whose data values lie *within* the specified range.

cull specifies whether the excluded (culled) points are to be invalidated or actually removed from the data set.

- 0: Returns the field with invalid positions and invalid connections, invalid faces, or invalid polylines components. Since it is unnecessary to remove invalid positions, connections, faces, or polylines, in order to have them treated as invalid by other modules, this is usually the preferred setting.

  If the module removes points from data with regular connections (e.g., cubes or quads), the connections become irregular.
- 1: Returns the field with invalid positions and connections removed.

pointwise if set to 1, the connections of **data** are removed before the points are selected.

See “Invalid Positions and Invalid Connections Components” on page 23 in *IBM Visualization Data Explorer User’s Guide* for further discussion of invalid data.

Note: If **data** is a vector field and **min** and **max** are:

- scalars, then the module uses **min** and **max** to exclude points whose magnitudes fall inside (or outside) the limits.
- vectors matching the shape of the data field, then only those vector values lying between the corresponding limits are retained.

  For example, if the data are 3-D vectors, the values retained are those for which $\text{min}_x \le \text{data}_x \le \text{max}_x$ and $\text{min}_y \le \text{data}_y \le \text{max}_y$ and $\text{min}_z \le \text{data}_z \le \text{max}_z$ are retained (assuming **exclude** = 0).

Components

Modifies the “data,” “positions,” and “connections” components and any components that depend on “positions” or “connections.” Adds an “invalid positions,” “invalid connections,” or “invalid faces” component if **cull** is set to 0. Removes the “connections” component if **pointwise** is set to 1. All other components are propagated to the output.

Example

In the following example, the gradient of the electron density has been mapped onto an isosurface. The Include module removes all points whose data values are greater than 1.5.

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
electrondensity = Partition(electrondensity);
gradientdensity = Gradient(electrondensity);
maggradient = Compute("mag($0)", gradientdensity);
isosurface = Isosurface(electrondensity, 0.3);
mappediso = Map(isosurface, maggradient);
camera = AutoCamera(isosurface);
included = Include(mappediso, 0, 1.5);
Display(included, camera);
```

Including Data Points in a Data Field: Consider a field containing position-dependent data and a “connections” component. If **pointwise**=0, after invalidating positions with data values outside the specified range, Include invalidates all connections that reference (include) an invalid position, and finally invalidates all positions not referred to by any valid connection. That is, no connection elements that include *any* invalid data values are retained—because the result of interpolation within such a connection element is not defined. For example, the following grid has valid data points (x), invalid points (i), and quad connections.

```
x--------x--------i--------i--------x
|        |        |        |        |
|        |        |        |        |
|        |        |        |        |
x--------x--------i--------i--------x
|        |        |        |
|        |        |        |
|        |        |        |
x--------x--------x--------x
|        |        |        |
|        |        |        |
|        |        |        |
x--------x--------x--------x
```

The removal of invalid points leaves three invalid connection elements (the quads marked with “o”) and two data positions that have no valid connection to any other data point:

```
x--------x--------                  x
|        |
|        |   o
|        |
x--------x--------                  x
|        |        |        |
|        |   o    |   o    |
|        |        |        |
x--------x--------x--------x
|        |        |        |
|        |        |        |
|        |        |        |
x--------x--------x--------x
```

The field returned by Include is represented by the grid shown below.

```
x--------x
|        |
|        |
|        |
x--------x
|        |
|        |
|        |
x--------x--------x--------x
|        |        |        |
|        |        |        |
|        |        |        |
x--------x--------x--------x
```

Example Visual Programs

```
GeneralImport1.net
Thunder_cellcentered.net
SIMPLE/Include.net
```

See Also

Regrid

Input

Category

Special

Function

Defines an input parameter for a macro.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`parameter`	object	input value to macro

Functional Details

Use the configuration dialog box of this module to specify the parameter name, description, default value (if any), and tab position on the macro icon. This module is used inside a macro: its output is the macro input required by other modules in the macro. For additional information about the Input tool, see "Creating Macros" on page 149 in *IBM Visualization Data Explorer User's Guide*.

Example Macro and Program

`MakeLineMacro.net` is used by the visual program `PlotLine2.net`

See Also

DXLInput, Output, DXLOutput, DXLInputNamed

Inquire

Category

Structuring

Function

Returns information about an object.

Syntax

```
output = Inquire(input, inquiry, value);
```

Inputs

Name	Type	Default	Description
input	object	none	the subject of the inquiry
inquiry	string	"is null"	a request for particular information
value	string	none	additional qualifiers

Outputs

Name	Type	Description
output	integer or object	Examples include 1 and 0 (for yes/no inquiries), arrays, fields, groups, and vectors.

Functional Details

input	specifies the subject of the module's inquiry.
inquiry	specifies the query (see Notes on page 178). Some queries require an additional parameter (see value).
value	specifies an item of information needed for identifying the appropriate subject of the query. For example, the "attribute" inquiry requires the name of the attribute.

Notes:

1. Table 3 on page 179 through Table 6 on page 183 list the "inquiries" that can be made, the type(s) of object appropriate to each, an explanatory version of each, and the answer or type of answer returned.
2. Inquiries that start with "is" are true/false queries, returning 1 (one) for "yes" and 0 (zero) for "no." Such queries can be reversed, however, by inserting "not" after the initial "is": An "is not" query also returns 1 (one) for "yes" and 0 (zero) for "no." For example, if the subject of the inquiry is a scalar field, "is scalar" returns 1 (one) and "is not scalar" returns 0 (zero).
3. Any "is" query about a group whose members contain different types of objects will return 0 (zero) unless all members have the characteristic specified in the query. For example, if a group is composed of two fields, one scalar and one vector, the answer to the question "is scalar" will be 0 (zero).

4. Any inquiry that returns a scalar integer value can be made to return a value that is “1” higher or lower than that value. For example, “is array + 1” will return 2 (instead of 1) or 1 (instead of 0).
5. Capitalization of an inquiry is optional and its words may be separated or run together.

Table 3 (Page 1 of 2). Inquiries about objects

Inquiry	**Input operated on**	**Question**	**Answer**
"is array"	Any object	Is the input an array?	1 or 0
"is byte"	Any object	Is the data component unsigned byte?	1 or 0
"is camera"	Any object	Is the input a camera object?	1 or 0
"is clipped"	Any object	Is the input a clipped object?	1 or 0
"is composite field"	Any object	Is the input a composite field?	1 or 0
"is constant array"	Any object	Is the input a constant array?	1 or 0
"is double"	Any object	Is the data component double-precision floating point?	
"is empty"	Any object	Is the input an empty field, an empty group, or an array with no items?	1 or 0
"is field"	Any object	Is the input a field?	1 or 0
"is float"	Any object	Is the data component floating point?	1 or 0
"is generic array"	Any object	Is the input a generic array?	1 or 0
"is generic group"	Any object	Is the input a generic group?	1 or 0
"is group"	Any object	Is the input a group of any kind?	1 or 0
"is image"	Any object	Is the object an image?	1 or 0
"is int"	Any object	Is the data component integral?	1 or 0
"is integer"	Any object	Is the data component integral?	1 or 0
"is irregular array"	Any object	Is the input an irregular array?	1 or 0
"is light"	Any object	Is the input a light object?	1 or 0
"is line"	Any object	Is the connections component 1-dimensional (i.e., a line)?	1 or 0
"is matrix"	Any object	Is the data component a matrix?	1 or 0
"is mesh array"	Any object	Is the input a mesh array?	1 or 0
"is multigrid"	Any object	Is the input a multigrid?	1 or 0
"is null"	Any object	Is the input null?	1 or 0
"is object"	Any object	Is the input an object?	1 or 0
"is path array"	Any object	Is the input a path array?	1 or 0
"is private"	Any object	Is the input a private object?	1 or 0
"is product array"	Any object	Is the input a product array?	1 or 0
"is regular array"	Any object	Is the input a regular array?	1 or 0
"is scalar"	Any object	Is the data component scalar?	1 or 0
"is screen"	Any object	Is the input a screen object?	1 or 0
"is series"	Any object	Is the input a series?	1 or 0

Table 3 (Page 2 of 2). Inquiries about objects

Inquiry	**Input operated on**	**Question**	**Answer**
"is short"	Any object	Is the data component short integral?	1 or 0
"is string"	Any object	Is the input a string?	1 or 0
"is surface"	Any object	Are the connections 2-dimensional?	1 or 0
"is transform"	Any object	Is the input a transform object?	1 or 0
"is xform"	Any object	Is the input a generic array?	1 or 0
"is vector"	Any object	Is the data component vectorial?	1 or 0
"is volume"	Any object	Are the connections 3-dimensional?	1 or 0
"is 2vector"	Any object	Is the data component 2-vectorial?	1 or 0
"is 3vector"	Any object	Is the data component 3-vectorial?	1 or 0
"is 1D connections"	Any object	Does the object have 1-dimensional connections?	1 or 0
"is 2D connections"	Any object	Does the object have 2-dimensional connections?	1 or 0
"is 3D connections"	Any object	Does the object have 3-dimensional connections?	1 or 0
"is 1D positions"	Any object	Does the object have 1-dimensional positions?	1 or 0
"is 2D positions"	Any object	Does the object have 2-dimensional positions?	1 or 0
"is 3D positions"	Any object	Does the object have 3-dimensional positions?	1 or 0
"is 4D positions"	Any object	Does the object have 4-dimensional positions?	1 or 0
"is 1D grid connections"	Any object	Does the object have 1-dimensional regular connections?	1 or 0
"is 2D grid connections"	Any object	Does the object have 2-dimensional regular connections?	1 or 0
"is 3D grid connections"	Any object	Does the object have 3-dimensional regular connections?	1 or 0
"is 1D grid positions"	Any object	Does the object have 1-dimensional regular positions?	1 or 0
"is 2D grid positions"	Any object	Does the object have 2-dimensional regular positions?	1 or 0
"is 3D grid positions"	Any object	Does the object have 3-dimensional regular positions?	1 or 0
"is 4D grid positions"	Any object	Does the object have 4-dimensional regular positions?	1 or 0
"primitives"	Any object	How many of each type of primitive does the object contain?	string list
"object tag"	Any object	Returns the unique object identifying tag	integer

Table 4 (Page 1 of 2). Inquiries about particular types of objects

Inquiry	Input operated on	Question	Answer
"category"	Array or list	What category does the input belong to?	string (see 1 on page 182)
"connection counts"	array or list	How many connections are there?	integer
"connection type"	Field	What is the element type?	string
"count"	Array or list	How many items does the input contain?	integer
"data counts"	Array or list	How many data items are there?	integer
"deltas"	Array or list	What are the deltas in each dimension?	vector list
"grid counts"	Array or list	What are the counts in each dimension?	integer
"is empty array"	Array or list	Is the input an array with no items?	1 or 0
"items"	Array or list	How many items does the input contain?	integer
"origin"	Array or list	What is the origin of the input?	vector
"position counts"	Array or list	How many positions are there?	integer
"rank"	Array or list	What is the rank of the input?	integer
"shape"	Array or list	What is the shape of the input?	vector
"type"	Array or list	What is the type of the input?	string (see 2 on page 182)
"camera angle"	Camera	What is the camera angle?	scalar
"camera aspect"	Camera	What is the aspect ratio of the input camera?	scalar
"camera background"	Camera	What is the background color of the input camera?	3-vector
"camera fieldofview"	Camera	What is the field of view of the input camera?	scalar
"camera from"	Camera	What is the “from” point of the input camera?	vector
"camera matrix"	Camera	What is the matrix of the input camera?	matrix
"camera perspective"	Camera	What is the value of the perspective option?	0 or 1
"camera resolution"	Camera	What is the resolution of the input camera?	integer
"camera to"	Camera	What is the “to” point of the input camera?	vector
"camera transform"	Camera	What is the matrix of the input camera?	matrix
"camera up"	Camera	What is the up direction of the input camera?	vector
"camera width"	Camera	What is the width of the input camera?	scalar
"is camera orthographic"	Camera	Is the input camera orthographic?	1 or 0

Table 4 (Page 2 of 2). Inquiries about particular types of objects

Inquiry	Input operated on	Question	Answer
"is camera perspective"	Camera	Is the input camera perspective?	1 or 0
"clipped object"	Clipped	What is the clipped object?	object
"clipping object"	Clipped	What is the clipping object?	object
"component count"	Field	How many components does the field contain?	integer
"component names"	Field	What are the names of the field components?	string list
"connection gridcounts"	Field	What are the connection counts in each dimension?	integer vector
"is empty field"	Field	Is the input an empty field (i.e., with no components, “positions” component, or position items)?	1 or 0
"is regular"	Field	Does the field have regular positions and connections?	1 or 0
"is regular connections"	Field	Does the field have regular connections?	1 or 0
"is regular positions"	Field	Does the field have regular positions?	1 or 0
"is empty group "	Group	Is the input a group with no members?	1 or 0
"member count"	Group	How many members belong to the group?	integer
"member names"	Group	What are the names of the group members?	string list
"position gridcounts"	Field	What are the position counts in each dimension?	integer vector
"product terms"	Product Array	What are the individual product terms?	Group of Arrays
"mesh terms"	Mesh Array	What are the individual mesh terms?	Group of Arrays
"screen depth"	Screen	What is the screen depth?	integer
"screen object"	Screen	What is the screen object?	object
"screen position"	Screen	What is the screen position?	integer
"member positions"	Series	What are the series positions?	scalar list
"series positions"	Series	What are the series positions?	scalar list
"transform matrix"	Transform	What is the transform matrix?	matrix
"transform object"	Transform	What is the object to be transformed?	object
"valid count"	Any object	How many valid data items are there?	integer
"invalid count"	Any object	How many invalid data items are there?	integer

Notes:

1. Possible categories: “real,” “complex.”
2. Possible types: “signed byte,” “unsigned byte,” “short,” “unsigned short,” “integer,” “unsigned integer,” “float,” “double,” and “string.”

Table 5. Inquiries that take a value parameter

Inquiry	Input operated on	Value	Question	Answer
"attribute"	Any object	attribute name	What is the value of the specified attribute?	string
"counts"	Field	component name	How many items does the specified component contain?	integer
"grid counts"	Field	component name	What are the counts in each dimension of the specified component?	integer vector
"has attribute"	Any object	attribute name	Does the object have the specified attribute?	1 or 0
"has component"	Field	component name	Is the specified component present?	1 or 0
"has member"	Group	member name	Is the specified member present?	1 or 0
"is connection"	Field	element type	Are the connections of the specified type?	1 or 0
"is data dependent"	Any object	component name	Is the data dependent on the specified component?	0 or 1
"is regular"	Field	component name	Is the specified component regular?	1 or 0
"items"	Field	component name	How many items in the specified component?	integer
"member attribute"	Group	attribute name	What is the value of the specified attribute?	string, list
"member attributes"	Group	attribute name	What is the value of the specified attribute?	string, list
"string match"	String	character string	Does the output match the specified string?	1 or 0

Table 6. Miscellaneous Inquiries

Inquiry	Question	Answer
"processors"	How many processors is Data Explorer Using?	integer

Example Visual Programs

```
MultipleDataSets.net
PlotTwoLines.net
UsingAttributes.net
UsingTextandTextGlyphs.net
WindVorticity.net
```

Integer

Category

Interactor

Function

Generates an integer within a specified range of values.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
`data`	object	no default	object from which interactor attributes can be derived
`refresh`	flag	0	reset the interactor
`min`	integer	minimum data value	minimum output integer
`max`	integer	maximum data value	maximum output integer
`delta`	scalar	input dependent	increment between successive integer outputs
`method`	string	input dependent	defines interpretation of delta input
`label`	string	"Integer"	global name applied to interactor stand-ins

Outputs

Name	Type	Description
`output`	integer	interactor output

Functional Details

This interactor provides incremental control of various functions in a visual program (e.g., the number of contour lines currently displayed on an isosurface in the Image window). The integer range over which the module acts is governed by its attributes (e.g., minimum, maximum, and delta), which in turn are either (1) specified by the parameter values in its `Set Attributes...` dialog box or (2) determined from input to the module (e.g, a data field). In the second case, the interactor is said to be “data driven.”

Note: The interactor is invoked by double-clicking on its icon in the VPE window. Its configuration dialog box is accessed from the `Edit` pull-down menu in the same window.

`data` is the object (usually a data field) from which the interactor can derive any or all of the minimum, maximum, and delta attributes when the corresponding input tabs are up.

refresh	resets the interactor so that the output is computed from the current input. If **refresh** = 0 (the default), the output is recomputed only if the current output does not lie within the range of the current **data**. The default for the output of the interactor is the midpoint of **min** and **max**.
min and **max**	specify the minimum and maximum values of the interactor's integer output. If set, these values override those implied by **data**. If neither **min** nor **data** is specified, the interactor uses the minimum set in the **Set Attributes...** dialog box. If neither **max** nor **data** is specified, the interactor uses the maximum set in the **Set Attributes...** dialog box.
delta	specifies a scalar value as a factor for calculating the increment between successive outputs over the specified range. The actual value depends on the interpretation specified by **method** (see below).
method	specifies the interpretation of **delta**: • “rounded”: the increment (**max** – **min**) × **delta** is rounded to a “nice” number. The spacing between successive values will approximate the interval specified by **delta**. (For example, the default value of 0.01 specifies an interval of 1/100 of the specified range.) • “relative”: the interpretation is the same as for “rounded,” but the increment is *not* rounded. • “absolute”: **delta** is the absolute value of the interval. (If **delta** has not been specified, its default is 1.) The default value for **method** depends on other input. The default is: – “rounded” if **data** is specified *or* if both **min** and **max** are specified. – “absolute” in all other cases.
label	is the global label of all instances of the corresponding interactor stand-in. An interactor instance's local label (set from the Control Panel) overrides a global label. By default, the global label is set by the user interface.

Example Visual Programs

Many example visual programs use Integer interactors. An example program that uses a data-driven integer interactor is `WindVorticity.net`

See Also

IntegerList, Scalar, ScalarList, Vector, VectorList

IntegerList

Category

Interactor

Function

Generates a list of integers within a specified range of values.

Syntax

Available only through the user interface.

Inputs

Name	**Type**	**Default**	**Description**
`data`	object	no default	object from which interactor attributes can be derived
`refresh`	flag	0	reset the interactor
`min`	integer	minimum data value	minimum output integer
`max`	integer	maximum data value	maximum output integer
`delta`	scalar	input dependent	increment between successive integer outputs
`method`	string	input dependent	defines interpretation of delta input
`nitems`	integer	11	number of items in the initial list
`label`	string	"IntegerList"	global name applied to interactor stand-ins

Outputs

Name	**Type**	**Description**
`output`	integer list	interactor output

Functional Details

This interactor provides incremental control of various functions in a visual program (e.g., the number of contour lines currently displayed on an isosurface in the Image window). But it creates a list of integers, rather than a single integer, as Integer does.

The integer range over which the module acts is governed by its attributes (e.g., minimum, maximum, and delta), which in turn are either (1) specified by the parameter values in its configuration dialog box or (2) determined from input to the module (usually a data field). In the second case, the interactor is said to be "data driven."

If an interactor is not data driven, then the attributes (e.g., minimum, maximum, delta, etc.) are taken from the interactor's **`Set Attributes...`** dialog box (which is accessed from the **`Edit`** pull-down menu in the Control Panel).

Note: The interactor is invoked by double-clicking on its icon in the VPE window. Its configuration dialog box is accessed from the **Edit** pull-down menu in the same window.

data	is the object (usually a data field) from which the interactor can derive any or all of the minimum, maximum, and delta attributes when their corresponding input tabs are up.
refresh	resets the interactor so that the output is computed from the current input. If **refresh** = 0 (the default), the output is recomputed only if the current output does not lie within the range of the current **data**.
min and **max**	specify the minimum and maximum values of the interactor's integer output. If set, these values override those implied by **data**. If neither **min** nor **data** is specified, the interactor uses the minimum set in the **Set Attributes...** dialog box. If neither **max** nor **data** is specified, the interactor uses the maximum set in the **Set Attributes...** dialog box.
delta	specifies a scalar value as a factor for calculating the increment between successive outputs over the specified range. The actual value depends on the interpretation specified by **method** (see below).
method	specifies the interpretation of **delta**: • "rounded": the increment (**max** – **min**) × **delta** is rounded to a "nice" number. The spacing between successive values will approximate the interval specified by **delta**. (For example, the default value of 0.01 specifies an interval of 1/100 of the specified range.) • "relative": the interpretation is the same as for "rounded," but the increment is *not* rounded. • "absolute": **delta** is the absolute value of the interval. (If **delta** has not been specified, its default is 1.) The default value for **method** depends on other input. The default is: – "rounded" if **data** is specified *or* if both **min** and **max** are specified. – "absolute" in all other cases.
nitems	specifies the number of items in a newly created list. These are evenly spaced between the minimum and maximum values (see above). For example, if this parameter is given a value of 5, and the range is 0–100, the output list will be {0, 25, 50, 75, 100}. **Notes:** 1. If **nitems** changes, a new list is computed. 2. This parameter takes affect only if the minimum and maximum are set with the **min** and **max** parameters or with **data** parameter.
label	is the global label of all instances of the corresponding interactor stand-in. An interactor instance's local label (set from the Control Panel) overrides a global label. By default, the global label is set by the user interface.

See Also

Integer, Scalar, ScalarList, Vector, VectorList

Isolate

Category

Realization

Function

“Shrinks” the connection elements of a specified field, creating new positions.

Syntax

```
output = Isolate(field, scale);
```

Inputs

Name	Type	Default	Description
field	field	none	field whose connections are to be isolated
scale	scalar	0.5	shrinkage factor

Outputs

Name	Type	Description
output	field	field of isolated connections

Functional Details

This module moves all positions closer to the center of the connection elements to which they belong, by an amount specified by a scaling parameter (0 ≤ **scale** ≤ 1). Each original position yields as many new positions as there are connection elements that include it. This “shrinkage” makes it possible to view the connection elements of a field (triangles, quads, cubes, or tetrahedra) individually.

Using Isolate followed by ShowBoundary on volumetric data can provide a way to see values throughout a volume.

Components

Creates new positions and connections components. Items belonging to position-dependent components will be duplicated as necessary for the new positions.

Example Visual Program

`Isolate.net`

See Also

ShowConnections

Isosurface

Category

Realization

Function

Computes isosurfaces and contours.

Syntax

surface = Isosurface(**data, value, number, gradient, flag, direction**);

Inputs

Name	Type	Default	Description
data	scalar field	none	field from which one or more surfaces are to be derived.
value	scalar or scalar list	data mean	isosurface value or values
number	integer	no default	number of isosurfaces or contours to be computed
gradient	vector field	no default	gradient field
flag	flag	1	0: normals not computed 1: normals computed
direction	integer	–1	orientation of normals

Outputs

Name	Type	Description
surface	field or group	isosurface

Functional Details

This module computes any of the following:

- points (for an input field consisting of lines)
- lines (for a surface input field)
- surfaces (for a volumetric input field).

All positions in the output field are isovalues (i.e., they match a specified value or values).

The module also adds a default color to the output (gray-blue for isosurfaces and yellow for contour lines and points) if the input object is uncolored. If the object is colored, its colors are interpolated in the output object.

A “data” component with the same value as the input **value** is added to the output field.

data is the data object for which an isosurface or contour is to be created.

value	is the isovalue or isovalues to be used for computing the isosurface(s) or contour(s). If this parameter is not specified, the module bases it calculations on the value specified by **number** (see below). If neither parameter is specified, the module uses the arithmetic mean of the data input as a default.
number	is ignored if **value** has been specified. If that parameter is not specified, the module uses the value of **number** to compute a set of isosurfaces or contours with the following isovalues: `min + delta, min + (2*delta),..., min − delta` where delta = (max – min)/(**number** + 1), and “max” and “min” are the maximum and minimum data values of the input field.
gradient	is the gradient field used to compute normals for shading (see “Gradient” on page 155). If this parameter is not specified, the module adds normals by computing the gradient internally (**flag** can nullify this behavior; see below). **Note:** If only one isosurface is to be computed, it is probably more efficient to have module compute the gradient internally. If many are to be generated, it is probably more efficient to compute the gradient of the entire field once, so that the system can use it for every isosurface.
flag	specifies whether normals are to be computed for shading. A setting of 0 (zero) prevents the computation of normals. The default is 1 (one)
direction	specifies whether the normals should point against (0, the default) or with (1) the gradient.

Notes:

1. This module adds an attribute called “Isosurface value,” which has as its value the isovalue(s) used. To extract this attribute (e.g., for use in a caption for an image), use the Attribute module.
2. For contour lines, this module adds a “fuzz” attribute so that the line will be rendered slightly in front of a coincident surface (see Display).
3. A surface or contour is considered to be undefined if every point in the input volume or surface, respectively, is equal to **value**. In such cases, the module output is an empty field.
4. Isosurface does not accept connection-dependent data.
5. With disjoint data fields, there may be no data crossings (i.e., points along a connection element where the interpolated data value equals the isovalue), even though the isovalue itself falls in the range of the actual data.

Components

Creates new “positions” and “connections” components. For surfaces output, the default is to create a “normals” component. Any component dependent on “positions” is interpolated and placed in the output object.

Example Visual Programs

Many example visual programs use the Isosurface module, including:

```
AlternateVisualizations.net
ContoursAndCaption.net
InvalidData.net
MappedIso.net
Sealevel.net
UsingIsosurface.net
SIMPLE/Isosurface.net
```

See Also

Band, Color, Gradient, Map, SimplifySurface

KeyIn

"Debugging" on page 4

Function

Waits for a line of input from the terminal.

Syntax

```
KeyIn(prompt);
```

Inputs

Name	Type	Default	Description
prompt	string	"Type <ENTER> to continue"	string printed

Functional Details

This module delays execution of a script until it receives a line of input (as signaled by a return character) from the workstation. If **prompt** is specified, it will be printed. Otherwise the default string is printed.

Note: KeyIn can be used only in script mode, and the Data Explorer executive must also be running on the local machine.

Script Language Example

In this example, the first image is displayed. The second image is computed, but Data Explorer does not display it until you type the return character.

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
electrondensity = Partition(electrondensity);
isosurface = Isosurface(electrondensity, 0.3);
camera = AutoCamera(isosurface);
Display(isosurface, camera);
isosurface = Isosurface(electrondensity, 0.5);
camera = AutoCamera(isosurface);
KeyIn("press enter to continue");
Display(isosurface, camera);
```

Legend

Category

Annotation

Function

Creates a legend

Syntax

```
legend = Legend(stringlist,colorlist, position, shape,
                horizontal, label, colors, annotation, labelscale, font);
```

Inputs

Name	Type	Default	Description
stringlist	string list	none	list of strings for legend
colorlist	field, vector list, or string list	none	list of colors for legend
position	vector	[0.95, 0.95]	the position of the color bar (in viewport-relative coordinates)
shape	vector	[300 25]	length and width of the color bar (in pixels)
horizontal	flag	0	0: vertical orientation 1: horizontal orientation
label	string	no defaults	label for color bar
colors	vector list or string list	appropriate	colors for annotation
annotation	string list	"all"	annotation objects to be colored
labelscale	scalar	1.0	scale factor for labels
font	string	standard	font for labels

Outputs

Name	Type	Description
legend	color field	the legend

Functional Details

This module creates a legend associating a set of strings with a set of colors. The legend generated by this module can be collected with the rest of the objects in a scene (by using a Collect module) and incorporated into an image.

stringlist is a list of strings for the legend

colorlist is a colormap, a list of rgb vectors, or a list of color name strings (see “Color” on page 75 for a discussion of valid color name strings). If **colorlist** is a list of rgb vectors or a list of color name strings, then the length of **colorlist** must be the same as the length of **stringlist**. If **colorlist** is a colormap (see “Color” on page 75 for a description of a color map), then the colors are taken to be the values in **colorlist** corresponding to the integers 0, ... n-1 where n is the number of items in **stringlist**. Thus, for example, AutoColor or Color can be applied to a categorized string data set (see “Categorize” on page 55), and the colormap used can be directly passed to Legend to associate the appropriate colors with the categorized strings.

position is a 2-dimensional vector (or a 3-dimensional vector whose *z*-component is ignored) indicating the position of the legend in the final image. In viewport-relative coordinates, [0 0] places the legend at the lower left, and [1 1] at the upper right. These same coordinates determine the reference point that is used to position the legend relative to its placement in the image (e.g., for **position** = [0 0], the lower left corner of the legend is placed in the lower left corner of the image).

shape is a 2-vector that specifies the length and width of the legend , in pixels. For both horizontal and vertical orientations, the first element of the vector is the length and the second is the width.

horizontal determines whether the orientation of the legend is vertical (0) or horizontal (1).

label specifies a user-supplied label for the legend.

colors and **annotation**
set the colors of certain components of the legend.

colors can be a single color (RGB vector or color-name string) or a list. The color-name string must be one of the defined color names (see “Color” on page 75).

annotation can be a single string or a list of strings, chosen from the following: “all,” “frame,” “labels,” and “ticks.”

If **annotation** is not specified or is “all”—*and* if **colors** is a single string—then **colors** is used for all color-bar annotation. Otherwise the number of colors must match the number of annotation strings exactly. The default frame color is “clear.”

labelscale determines the size of the axes and tick-mark labels. For example, **labelscale** = 2.0 will display the labels at double their default size.

font specifies the font used for axes and tick-mark labels. You can specify any of the defined fonts supplied with Data Explorer. These include a variable-width font (“variable”, the default for axes labels) and a fixed-width font (“fixed”, the default for tick-marks labels).

area	gothicit_t	pitman	roman_ext
cyril_d	greek_d	roman_d	script_d
fixed	greek_s	roman_dser	script_s
gothiceng_t	italic_d	roman_s	variable
gothicger_t	italic_t	roman_tser	

For more information, see Appendix E, “Data Explorer Fonts” on page 307 in *IBM Visualization Data Explorer User’s Guide*.

Example Visual Programs

```
HomeOwn.net
Legend.net
SalesOnStates.net
```

See Also

ScaleScreen, Color, ColorBar, Categorize

Light

Category

Rendering

Function

Creates a distant light source.

Syntax

light = Light(**where, color, camera**);

Inputs

Name	Type	Default	Description
where	vector or camera	[0 0 1]	position or direction of light
color	vector or string	[1 1 1]	color and intensity of light
camera	flag	0	0: fixed direction 1: direction relative to camera

Outputs

Name	Type	Description
light	light	a distant light

Functional Details

where specifies the direction to the light source. If this parameter is specified as a camera rather than as a vector, the module positions the light source behind the camera and to its left (this is also the default light supplied by Data Explorer).

color specifies the color and intensity of the light either as an RGB vector or as a color-name string. The color name must be one of the defined color-name strings (see "Color" on page 75).

camera specifies that **where** is to be interpreted relative to the camera coordinate system rather than the world coordinate system.

If **where** is a camera, then the specification **camera** = 0 is an error.

Use the Collect module to incorporate the output light in the scene given to the Render, Display, or Image.

If no light is present in the object passed to the Render, Display, or Image tool, a distant light of color [1 1 1] (behind and 45 degrees to the left of the viewer) is automatically incorporated in the scene. A small amount of ambient light is also used. Data Explorer removes these default lights if any light has been specified. The effect of the system's default lights by combining an ambient light (generated by using the default settings of AmbientLight) with the distant light generated by the Light module when a camera is specified for the **where** parameter.

Lights have no effect on volume-rendered objects.

Example Visual Programs

```
ThunderGlyphSheet.net
UsingLight.net
```

See Also

AmbientLight, Collect, Color, Convert

List

Category

Structuring

Function

Creates a list.

Syntax

```
list = List(object1, object2,...]);
```

Inputs

Name	Type	Default	Description
object1	value, value list, or string list	no default	list item
object2, ...	**...**	**...**	more list items to concatenate

Outputs

Name	Type	Description
list	value list or string list	the list of objects

Functional Details

This module creates a list from specified objects, which can be single values, strings or lists themselves. The objects specified (**object1, object2, ...**) must all be the same type or be convertible to the same type. The output **list** is a list of the concatenated items.

To select items from the list, use the Select module.

A maximum of 21 objects can be concatenated with single call to the List module. In the user interface, the default number of enabled **string** tabs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Example Visual Programs

```
AnnotationGlyphs.net
ContoursAndCaption.net
MappedIso.net
PlotTwoLines.net
UsingAttributes.net
```

See Also

Select, Enumerate

Lookup

Category

Transformation

Function

Syntax

```
output = Lookup(input, table, data, lookup, value, destination, ignore);
```

Inputs

Name	Type	Default	Description
input	field, string list, value, value list	(none)	object to lookup
table	field, string, value list	(dataname lookup)	lookup table
data	string	“data”	component of **input** to lookup
lookup	string	“positions”	component of **table** in which to look
value	string	“data”	component in **table** containing lookup results
destination	string	“lookedup”	component of **input** in which to put result
ignore	string list	(no default)	Properties to ignore in string lookup

Outputs

Name	Type	Description
output	field or array	field or array with looked-up values

Functional Details

input	field containing the component to use as lookup
table	field containing the lookup table components
data	component in input to use for looking up
lookup	component in table that the data component is to match
value	component in table to use as the looked up result
destination	component of input in which to put the looked up results
ignore	properties to ignore in string lookup. Can be one of “case”, “space”, “lspace”, “rspace”, “lrspace”, “punctuation”. “case” means to ignore the case of the characters, “space” means to ignore all white space (spaces, tabs, ...), “lspace” means to ignore white space on the left side of the string, “rspace” means to ignore white space on the right side of the string, “lrspace” means to ignore white space on both

the left and right sides (but not internal white space), and "punctuation" means to ignore all punctuation characters (anything other than alphabetic and numeric characters and white space). **ignore** can be a list of strings; for example, a commonly used combination is {"space", "case", "punctuation"}.

Lookup uses one component to find another by "looking up" the corresponding value in a lookup table. Lookup serves to convert a categorized component back to its original form, or more generally to provide arbitrary associations of unrelated object types. The lookup can occur entirely within the input field when it contains all components necessary to do the lookup, or alternatively the lookup table can be provided as a separate field in **table**.

If **table** is an array, Lookup treats it as if it has an implicit "positions" component with values from 0 to n-1, where n is the number of items in the array. Thus if the **data** component is integer, and **lookup** is an array of strings, **data** can be used to lookup a matching value in the implicit positions component and return the string. Alternatively, if **data** were also a string array, Lookup could lookup a matching value in the **table** array and return the corresponding implicit position.

Note: The lookup is done using a binary search of the lookup component. If this component is not already sorted, it will be internally sorted in order to perform the lookup. Connections are not used by this module. Since Data Explorer does not support string positions, if lookup is done using a string data type the table input must have a separate float positions component. If the lookup component has duplicate values, the value corresponding to the first one found will be used.

Components

Adds a new component as specified by the **destination** parameter containing the looked-up values.

Example Visual Programs

`Duplicates.net`

See Also

Categorize, Map, CategoryStatistics

ManageColormapEditor

Category

Interface Control

Function

Allows colormap editors to be opened and closed from within a visual program.

Syntax

Available only in the user interface.

Inputs

Name	Type	Default	Description
name	string	no default	name of the colormap editor(s) to be opened or closed
open	flag, flag list, or string list	0	0: close the colormap editor(s) 1: open the colormap editor(s)
how	string or string list	"title"	open or close the colormap editor(s) by title or label

Functional Details

name specifies the name(s) of the colormap editor(s) to be opened.

open determines whether the specified editor(s) will be opened (1) or closed (0). The effect of this parameter depends on the type of argument supplied:

- If the argument is a *single integer*, all the specified Colormap editors will either be opened or they will all be closed.
- If the argument is an *integer list*, it must have the same number of items as the list of strings given in **name**, and the flags are associated one-to-one with the named editors.
- If the argument is a *string list*, the Colormap editors listed in **open** will be opened, and those listed in **name** but not in **open** will be closed.

how if specified, determines how the **name** parameter is to be interpreted:

"title": the name specified is the title of the colormap editor (accessible through the **Change Colormap Name** option in the **Option** pull-down menu of the Colormap Editor) or the **title** parameter of the Colormap tool.

"label": the name specified is the label of the colormap editor (accessible the **Notation** field of the Colormap Editor configuration dialog box).

Example Visual Program

InterfaceControl1.net

See Also

Execute, ManageControlPanel, ManageImageWindow, ManageSequencer

ManageControlPanel

Category

Interface Control

Function

Allows control panels to be opened and closed from within a visual program.

Syntax

Available only in the user interface.

Inputs

Name	Type	Default	Description
name	string or string list	no default	name of the control panel(s) to be opened or closed
open	flag, flag list, or string list	0	0: close the control panels(s) 1: open the control panels(s)

Functional Details

name specifies the control panel(s) to be opened.

open determines whether the specified control panel(s) will be opened (1) or closed (0). The effect of this parameter depends on the type of argument supplied:

- If the argument is a *single integer*, all the specified panels will be opened or they will all be closed.
- If the argument is an *integer list*, it must have the same number of items as the list of strings given in **name**, and the flags are associated one-to-one with the named panels.
- If the argument is a *string list*, the panels listed in **open** will be opened, and those listed in **name** but not in **open** will be closed.

Example Visual Program

InterfaceControl1.net

See Also

Execute, ManageColormapEditor, ManageImageWindow, ManageSequencer

ManageImageWindow

Category

Interface Control

Function

Allows Image or Display windows to be opened and closed from within a visual program.

Syntax

Available only in the user interface.

Inputs

Name	Type	Default	Description
name	string or string list	no default	name of the windows to be opened or closed
open	flag, flag list, or string list	0	0: close the windows 1: open the windows
how	string	"title"	open or close the Image or Display windows by title, label, or window

Functional Details

name specifies the windows to be opened.

open determines whether the specified windows will be opened (1) or closed (0). The effect of this parameter depends on the type of argument supplied:

- If the argument is a *single integer*, all the specified windows will be opened or they will all be closed.
- If the argument is an *integer list*, it must have the same number of items as the list of strings given in **name**, and the flags are associated one-to-one with the named windows.
- If the argument is a *string list*, the windows listed in **open** will be opened, and those listed in **name** but not in **open** will be closed.

how if specified, determines how the **name** parameter is to be interpreted:

"title" the name specified is the title of the window (accessible through the **Image Name** option in the **Options** pull-down menu of the Image window, the **title** parameter of the Image tool or the **where** parameter of the Display module).

"label" the name specified is the label of the window (accessible through the **Notation** field of the configuration dialog box of the Image or Display module).

"window"
the window identifier. If using SuperviseWindow and SuperviseState, then **name** should be the **where** output of SuperviseWindow.

Note: If you use this module to close a window, you must also turn off rendering to that window, using the Route module. Otherwise the window will open and then immediately close.

Example Visual Program

InterfaceControl2.net

See Also

Execute, ManageColormapEditor, ManageControlPanel, ManageSequencer, Route,SuperviseWindow , SuperviseState

ManageSequencer

Category

Interface Control

Function

Determines whether the Sequence control panel is displayed or not.

Syntax

Available only in the user interface.

Inputs

Name	Type	Default	Description
open	flag	1	0: close the sequencer 1: open the sequencer

Functional Details

This module allows a sequencer control panel to be opened or closed from within a visual program (without use of the **Sequencer** option in an **Execute** pull-down menu).

See Also

Execute, ManageColormapEditor, ManageControlPanel, ManageImageWindow

Map

Category

Transformation

Function

Applies a map to a field or value list.

Syntax

```
output = Map(input, map, source, destination);
```

Inputs

Name	Type	Default	Description
input	field, value, or value list	none	field to be mapped
map	scalar, vector, or field	identity	map to be used
source	string	"positions"	component used as index into map
destination	string	"data"	component in which to place the interpolated data.

Outputs

Name	Type	Description
output	field, value, or value list	mapped input field

Functional Details

Map is a general purpose module which maps from one field to another.

input is the field or value list to which **map** is applied.

map is the map to be applied to the input object. Unless this parameter specifies a single value, it should contain both a “positions” and a “data” component.

Because the module performs interpolation, the map it applies to the input must also contain either (1) a “connections” component or (2) “faces,” “loops,” and “edges” (see “Faces, Loops, and Edges Components” on page 24 in *IBM Visualization Data Explorer User’s Guide*). The “data” component may be dependent on “positions,” “connections,” or “faces.” (If the field contains faces, loops, and edges, the data *must* be dependent on faces.)

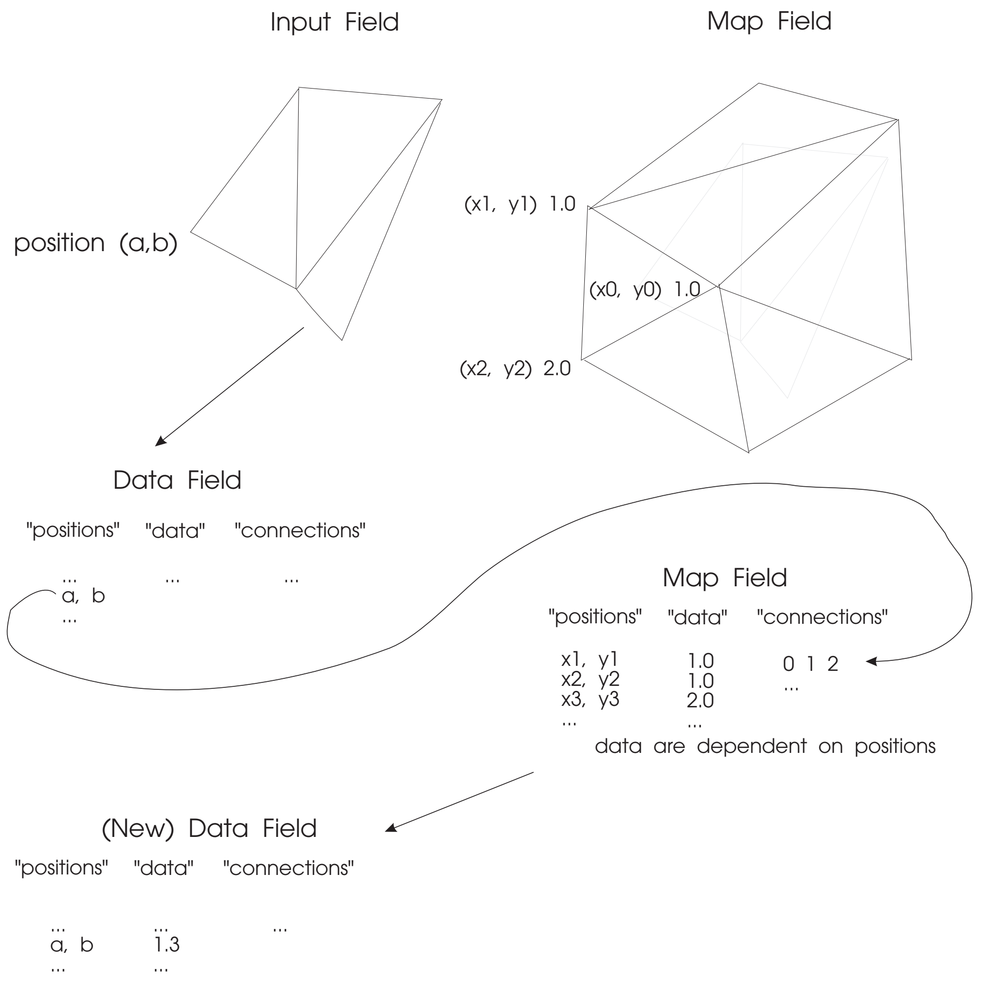
source specifies the component of **input** that is to be used for indexing into the “positions” component of **map**. The value of the corresponding “data” in **map** is determined by interpolation between the “positions” in **map**, using the interpolation elements of **map** (either “connections” or “faces”, “loops”, and “edges”). **source** is ignored if **input** is a value list.

`destination` specifies the **`output`** component in which interpolated values should be placed. If **`input`** is a value list, then the interpolated values simply replace the values in **`input`**.

The Map module steps through the **`source`** component of **`input`**. For each item in that component, the module looks up that value in the "positions" component of **`map`** and finds the corresponding value in the "data" component of **`map`**, interpolating if necessary. The resulting value is placed in the **`destination`** component of **`output`** (see Figure 3).

Figure 3. Mapping from one field to another. This figure shows an input field and a map field, both with two-dimensional positions and triangle connections. The figure shows how a data value is found for the position (a,b) in the input field by interpolating in the map field when the parameters to the Map module are **`input`***= Input Field,* **`map`***= Map Field, and* **`source`** *and* **`destination`** *default to "positions" and "data" respectively.*

As with all maps in Data Explorer, the map field must have "positions", "data", and "connections". Since the **`source`** *component is "positions", the "positions" component of* **`input`** *is used to index into the "positions" component of* **`map`***. Thus we lookup the position (a,b) in the map field. This leads us to the triangle connecting positions 0, 1, and 2 in the map field. The data values corresponding to positions 0, 1, and 2 are interpolated to yield the result 1.3, which is then placed in the "data" component of the output of Map (since* **`destination`** *is "data").*



So, for example, if **source** is “positions,” **destination** is “data,” **input** is an isosurface, and **map** is a 3-D field with temperature values, then Map steps through the “positions” component of the isosurface and finds the temperature value for each position, interpolating if necessary. The resulting temperature value is placed in the “data” component of the output field.

If **map** is a value, the Map module adds to **output** a **destination** component that (1) contains one element for each element of the **source** component and (2) derives its dependency from the **source** component. All elements in this component have the value given them by **map**.

The following table summarizes some of the uses of Map.

Source	Destination	Use
“data”	“colors”	color mapping
“positions”	“data”	data mapping
“data”	“data”	arbitrary tabular function

Notes:

1. The dimensionality of the positions in **input** and in **map** must agree and must also match the dimensionality of the connections in **map**. That is, if **map** has 2-D connections (quads or triangles), the positions must also be 2-D.
2. There are few constraints on Map’ functionality. For instance, the Color module is generally preferable for performing mapping, because that module prevents the formation of invalid colors, whereas Map does not.

Components

If **input** is a field, a new component name, specified by **destination**, is created. All other input components are propagated to the output.

If **input** is an array, the output is an array.

If any **source** values cannot be interpolated, an “invalid positions” or “invalid connections” component (depending on the dependency of the **source** parameter) will be created, and values that are not interpolated will be marked invalid.

Example Visual Programs

```
AlternateVisualizations.net
Interop.net
ManipulateGroups.net
MappedIso.net
PlotTwoLines.net
Thunder_cellcentered.net
UsingMap.net
SIMPLE/Map.net
```

See Also

AutoColor, Color, MapToPlane

MapToPlane

Category

Realization

Function

Maps a 3-dimensional field onto a plane.

Syntax

```
plane = MapToPlane(data, point, normal);
```

Inputs

Name	Type	Default	Description
data	field	none	data to be mapped
point	vector	center of object	a point on the map plane
normal	vector	[0 0 1]	normal to the map plane

Outputs

Name	Type	Description
plane	field	mapped plane

Functional Details

This module creates an arbitrary cutting plane through 3-dimensional space and interpolates data values onto it.

data	must be a field with 3-dimensional connections (i.e., cubes or tetrahedra).
point	is a vector value specifying a point on the cutting plane. If this parameter is not specified, Data Explorer uses the center of the bounding box of **data**.
normal	is a vector value specifying the normal to the map plane and is interpreted as the end point of a vector from the origin (not from **point**). The parameter defaults to [0 0 1].

Notes:

1. To create a plane parallel to one of the axes along a connections boundary (for regular data) it is more efficient to use the Slab module with zero thickness, because it performs no interpolation.
2. MapToPlane (unlike Slab) adds a "normals" component to the plane, so that the result is shaded. To eliminate the shading, remove the "normals" component with the Remove module or turn off the shading with the Shade module.
3. If the specified plane is precisely at the edge of the data, the output may or may not appear, depending on the direction of **normal**.

Components

Creates new “positions,” “connections,” and “normals” components. New “components” of all other input components are created (e.g., the “data” component) and contain values interpolated from the originals.

Example Visual Programs

```
AlternateVisualizations.net
ContoursAndCaption.net
ThunderGlyphSheet.net
SIMPLE/MapToPlane.net
```

See Also

AutoColor, Map, Remove, RubberSheet, Shade, Slab

Mark

Category

Structuring

Function

Marks a component.

Syntax

```
output = Mark(input, name);
```

Inputs

Name	Type	Default	Description
input	field	none	the field with a component to be marked
name	string	none	the component to be marked

Outputs

Name	Type	Description
output	field	the field with the named component marked

Functional Details

This module marks a specified component of a specified input as “data” (without moving the “marked” component from its original position).

input	is the field that contains the component to be marked.
name	specifies the component to be marked.

Once a component has been marked, all modules that operate on the “data” component will now operate on the **name** component. If a “data” component already exists in **input**, it is saved as the “saved data” component. If a “saved data” component already exists, the module returns an error.

Notes:

1. Many modules operate only on the “data” component of a field. The functional scope of such modules can be expanded by using the Mark module to mark, for example, “positions” or “colors” components.
2. Mark adds an attribute, called “marked component,” that lists the name of the component that was marked. The Unmark module will use this attribute if no component is specified for unmarking.

Components

Moves the “data” component to the “saved data” component and copies the **`name`** component to the “data” component. All other input components are propagated to the output.

Example Visual Programs

```
MakeLineMacro.net
PlotLine.net
PlotLine2.net
PlotTwoLines.net
Sealevel.net
UsingMap.net
WarpingPositions.net
SIMPLE/MarkUnmark.net
```

See Also

Compute, Extract, Options, Rename, Replace, Unmark

Measure

Category

Transformation

Function

Performs length, area, and volume measurements on an input object.

Syntax

```
output = Measure(input, what);
```

Inputs

Name	Type	Default	Description
`input`	field	none	data to be measured
`what`	string	input dependent	measurement to be performed

Outputs

Name	Type	Description
`output`	value	result of the measurement

Functional Details

`input` is the field to be measured. It is expected to have a “connections” component consisting of lines, surfaces, or volumes.

`what` specifies the type of measurement to be performed.

- Unless the parameter value is “element,” the module performs a measurements of the field as a whole. The structure of the output is identical to that of the input, with each field (or composite field) replaced by an array containing a simple scalar floating-point measurement of that field.

  The default value of **`what`** is determined by the “connections”-component element type. If the connections element type is:

  “lines,” the parameter defaults to “length.”
  “triangles” or “quads,” the parameter defaults to “area.”
  is “cubes” or “tetrahedra,” the parameter defaults to “volume.”

- If **`what`** is “element,” the module performs element-by-element measurement on **`input`**, replacing the “data” component with scalar arrays that are connection dependent and contain the measurement of each connection element. If the connections element type is:

  “lines,” the resulting “data” component will contain the length of each line segment.
  “triangles” or “quads,” the resulting “data” component will contain the area of each triangle or quad.

is "cubes" or "tetrahedra," the resulting "data" component will contain the volume of each cube or tetrahedron.

The options for **what** are shown in the following table:

Table 7. Options for Measure's What Parameter

Connection element	What	Measurement
lines	"length"	total length of line segments
	"element"	length of each line segment
	"area"	2-D area enclosed by connected line segments. If a series of segments is not closed, a segment connecting the first and final points of the sequence is added.
faces	"area"	total area of all faces
	"element"	area of each face
triangles, quads	"area"	total area of all surface elements
	"element"	area of each surface element
	"volume"	3-D volume enclosed by connected sets of surface elements. If a connected set of surface elements is not closed, the approximate closing surface(s) is(are) found by triangulating the openings in the surface (defined by loops of unshared edges).
polylines	"length"	total length of all lines
	"element"	length of each line
tetrahedra, cubes	"element"	volume of each 3-D element
	"volume"	total volume of all 3-D elements

Notes:

1. The default value for lines and polylines is "length"; "area" is the default for triangles, quads, and faces; and "volume" is the default for tetrahedra and cubes.
2. All sequences of line segments are handled independently when measuring area. Sequences are defined by following exactly identical coordinate points, rather than by following identical point indices. Overlapping volumes are not detected. Similarly for volumes enclosed by surface elements, a volume is defined by triangles consisting of identical vertex positions, rather than by triangles consisting of identical point indices. Overlapping volume is not detected.
3. Be aware of the following: Curves are closed by straight lines regardless of the geometry from which they were derived. The surfaces added to close volumes are triangulated as if planar. This module ignores any transforms in the object. For example, if the object is scaled by 2 (with the Scale module), the volume will *not* increase.

Example Visual Program

```
ThunderGlyphSheet.net
```

See Also

Compute

Message

Category

Debugging

Function

Displays a message to the user.

Syntax

`Message(`**`message, type, popup`**`)`

Inputs

Name	Type	Default	Description
`message`	string	none	message to be displayed
`type`	string	"message"	type of information: error, warning, or message
`popup`	flag	0	0: Do not display message in pop-up window 1: Display message in pop-up window

Functional Details

`message` specifies the message to be displayed.

`type` specifies the kind of message. Information messages are presented unaccompanied, but error messages are preceded by **`ERROR`**, and warnings by **`WARNING`**.

`popup` specifies whether the message is to be displayed in a pop-up window as well as in the Message Window. In script mode, messages are sent to the xterm, regardless of the pop-up setting.

Note: The display of any or all categories of message can be prevented by deactivating the appropriate toggle buttons in the **`Options`** pull-down menu of the Message Window.

Example Visual Program

`UsingMessage.net`

See Also

Echo, Format, Print

Morph

Category

Transformation

Function

Applies a binary morphological operator to a field.

Syntax

`output` `= Morph(`**`input, operation, mask`**`);`

Inputs

Name	Type	Default	Description
`input`	field	none	input data
`operation`	string	"erode"	the operation to be applied
`mask`	value or string	"box"	the mask element

Outputs

Name	Type	Description
`output`	field	morphologically transformed input

Functional Details

This module applies one of the following binary morphological operators to input data:

“`close`” “`dilate`” “`erode`” “`open`”

The input data are treated much as booleans are in the C language (e.g., 0 = FALSE and nonzero = TRUE).

The **`mask`** parameter (see below) is centered on each position in **`input`** in turn, and all positions corresponding to “1” in the mask are considered for **`operation`** (see below).

`input` is the input data to be operated on.

`operation` specifies one of the following:

- “dilate”: The output corresponding to a given data value is true (1) if any data value in **`input`** corresponding to a “1” in **`mask`** is true. Otherwise, the output is false.
- “erode”: The output corresponding to a given data value is true (1) if all the data values corresponding to a “1” in **`mask`** are true. Otherwise, the output is false.
- “close”: is dilation followed by erosion.
- “open” is erosion followed by dilation.

Note: Successive openings or closings have no additional effect.

mask specifies one of the following:

- a Filter name (see "Filter" on page 137)
- an explicit matrix.

The function of both is to identify potential operands for **operation** (see above). The dimensions must be odd, and the default is the 3 × 3 box filter of Filter.

The module supports all data types and, like Filter, requires regular connections (quads or cubes). It handles both position- and connection-dependent data.

If the data are vectors, each element of a vector is transformed independently. Because the module returns a 0/1 output, its output is always TYPE_UBYTE.

Note: Data along the boundary are replicated to fill the overlap region for the filter.

Components

Modifies the "data" component. All other components are propagated to the output.

Example Visual Program

`UsingMorph.net`

See Also

Compute, Filter

Normals

Category

Rendering

Function

Computes normals for shading a specified surface.

Syntax

```
normals = Normals(surface, method);
```

Inputs

Name	Type	Default	Description
surface	geometry field	none	surface on which to compute normals
method	string	"positions"	component on which to base normals

Outputs

Name	Type	Description
normals	field	the surface with normals

Functional Details

surface is the surface for which normals are to be created.

method is the computational method used to create normals for the specified surface. If the parameter specifies:

- “connections,” the normal for each connection in the field is computed as the perpendicular to that element.
- “faces,” the normal for each face is computed as the perpendicular to that face. This method is intended for use with faces, loops, and edges data (see “Faces, Loops, and Edges Components” on page 24 in *IBM Visualization Data Explorer User’s Guide*).
- “positions,” the normal at each position in the field is determined by averaging the face normals of the quads or triangles incident on the point. Each face normal is weighted by the distance between the position and the centroid of the face.

Notes:

1. The Normals module assumes that the triangles or quads have consistent point orderings so that the average of the face normals at a given point is meaningful.
2. The Shade module also adds normals to a surface.
3. Smooth shading (**method**=“positions”) is not supported for faces, loops, and edges data. However, you can convert faces, loops, and edges data to triangles using Refine, and then perform smooth shading.

Components

Creates a “normals” component that is position dependent. All other input components are propagated to the output.

Example Visual Programs

`WarpingPositions.net`

See Also

FaceNormals, Shade, Display, Render, Image

Options

Category

Structuring

Function

Associates one or more attributes with a specified object.

Syntax

```
output = Options(input, attribute, value, ...);
```

Inputs

Name	Type	Default	Description
input	object	none	object whose attributes are to be set
attribute	string	no default	attribute to be added
value	value or string or object	no default	value of the attribute
...			additional attribute-value pair(s)

Outputs

Name	Type	Description
output	object	the object with attributes added

Functional Details

This module associates attributes with an object. (Attributes can be extracted from an object with the Attribute module.) Attributes are used by some modules to determine the behavior of a particular input object. For example, you can add attributes to objects to tell the Plot module how to draw markers on particular lines (see Table 8 on page 225 and Plot). While there is a set of attributes which Data Explorer modules understand, you can also add your own attributes to objects, to be interpreted by user-written modules.

You can remove an already present attribute by setting its value to null.

input	names the object to which one or more attributes are to be added.
attribute	names the attribute to be added.
value	specifies the value of the attribute
...	One or more additional pairs of parameters, each specifying an **attribute** and a **value** to be associated with **input**.

A single Options module can specify a maximum of 21 **attribute-object** pairs. In the user interface, the default number of enabled tab-pairs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Table 8. Attributes which have predefined meanings in Data Explorer

Attribute name	**Attribute values**	**Relevant module**
label	any user-supplied label	Plot
scatter	1 or 0	Plot
mark	“circle”, “diamond”, “dot”, “square”, “star”, “triangle”, “x”	Plot
mark every	positive integer	Plot
mark scale	positive scalar	Plot
fuzz	integer	Display, Render, Image
ambient	positive scalar	Display, Render, Image
diffuse	positive scalar	Display, Render, Image
specular	positive scalar	Display, Render, Image
shininess	positive integer	Display, Render, Image
shade	0 or 1	Display, Render, Image
opacity multiplier	positive scalar	Display, Render, Image (for volume rendering)
color multiplier	positive scalar	Display, Render, Image (for volume rendering)
texture	a texture map	Display, Image
antialias	“lines”	Display, Image
line width	positive integer	Display, Image
direct color map	0 or 1	Display
cache	0 or 1	Display, Image
rendering mode	software or hardware	Display, Image
rendering approximation	“none”, “box”, “dots”, or “wireframe”	Display, Image
render every	positive integer	Display, Image
pickable	0 or 1	Pick
marked component	string	Mark, Unmark

Attributes may also be added to arrays to add information which may be needed to interpret an array as a Data Explorer component. For example, you can add a “ref” attribute with a value of “positions” to an integer list which you intend to be used as a connections component in a field. This use of **Options** should be done only with a solid understanding of the Data Explorer data model. See “Standard Attributes” on page 25 in *IBM Visualization Data Explorer User’s Guide*.

Components

The output object is the same as the input object except for the added attributes. All input components are propagated to the output.

Options

Example Visual Programs

```
PlotTwoLines.net
FatLines.net
```

See Also

Attribute, AutoAxes, Display, Plot, Render

Output

Category

Special

Function

Defines an output parameter for a macro.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
`parameter`	object	none	output value of macro

Functional Details

Use the configuration dialog box of this module to specify the parameter name, add a brief description, and specify the tab position on the macro icon. For additional information, see “Creating Macros” on page 149 in *IBM Visualization Data Explorer User’s Guide*.

Example Macro and Program

`MakeLineMacro.net` is used by the visual program `PlotLine2.net`

See Also

DXLOutput, Input, DXLInput, DXLInputNamed

Overlay

Category

Rendering

Function

Overlays one image with another.

Syntax

combined = Overlay(**overlay, base, blend**);

Inputs

Name	Type	Default	Description
overlay	image	none	overlay image
base	image	none	base image
blend	scalar, vector, field, or string	.5	0: for base image only 1: for overlay image only

Outputs

Name	Type	Description
combined	image	combined image

Functional Details

overlay is the overlay image.

base is the base image under **overlay**.

blend determines how the module combines the two images and what the output image will be. If the parameter is:

- *a scalar value*: the resulting image (**combined**) is equal to

  ((1 - **blend**) × **base**) + (**blend** × **overlay**)

- *a vector or string*: this value will be interpreted as an RGB color. The **combined** image is the base image *except* for the pixels where the base image is equal to **blend**. These pixels will be taken from the overlay image. This specification allows you to perform chromakeying.

  Two pixels are considered to have the same color if the corresponding component colors (i.e., red, green, blue) of one are each within 0.1% of the other.

  The **blend** can also be specified (in a string) as one of the standard X Window System** colors (see "Color" on page 75).

- *a field of position-dependent data*: each data value in the field is used as a blending value to overlay the two pixels (one from each image) corresponding to that position. That is, the blending is pixel by pixel. The "blending" field must have a grid that is compatible with both the overlay and the base image.

The **blend** values in the field must be scalar values between 0.0 and 1.0.

Notes:

1. The images **base** and **overlay** must be the same size and have the same grid positions. The “colors” component must be dependent on “positions.” The **base** and **overlay** images must both have the same partitioning (or none).
2. To add two images together, or subtract one from the other, use the Compute module. First mark the “colors” component in each image (with the Mark module), then, use Compute to perform the desired operation on each. Finally, use Unmark to return the modified data to the “colors” component.

Components

Modifies the “colors” component. All other input components are propagated to the output.

Example Visual Program

`UsingOverlay.net`

See Also

Arrange, Color, Compute, Display, Render

Parse

Category

Annotation

Function

Extracts values from an input string.

Syntax

```
value, ... = Parse(input, format);
```

Inputs

Name	Type	Default	Description
input	string	none	string to be operated on
format	string	"%s"	format control string

Outputs

Name	Type	Description
value	value or string	value extracted from input string
...	value or string	additional values extracted

Functional Details

This module uses a format-control string (**format**) to extract values from a specified input string (**input**).

Note: The control string resembles a C-language scanf format string.

The “%” symbol in the control string specifies that extraction is to begin at the corresponding position in the input string. The character immediately following this symbol specifies the type of value to be extracted:

c: single character
d: integer
f: floating point (with a fixed number of digits after the decimal point)
g: general (scientific notation if appropriate)
s: string.

The number of outputs is equal to the number of extractions specified in **format**.

Example

Given the statements:

```
string = "temperature = 45.8  index = 4 color = red"
format = "temperature = %f index = %d color = %s"
output1, output2, output3 = Format(string, format);
```

the three outputs of parse will be 45.8, 4, and “red.”

Example Visual Program

`UsingParse.net`

See Also

Format

Partition

Category

Import and Export

Function

Partitions a data set for parallel processing.

Syntax

```
partitioned = Partition(input, n, size);
```

Inputs

Name	Type	Default	Description
input	field or group	none	field to be partitioned
n	integer	machine dependent	maximum number of subparts
size	integer	one primitive	threshold for partitioning

Outputs

Name	Type	Description
partitioned	field or group	set of partitioned fields

Functional Details

This module partitions a data set for parallel processing on an SMP multiprocessor machine. (You must be using Data Explorer SMP to take advantage of this feature.) Its output is a composite field, which is treated as a single entity by other modules.

input is the input object to be partitioned. If this parameter specifies a group, each group member is partitioned with the same **n** and **size** parameters.

n is the approximate number of partitions to be created. However, the module will not create partitions smaller than **size** (see below).

size is the minimum number of connection elements per partition.

If **n** × **size** is larger than the total number of points, the output number of partitions may be smaller than **n**

If you do not specify **n** or **size**, appropriate default values are supplied, depending on the number of processors available.

Note: On a uniprocessor machine, n = 1 by default. Consequently, the same programs can be run on uniprocessor and multiprocessor machines without modification.

Components

All components in the input are propagated to the output.

Example Visual Programs

`ExampleSMP.net`

See Also

Import

Pick

Category

Special

Function

Outputs a pick structure.

Syntax

```
picked = Pick(pickname, imagename, locations, reexecute, first,
              persistent, interpolate, object, camera);
```

Inputs

Name	Type	Default	Description
pickname	string	none	name of cached picks
imagename	string	none	name of cached scene
locations	vectorlist	no default	2-D screen coordinate pick positions
reexecute	flag	none	cause reexecution whenever pick list is reset
first	flag	1	0: include all “picks” 1: include only first “picks”
persistent	flag	1	0: “picks” not saved 1: “picks” saved
interpolate	integer	0	0: no data interpolation 1: nearest vertex interpolation 2: interpolate
object	object	no default	object to be “poked”
camera	camera	none	camera used to create scene picked in

Outputs

Name	Type	Description
picked	field	pick structure

Functional Details

Picking involves using the mouse to determine information about an object rendered in an image. The user may select one or more points on the screen by positioning the mouse and pressing the left button; each such action is referred to as a “poke.” Each poke may intersect the object at one or more places or may miss the object altogether. Each intersection is called a “pick.” For example, a poke on a spherical isosurface will result in two picks, one on the front surface and one on the rear.

Individual subobjects in a structured scene may be made unpickable by attaching an attribute named “pickable”, with a value of 0 (zero), to the root of the subobject, using the Options module.

The information returned by Pick in **picked** includes the positions in the (xyz) world-coordinate system at which the picks occurred. These positions (contained in the "positions" component of **picked**) may be used to determine the coordinates of points on objects, to start streamlines, or to label points on the surface (by using AutoGlyph appropriately). These kinds of operations can be done directly in a visual program, without writing a special module.

In addition to the pick positions, **picked** contains information identifying the individual elements of the object structure that was picked. This information can be used to select elements of the object structure at any level down to the connections element and vertex closest to the pick point. See *IBM Visualization Data Explorer Programmer's Reference* for an example module that uses the position information contained in the pick structure.

pickname is used only by the user interface and is not intended to be set by users.

imagename is used only by the user interface and is not intended to be set by users.

locations is a list of 2 dimensional screen coordinates pixel positions identifying the picks. If you are using the Image tool, this parameter is set automatically for you by the user interface. If you are using SuperviseWindow and SuperviseState, then the **events** output of SuperviseState should be used to provide this input.

reexecute is used only by the user interface and is not intended to be set by users.

first specifies whether all picks generated by a poke intersecting an object are to be added to the pick structure or only the first pick (i.e., the pick closest to the viewpoint):

0 All picks are added to the output.

1 Only the first pick is reported for each poke.

persistent specifies whether or not picks are saved between executions:

0 Picks are not saved.

1 Picks are saved and are deleted only when the user makes new pokes.

Note: When new picks are made, the previous pick points are discarded.

interpolate specifies whether a "data" component should be created in **picked** and, if so, how the interpolation of values for that component should be performed.

If the **interpolate** parameter is set to 1 or 2, the Pick output object will contain a set of components matching the set of dependent components in the picked object. Each of the components in the output object is dependent on the "positions" component of **picked**. For components that are dependent on connections in the picked object, the data for the picked element will be placed in the output component.

For components that are dependent on positions in the picked object, the data placed in the output object is determined by one of two options:

1 = "nearest vertex"
: The data corresponding to the vertex nearest the pick point is extracted.

2 = "interpolated data"
: The data is interpolated from the vertex data of the picked element.

In either case, an additional component is created (called "closest vertex") that receives the coordinates of the vertex closest to the pick point among the vertices of the picked element.

Note: Regardless of which of the two options is selected, if different picks result in different sets of components or different data types in components of the same name, an error results.

`object` allows the user to override the object being picked. By default, this object is the one rendered in the Image window associated with the previous execution of the graph. If the scene consists of several objects collected before rendering, and if only some of them are to be made pickable, the user can pass those objects directly into Pick. The **`object`** parameter also makes it possible to pick in an object that lies in the same coordinate space as the scene does but is not in fact a part of the scene.

`camera` sets the camera used to create the scene being picked in. If you are using the Image tool, this parameter is set automatically for you by the user interface. If you are using SuperviseState and SuperviseWindow, then you must provide the camera used to render the scene.

Note: Pick currently supports surfaces, lines, and points. Picking does not support volume elements; if a poke is made on a volume object in the image, no picks will result.

Example Visual Program

`PickStreamline.net`

See Also

SuperviseWindow, SuperviseState

"Using Pick" on page 87 in *IBM Visualization Data Explorer User's Guide*.

Plot

Category

Annotation

Function

Creates a 2-dimensional plot.

Syntax

```
plot = Plot(input, labels, ticks, corners, adjust, frame, type, grid,
            aspect, colors, annotation, labelscale, font, input2,
            label2, ticks2, corners2, type2,
            xticklocations, y1ticklocations, y2ticklocations,
            xticklabels, y1ticklabels, y2ticklabels);
```

Inputs

Name	Type	Default	Description
input	field or group	none	data to be plotted
labels	string list	{“x,” “y”}	axis label
ticks	integer list	10	approximate number of tick marks
corners	vector list or object	{[xmin, ymin], [xmax, ymax]}	plot limits
adjust	flag	input dependent	0: end points not adjusted 1: end points adjusted to match tick marks
frame	integer	0	framing style for plot
type	string list	"lin"	plot type
grid	integer	0	grid style
aspect	scalar or string	1.0	y:x (aspect) ratio of resulting plot
colors	vector list or string list	appropriate	colors for annotation
annotation	string list	"all"	annotation objects to be colored
labelscale	scalar	1.0	scale factor for labels
font	string	standard	font for labels
input2	field or group	no default	second set of data to be plotted
label2	string	{“y2”}	label for second y-axis
ticks2	integer	input dependent	approximate number of tick marks for second y-axis.
corners2	vector or object	{[ymin, ymax]}	plot limits for second y-axis
type2	string	"lin"	plot type for second y-axis
xticklocations	scalar list	appropriate	x tick locations
y1ticklocations	scalar list	appropriate	y1 tick locations

Name	Type	Default	Description
y2ticklocations	scalar list	appropriate	y2 tick locations
xticklabels	string list	xticklocations	x tick labels
y1ticklabels	string list	y1ticklocations	y1 tick labels
y2ticklabels	string list	y2ticklocations	y2 tick labels

Outputs

Name	Type	Description
plot	field	the plot

Functional Details

This module creates a 2-dimensional plot from a line or set of lines. The following plot characteristics can be set on the line or lines with the Options module (in the **Structuring** category) before they are passed to Plot: line label, line marker type, marker size, number of markers, and the choice of creating a scatter plot or not (see “Creating a Legend” on page 240, “Using Line Markers” on page 240, and “Scatter Plots” on page 241).

input is a field or group of fields, where each field has 1-dimensional positions representing the “x”-values, and 1-dimensional data representing the “y”-values. If the input is a group of Fields, each Field is plotted as a separate line.

There can be a “connections” component of element type “lines” connecting the positions; if a connections component for a given Field does not exist, the module adds one. The input can have a “colors” component, in which case the output plot preserves those colors. If the input does not have a “colors” component, the line is colored white.

labels specifies the labels for the axes of the plot.

ticks specifies the number of tick marks to be placed on the plot axes. The default is 10.

If the parameter value is a single integer, then approximately that many tick marks are placed along the axes. If the parameter list is a two-element integer list, the first element is interpreted as the approximate number of tick marks to be placed on the x-axis; the second element, the approximate number to be placed on the y-axis.

corners specifies the limits of the plot axes. (The default limits are set by the input line or lines.) The specification can be a vector list or an object. If the specification is a vector list, then all points that lie outside of **corners** are eliminated from the plot. If the specification is an object (such as a group of lines), the module determines the minimum and maximum x- and y-values for the entire object and uses those to set the plot limits.

adjust determines whether the axes end at tick marks or not.

frame	determines how the plot is framed. In all cases, axes and tick labels are drawn on the left side and bottom of the plot. For **frame** = 1, additional lines (without ticks) are drawn on the right side and top of the plot. For **frame** = 2, lines and major and minor ticks are drawn on the right side and top of the plot.
type	specifies the plot type. It is a string or string list and must be: 1. "lin" (linear plot) 2. "log" (logarithmic plot) 3. any two-element combination of these (e.g., {"lin," "lin"} or {"log," "lin"}). If **type** is a single string, both the *x*- and the *y*-axis are of that type. If type is two strings, then the first string applies to the *x*-axis while the second string applies to the *y*-axis.
grid	specifies whether and how to draw grid lines along major ticks. If **grid** = 0, no grid lines are drawn. If **grid** = 1, then grid lines are drawn horizontally along major ticks. If **grid** = 2, then grid lines are drawn vertically along major ticks. If **grid** = 3, grid lines are drawn both horizontally and vertically along major ticks. If **grid** is not equal to 0, **adjust** is set to 1.
aspect	allows you to set the approximate *y:x* aspect ratio of the resulting plot. You can also use the Scale module before Plot to adjust the aspect ratio. The default is an aspect ratio of 1.0. If you want Plot to simply use the actual aspect ratio of the data, set **aspect** to "inherent". Note that if a second plot is drawn (i.e. **input2** is provided), **aspect** is set to 1.0 even if "inherent" is specified.
colors	together with **annotation** (see below), can be used to set the color of one or more components of the plot. **colors**: specifies a single color (an RGB vector or color-name string) or a list of colors. Color-name strings must be from the list of defined color strings (see "Color" on page 75). If **colors** is a single string and **annotation** is not specified or is "all," then that color is used for all axes annotation. Otherwise, the number of colors in **colors** must match the number of **annotation** strings exactly and in one-to-one correspondence. By default, a background is not drawn.
annotation	can be one of the following: "all," "axes," "background," "grid," "labels," or "ticks."
labelscale	allows you to change the size of the axes and tick labels. For example, to make the labels twice as large, specify **labelscale** = 2.
font	specifies the font used for labels. The default is "standard," which uses a fixed font for the tick-mark labels and a variable font for the axes labels. You may specify **font** as any of the defined fonts supplied with Data Explorer:

area	gothicit_t	pitman	roman_ext
cyril_d	greek_d	roman_d	script_d
fixed	greek_s	roman_dser	script_s
gothiceng_t	italic_d	roman_s	variable
gothicger_t	italic_t	roman_tser	

For more information, see Appendix E, "Data Explorer Fonts" on page 307 in *IBM Visualization Data Explorer User's Guide*.

The Plot module can also generate a second set of lines and a second y-axis on the right side of the plot. If a second plot is drawn, the **aspect** ratio defaults to 1. For this second plot and axis, you must repeat several specifications:

input2 is a second field or group of fields (see **input** above).

label2 is the label for the second *y*-axis.

ticks2 is the approximate number of ticks for the second *y*-axis. A negative integer specifies outward, as opposed to inward, pointing ticks.

corners2 (see **corners on page 238**)
specifies the **min** and **max** for the second “y”-axis.

type2 specifies the type of second *y*-axis and must be either “log” or “lin.”

xticklocations
explicit set of tick locations for the x axis. If specified, overrides the values for tick locations as determined from **ticks**.

y1ticklocations
explicit set of tick locations for the first y axis. If specified, overrides the values for tick locations as determined from **ticks**.

y2ticklocations
explicit set of tick locations for the second y axis. If specified, overrides the values for tick locations as determined from **ticks**.

xticklabels list of labels to be associated with the specified **xticklocations**. If **xticklabels** is specified and **xticklocations** is not specified, then **xticklocations** to the integers 0 to n-1, where n is the number of items in **xticklabels**.

y1ticklabels list of labels to be associated with the specified **y1ticklocations**. If **y1ticklabels** is specified and **y1ticklocations** is not specified, then **y1ticklocations** to the integers 0 to n-1, where n is the number of items in **y1ticklabels**.

y2ticklabels list of labels to be associated with the specified **y2ticklocations**. If **y2ticklabels** is specified and **y2ticklocations** is not specified, then **y2ticklocations** to the integers 0 to n-1, where n is the number of items in **y2ticklabels**.

Creating a Legend: To create a legend for the plot, use Options to set a “label” attribute on the incoming lines. Plot will then create an appropriate plot legend and place it in the upper right corner (see the first example below). The legend contains a short line segment with a color matching that of the line it corresponds to, along with the label for the line. The “label” attribute can also be set in the Data Explorer file format (see Appendix B, “Importing Data: File Formats” on page 241 in *IBM Visualization Data Explorer User’s Guide*).

Using Line Markers You may also choose markers for your plot lines. You may set a “mark” attribute on the input lines using the Options module. “mark” should have as a value one of the following: “circle,” “diamond,” “dot,” “square,” “star,” “triangle,” or “x.” By default, a mark is placed at every point on the line. You can also set a “mark every” attribute with an integer as a value. For example, the value 3 would cause every third point to be marked. You can control the size of the mark using the “mark scale” attribute, which applies a scale factor to the default mark size. For example, a “mark scale” value of 2 would make the marks twice as large. If you are using the “label” attribute, the mark will also appear in the legend.

Note: The “dot” mark type will only be visible if you also set the “scatter” attribute to 1 (see “Scatter Plots”).

Different colored lines If you want your lines to have different colors, simply pass them through the Color module before passing them to Plot.

Scatter Plots To create a scatter plot (with no line drawn between markers), set the “scatter” attribute on the line to 1 (one). (The default is a line between markers.)***:***

Notes:

1. If **`corners`** is more restrictive than the **`ticklocations`**, then the given locations outside the corners are not shown.
2. If **`corners`** is less restrictive than the given **`ticklocations`**, or if **`corners`** is not specified, then all given tick locations are shown, whether or not there is data there.
3. If **`ticklocations`** is specified, then the data range determines the extent of the axes, unless **`corners`** is specified, in which case the given corners are used.

Bar Charts

If the input to Plot has a “data” component that is dependent on “connections” rather than “positions,” a bar chart is created.

Components

Creates “positions,” “connections,” and “colors” components.

Example Visual Programs

```
PlotLine.net
PlotLine2.net
PlotTwoLines.net
PlotSpecifyTicks.net
```

See Also

Compute, Construct, Histogram, Options

Post

Category

Transformation

Function

Changes the data dependency of a field.

Syntax

```
output = Post(field, dependency);
```

Inputs

Name	Type	Default	Description
input	field	none	field whose data dependency is to be changed
dependency	string	"positions"	data dependency desired

Outputs

Name	Type	Description
output	field	field with data dependency changed

Functional Details

input is a field whose data dependency on one component is to be changed to dependency on another component.

dependency is the output dependency desired and must be “positions” or “connections.”

If you are changing the dependency from “positions” to “connections,” all components that are dependent on “positions” (but do not contain references to that or any other component, such as colors, data, and normals) are changed to “connections” dependency. The Post module uses an average of the vertex values for each connection element to determine the new value for the component.

Similarly, when changing from connection dependence to position dependence, the Post module associates with a vertex the average of the values for each connection touching that vertex.

Components

Creates a new component for each input component that is either dependent on connections if **dependency** is “positions” or that is dependent on positions if the **dependency** is “connections.” The dependencies of the new components are set to **dependency**.

Example Visual Programs

```
Thunder_cellcentered.net
SIMPLE/Post.net
```

Print

Category

Debugging

Function

Prints an object.

Syntax

```
Print(object, options, component);
```

Inputs

Name	Type	Default	Description
object	object	none	object to print
options	string	"o"	printing options
component	string or string list	all components	component or components to print

Functional Details

This module prints **object** according to the specifications in **options**. Each character in the options string specifies printing a particular portion of the information about the object:

r	recursively traverse the object
o	print only the top level of the object
d	print first and last 25 items in arrays, as well as headers
D	print all the items in arrays as well as headers
x	for debugging, print in expanded form (lists the object address, the object tab, and the reference counts of the object)
n	print object to ***n*** levels.

The **component** parameter controls which components of **object** are printed. By default, all components are printed. All values are printed in decimal format except for byte array data, which are printed in hexadecimal.

In the user interface, the output of the Print module appears in the Message window.

Script Language Example

In the following example, the first call to Print does not recurse through the structure but simply reports (prints) that **both** is a group with two members. The second call recurses through the group and reports that each member of the group has six components, what those components are, and how many items there are in each. The third call provides this information and, in addition, the first and last 25 items in the “data” component of each field. The final call prints out all of the items in both the “positions” and “box” components.

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
electrondensity = Partition(electrondensity);
iso1 = Isosurface(electrondensity,0.1);
iso2 = Isosurface(electrondensity,0.3);
both = Collect(iso1,iso2);
Print(both);
Print(both,"r");
Print(both,"d", "data");
Print(both,"D", {"positions", "box"});
```

See Also

Echo, Describe

Probe

Category

Special

Function

Outputs an x,y,z point.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`point`	vector	x,y,z position of probe point

Functional Details

This tool generates an x,y,z position from mouse-driven input. see "Using Probes (Cursors)" on page 85 in *IBM Visualization Data Explorer User's Guide*.

Note: Because the output is in the units of the rendered object, the result may be unexpected if you have scaled or translated the object before rendering it, since the x,y,z point will be the transformed units.

Example Visual Programs

```
AnnotationGlyphs.net
PlotLine2.net
ProbeText.net
UsingClipPlane.net
UsingProbes.net
UsingStreakline.net
```

See Also

Pick, ProbeList

ProbeList

Category

Special

Function

Outputs a list of x,y,z points.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`point`	vector list	x,y,z positions of probe points

Functional Details

This tool generates a list of x,y,z positions from mouse-driven input. For more information about probes, see "Using Probes (Cursors)" on page 85 in *IBM Visualization Data Explorer User's Guide*.

Note: Because the output is in the units of the rendered object, the result may be unexpected if you have scaled or translated the object before rendering it, since the x,y,z point will be the transformed units.

Example Visual Programs

```
AnnotationGlyphs.net
PlotLine2.net
ProbeText.net
UsingClipPlane.net
UsingProbes.net
UsingStreakline.net
```

See Also

Pick, Probe

QuantizeImage

Category

Transformation

Function

Converts an RGB image to a byte image with a colormap

Syntax

```
images = QuantizeImage(Images, nColors);
```

Inputs

Name	Type	Default	Description
images	image, image series	(none)	image(s) to quantize
nColors	integer	256	maximum number of colors to use

Outputs

Name	Type	Description
images	image, image series	resulting quantized image(s)

Functional Details

QuantizeImage converts an image from a format in which the “colors” component is a list of RGB (red, green, blue) vectors, with as many entries as there are pixels in the image, to a format in which the “colors” component is simply a list of unsigned bytes which are interpreted as pointers into a color table.

QuantizeImage sets a “direct color map” attribute of 1 on its output. When the output **images** is passed to Display, this will force Display to use the colormap specified by QuantizeImage, rather than a default colormap (see “Using Direct Color Maps” on page 111). This will be the case even if you set the environment variable DX8BITCMAP to 1. You can use the Options module to set a “direct color map” attribute of 0 on the output of QuantizeImage to force Display to use a shared colormap.

images	is the image or series of images to be quantized.
nColors	is the maximum number of entries in the color table.

The output **images** will be a new image or series of images in which the color table is attached as a component called “color map”.

The advantage of using QuantizeImage is that the image will consume much less memory.

Components

Changes the “colors” component. Adds a “color map” component.

Example Visual Programs

`SIMPLE/QuantizeImage.net`

See Also

Render, ReadImage, WriteImage, Display, Options

ReadImage

Category

Import and Export

Function

Reads an image from an image file.

Syntax

image = ReadImage(**name, format, start, end, delta, width, height**);

Inputs

Name	Type	Default	Description
name	string	"image"	file name
format	string	"rgb" or input dependent	file format
start	integer	first frame	first movie frame
end	integer	last frame	last movie frame
delta	integer	1	delta of images to be read
width	integer	input dependent	width of image
height	integer	input dependent	height of image
delayed	flag	environment dependent	use delayed colors if present in file
colortype	string	environment dependent	data type for colors

Outputs

Name	Type	Description
image	image or image series	resulting image

Functional Details

This module supports four basic file formats: RGB and TIFF (Tag Image File Format), GIF (Graphics Interchange Format), and MIFF.

name is the name of the image file.

If **name** contains a series, the parameters **start**, **end**, and **delta** can be used to read a subset of the images (see parameter descriptions).

format specifies the format of the image file. This parameter is not required if **name** includes a file extension defining the format (see following table). If **name** is specified without the appropriate extension, then the file format must be specified. If **format** is specified as “rgb,” and **name** is “image,” the module will first try to open image.rgb. If that fails, it will then try to open image.

File Type	Format Specifier	Expected File Extension(s)	Multiframe/Series Data
RGB	"rgb" “r+g+b”	.rgb and .size .r, .g, .b, .size,	Yes
TIFF	"tiff"	.tiff	Yes
GIF	"gif"	.gif	No
MIFF	"miff"	.miff	Yes

start and **end** specify the first and last frame to be read from an image file containing a series.

delta specifies the increment in counting the frames in the range from **start** to **end**. For example, if the first and last frames are 10 and 20 respectively, and **delta** = 2, the output (**image**) is a series group with six members (indexed from 0 to 5). Frame numbers (10, 12,..., 20 in this example) are preserved as the series position number.

width and **height**
are used only for RGB format files. The module obtains information about the size and number of images from **name.size**, an ASCII file containing the string “w×h×f,” (where w and h are pixel width and height of image respectively, and f is the number of frames).

If the **.size** file is not available, then **width** and **height** can be used to specify the size of the image(s).

delayed specifies whether ReadImage should create a “delayed color” image if the image file is stored in an “image with colormap” format. By default, ReadImage will create a “delayed color” image if possible, unless the environment variable DXDELAYEDCOLORS is set to 0, or the **delayed** parameter is set to 0.

colortype specifies whether the colors in the image should be byte or floating point. By default, ReadImage will create byte colors unless the DXPIXELTYPE environment variable is set to DXFloat, or **colortype** is set to “float”.

Notes:

1. RGB format files can be either “rgb” or “r+g+b.” The file format “rgb” contains the image with the bytes for the red, green, and blue values interleaved. An alternate file format is “r+g+b,” where three output files contain the non-interleaved image: “**name.r**,” “**name.g**,” and “**name.b**.”

   TIFF format files must be either full color RGB images (TIFF Class R) or palette color RGB images (TIFF Class P), uncompressed (“Compression=1”), and interleaved (“PlanarConfiguration=1”). If the extension is not part of the name, the format must be set to “tiff.” Information about the size of the image is obtained from the file itself, since TIFF files are self-describing. (For more information on file formats, see “WriteImage” on page 374.)

2. The default behavior of ReadImage is different in Data Explorer version 3.1.4 and above in terms of the type of image it creates internally. If you do not specify the environment variable DXPIXELTYPE as “DXFloat”, ReadImage will

create byte colors internally. This will result in reduced storage required for the image, and will only affect networks that require the colors to be of type float, in which case the ReadImage parameter “colortype” should be set to “float”, or the `DXPIXELTYPE` environment variable should be set to “DXFloat”.

3. ReadImage will also maintain the delayed color status of an image, as will be the case for GIF, some TIFF, and some MIFF format files. If the visualization program requires full colors for each pixel, set the ReadImage “delayed” parameter to false (0) or set the environment variable `DXDELAYEDCOLORS` to 0.
4. If you want to convert an image file that does not already have delayed colors to one that does, use ReadImage to load it in and then use QuantizeImage to make it delayed color.

Components

Creates an output with “positions,” “connections,” and “colors” components.

Example Visual Programs

```
ReadImage.net
UsingFilter.net
```

See Also

WriteImage

ReadImageWindow

Category

Windows

Function

Retrieves the contents of a window

Syntax

```
image = ReadImageWindow(where);
```

Inputs

Name	Type	Default	Description
where	window	none	window identifier

Outputs

Name	Type	Description
image	field	image found in window

Functional Details

ReadImageWindow retrieves the contents of a window.

where	is the window identifier, and should be the **where** output of either Display or Image.

Note that on some platforms, if you call ReadImageWindow on a window that is partially obscured (by another window for example), some pixels may be missing from the output **image**.

See Also

Display, Image

Receiver

Category

Special

Function

Receives an object from a Transmitter.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`object`	object	the object received

Functional Details

To maintain the modularity and readability of large programs, Data Explorer provides two tools that allow connections between input and output tabs of separate modules without the use of a visible connecting line. These tools, Transmitter and Receiver, allow you to separate visual programs into logical blocks. For example, the output of several logical blocks can be transmitted to another block that receives them, collects them, and produces an image.

Receivers and Transmitters can also be used to communicate information between pages in the Visual Program Editor (see "Creating pages in the VPE" on page 115 in *IBM Visualization Data Explorer User's Guide*). Pages are a valuable way of structuring complex visual programs into logical blocks.

Note: Macros provide another way of structuring visual programs in logical blocks (see 7.2, "Creating and Using Macros" on page 149 in *IBM Visualization Data Explorer User's Guide*).

To remotely connect input and output tabs:

1. Select a Transmitter tool (in the Special category in the tool palette) and place it near the output tab of the module that is to be "connected."
2. Connect the module's output tab to the Transmitter's input tab.
3. Select a Receiver tool (also in the Special category), and place it near the input tab of the other module that is to be "connected."
4. Connect the Receiver's output tab to the second module's input tab.

   The Transmitter and Receiver are now connected.

The Receiver automatically assumes the same name as the Transmitter. More than one Receiver can be connected to a single Transmitter and they assume the same name until a new Transmitter is placed on the VPE canvas.

Notes:

1. To change the name of a Transmitter and Receiver, use the **`Notation`** field of the appropriate configuration dialog box (see "Entering Values in a Configuration Dialog Box" on page 107 in *IBM Visualization Data Explorer User's Guide*). Changing the name of a Transmitter changes the name of all the Receivers connected to it. Changing the name of a Receiver affects only that receiver.
2. For more information see "Using Transmitters and Receivers" on page 106 in *IBM Visualization Data Explorer User's Guide*.

Example Visual Programs

Many of the example visual programs use receivers and transmitters, including:

```
AlternateVisualizations.net
Imide_potential.net
```

See Also

Transmitter

Reduce

Category

Import and Export

Function

Computes a reduced-resolution version (or group of versions) of a field.

Syntax

`reduced` `= Reduce(`**`input, factor`**`);`

Inputs

Name	Type	Default	Description
`input`	field	none	field to be reduced
`factor`	scalar list or vector list	2	reduction factor(s)

Outputs

Name	Type	Description
`reduced`	field or group	set of reduced-resolution data

Functional Details

Reduces the resolution of regularly connected grids. Box filtering is applied to minimize aliasing effects in the components that are associated with the grid. It filters and reduces the number of items in the "data" component of the input field and also in any components that are position or connection dependent.

`input`	is the field to be reduced. It must have regular connections.
`factor`	specifies the reduction factor(s) to be used in reducing the resolution of the field.

If this parameter specifies:

- *a single scalar value:* the output is reduced by that factor in all dimensions.
- *a vector:* each vector element is applied to the corresponding dimension.
- *a list of scalars or vectors:* the output is a group of fields, each with the corresponding factor applied.

Components

All components in the input are propagated to the output. Components dependent on "positions" or "connections" are filtered.

Example Visual Programs

```
GeneralImport2.net
ManipulateGroups.net
VolumeRendering.net
SIMPLE/Reduce.net
```

See Also

Refine, Select, Display

Refine

Category

Import and Export

Function

“Refines” a grid or changes its element type.

Syntax

```
refined = Refine(input, level);
```

Inputs

Name	Type	Default	Description
input	field	none	field to be “refined”
level	integer or string	1	level of refinement

Outputs

Name	Type	Description
refined	field	"refined" input

Functional Details

This module resamples a grid at a finer resolution or changes the element type.

input is the object to be refined. It must have “connections” or “faces.”

level specifies either the level or the type of refinement:

- *an integer* specifies the number of levels (in powers of 2) of refinement. New positions and connections are added, and any components that are position or connection dependent are interpolated linearly and placed in the output.

  For quads and triangles, one level of refinement increases the number of elements by a factor of 4, two levels by a factor of 16, and so on. For cubes and tetrahedra, one level of refinement increases the number of elements by a factor of 8, two levels by a factor of 64, and so on.
- *a string* has two values: “tetrahedra” refines cubes to tetrahedra; “triangles” refines quads or faces to triangles.

Components

All components in the input are propagated to the output. The “connections” component is modified.

Example Visual Programs

```
ManipulateGroups.net
SIMPLE/Refine.net
```

See Also

Reduce, Display

Regrid

Category

Realization

Function

Maps scattered points onto a grid.

Syntax

```
output = Regrid(input, grid, nearest, radius, exponent, missing);
```

Inputs

Name	Type	Default	Description
`input`	field or vector list	none	field with positions to regrid
`grid`	field	none	grid to be used as template
`nearest`	integer or string	1	number of nearest neighbors to use
`radius`	scalar or string	"infinity"	radius from grid point
`exponent`	scalar	1.0	weighting exponent
`missing`	value	no default	missing value to be inserted if necessary

Outputs

Name	Type	Description
`output`	field	regridded field

Functional Details

This module uses a specified set of scattered points (**`input`**) to assign data values to every position of a specified grid.

`input` should be either (1) a field with a 1-, 2-, or 3-dimensional "positions" component or (2) a list of 1-, 2-, or 3-dimensional vectors. In the second case, the vectors are interpreted as positions.

`grid` is required. It specifies the grid to be used as a base for creating a "connections" component. The dimensionality of positions in this grid must match that of the positions in **`input`**. The specified **`grid`** could be created with the Construct module.

`nearest` must be an integer or the string "infinity." An integer value specifies the number of nearest points (to each grid point) to be used in computing an average data value for that grid point.

`radius` specifies the maximum radius (from the grid point) within which the nearest neighbors can be found. The parameter must specify a scalar value or the string "infinity."

exponent	The averaging method is a weighted average. The expression for this average is 1/radius$^{(\mathbf{exponent})}$. The default value is 1.0, reducing the expression to the reciprocal of the radius.
missing	is used when **radius** is set to a value other than “infinity.” The parameter specifies how to treat those grid points for which no points in **input** occur within the specified radius.
	If **missing** *is not set*, the module creates an “invalid positions” component, and grid points with no assigned data value are invalidated. If **missing** is set, the data value is inserted for the missing data values. It must match the data component of **input** in rank, type, and shape.
	All components that are position-dependent are treated in the same way as the “data” component.

Note: To remove invalidated positions, use the Include module. However, it is not necessary to remove invalidated positions in order to have them treated as invalid by other modules.

Components

Adds a “connections” component. The “positions” and “connections” components are those of **grid** while all components in **input** that depend on “positions” will be present in the output, modified by averaging.

Example Visual Program

`SIMPLE/Regrid.net`

See Also

Connect, Construct, Include, AutoGrid

Remove

Category

Structuring

Function

Removes components from a field.

Syntax

```
output = Remove(input, name);
```

Inputs

Name	Type	Default	Description
input	field	none	the field from which to one or more components are to be removed
name	string or string list	none	the component(s) to be removed

Outputs

Name	Type	Description
output	field	the field without the named component(s)

Functional Details

This module creates an **output** field containing all components of the **input** field except those specified by **name**.

Components

All input components are propagated to the output except those specified by **name**.

See Also

Rename, Shade

Rename

Category

Structuring

Function

Renames a component in a field.

Syntax

```
output = Rename(input, oldname, newname);
```

Inputs

Name	Type	Default	Description
input	field	none	the field containing the component to be renamed
oldname	string	none	the original name of the component
newname	string	"data"	the new name assigned to the component

Outputs

Name	Type	Description
output	field	field containing the renamed component

Functional Details

This module creates a field (**output**) in which all occurrences of a specified component (**oldname**) in **input** have been assigned a new name (**newname**).

Components

All input components are propagated to the output. The component **oldname** is renamed to **newname**.

Example Visual Programs

```
AlternateVisualizations.net
ReadImage.net
UsingMorph.net
```

See Also

Extract, Mark, Remove, Replace, Unmark

Render

Category

Rendering

Function

Renders an object.

Syntax

```
image = Render(object, camera, format);
```

Inputs

Name	Type	Default	Description
`object`	object	none	object to be rendered
`camera`	camera	none	camera to be used for rendering
`format`	string	standard	format of resulting image

Outputs

Name	Type	Description
`image`	image	resulting image

Functional Details

This module uses a specified camera to create an image of an object.

`object` specifies the object to be rendered, which can contain surfaces, volumes, or combinations of surfaces and volumes.

Note: The current algorithm does not support coincident volumes or volumes in perspective. (See "Display" on page 109 for a discussion of Data Explorer rendering capabilities.)

The specified object must contain a "colors," "front colors," or "back colors" component. Many modules add a default color. In addition, volume rendering (i.e., of cubes or tetrahedra) requires an "opacities" component. For surfaces, the lack of an "opacities" component implies an opaque surface.

`camera` is the camera used to create the image.

Note: A transformed camera cannot be used for this parameter.

`format` is reserved for future use.

Notes:

1. The rendering properties of an object (e.g., shading) can be changed with the Options or Shade module (see "Display" on page 109). Render always invokes the software renderer and so ignores the "render mode" attribute.
2. Render creates 24-bit images if the DXPIXELTYPE environment variable is set to DXByte (the default is 96-bit images).

3. If you use the Display or Image tool rather than the Render module, Data Explorer will automatically choose a hardware-appropriate format for you. It is generally preferable to use one of these tools unless you want to operate on the image itself. For example, filtering the image or arranging several images together requires the Render module.
4. The interactive image-manipulation options provided by Data Explorer in the user interface require the Image tool. See “Controlling the Image: View Control...” on page 74 in *IBM Visualization Data Explorer User’s Guide*.

   See Display for information on:

   “Changing Rendering Properties” on page 112
   “Differences between Hardware and Software Rendering” on page 115
   “Shading” on page 114
   “Object fuzz” on page 114
   “Coloring Objects for Volume Rendering” on page 113

Components

Creates “positions,” “connections,” and “colors” components for the resulting image.

Example Visual Programs

```
PlotLine.net
UsingCompute.net
UsingMorph.net
SIMPLE/Arrange.net
```

See Also

Arrange, Compute, Display, Filter, Image, Options, WriteImage

Reorient

Category

Rendering

Function

Changes the orientation of an image or group of images.

Syntax

```
image = Reorient(image, how);
```

Inputs

Name	Type	Default	Description
`image`	field or group	none	image(s) to be reoriented
`how`	integer	none	specific change of orientation

Outputs

Name	Type	Description
`image`	image	reoriented image

Functional Details

This module rotates or inverts an image. (Use Refine or Reduce to change the size of an image.)

Note: This module is intended for images that will be displayed directly without rendering (i.e., using Display without a camera). So if you are rendering an object (using Image, Display with a camera, or Render), you should instead use Transpose, Rotate, Scale, and Translate to reorient that object before rendering it.

`image`	is an image or group of images. An image is (1) a regular 2-dimensional field *or* the output of Render or ReadImage and (2) must have the following characteristics. • regular 2-dimensional positions and quad connections • position-dependent colors • origin at [0, 0].
`how`	specifies one of several possible reorientations. Allowed values are 0–7 (see figure).

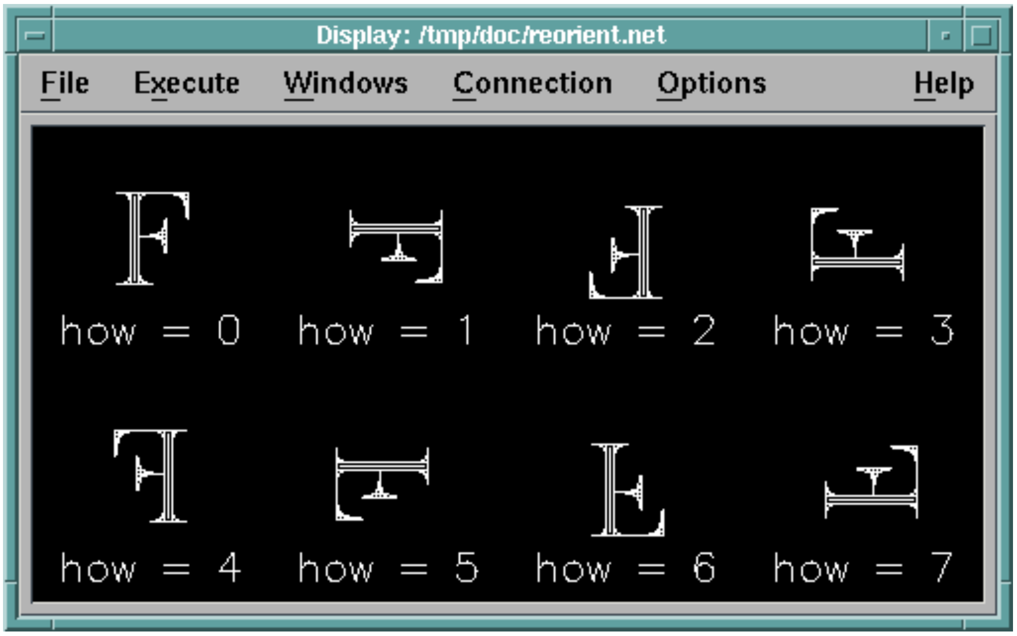

Display: /tmp/doc/reorient.net
File Execute Windows Connection Options Help
how = 0
how = 1
how = 2
how = 3
how = 4
how = 5
how = 6
how = 7

Figure 4. Reorientation of the letter F.

If **how** is set to 0, the image's appearance does not change. However, if the origin or deltas are not in the preferred image order (i.e., origin at [0, 0] and x varying fastest), the internal layout is altered to the preferred order. The result is a more efficient display of the image.

Thus Reorient can be used to align images from two sources so that the pixels are in one-to-one correspondence. A tool like Compute can then operate on corresponding pixels from the two images.

Components

Modifies the positions and connections components and reorders the position-dependent components.

Example Visual Program

`Topo.net`

See Also

Display, Overlay, ReadImage, Reduce, Refine, Render

Replace

Category

Structuring

Function

Replaces a component in a field or inserts a value list as a component.

Syntax

```
out = Replace(srcfield, dstfield, srcname, dstname);
```

Inputs

Name	Type	Default	Description
srcfield	field or value list	none	the field containing the replacement component
dstfield	field	none	the field in which the replacement component is to be placed
srcname	string	"data"	the component of **srcfield** that is to be placed in **dstname**
dstname	string	"data"	the new name of the replacement component

Outputs

Name	Type	Description
out	field	**dstfield** with component from **srcfield**

Functional Details

If **srcfield** is a field: the module puts the specified replacement component (**srcname**) from **srcfield** into **dstfield** and gives it a new component name (**dstname**).
If **srcfield** is a value list: the module inserts **srcfield** into **dstfield** as the **dstname** component (**srcname** is ignored).
If **dstfield** is a group: the hierarchy of **srcfield** must match exactly.

Components

All components of **dstfield** are propagated to the output. The **srcname** component of **srcfield** replaces or is added as the **dstname** component in the output.

Example Visual Programs

```
AlternateVisualizations.net
Imide_potential.net
UsingStreakline.net
```

See Also

Extract, Mark, Rename, Unmark

Reset

Category

Interactor

Function

Outputs one of two values.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`output`	value or string	output of Reset

Functional Details

This module outputs one ("set") value on the first execution after the Reset toggle has been activated and another ("unset") on all subsequent executions. The module will continue to output the "unset" value in subsequent executions until the toggle is activated again. The default values are "1" (set) and "0" (unset), but these can be modified to any value or string with **`Set Attributes...`** dialog box (selected from the **`Edit`** pull-down menu of the Interactor control panel).

Possible uses of the Reset interactor include driving the reset toggle of the Get module to initialize a program state and restarting an external simulator.

Note: This interactor cannot be data driven.

Example Visual Programs

`MultipleDataSets.net`
`UsingSwitchAndRoute.net`

See Also

GetLocal

Ribbon

Category

Annotation

Function

Produces a ribbon of specified width from a specified line.

Syntax

```
ribbon = Ribbon(line, width);
```

Inputs

Name	Type	Default	Description
`line`	field	none	line to be drawn as a ribbon
`width`	scalar	input dependent	ribbon width

Outputs

Name	Type	Description
`ribbon`	field	the line turned into a ribbon

Functional Details

This module is intended for use with any module that creates lines.

`line` specifies a line to be turned into a ribbon. If this line has a "normals" component (as would occur, for example, if the input field **`curl`** were used with Streamline), the resulting ribbon shows a twist corresponding to this component (at each point of the input line, the ribbon is oriented in the direction pointed to by the normal).

If a "normals" component is not found but the input contains a "binormals" component, normals are derived by crossing the binormals with the approximated tangents.

If no "normals" component is present, the ribbon's orientation is based on the path of the line.

`width` specifies the width of the ribbon in the same units as those of the original space. If this parameter is not specified, the system provides an appropriate value (1/50 of the diagonal of the bounding box of **`line`**).

The value used is attached to the output object as a "Ribbon width" attribute (which can be extracted with the Attribute module).

Note: Different colors can be specified for the two sides of the ribbon. Use the Color module and specify "front colors" and "back colors."

Components

Creates new "positions," "connections," and "normals" components. All other components are propagated to the output.

Example Visual Programs

```
ThunderStreamlines.net
UsingStreakline.net
```

See Also

Color, Streakline, Streamline, Tube

Rotate

Category

Rendering

Function

Rotates an object.

Syntax

```
output = Rotate(input, axis, rotation, ...);
```

Inputs

Name	Type	Default	Description
`input`	object	none	object to be rotated
`axis`	integer or string	"y"	axis of rotation
`rotation`	scalar	0	amount of rotation (in degrees)
`...`			additional axis-rotation pair(s)

Outputs

Name	Type	Description
`output`	object	object marked for rotation

Functional Details

This module prepares a specified object for one or more rotations about a specified axis.

Note: A Transform object containing the specified transformation matrix is inserted at the root of the object. This transform is applied during rendering.

Each rotation is specified by an axis-rotation parameter pair.

`input` specifies the object to be rotated.

`axis` specifies the axis of rotation. Allowed values are strings (“x,” “y,” or “z”) or integers (0, 1, or 2).

`rotation` specifies the extent of rotation (in degrees) about the specified axis.

Note: Rotation is counterclockwise in a right-handed coordinate system and relative to the origin of world-coordinate space.

A single Rotate module can specify a maximum of 22 rotations (i.e., **`axis-rotation`** pairs). In the user interface, the default number of corresponding tab pairs is two. (Tabs can be added to the module icon and removed with the appropriate **`...Input Tab`** options in the **`Edit`** pull-down menu of the VPE.)

Note: If you want to use the mouse to rotate the object in the Image window, see “Rotating the Object” on page 77 in *IBM Visualization Data Explorer User’s Guide*.

Components

All input components are propagated to the output.

Example Visual Programs

`Imide_potential.net`
`UsingLights.net`

See Also

Scale, Translate

Route

Category

Flow Control

Function

Routes an object through selector-specified output paths.

Syntax

```
output = Route(selector, input);
```

Inputs

Name	Type	Default	Description
`selector`	integer or integer list	0	paths for routing the input object
`input`	value list, string list, or object	no default	object to be routed

Outputs

Name	Type	Description
`output`	object	a possible path for routing the input object
`...`		additional paths

Functional Details

This module determines which of several output paths are executed.

`selector` specifies the output paths that are to be executed. If the specified value is:

- NULL or 0 (zero): none of the modules that use Route output are executed.
- *n*: specifies the consumer that is executed. If *n* = 1, the modules that use Route's first output are executed; if *n* = 2, the modules that use Route's second output are executed; and so on.

This parameter may also specify a list of integers, allowing multiple output paths to execute.

`input` specifies an object to be routed to the output paths that are to be executed.

A single Route module can specify a maximum of 21 outputs. In the user interface, the default number of enabled tabs is two. (Tabs can be added to the module icon and removed with the appropriate **`...Input Tab`** options in the **`Edit`** pull-down menu of the VPE.)

Notes:

1. Modules that use outputs that are not specified in the **`selector`** list are said to be "killed" and are not executed. If there are no outputs associated with **`selector`** (e.g., if **`selector`** is NULL or 0), none of the modules that use Route output is allowed to execute: all are killed.

   In general, modules that use the results of a killed module are also killed. An exception is Collect, which runs unless all inputs are killed (either by Route or by errors in modules that produce its inputs). .

2. In the scripting language, if the Route module is to work properly, it and modules downstream of it must be executed as part of a macro.

Components

All input components are propagated to the selected outputs.

Example Visual Programs

```
UsingSwitchAndRoute.net
UsingMessage.net
SIMPLE/Route.net
```

See Also

Collect, Switch

Chapter 4, "Data Explorer Execution Model" on page 37 in *IBM Visualization Data Explorer User's Guide*.

RubberSheet

Category

Realization

Function

Deforms a surface, using the data values of that surface.

Syntax

graph = RubberSheet(**data, scale, min, max**);

Inputs

Name	Type	Default	Description
data	scalar field	none	the field defining the surface to be deformed.
scale	scalar	input dependent	displacement scaling
min	scalar or field	0.0	offset to be applied to the surface data values
max	scalar or field	maximum data value	value used for setting the maximum displacement

Outputs

Name	Type	Description
graph	scalar field	the deformed field

Functional Details

This module takes a specified scalar surface or line field and displaces each point by an amount based on the data value at that point, as follows:

displacement = **scale** × (**data** – **min**)

(see parameter descriptions).

data specifies the surface to be deformed.

scale is the scale factor used in calculating the displacement.

If this parameter is not specified, the module provides a scale factor calculated as follows:

scale = (0.1 * diagonal of **data** boundary box) ÷ (**max** – **min**)

If **min** is a field, the value used is the minimum data value of that field.

The resulting scale factor is attached to the output object as a “RubberSheet scale” attribute (which can be extracted with the Attribute module).

min is the offset applied to the data values before they are scaled.

max is used to determine a scale factor according to the formula shown above, if **scale** is not specified.

Notes:

1. If the input data have no "normals" component (e.g., the output of Slab), the perpendicular to the first element is used as the displacement direction. In that case, for a positive scale factor and a positive data value, the surface is displaced in the direction given by the right-hand rule applied to the connections at the surface (i.e., with the thumb representing the direction of displacement and the fingers following the order of points in the connections).

2. If the input does have a "normals" component, the displacement at the surface is determined by the dependencies of the data and of the normals:

   - If both the data and the normals are position dependent, the surface is displaced in the direction of the normal at each point.
   - If both are connection dependent, each face of the input is displaced as a whole in the direction indicated by the normal for that face. The amount of displacement is proportional to the data value for that face. Additional sides are added to each displaced face to complete the "box"; box sides are not shared between neighboring faces.
   - If the data are position dependent and the normals are connection dependent, each face is displaced in the direction indicated by the normal for that face, but the amount of displacement varies across the face in proportion to the data value. Additional sides are added to each displaced face to complete the "box"; box sides are not shared between neighboring faces.
   - If the data are dependent on connections and the normals are dependent on positions, the displaced faces will be parallel to the original faces, displaced by an amount proportional to the data value for that face. Additional sides are added to each displaced face to complete the "box"; box sides of neighboring faces are coincident.

3. If the input data are 1- or 2-dimensional, an additional dimension is added in the resulting graph.

4. If no colors are present, the module adds a default color.

5. RubberSheet automatically adds shading to the output **graph**.

Components

Creates new "positions" and "connections" components. All other components, with the exception of "normals," are propagated to the output.

Example Visual Programs

```
AlternateVisualizations.net
RubberTube.net
ThunderGlyphSheet.net
UsingSwitchAndRoute.net
SIMPLE/Rubbersheet.net
```

See Also

FaceNormals, MapToPlane, Normals, Slab

Sample

Category

Realization

Function

Samples the points of a field (surface or volume).

Syntax

```
samples = Sample(manifold, density);
```

Inputs

Name	Type	Default	Description
manifold	field	none	the surface or volume to be sampled
density	integer	100	approximate number of samples

Outputs

Name	Type	Description
samples	field	a set of samples of the field

Functional Details

This module samples a specified field at a set of more or less uniformly spaced points.

manifold	is the field to be sampled.
density	is the approximate number of output samples.

Note: Any components in the original field (**manifold**) that are position or connection dependent are mapped onto the output samples. In either case, the component in the output field will be position dependent ("dep" "positions"). (Because the output of Sample is a set of unconnected positions, you cannot use mapped samples as input to modules that need to interpolate values, such as DivCurl or MapToPlane).

Components

Creates a new "positions" component. All other components, except for "connections," are propagated to the output.

Example Visual Programs

```
MappedIso.net
RubberTube.net
Streamline.net
UsingCompute2.net
UsingGlyphs.net
```

See Also

AutoGlyph, Glyph, Map, Streakline, Streamline

Scalar

Category

Interactor

Function

Generates successive scalar values over a specified range.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
data	object	no default	object from which interactor attributes can be derived
refresh	flag	0	reset the interactor
min	scalar	minimum data value	minimum output value
max	scalar	maximum data value	maximum output value
delta	scalar	input dependent	increment between successive scalar outputs
method	string	input dependent	defines interpretation of delta input
decimals	integer	input dependent	number of decimal places to be displayed in output values
label	string	"Scalar"	global name applied to interactor stand-ins

Outputs

Name	Type	Description
output	scalar	interactor output

Functional Details

This interactor allows the user to interactively change a scalar value. The range of values over which the module acts is governed by its attributes (e.g., minimum, maximum, and delta), which in turn are either (1) specified by the parameter values in its **Set Attributes...** dialog box or (2) determined from input to the module (e.g., a data field). In the second case, the interactor is said to be “data driven.”

If the interactor is not data-driven, its attributes are taken from its **Set Attributes...** dialog box (accessed from the **Edit** pull-down menu in the Control Panel).

Because the module is interactive, the user can change the current controlling value directly in Scalar’s control panel.

Note: The module's control panel is invoked by double-clicking on its icon in the VPE window. Its configuration dialog box is accessed from the **`Edit`** pull-down menu in the same window.

`data` is the object (usually a data field) from which the interactor can derive any or all of the minimum, maximum, and delta attributes when their corresponding input tab is up.

`refresh` resets the interactor so that the output is computed from the current input. If **`refresh`** = 0 (the default), the output is recomputed only if the current output does not lie within the range of the current **`data`**. The default for the output of the interactor is the midpoint of **`min`** and **`max`**.

`min` and **`max`** specify the minimum and maximum values of the interactor's scalar output. If set, these values override those implied by **`data`**.

If neither **`min`** nor **`data`** is specified, the interactor uses the minimum set in the **`Set Attributes...`** dialog box.

If neither **`max`** nor **`data`** is specified, the interactor uses the maximum set in the **`Set Attributes...`** dialog box.

`delta` specifies a scalar value as a factor for calculating the increment between successive outputs over the specified range. The actual value depends on the interpretation specified by **`method`** (see below).

`method` specifies the interpretation of **`delta`**:

- "rounded": the increment (**`max`** – **`min`**) × **`delta`** is rounded to a "nice" number. The spacing between successive values will approximate the interval specified by **`delta`**. (For example, the default value of 0.01 specifies an interval of 1/100 of the specified range.)
- "relative": the interpretation is the same as for "rounded," but the increment is *not* rounded.
- "absolute": **`delta`** is the absolute value of the interval. (If **`delta`** has not been specified, its default is 1.)

  The default value for **`method`** depends on other input. The default is:
  - "rounded" if **`data`** is specified *or* if both **`min`** and **`max`** are specified.
  - "absolute" in all other cases.

`decimals` specifies the number of decimal places displayed in the interactor. If neither **`data`** nor **`delta`** is specified, the interactor uses the value in its own **`Set Attributes...`** dialog box.

`label` is the global label of all instances of the corresponding interactor stand-in. An interactor instance's local label (set from the Control Panel) overrides a global label. By default, the global label is set by the user interface.

Example Visual Programs

Many example visual programs use the Scalar interactor. `DataDrivenInteractors.net` uses a data-driven scalar interactor.

See Also

Integer, IntegerList, ScalarList, Vector, VectorList

ScalarList

Category

Interactor

Function

Generates a list of scalar values.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
data	object	no default	object from which interactor attributes can be derived
refresh	flag	0	reset the interactor
min	scalar	minimum data value	minimum output value
max	scalar	maximum data value	maximum output value
delta	scalar	input dependent	increment between successive scalar outputs
method	string	input dependent	defines interpretation of delta input
decimals	integer	input dependent	number of decimal places to be displayed in output values
nitems	integer	11	number of items in the initial list
label	string	"ScalarList"	global name applied to interactor stand-ins

Outputs

Name	Type	Description
output	scalar list	interactor output

Functional Details

This interactor allows the user to interactively change a list of scalar values. The range over which the module acts is governed by its attributes (e.g., minimum, maximum, and delta), which in turn are either (1) specified by the parameter values in its **Set Attributes...** dialog box or (2) determined from input to the module (e.g., a data field). In the second case, the interactor is said to be “data driven.”

If an interactor is not data-driven, its attributes are taken from its **Set Attributes...** dialog box (accessed from the **Edit** pull-down menu in the Control Panel).

Because the module is interactive, the user can change the current controlling value directly in the control panel.

Note: The module's control panel is invoked by double-clicking on its icon in the VPE window. Its configuration dialog box is accessed from the **`Edit`** pull-down menu in the same window.

`data`	is the object (usually a data field) from which the interactor can derive any or all of the minimum, maximum, and delta attributes when the corresponding input tabs are up.
`refresh`	resets the interactor so that the output is computed from the current input. If **`refresh`** = 0 (the default), the output is recomputed only if the current output does not lie within the range of the current **`data`**.
`min` **and** **`max`**	specify the minimum and maximum values of the interactor's scalar output. If set, these values override those implied by **`data`**. If neither **`min`** nor **`data`** is specified, the interactor uses the minimum set in the **`Set Attributes...`** dialog box. If neither **`max`** nor **`data`** is specified, the interactor uses the maximum in the **`Set Attributes...`** dialog box. This value overrides the value implied by **`data`**.
`delta`	specifies a scalar value as a factor for calculating the increment between successive outputs over the specified range. The actual value depends on the interpretation specified by **`method`** (see below).
`method`	specifies the interpretation of **`delta`**: • “rounded”: the increment (**`max`** – **`min`**) × **`delta`** is rounded to a “nice” number. The spacing between successive values will approximate the interval specified by **`delta`**. (For example, the default value of 0.01 specifies an interval of 1/100 of the specified range.) • “relative”: the interpretation is the same as for “rounded,” but the increment is *not* rounded. • “absolute”: **`delta`** is the absolute value of the interval. (If **`delta`** has not been specified, its default is 1.) The default value for **`method`** depends on other input. The default is: – “rounded” if **`data`** is specified *or* if both **`min`** and **`max`** are specified. – “absolute” in all other cases.
`decimals`	specifies the number of decimal places displayed in the interactor. If neither **`data`** nor **`delta`** is specified, the interactor uses the value in its own **`Set Attributes...`** dialog box.
`nitems`	specifies the number of items in the interactor list. These are evenly spaced between the minimum and maximum values (see above). For example, if this parameter is given a value of 5, and the range is 0–100, the output list will be {0, 25, 50, 75, 100 }. **Note:** If **`nitems`** changes, a new list is computed.

`label`	is the global label of all instances of the corresponding interactor stand-in. An interactor instance's local label (set from the Control Panel) overrides a global label. If not specified, the global label is set by the user interface.

Example Visual Program

`ContoursAndCaption.net`

See Also

Integer, IntegerList, Scalar, Vector, VectorList

Scale

Category

Rendering

Function

Scales a specified object.

Syntax

```
output = Scale(input, scaling);
```

Inputs

Name	Type	Default	Description
input	object	none	object to be scaled
scaling	scalar or vector	[1 1 1]	scaling factor along *x*, *y*, and *z* axes

Outputs

Name	Type	Description
output	object	object marked to be scaled

Functional Details

This module prepares a specified object for scaling along all three axes.

Note: A Transform object containing the specified transformation matrix is inserted at the root of the object. This transform is applied during rendering.

input specifies the object to be scaled.

scaling is the scaling factor. If the parameter value is scalar, the object is scaled by that amount along each axis. The default value is [1 1 1], which produces no change in the input object.

Note: Scaling is relative to the origin of world-coordinate space.

Components

All input components are propagated to the output.

Example Visual Program

`AnnotationGlyphs.net`

See Also

Plot, Rotate, Transform, Translate

ScaleScreen

Category

Rendering

Function

Increases or decreases size of all screen objects (e.g. captions and colorbars) by specified factor.

Syntax

```
output, newcamera = ScaleScreen(object, scalefactor,
                                finalres, currentcamera);
```

Inputs

Name	Type	Default	Description
object	field	none	object to scale
scalefactor	scalar	1	scale factor for screen objects
finalres	integer	no default	final (x) resolution of desired image
currentcamera	camera	no default	current camera used to view object

Outputs

Name	Type	Description
output	object	object with screen objects scaled as specified
newcamera	camera	current camera updated by specified scale factors

Functional Details

The ScaleScreen module is used to scale all the screen objects within **object** by a specified amount. It does not affect the size of any other objects in the input **object**. A typical use of ScaleScreen is when you have created an image for display using Render, and you wish to re-render it at a higher resolution for printing, for example. You would need to use ScaleScreen to increase the pixel size of any screen objects (such as captions and color bars) in the object before re-rendering. Note that ScaleScreen is called implicitly when you use the “Allow Rerendering” option of the Save/Print Image dialogs of the Image window, so you do not need to use it. You would only need to use ScaleScreen if you are doing the re-rendering yourself using Render or Display.

object	specifies the object containing screen objects to be scaled.
scalefactor	specifies the scale factor you intend to use on re-rendering; for example **scalefactor** should be set to 2 if you want the re-rendered image to be twice as large.

finalres	is used as an alternative to **scalefactor**, if, for example, you know that you want your final resolution of your image to be 1000 pixels. If you specify **final_res**, you must also provide the **currentcamera** input.
currentcamera	is the camera you are currently using to view your object. This can be the output of AutoCamera or Camera, or the **camera** output of Image.

The output **output** is the object with all screen objects scaled. If you specify **currentcamera**, then the module will, in addition, output **newcamera**, which is a new camera for you to use with Render or Display.

Components

All components in the input are propagated to the output. Only screen objects are modified.

Example Visual Programs

`SIMPLE/ScaleScreen.net`

See Also

Render, AutoCamera, Camera, Image

Select

Category

Structuring

Function

Selects a member of a group or a list.

Syntax

```
object = Select(input, which);
```

Inputs

Name	Type	Default	Description
input	group, series, value list, string list	none	the object from which one or more members are to be selected
which	integer, integer list, string list	0	the member(s) to be selected
except	flag	0	0: copy all listed members 1: copy all but listed members

Outputs

Name	Type	Description
object	object	the selected member(s)

Functional Details

input is a series or group from which items are to be selected.

which specifies the item(s) to be selected.

If **input** is:

- *a series or group:* the module selects items
  - by name if **which** is a string or string list.
  - by index in the group if **which** is an integer or integer list.
- *a list:* **which** must be an integer or integer list, and the module selects the corresponding items.

If this parameter is not specified, the module selects the first (0th) object.

Note: Members of a series can be selected only by ordinal number, not by series position. For both groups and lists, counting begins at 0 (zero).

except specifies whether **which** is to be interpreted as an inclusive or exclusive selection list.

Notes:

1. Since the components of fields are typically arrays (lists), you can select a particular position as follows:

   Use Extract to extract the “positions” component.

   Use Select on that array, setting **`which`** to the appropriate value (e.g., 7 for the eighth position).

   You can also use the Select module to select individual frames from a data series, passing the Sequencer output to **`which`**.

2. If you pass a series to Select, and that series has groups of fields as members, the output of Select is one of the groups of fields. Selecting a single field from one of these groups requires two calls to Select: the first to select the group of fields, and the second to select the individual field.

3. If **`which`** is a string, Select finds the object with that name only if it lies at the top hierarchical level of the **`input`** passed to Select.

Components

All input components are propagated to the output.

Example Visual Programs

```
ConnectingScatteredPoints.net
GeneralImport1.net
GeneralImport2.net
PlotTwoLines.net
UsingTextAndTextGlyphs.net
```

See Also

Collect, CollectNamed, CollectSeries, List

Selector

Category

Interactor

Function

Generates a value and a string based on user input.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
`stringdata`	string list or group	no default	specifies or derives a list of potential output strings
`valuelist`	integer or string list or value list	0-based list	list of potential output values
`cull`	flag	0	determines whether zero-length strings are removed (1) or not (0)
`label`	string	"Selector"	global name applied to interactor stand-ins

Outputs

Name	Type	Description
`value`	string or value	value
`name`	string	string name

Functional Details

This module allows the user to interactively select *one* item from a list. Through inputs to the module (outputs from other tools or values set in its configuration dialog box) the interactor can be "data driven."

If the interactor is not data driven, its attributes (e.g., **`stringdata`** or **`valuelist`**) are taken from its **`Set Attributes...`** dialog box (accessed from the **`Edit`** pull-down menu in the Control Panel).

This interactor requires a list of strings and a list of values or strings that are then paired up one-to-one and used as selectable outputs. The inputs to Selector are used to generate both lists. If there are no inputs, the module uses the values in its **`Set Attributes...`** dialog box.

`stringdata` controls the string values that appear as output in **`name`**. It is required if the interactor is data driven.

If the parameter value is:

- *a string list:* the individual strings are used as potential outputs.

- *a group:* the names of the group members (or member1, member2, ...) are used as possible output names.
- *a series group:* the names are “position = *n*,” (where *n* is the series position of each series member).

In every case, the items in the resulting string list are displayed in the list of selections in the corresponding Selector interactor.

`valuelist` is a list of potential output values.

If this parameter is a value or string list, these values are mapped one-to-one onto the string list that results from **`stringdata`** input.

If the interactor is data-driven, the parameter value can also be a single integer used to generate a list of integers beginning with that one.

If the parameter is defaulted and the interactor is data-driven, the value list is a list of integers starting at 0 (zero).

`cull` is specified only when the interactor is data-driven. It determines whether or not empty strings are culled from the string list that results from **`stringdata`** input. If set to 1, empty strings are removed.

`label` is the global label of all instances of the corresponding interactor stand-in. An interactor instance’s local label (set from the Control Panel) overrides a global label. If not specified, the global label is set by the user interface.

Example Visual Programs

Many example visual programs use the Selector interactor. Example programs that use a data-driven Selector interactor are:

```
DataDrivenSelector.net
UsingAttributes.net
```

See Also

Integer, IntegerList, Scalar, ScalarList, Vector

SelectorList

Category

Interactor

Function

Generates a list of values and a strings based on user input.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
stringdata	string list or group	no default	specifies or derives a list of potential output strings
valuelist	integer or string list or value list	0-based list	list of potential output values
cull	flag	0	determines whether zero-length strings are removed (1) or not (0)
label	string	"Selector"	global name applied to interactor stand-ins

Outputs

Name	Type	Description
value	string or value	value
name	string	string name

Functional Details

This module allows the user to interactively select zero, one, or more items from a list. Through inputs to the module (outputs from other tools or values set in its configuration dialog box) the interactor can be "data-driven."

If the interactor is not data-driven, its attributes are taken from its **Set Attributes...** dialog box (accessed from the **Edit** pull-down menu in the Control Panel).

This interactor requires a list of strings and a list of values or strings that are then paired up one-to-one and used as selectable outputs. The inputs to SelectorList are used to generate both lists. If there are no inputs, the module uses the values in its **Set Attributes...** dialog box.

stringdata controls the string values that appear as output in **name**. It is required if the interactor is data-driven.

If the parameter value is:

- *a string list:* the individual strings are used as potential outputs.

- *a group:* the names of the group members (or member1, member2, ...) are used as possible output names.
- *a series group:* the names are “position = *n*,” (where *n* is the series position of each series member).

In every case, the items in the resulting string list are displayed in the list of selections in the corresponding SelectorList interactor.

valuelist is a list of potential output values.

If this parameter is a value or string list, these values are mapped one-to-one onto the string list that results from **stringdata** input.

If the interactor is data-driven, the parameter value can also be a single integer used to generate a list of integers beginning with that one.

If the parameter is defaulted and the interactor is data-driven, the value list is a list of integers starting at 0 (zero).

cull is specified only when the interactor is data-driven. It determines whether or not empty strings are culled from the string list that results from **stringdata** input. If set to 1, empty strings are removed.

label is the global label of all instances of the corresponding interactor stand-in. An interactor instance’s local label (set from the Control Panel) overrides a global label. If not specified, the global label is set by the user interface.

Example Visual Program

`InterfaceControl3.net`

See Also

Integer, IntegerList, Scalar, ScalarList, Vector

Sequencer

Category

Special

Function

Generates a sequence of integers.

Syntax

The sequencer is available in script mode and in the user interface. However, it can be data driven only in the user interface (see Chapter 10, “Data Explorer Scripting Language” on page 187 in *IBM Visualization Data Explorer User’s Guide*, and the example later in this section.)

Inputs

Name	Type	Default	Description
min	integer	1	minimum of integer sequence
max	integer	100	maximum of integer sequence
delta	integer	1	numerical interval between successive integers in the sequence

Outputs

Name	Type	Description
frame	integer	frame number

Functional Details

This module allows a user to “animate” a visualization. Through inputs to the module (outputs from other tools or values set in its configuration dialog box) the interactor can be “data driven.”

If the interactor is not data-driven, its attributes are taken from its **Frame Control** dialog box (accessed from the **...** button).

The configuration dialog box for Sequencer can be accessed by selecting the Sequencer icon in the VPE and then choosing **Configuration** in the **Edit** menu.

min and **max**	specify the minimum and maximum integer values (frame numbers) for an “animation” sequence. By default, **min** = 1 and **max** = 100.
delta	specifies the increment between integers in the output sequence. If **min** and **max** are equal, **delta** is ignored. By default, **delta** = 1.

Note: If **min** is specified, and either **min** or **max** change, the output value (**frame**) is set to the new **min** value and the Start and Stop values in the Frame Control dialog box are set to the new minimum and maximum, respectively.

However, if **min** is not specified and **max** is and then changes on the next execution, the output frame value becomes the current Start value as set in the Frame Control dialog box.

Script Language Example

The following example uses the Sequencer in script mode to display a series of isosurfaces. The input to the macro `do_each_frame` is the frame number, which is converted into an isosurface value by multiplying it by 0.05. The initialization values are set by `@startframe`, `@endframe`, and `@nextframe`. The sequence command invokes the macro for each step of the sequence. Finally, the play command starts the Sequencer running. For more information about using the sequencer in script mode, see Chapter 10, “Data Explorer Scripting Language” on page 187 in *IBM Visualization Data Explorer User’s Guide*.

```
macro do_each_frame(frame)
{
   isovalue = frame*.05;
   isosurface = Isosurface(data,isovalue);
   Display(isosurface,camera);
}

data = Import("/usr/lpp/dx/samples/data/cloudwater");
camera = AutoCamera(data);
@startframe = 1;
@endframe = 5;
@nextframe = @startframe;
sequence do_each_frame(@frame);
play;
```

Example Visual Programs

Many of the example visual programs use a sequencer, including:

```
ContoursAndCaption.net
GeneralImport1.net
MovingCamera.net
```

The following example visual program uses a data-driven Sequencer:

```
Imide_potential.net
```

SetGlobal

Category

Flow Control

Function

Places an object in the cache.

Syntax

```
SetGlobal(object, link);
```

Inputs

Name	Type	Default	Description
object	object	no default	object to be cached
link	string	no default	link to corresponding GetGlobal module

Functional Details

This module works with GetGlobal to place and retrieve objects from the cache: SetGlobal places an **object** in the cache, where GetGlobal can retrieve it. SetGlobal should always be used with GetGlobal, not with GetLocal.

Note: The corresponding GetGlobal module must be executed on the same machine (i.e., it cannot be distributed to another machine). See “GetGlobal” on page 149 and “GetLocal” on page 151 for a discussion of the differences between the Global Get/Set pair and the Local Get/Set pair.

object	is the object to be placed in the cache.
link	specifies the GetGlobal module that corresponds to the SetGlobal module. In the VPE, this link would be created by dragging an arc from the **link** output of GetGlobal to the **link** input of SetGlobal.

A detailed description of the behavior and use of the GetLocal, GetGlobal, SetLocal, and SetGlobal modules can be found in 4.6, “Preserving Explicit State” on page 45 in *IBM Visualization Data Explorer User’s Guide*.

Example Visual Programs

`SIMPLE/GetSet.net`

See Also

GetGlobal, GetLocal, SetLocal

Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

SetLocal

Category

Flow Control

Function

Places an object in the cache.

Syntax

```
SetLocal(object, link);
```

Inputs

Name	Type	Default	Description
`object`	object	no default	object to be cached
`link`	string	no default	link to corresponding GetLocal module

Functional Details

This module works with GetLocal to place and retrieve objects from the cache: SetLocal places an **object** in the cache, where GetLocal can retrieve it. SetLocal should always be used with GetLocal, not with GetGlobal.

Note: The corresponding GetLocal module must be executed on the same machine (i.e., it cannot be distributed to another machine). See "GetGlobal" on page 149 and "GetLocal" on page 151 for a discussion of the differences between the Global Get/Set pair and the Local Get/Set pair.

object	is the object to be placed in the cache.
link	specifies the GetLocal module that corresponds to the SetLocal module. In the VPE, this link would be created by dragging an arc from the **link** output of GetLocal to the **link** input of SetLocal.

A detailed description of the behavior and use of the GetLocal, GetGlobal, SetLocal, and SetGlobal modules can be found in 4.6, "Preserving Explicit State" on page 45 in *IBM Visualization Data Explorer User's Guide*.

Example Visual Programs

```
Accumulate.net
Bounce.net
SimpleGetSetLoop.net
```

See Also

GetLocal, GetGlobal, SetGlobal

Chapter 4, "Data Explorer Execution Model" on page 37 in *IBM Visualization Data Explorer User's Guide*.

Shade

Category

Rendering

Function

Specifies the shading attributes of an object.

Syntax

```
output = Shade(input, shade, how, specular, shininess, diffuse,
               ambient);
```

Inputs

Name	Type	Default	Description
input	object	none	object to be shaded
shade	flag	1	0: object not shaded 1: object shaded
how	string	none	(“smooth” or “faceted”)
specular	scalar	none	specular coefficient (standard = 0.5)
shininess	integer	none	shininess exponent (standard = 10)
diffuse	scalar	none	diffuse coefficient (standard = 0.7)
ambient	scalar	none	ambient coefficient (standard = 1.0)

Outputs

Name	Type	Description
output	field	shaded object

Functional Details

This module is applicable only to surface objects (i.e., objects with connections of type “triangle” or “quad”).

input specifies the object whose shading attributes are to be modified.

shade turns shading on or off. By default, the module turns shading on.

how specifies whether the shading is to be smooth or faceted. If this parameter is not set, and:

- if the input object is already shaded, the shading method is not changed.
- if the object is not shaded, the choice between smooth and faceted shading is determined by the dependency of the “data” component, if it has a data dependency:

  smooth shading is used for position-dependent data.

faceted shading is used for connection-dependent data.

- if no data are present, smooth shading is used.

An explicit specification of “smooth” or “faceted” shading will apply to all objects. However, if a “normals” component of the requested type (“dep” “positions” or “dep” “connections”) is already present, the normals are not recomputed.

The next four parameters specify the particular shading attributes of the object. If any of these parameters are not explicitly set, the corresponding attributes remain unchanged.

`specular`	specifies the amount of light reflected by the object (as from a very smooth, highly reflective surface). The specified value must be between 0 and 1. The default is 0.5.
`shininess`	specifies how sharp the specular highlight is. The specified value must be an integer. Larger numbers result in a smaller, sharper, specular highlight. The default is 10.
`diffuse`	specifies how much light is diffused by the object (as from a rough non-reflective surface). The specified value must be between 0 and 1. The default is 1 (one).
`ambient`	specifies how much light is reflected equally in all directions. The specified value must be between 0 and 1.

Notes:

1. The Shade module will also check whether the orientation of the “connections” component is consistent with the directions of the “normals”. If they are not consistent, it will modify the directions of the normals.
2. Smooth shading is not supported for faces, loops, and edges data. However, faces, loops, and edges can be converted to triangles using Refine, and can then be smooth-shaded.

Components

May add or modify the “normals” component. All other components are unchanged.

Example Visual Programs

```
AlternateVisualizations.net
Imide_potential.net
UsingShade.net
```

See Also

Render, Display, Image

ShowBoundary

Category

Realization

Function

Shows the boundary of a field.

Syntax

```
output = ShowBoundary(input, validity);
```

Inputs

Name	Type	Default	Description
input	field	none	field whose boundary is to be shown.
validity	flag	0	0: create boundary of all data 1: create boundary of valid data only

Outputs

Name	Type	Description
output	color field	renderable boundary of the input field

Functional Details

The ShowBoundary module creates a field containing only the exterior faces of the input object.

input	is the object whose boundary is to be "shown" by the module.
validity	determines whether validity of data (as specified by "invalid position" and "invalid connections" components) is to be used in constructing the boundary. By default, such information is not used, and the information is simply passed through to the output boundary. Display, Image, or Render will not render any invalid faces. (See "Invalid Positions and Invalid Connections Components" on page 23 in *IBM Visualization Data Explorer User's Guide* for a discussion of invalid data.)

Notes:

1. Any colors or data in the input object are passed through to **output**. A default color is added if a "colors" component is not already present.
2. The boundaries of a volumetric field are a surface or a set of surfaces; the boundary a 2-dimensional field is a line or a set of lines.

Components

Creates new “positions,” “connections,” and “normals” components. All other components are propagated to the output. If **`input`** does not have a “colors” component, one is added. Adds a “normals” component if the input is a volumetric field.

Example Visual Programs

```
InvalidData.net
Thunder_cellcentered.net
UsingCompute.net
SIMPLE/ShowBoundary.net
```

See Also

ShowBox, ShowConnections, ShowPositions

ShowBox

Category

Realization

Function

Draws the bounding box of a field.

Syntax

```
box, center = ShowBox(input);
```

Inputs

Name	Type	Default	Description
`input`	field	none	the field for which a bounding box is to be shown

Outputs

Name	Type	Description
`box`	color field	renderable bounding box of input field
`center`	vector	center of bounding box

Functional Details

This module produces a set of lines representing the edges of the bounding box of a specified input object. It also generates a 3-vector for the position of the center of the bounding box.

`input`	specifies the object for which ShowBox generates a bounding box.

Notes:

1. The module adds a default color to the output box.
2. Unlike ShowConnections, ShowBoundary, and ShowPositions, this module does not pass the colors and data of the input through to the output.
3. Any invalid data in **`input`** is ignored in creating the bounding box. (See “Invalid Positions and Invalid Connections Components” on page 23 in *IBM Visualization Data Explorer User’s Guide*.)

Components

Creates new “positions,” “connections,” and “colors” components.

Example Visual Programs

```
ContoursAndCaption.net
UsingSwitchAndRoute.net
SIMPLE/MapToPlane.net
```

See Also

ShowBoundary, ShowConnections, ShowPositions

ShowConnections

Category

Realization

Function

Draws the connective elements of a field.

Syntax

```
output = ShowConnections(input);
```

Inputs

Name	Type	Default	Description
`input`	field	none	field whose connection elements are to be shown.

Outputs

Name	Type	Description
`output`	color field	renderable connections of input field

Functional Details

This module creates a set of renderable lines representing the "connections" component of a specified input.

`input` specifies the object whose connection elements are to be shown.

Notes:

1. If the input object lacks a "colors" component, the module adds a default color to the output.
2. Position-dependent color and data components are passed to the output. Connection-dependent color and data components are not passed to the output, because their values along a connection line are ill defined. To make the data or colors position-dependent, use the Post module.

Components

Creates a new "connections" component and adds a "colors" component if the input does not already have one. Connection-dependent components are not propagated to the output; all other components are.

Example Visual Programs

```
SIMPLE/Refine.net
ConnectingScatteredPoints.net
ImportExternalFilter.net
PlotLine.net
Isolate.net
UsingStreakline.net
```

See Also

Post, ShowBoundary, ShowBox, ShowPositions

ShowPositions

Category

Realization

Function

Shows the positions of a field.

Syntax

```
output = ShowPositions(input);
```

Inputs

Name	Type	Default	Description
input	field	none	field whose positions are to be shown
every	scalar	1.0	factor used to reduce the number of positions shown

Outputs

Name	Type	Description
output	color field	renderable positions of input field

Functional Details

This module creates an output field that shows the positions of a specified input.

input specifies the object whose positions are to be shown.

every determines the relative number of input positions shown in the output. By default, all positions are shown.

The multiplicand used is the reciprocal of the parameter value (e.g., given a value of 2, the output shows half the positions in the input).

Notes:

1. If the input object lacks a "colors" component, the module adds a default color to the output.
2. Position-dependent color and data components are passed to the output.

Components

Adds a "colors" component if the input does not already have one. All input components, except "connections," are propagated to the output. Invalid positions are not passed through to the output.

Example Visual Programs

```
MovingCamera.net
UsingSwitchAndRoute.net
```

See Also

ShowBoundary, ShowBox, ShowConnections

SimplifySurface

Category

Transformation

Function

Simplifies a triangulated surface and resamples data attached to the surface.

Syntax

```
simplified = SimplifySurface(original_surface, max_error, max_data_error,
                             volume, boundary, length, data, stats);
```

Inputs

Name	Type	Default	Description
original_surface	field	(none)	triangulated surface
max_error	scalar	input dependent	maximum positional error
max_data_error	scalar	input dependent	maximum data error
volume	flag	1	1: move vertices to preserve volume 0: do not move vertices
boundary	flag	0	1: simplify surface boundaries 0: do not simplify surface boundaries
length	flag	1	1: move vertices to preserve the length of boundaries 0: do not move vertices
data	flag	1	1: use data dependent on “positions” to constrain simplification 0: ignore data for simplification
stats	flag	0	1: provide simple statistics. 0: do not provide statistics

Outputs

Name	Type	Description
simplified	field	simplified triangulated surface

Functional Details

SimplifySurface builds a **simplified** surface that is guaranteed to deviate from **original_surface** by less than **max_error**. This means that each vertex of **simplified** as well as each point inside a triangle of **simplified** is at a Euclidean distance no further than **max_error** from **original_surface**. Similarly, each vertex of **original_surface** as well as each point inside a triangle of **original_surface** is at a distance no further than **max_error** from the simplified surface.

In addition, if a “data” component of **original_surface** is dependent on the “positions” component and is not **TYPE_STRING**, SimplifySurface will perform a data dependent simplification: SimplifySurface will resample “data” on the simplified surface and guarantee that the maximum deviation between the original and re-sampled data is less than **max_error_data**. For efficiency reasons, the dimensionality of the “data” component is currently limited to 3: for instance, it will work for RGB colors or for gradient values in 3-dimension. Note that data dependent on connections will **not** constrain simplification.

Vertex normals for the **simplified** surface are automatically computed.

Components dependent on “positions” or “connections” of **original_surface** are added to **simplified** and re-sampled. For components dependent on “connections”, the triangle areas are used to weight the resampling.

The following components are not re-sampled: “positions”, “connections”, “invalid positions”, “invalid connections” “normals”, “neighbors”, and “positional error”.

original_surface
must have triangular connections. **original_surface** is the field that is being simplified.

max_error
maximum distance (in the units of the “position” component) between **simplified** and **original_surface**. The default value for **max_error** is 1% of the diagonal of the bounding box of **original_surface**. You may decide to set **max_error** to a lower or higher value than the default.

max_data_error
maximum deviation between data attached to **original_surface** vertices, and the resampling of that data on the **simplified** surface. (Again the maximum deviation also holds for points inside triangles, not just vertices).

The default value for the maximum error on the data is set to 10% of the diagonal of the bounding box in data space. If the data is one dimensional, the default error is (max_data-min_data)/10.

max_data_error is ignored if **data** = 0 or if the data are connection-dependent.

volume
specifies whether the volume enclosed by the surface should be preserved or not. If set to 1, SimplifySurface will move the surface vertices in order to preserve the volume while the number of vertices is being reduced. In this case, the volume is preserved to within floating point or higher accuracy. If set to 0, the surface vertices will not be moved. The default is 1.

The volume is only truly defined for a surface that does not have a boundary (closed surface). Setting **volume** to 1 on a surface with boundary will have the effect of preventing shrinkage and producing more regular triangles.

Note that if several surfaces share the same boundary, as when Isosurface is used on partitioned data, the volume can still be preserved with **volume** set to 1 if the boundary is left intact (**boundary** set to 0).

With surfaces that present sharp angles, such as CAD data, setting **volume** to 1 might not work well. In general, we recommend setting **volume** to zero when operating on CAD data.

boundary	specifies whether the boundary of a surface should be simplified or left intact. If set to 1, then the boundary is simplified while respecting the same errors **max_error** and **max_data_error**. If set to 0, then the boundary is left intact. The default is 0. If **original_surface** has no boundary, this option is ignored.
length	if **boundary** set to 1, specifies whether the length of the simplified boundary should be preserved. If set to 1, in a manner analogous to volume preservation, SimplifySurface will move the boundary vertices in order to preserve the boundary length while the number of boundary vertices is being reduced. If set to 0, the boundary vertices will not be moved. The default is 1. If **boundary** is set to 0 then the value of **length** is ignored.
data	if set to 1, instructs SimplifySurface use **max_data_error** to constrain simplification if data are dependent on "positions" and are not **TYPE_STRING**, and if the dimensionality of such data is 3 or lower. If set to 0, simplification will not be constrained by data. In any case, SimplifySurface will resample data after simplification. The default is 1.
stats	if set to 1, instructs SimplifySurface to write simple statistics: number of vertices and triangles in original_surface, number of vertices and triangles in simplified, and percentage of original numbers of vertices and triangles. This information will appear in the Message window. If set to 0, SimplifySurface will not provide statistics. The default is 0.
simplified	simplified surface.

Generating surface levels of detail.

SimplifySurface adds a "positional error" component to the **simplified** surface. "positional error" is a component dependent on "positions" that provides for each vertex of **simplified** a positive number. This positive number, the error value, represents the radius of a sphere centered on the vertex, that is guaranteed to intersect the original surface. The union of such spheres represents the error volume of the simplified surface, which is guaranteed to enclose the original surface. Points inside triangles are assigned an error value that is interpolated from the error values at the triangle vertices using barycentric coordinates.

Using the "positional error" component, a simplified surface can be re-simplified to any **max_error** while still guaranteeing a bound with respect to the original surface. This is useful for generating successive levels of detail.

If the same error bound is used, a marginal additional simplification can be observed: the algorithm implements a greedy method, and does not guarantee to find the minimum number of triangles that respects a given error bound.

The use of the "positional error" component by SimplifySurface is transparent to you, but you may occasionally want to visualize the simplification error, using the Mark module. An example is provided in SimplifySurface.net.

Components

SimplifySurface adds a “positional error” component that is dependent on “positions” (see “Generating surface levels of detail.” on page 313).

In general **`simplified`** has the same components as **`original_surface`** except for “invalid positions”, “invalid connections”, “neighbors”, and components that are **`TYPE_STRING`**.

Example Visual Programs

`SimplifySurface.net`

See Also

Refine, Reduce, Isosurface, Map

Slab

Category

Import and Export

Function

Creates a “slab” of data.

Syntax

output = Slab(**input, dimension, position, thickness**);

Inputs

Name	Type	Default	Description
input	field	none	object to be “slabbed”
dimension	integer or string	0	spatial orientation of the slab
position	integer or integer list	all	starting position(s)
thickness	integer	0 or 1	thickness of slab (in number of elements)

Outputs

Name	Type	Description
output	field or field series	slabbed data

Functional Details

This module creates a multidimensional object consisting of a selected subset of input data.

data specifies the data field to be slabbed. This field must have regular connections.

dimension specifies the dimension in which the slab should be oriented relative to the input object (the output slab will have the same orientation). The *n* dimensions of the object can be specified by number (from 0 to *n* – 1). The first three dimensions can also be specified as *x*, *y*, and *z*, respectively.

The default for this parameter is dimension 0 (zero).

Notes:

1. The **dimension** number refers to the ordering of positions in the “connections” component. Thus, *x* corresponds to 0 only if the positions have been specified as *x* varies slowest.
2. Slab can also be used on deformed grids, in which case the connections do not align with any particular axis.
3. Transposing the positions (i.e., with Transpose) does *not* change the order in the “connections” component.

position	specifies the position from which the slab is generated. If the specified value is a single integer, the module creates a single field, with the slab oriented in the specified dimension and containing the number of volume elements specified by **thickness** (see below). If **position** is a list of integers, the module creates a series of specified slabs, and the default value of **thickness** is 1 (one). If this parameter is not specified, the output is a series of slabs (**thickness** = 1) that begin at every grid position along **dimension**.
thickness	specifies the thickness of the slab (in volume elements). A specified value of 0 (the default when **position** is a single number) will produce a 2-dimensional slice in a 3-dimensional object.

Notes:

1. This module performs no interpolation, and only data with regular connections can be slabbed.
2. If the input object is already colored, the colors are passed through unaltered.
3. If the input object is a volume and the requested slab is much thinner than the input object, the colors for volume rendering may be dim.
4. If the data are connection dependent, a request for a slab of **thickness** = 0 is ill-defined and the module returns an error.

Components

All input components are propagated to the output.

Example Visual Programs

```
Imide_potential.net
RubberTube.net
Streamline.net
SIMPLE/Slab.net
```

See Also

MapToPlane, Select, Slice

Slice

Category

Import and Export

Function

Slices a multidimensional object.

Syntax

```
output = Slice(input, dimension, position);
```

Inputs

Name	Type	Default	Description
`input`	field	none	object to be sliced
`dimension`	integer or string	0	dimension to be eliminated
`position`	integer list	all	starting positions

Outputs

Name	Type	Description
`output`	field or field series	sliced data

Functional Details

This module creates one or more "slices" of data from a multidimensional object, each slice containing a subset of input data. It differs from the Slab module in reducing by one the dimensionality of the object passed to **`output`**.

`input` specifies the data field to be sliced. This field must have regular connections.

`dimension` specifies the dimension to be eliminated. If **`input`** has *n* dimensions, the output will have *n* – 1. These dimensions can be specified by number (from 0 to *n* – 1). The first three can also be specified as *x*, *y*, and *z*, respectively.

The default for this parameter is dimension 0 (zero).

Notes:

1. The **`dimension`** number refers to the ordering of positions in the "connections" component. Thus, *x* corresponds to 0 only if the positions have been specified as *x* varies slowest.
2. Slice can also be used on deformed grids, in which case the connections do not align with any particular axis.
3. Transposing the positions (i.e., with Transpose) does *not* change the order in the "connections" component.

`position` specifies the position from which the slice is generated.

If the specified value is a single value, the module creates a single field sliced that position along the eliminated dimension.

If **position** is a list of integers, the module creates a field series, sliced at each position in the list. The series position of each series member is equal to the value of the origin of that slice along the sliced axis.

If this parameter is not specified, the output is a series of slices that begin at every grid position along **dimension**.

Notes:

1. This module produces 2-dimensional data from 3-dimensional data. To create a 2-dimensional slice in 3-dimensional space, use Slab.
2. Slice performs no interpolation, and only data with regular connections can be sliced.
3. If the data are connection dependent, slicing is ill-defined and the module returns an error.

Components

All input components are propagated to the output.

Example Visual Program

`WarpingPositions.net`

See Also

Select, Slab, Stack

Sort

Category

Transformation

Function

Sorts the values of a specified list or field in a specified order.

Syntax

`result = Sort(field, descending)`

Inputs

Name	Type	Default	Description
`field`	scalar list or field	none	object to be sorted
`descending`	flag	0	0: sort in descending order 1: sort in ascending order

Outputs

Name	Type	Description
`result`	scalar list or field	sorted object

Functional Details

`field` specifies the field or list to be sorted.

If the parameter is a scalar list, its values must be scalars or 1-vectors. If the parameter is a field, the sorting is performed on the "data" component; other components are reordered and renumbered as appropriate.

`descending` specifies the order of sorting: descending (0) or ascending (1).

Components

The "data" and data-dependent components are reordered according to their data values. Similarly, components on which data depends are reordered. Components that reference reordered components are renumbered.

Example Visual Program

`Sort.net`

Stack

Category

Import and Export

Function

Stacks fields or series of fields.

Syntax

```
output = Stack(input, dimension);
```

Inputs

Name	Type	Default	Description
`input`	field, field series, or group	none	data to be stacked
`dimension`	integer or string	0	dimension in which the data is to be stacked

Outputs

Name	Type	Description
`output`	field	stacked data

Functional Details

This module stacks a series of *n*-dimensional fields to form a single (*n*+1)-dimensional field. You can also use Stack to add a dimension to a single field (e.g., “stack” it from 2 dimensions to 3 dimensions). Note that Stack does *not* accept partitioned data as **`input`**.

`input` specifies the object(s) to be stacked. The field(s) must have regular connections.
If the parameter specifies a series, the location of the stacked field in the new **`dimension`** is equal to the series position of that field. If the parameter specifies a group, then the location of each member of the group in the new **`dimension`** will be 0, 1, ... If the parameter specifies a field (rather than a field series), the *n*-dimensional data is converted to (*n*+1)-dimensional data, with the value of the extra dimension set to zero.
A dimension can be specified by number (from 0 to *n* – 1). The first three can also be specified as *x*, *y*, and *z*, respectively. The default value is dimension 0 (zero).

Components

All input components are propagated to the output.

Example Visual Program

`ManipulateGroups.net`

See Also

CollectSeries, Slice, Transpose

Statistics

Category

Transformation

Function

Computes statistical characteristics of a field or list.

Syntax

```
mean, sd, var, min, max = Statistics(data);
```

Inputs

Name	Type	Default	Description
data	field or value list	none	data set used in statistical computations

Outputs

Name	Type	Description
mean	scalar	the mean
sd	scalar	the standard deviation
var	scalar	the variance
min	scalar	minimum value
max	scalar	maximum value

Functional Details

This module finds the mean, standard deviation, variance, minimum, and maximum of the value list, or the "data" component if a field is input.

data specifies the field or list of values on which the module performs statistical computations.

For vector fields, the computations are based on the magnitude of the vector. For matrices, the computations are based on the determinant of the matrix.

Example Visual Program

`AlternateVisualizations.net`

See Also

Compute, Histogram

Streakline

Category

Realization

Function

Computes streaklines to represent the movement of particles through changing vector fields.

Syntax

```
line = Streakline(name, data, start, time, head, curl, flag, stepscale);
```

Inputs

Name	Type	Default	Description
name	string or object	no default	name of streakline
data	vector field or vector field series	none	vector series time-step or vector field series
start	vector list or geometric field	center of the boundary box of the first series member	starting point(s)
time	scalar list	series start time(s)	starting time(s)
head	integer	final series frame number	series time-step at which the streaklines end
frame	integer	current frame	current frame
curl	vector field or vector field series	no curl	vorticity of vector field(s)
flag	flag	input dependent	0: curl not used 1: curl used to generate twist of streakline(s)
stepscale	value	0.1	step length as a fraction of the size of the connection element

Outputs

Name	Type	Description
line	field or group	one or more streaklines

Functional Details

This module traces the path of a particle through a changing vector field in discrete steps. The visual form of this trace is a streakline, which starts at a specified time and continues until the particle it represents exits from the field or until the field "expires."

name is required only if more than one invocation of Streakline in the same visual program uses the same **start**, **time**, **head**, **curl**, and **flag** arguments with different vector fields (**data**).

The reason for this requirement is that, in order to function efficiently in movie applications, Streakline saves intermediate results from frame to frame. This information is stored internally in association with a key. Each invocation of the Streakline module in the same graph receives a unique name, which is constructed from the **name**, **start**, **time**, **head**, **curl**, and **flag** arguments. Thus, if more than one invocation of Streakline in the same graph uses the same arguments, varying only the vector field, a unique **name** string must be assigned to each.

If you have only one Streakline module or if any of the inputs to the module other than **data** are different between the two modules, you do not need to specify **name**.

data specifies a vector field or vector field series. Vector fields are limited to *n*-D vectors defined in *n* space; for example, 2-D vectors on a 2-D sheet or 3-D vectors defined in a 3-D volume.

The individual time-step vector fields are passed to Streakline as **data**, either combined in a single series object or one at a time in successive executions of the graph containing the Streakline module, as fields. Times are associated with each vector field by the "series position" attached to each member of the series group. Vector values in between these times are interpolated linearly from the prior and subsequent vector fields. The times associated with each field should increase monotonically as they are encountered by the Streakline module, either as the individual series members are indexed or as the individual series members are encountered in successive executions of the graph. Vector fields encountered out of order are ignored.

If **data** is a single field rather than a series group, then it must have a "series position" attribute for Streakline to accept it. This attribute is attached when a series member is selected with the Select module. The individual members of the series group can be extracted using the Select module or by requesting a single member of the series when using the Import module.

start specifies a set of points at which streaklines originate. It can be either a list of positions or a geometric field (e.g., the output of a Grid module).

time specifies a set of times at which streaklines begin to be traced. The units of time correspond to the units of the vector field. That is, a particle travelling at a unit velocity moves one unit of distance in one unit of time.

Both **time** and **start** can contain single or multiple elements. If either parameter contains a single element, that element is associated with each element of the other list. If both parameters contain multiple elements, they must contain the same number of elements, which are then matched in pairs. This allows either a single starting point to be associated with several different starting times, several starting points to be associated with a single starting time, or several starting points to be associated with individual starting times. If no **time** parameter is given, the series position of the first member of the vector field series (either the first series member or the first individual vector field encountered) is used. If no **start** parameter is given, the center point of the first member of the vector field series is used.

head specifies an expiration time for streaklines that have not otherwise terminated. The streakline points are computed iteratively, and the spacing between the points depends on the element size of the connections.

frame allows the Streakline module to incrementally generate streaklines when the entire series is input as **data**. For example, you could attach the Sequencer to **frame**, and as **frame** advanced, the streaklines would grow longer.

flag specifies whether the normals of the streaklines produced rotate according to the vorticity of the vector field or do not rotate at all.

curl specifies the curl of the vector field (see "DivCurl" on page 118).

If the vorticity of the vector field is supplied by this parameter, the curl is interpolated from it. In that case, the field must correspond to the vector field in type (either both must be a series or both must be a single field) and they should correspond in time. If **curl** is specified, the default value for the **flag** parameter is 1. If **curl** is not specified, the vorticity can be computed within Streakline, but at a considerable cost in time. In this case the default value for **flag** is 0, and if you want Streakline to compute the curl of the field, you must set **flag** to 1.

stepscale specifies the accuracy of the interpolation. Streakline traces the path of a particle through the vector field in discrete steps. These steps are determined by interpolating vectors from the prior and subsequent vector fields at the geometric position of the current end of the streakline, and then linearly interpolating between the results to produce a vector for the current position and time. A segment is then appended to the current streakline, which continues it in the direction of the resulting vector.

The length of the new segment is determined by two factors. First, the length of the projection of the segment along the edges of the cells of the prior and subsequent vector fields in which the starting point of the segment is found does not exceed a proportion (specified by the **stepscale** parameter) of the length of the edge. Second, if grid is irregular, the segment is truncated at the boundary of the element in which it begins. This allows the segment length to be determined anew for the next element.

Thus the **stepscale** parameter offers the user control over the accuracy of the streakline. If **stepscale** is a small value (perhaps in the range of 0.01 to 0.05), the resulting streakline contains a relatively large number of segments that closely trace small variations in the vector field. If **stepscale** is assigned a large number (0.5 to 1), the steps are larger and less accurate, but require less execution time. Note, however, that since the step is determined as a proportion of the size of the grid cell in which it is contained, the segments are small in areas in which the grid elements are small, and proportionally larger in areas in which the grid elements are large.

Streaklines trace particles through time. Associated with the streakline is a "time" component that indicates, for each position of the streakline, the time at which the particle reached that position. The initial value in this component, therefore, is equal to the starting time for that streak, which is by default the first series position, or settable by the user using the **time** parameter. The final value in the time component is equal to the time at which the particle exits from the vector field, either geometrically or for one of the following reasons:

- It has reached the time associated with the final member of the vector field series.
- It has reached the time associated with the current frame.
- It has reached the time associated with the series member indicated by the **head** parameter.

The output streaklines also contain a "data" component that indicates the velocity of the data field at each position along the streakline.

Note: Unlike other modules in the system, the Streakline module maintains information about previous executions. If errors occur during the execution of the visual program, it may be necessary to use the **Reset Server** option to reinitialize the system.

Components

Creates new "positions," "connections," "data," "time," and "colors" components. If the flag argument is set, "normals" and "binormals" components are also created.

Example Visual Programs

```
UsingCompute3.net
UsingStreakline.net
```

See Also

DivCurl, Ribbon, Streamline, Tube

Streamline

Category

Realization

Function

Computes streamlines to represent the movement of particles through static vector fields.

Syntax

line = Streamline(**data, start, time, head, curl, flag, stepscale**);

Inputs

Name	Type	Default	Description
data	vector field	none	vector field
start	vector list or geometric field	center of the first series member	starting point(s)
time	scalar list	series start time(s)	starting time(s)
head	integer	final series frame number	series time-step at which the streamlines end
curl	vector field	no curl	vorticity of vector field(s)
flag	flag	input dependent	0: curl not used 1: curl used to generate twist of streamline(s)
stepscale	value	0.1	step length as a fraction of the size of the connection element

Outputs

Name	Type	Description
line	field or group	one or more streamlines

Functional Details

This module traces the path of a particle through a static vector field in discrete steps. The visual form of this trace is a streamline, which starts at a specified time and continues until the particle it represents exits from the field or reaches a region of zero velocity.

data specifies a vector field. Vector fields are limited to *n*-D vectors defined in *n* space; for example, 2-D vectors on a 2-D sheet or 3-D vectors defined in a 3-D volume.

start specifies a set of positions at which streamlines originate. It can be either a list of positions or a geometric field (e.g., the output of a Grid module or even an isosurface).

time	specifies a set of times at which streamlines begin to be traced. The units of time correspond to the units of the vector field. That is, a particle travelling at a unit velocity moves one unit of distance in one unit of time.
	Both **time** and **start** can contain single or multiple elements. If either parameter contains a single element, that element is associated with each element of the other list. If both parameters contain multiple elements, they must contain the same number of elements, which are then matched in pairs.
	You can use the time parameter to start different streamlines at different times. Note that for the static vector fields used by Streamline, a particle at a given location will trace the same streamline, regardless of what time it is started.
head	specifies an expiration time for streamlines that have not otherwise terminated. The streamline points are computed iteratively, and the spacing between the points depends on the element size of the connections and the stepscale. If this maximum is reached, a warning is produced and the streamline is terminated. The maximum number of streamline points is 25,000.
curl	causes Streamline to produce normals and binormals components that represent the vorticity of the vector field. “DivCurl” on page 118). In that case, the default value for **flag** is 1. The “normals” and “binormals” components of the resulting streamlines rotate according to the vorticity of the vector field. This information is either interpolated from the vector field passed in as the **curl** parameter or is computed internally.
flag	specifies whether or not the twist specified by **curl** is enabled. A setting of 0 (zero) disables the twist.
	However, if **curl** is not specified, the default value for **flag** is 0, and no twist is produced. In that case you have the option of setting **flag** to 1, which causes Streamline to compute the curl internally.
stepscale	specifies the accuracy of the interpolation. Streamline traces the path of a particle through the vector field in discrete steps. These steps are determined by interpolating a vector from the vector field at the current end of the streamline and appending a new segment that extends the streamline in the direction of the vector.
	The length of the new segment is determined by two factors. First, the length of the projection of the segment along the edges of the cells of the prior and subsequent vector fields in which the starting point of the segment is found does not exceed a proportion (specified by the **stepscale** parameter) of the length of the edge. Second, if grid is irregular, the segment is truncated at the boundary of the element in which it begins. This allows the segment length to be determined anew for the next element.
	Thus the **stepscale** parameter offers the user control over the accuracy of the streamline. If **stepscale** is a small value (perhaps in the range of 0.01 to 0.05), the resulting streamline contains a relatively large number of segments that closely trace small variations in the vector field. If **stepscale** is assigned a large number (0.5 to 1), the steps are larger and less accurate, but

require less execution time. Note, however, that since the step is determined as a proportion of the size of the grid cell in which it is contained, the segments are small in areas in which the grid elements are small, and proportionally larger in areas in which the grid elements are large.

Streamlines trace particles through time. Associated with the streamline is a “time” component that indicates, for each position of the streamline, the time at which the particle reached that position. The initial value in this component, therefore, is equal to the starting time for that stream, which is by default the first series position, or settable by the user using the **time** parameter. The final value in the time component is equal to (1) the time at which the particle exits from the vector field or (2) the time specified by the **head** parameter.

The output streamlines also contain a “data” component that indicates the velocity of the data field at each position in the streamline.

Components

Creates new “positions,” “connections,” “data,” “time,” and “colors” components. If the flag argument is set, “normals” and “binormals” components are also created.

Example Visual Programs

```
Interop.net
InvalidData.net
RubberTube.net
Streamline.net
ThunderStreamlines.net
```

See Also

DivCurl, Glyph, Ribbon, Streakline, Tube

String

Category

Interactor

Function

Generates a string.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`output`	string	interactor output

Functional Details

The String interactor generates a string as output. For more information, see "String Interactor" on page 144 in *IBM Visualization Data Explorer User's Guide*.

Note: This interactor cannot be data driven.

Example Visual Program

`AnnotationGlyphs.net`

See Also

FileSelector, Integer, IntegerList, Scalar, ScalarList, StringList, Value, ValueList, Vector, VectorList

StringList

Category

Interactor

Function

Generates a string list.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`output`	string list	interactor output

Functional Details

The StringList interactor generates a string list as output. For more information, see “List Interactors” on page 145 in *IBM Visualization Data Explorer User’s Guide*.

Note: This interactor cannot be data driven.

See Also

FileSelector, Integer, IntegerList, Scalar, ScalarList, String, Value, ValueList, Vector, VectorList

SuperviseState

Category

Windows

Function

Manages the object and/or camera associated with an image window created using SuperviseWindow.

Syntax

```
object, camera, where, events = SuperviseState(where,defaultCamera, resetCamera,
                                               object, resetObject, size,
                                               events, mode, args);
```

Inputs

Name	Type	Default	Description
where	string	(none)	the window for which objects and cameras are to be manipulated
defaultCamera	camera	(none)	initial or default camera
resetCamera	flag	0	whether to reset camera to the default
object	object	(none)	initial or default object
resetObject	flag	0	whether to reset object to the default
size	vector or integer list	(none)	size
events	object	(none)	mouse or keyboard events from SuperviseWindow
mode	integer	(none)	specifies the mode, or which UserInteractor to enable
args	object	(none)	interactor arguments

Outputs

Name	Type	Description
object	object	current object
camera	camera	current camera
where	string	input where
events	string	unhandled events

Functional Details

Briefly, this module allows a user to specify the particular action that should be taken on mouse or keyboard events in a window created using SuperviseWindow. The SuperviseState module is expected to be used together with SuperviseWindow and Display to create and display a window. These modules are used as an alternative to the Image tool. See “SuperviseWindow” on page 336 for a more complete discussion of the use of the Supervise modules and the benefits one gets with respect to the simple use of the Image tool.

Actions are specified using user-supplied routines called UserInteractors. A single UserInteractor is the set of actions which take place for any mouse or keyboard event for a given **mode**. For example, a single **mode** may define different behaviors for left, middle, and right mouse actions. An arbitrary number of different modes can be defined, providing a limitless number of different user interactions with the image. Note that this implies that different actions (for example zoom and translate) can be implemented either as different modes using the same mouse button, as the same mode with different mouse buttons, or of course, different modes and different buttons.

where	the window for which objects and cameras are to be manipulated. This input should be supplied by the **where** output of SuperviseWindow.
defaultCamera	is the initial or default camera.
resetCamera	indicates whether to reset camera to the default
object	is the initial or default object
resetObject	indicates whether to reset object to the default
size	is the current size of the image, and should be supplied by the **size** output of SuperviseWindow
events	mouse or keyboard events. Should be supplied by the **events** output of SuperviseWindow
mode	specifies the mode, (which UserInteractor to enable). The set of user-defined UserInteractors is created as a table of callbacks. The **mode** value specifies which entry in that table is to be called.
args	interactor arguments as required by a UserInteractor.

The following are output parameters:

object	current object; should be passed to the **object** input of Display.
camera	current camera; should be passed to the **camera** input of Display
where	Window where parameter; should be passed to the **where** input of Display.
events	are unhandled events, that is any mouse or keyboard events that have been masked off within the routine InitMode, described in UserInteractors.

UserInteractors

UserInteractors consist of the following routines, the contents of which must be provided by the user. You specify the location of your custom interactor object files using the `DX_USER_INTERACTOR_FILE` environment variable (see "Other Environment Variables" on page 292 in *IBM Visualization Data Explorer User's Guide*).

Note: Default user interactors to implement rotation, pan, and zoom functionality are provided by Data Explorer if you do not provide your own custom interactors. Rotation (mode 0) is the same as the standard left-button rotation interaction of the Image tool. Pan (mode 1) operates differently than the Data Explorer pan mode of the Image tool; you simply drag on the object to move it in the desired direction.

Zoom (mode 2) operates as follows: drag upward to zoom in; drag downward to zoom out.

`void *InitMode(Object args, int width, int height, int *mask)`
Given an object containing **`args`** (which come in as input to SuperviseState and are up to the interactor builder to specify), and the current width and height of the window, returns a handle that is passed into all the other UserInteractor routines. This routine also sets the value of **`mask`** to reflect which events the particular interactor is interested in (for example, only left or right buttons). Once **`mask`** is set, only those events which have been specified as interesting to the interactor will cause the interactor to be called. The set of possible masks is:

```
DXEVENT_LEFT

DXEVENT_MIDDLE

DXEVENT_RIGHT

DXEVENT_KEYPRESS
```

`void EndMode(void *handle)`
Frees the space allocated in InitMode.

`void SetCamera(void *handle, float *to, float *from, float *up, int projection, float fov, float width)`
Passes current camera info from Data Explorer into the interactor. The interactor can extract whatever camera information it is interested in and put into its private **`handle`**. Note that if the camera is going to be modified, this handle must retain the entire camera state so that it can be passed back in GetCamera().

`void SetRenderable(void *handle, Object object)`
Passes the current object into the interactor. If the interactor is going to change the object, it must be retained in **`handle`**.

`int GetCamera(void *handle, float *to, float *from, float *up, int *projection, float *fov, float *width)`
Passes updated camera information from the interactor back to Data Explorer. If the interactor has not updated the camera information, return 0; otherwise set ALL the inputs and return 1.

`int GetRenderable(void *handle, Object *obj)`
Passes updated object information from the interactor back to Data Explorer. If the interactor has not updated the object, return 0, otherwise set **`obj`** to point to the updated object and return 1.

`void EventHandler(void *handle, DXEvent *event)`
Event handler. Receives the event in **`*event`** where a **`DXEvent`** is of type:

```
typedef union
{
  DXAnyEvent any;
  DXMouseEvent mouse;
  DXKeyPressEvent keypress;
}
```

and where `DXAnyEvent`, `DXMouseevent`, and `DXKeyPressEvent` are of type:

```
typedef struct
{
  int event
} DXAnyEvent
typedef struct
{
  int event;
  int x;
  int y;
  int state;
} DXMouseEvent
typedef struct
{
  int event;
  int x;
  int y;
  int key;
} DXKeyPressEvent
```

`event` is one of `DXEVENT_LEFT`, `DXEVENT_MIDDLE`, `DXEVENT_RIGHT`, or `DXEVENT_KEYPRESS`; `x` and `y` are the pixel location of the event; `state` is one of `BUTTON_DOWN`, `BUTTON_MOTION`, or `BUTTON_UP`; and `key` is the key that was pressed.

Doing Picking

If you want to do picking in a window created by SuperviseWindow, simply pass the **events**, **object**, and **camera** outputs of SuperviseState to the **locations**, **object**, and **camera** inputs of Pick. Pick will pull out only the locations of button-down events from **events** and use **camera** to do the picking.

Example Visual Programs

```
SIMPLE/Supervise.net
Image_wo_UI.net
IndependentlyArrange.net
InsetImage.net
```

Also see the demonstrations of custom interactors in `/usr/lpp/dx/samples/supervise`.

See Also

SuperviseWindow, Display, Image

SuperviseWindow

Category

Windows

Function

Creates a Display window for an image and captures mouse and keyboard events in that window.

Syntax

```
where, size, events = SuperviseWindow(name, display, size,
                                       offset, parent, depth, visibility,
                                       pick, sizeFlag, offsetFlag);
```

Inputs

Name	Type	Default	Description
name	string	(none)	unique name for the window
display	string	“local host”	display on which to create the window
size	integer vector	[640,480]	x,y size of the window
offset	integer vector	[0,0]	offset of window in parent
parent	string	(root window)	parent window
depth	integer	8	depth of window
visibility	integer	1	visibility of window
pick	flag	0	whether to consider only button-down events
sizeFlag	flag	0	whether to force the size of an already existing window
offsetFlag	flag	0	whether to force the offset of an already existing window

Outputs

Name	Type	Description
where	string	where parameter for window
size	integer vector or integer list	window size
events	object	mouse or keyboard events

Functional Details

SuperviseWindow, along with the associated module SuperviseState, provides the user with direct control over the effect of mouse and keyboard events in a window containing an image. SuperviseWindow and SuperviseState are used together with the Display tool, as an alternative to the Image tool.

When you include the Image tool in a visual program, mouse and keyboard events are interpreted in a particular way as described in 6.1, “Using the Image Window” on page 74 in *IBM Visualization Data Explorer User’s Guide*. In contrast, when the Supervise modules and the Display tool are used in a visual program, you are able to specify how a particular mouse or keyboard event should affect either the object being viewed or the camera used to view it. For example, you could specify that right-mouse drags in a horizontal direction should zoom in on an object, while right-mouse drags in a vertical direction should zoom out.

Not only does this allow you to specify the particular action that should take place given events in the display window, but it also allows your interaction with the image to be separated completely from the Data Explorer user interface. For example, a completely custom user interface could be created with a commercial GUI builder, bypassing the Data Explorer user interface entirely, but still allowing user-interaction with the image.

In order to specify custom interaction modes in a program, it is necessary for you to provide routines which indicate what action is to be taken for a given event. An action can affect either the object (for example rotating it or coloring it red), or the camera (for example zooming in), or both. These sets of user-provided routines are called UserInteractors. UserInteractors let you specify your own direct interactors by specifying a table of callbacks, each implementing a custom interactor.

To start Data Explorer using custom UserInteractors, it is necessary to identify the user-written UserInteractor routines to the Data Explorer executive. This is done by setting the environment variable `DX_USER_INTERACTOR_FILE` before starting Data Explorer. (You can also load UserInteractors after starting Data Explorer by using the Executive module (“Executive” on page 126 in *IBM Visualization Data Explorer User’s Reference*)).

If you do not set the `DX_USER_INTERACTOR_FILE` environment variable, a set of default interactors are provided automatically by Data Explorer. These interactors provide Rotate, Pan, and Zoom functions.

See “SuperviseState” on page 332 for more detailed information on how to specify UserInteractors. SuperviseWindow and SuperviseState should not be used in -image mode of Data Explorer. If used in -image mode, they will compete for ownership of the Image window with the User Interface. Use -edit or -menubar mode instead.

name	is a unique name for the window.
display	is the display on which to create the window and defaults to “local host”.
size	is the x,y size of the window, and defaults to 640x480. It can be specified either as an integer vector or as an integer list.
offset	is the offset of the window in the parent window. By default, **offset** is [0,0]. It can be specified either as an integer vector or as an integer list.
parent	is the parent window “where” parameter; **parent** defaults to the root window, but can also be an already created window if nested windows are desired.
depth	is the depth (in bits) of the window; **depth** defaults to 8.

`visibility`	indicates the visibility of the window: **0** window is closed **1** window is open **2** window is open and always on top
`pick`	indicates whether only button-down events should be considered. This is useful if the user wants to implement picking in the Display window.
`sizeFlag`	indicates whether to force the size of an already existing window. The default is 0 (false).
`offsetFlag`	indicates whether to force the offset of an already existing window. The default is 0 (false).
`where`	identifies the window. This output must be connected to the **`where`** parameter of the Display tool. It would also be used, for example, if nested windows are desired using a SuperviseWindow module with the **`parent`** parameter set to something other than the root window.
`size`	is the current size of the window, and may differ from the **`size`** input if you resize the window.
`events`	encodes mouse or keyboard events. It is an array of integer of integer 4-vectors, where the four integers represent the following:

- event
- x
- y
- state or keypress, depending on event

`event` is one of `DXEVENT_LEFT`, `DXEVENT_MIDDLE`, `DXEVENT_RIGHT`, or `DXEVENT_KEYPRESS`. **`x`** and **`y`** are the pixel locations of the event. For **`event`** = `DXEVENT_LEFT`, `DXEVENT_MIDDLE`, `DXEVENT_RIGHT`, the final integer is “state”, which is one of `BUTTON_UP`, `BUTTON_MOTION`, or `BUTTON_DOWN`. For **`event`** = `DXEVENT_KEYPRESS`, the final integer is “keypress” which is the character which was pressed. Note that

```
#define DXEVENT_LEFT            0x01
#define DXEVENT_MIDDLE          0x02
#define DXEVENT_RIGHT           0x04
#define DXEVENT_KEYPRESS        0x08
#define BUTTON_UP               1
#define BUTTON_DOWN             2
#define BUTTON_MOTION           3
```

For operations other than picking, the user simply passes **`events`** to the **`events`** input of SuperviseState, which interprets the mouse or keyboard events and calls the appropriate user routines. For picking, see “Doing Picking” on page 335.

Example Visual Programs

```
SIMPLE/Supervise.net
Image_wo_UI.net
IndependentlyArrange.net
InsetImage.net
```

Also see the demonstrations of custom interactors in `/usr/lpp/dx/samples/supervise`.

See Also

SuperviseState, Display, Image

Switch

Category

Flow Control

Function

Selects one input from a list.

Syntax

```
output = Switch(selector, input, ...);
```

Inputs

Name	Type	Default	Description
selector	integer	0	object to be selected
input	value list, string list, or object	no default	selectable object
...			more objects to be switched

Outputs

Name	Type	Description
output	object	the object switched

Functional Details

This module selects a specified object from **input** and passes it through to **output**.

selector specifies which of the *n* input objects are to be passed to **output**. If the specified value is 1, the first **input** object (the second input to the module itself) is passed through; if the specified value is 2, the second input object (the third input to the module) is passed through; and so on.

If the specified value is ≤ 0 or larger than *n*, **output** is NULL.

input specifies an input that may or may not be passed to output.

A single Switch module can accept a maximum of 21 input objects. In the user interface, the default number of enabled tabs is two. (Tabs can be added to the module icon and removed with the appropriate **...Input Tab** options in the **Edit** pull-down menu of the VPE.)

Note: In the user interface you may find it helpful to use the first output of the Selector interactor as the first input (**selector**) to Switch. See “Selector and SelectorList Interactors” on page 146 in *IBM Visualization Data Explorer User’s Guide* for more information.

Components

All input components are propagated to the output.

Example Visual Programs

```
AlternateVisualizations.net
ConnectingScatteredPoints.net
UsingSwitchAndRoute.net
SIMPLE/Switch.net
```

See Also

Collect, Route

Chapter 4, “Data Explorer Execution Model” on page 37 in *IBM Visualization Data Explorer User’s Guide*.

System

Category

Debugging

Function

Executes a system function.

Syntax

```
System(string);
```

Inputs

Name	Type	Default	Description
string	string	none	shell command to be executed

Functional Details

The System module uses the C library **system()** function to execute operating system commands.

If the characters %, \, or " occur in the command string, they must be escaped: the percent sign must be preceded by another percent sign; backslashes and double quotes must be preceded by a backslash.

Script Language Example

This example creates a sequence of captioned images using different isosurface values. The Format module creates a different image file name for each image. The System module executes the compress function to minimize the amount of disk space used.

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
electrondensity = Partition(electrondensity);
camera = AutoCamera(electrondensity,resolution=300,aspect=1,width=2.5);
macro makeiso(isoval)
{
    isosurface = Isosurface(electrondensity, isoval);
    caption = Format("isoval = %g", isoval);
    caption = Caption(caption);
    imagename = Format("iso%4.2f.rgb", isoval);
    collected = Collect(caption, isosurface);
    image = Render(collected, camera);
    Display(image);
    WriteImage(image, imagename);
    command = Format("compress %s", imagename);
    System(command);
}
makeiso(0.1);
makeiso(0.2);
makeiso(0.3);
makeiso(0.33);
makeiso(0.36);
makeiso(0.39);
```

Text

Category

Annotation

Function

Displays text.

Syntax

```
text = Text(string, position, height, font, direction, up);
```

Inputs

Name	Type	Default	Description
string	string or field	none	text to be displayed
position	vector	[0 0 0]	placement of displayed text
height	scalar	1	height of the text (in world coordinates)
font	string	"variable"	text font
direction	vector	[1 0 0]	orientation of the baseline
up	vector	perpendicular to baseline	orientation of vertical strokes of text font

Outputs

Name	Type	Description
text	color field	renderable string object

Functional Details

This module produces text that is displayed in space.

string specifies the text to be displayed. If the parameter value is a string, that string is displayed. If the parameter value is a field, its “data” component must be TYPE_STRING. The data may be position or connection dependent:

- position dependent: each string in the “data” component is displayed at its corresponding position.
- connection dependent: each string is displayed at the center of its corresponding connection element.

position specifies in world units the placement of the displayed text. It is used only when **string** is not a field.

height specifies the height of the text, in world units.

font specifies the font used for displayed text. You can specify any of the defined fonts supplied with Data Explorer. These include a variable-width font (“variable,” the default) and a fixed-width font (“fixed”):

`area`	`gothicit_t`	`pitman`	`roman_ext`
`cyril_d`	`greek_d`	`roman_d`	`script_d`
`fixed`	`greek_s`	`roman_dser`	`script_s`
`gothiceng_t`	`italic_d`	`roman_s`	`variable`
`gothicger_t`	`italic_t`	`roman_tser`	

For more information, see Appendix E, “Data Explorer Fonts” on page 307 in *IBM Visualization Data Explorer User’s Guide*.

direction specifies the orientation of the baseline (see Note in next description). For example, a value of 10 for this parameter specifies that the text is aligned with the x-axis.

up specifies the orientation of the vertical strokes of the font used for the displayed text.

Note: For string data in a field, the orientation of baseline and text can specified by using “tangents” and “binormals” components. In that way, each string can be oriented individually. These components should have the same dependency as “data” The “tangents” component corresponds to **direction** and “binormals” to **up**.

The parameter specifications will override those of the field components.

Components

Creates new “positions,” “connections,” and “colors” components.

Example Visual Program

`UsingTextAndTextGlyphs.net`

See Also

Caption, Color, Format

Toggle

Category

Interactor

Function

Selects between two possible outputs.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
setval	value, string	1	output value when set
unsetval	value, string	0	output value when set
label	string	no default	interactor label

Outputs

Name	Type	Description
output	value, string	selected toggle value

Functional Details

This interactor allows the user to interactively choose between two different outputs. Through inputs to the module (outputs from other tools or values set in its configuration dialog box) the interactor can be "data driven."

If the interactor is not data driven, its attributes (i.e., the two possible output values) are taken from its **Set Attributes...** dialog box (accessed from the **Edit** pull-down menu in the Control Panel).

Note: The module's control panel is invoked by double-clicking on its icon in the VPE window. Its configuration dialog box is accessed from the **Edit** pull-down menu in the same window.

setval	is the value put out when the toggle is set.
unsetval	is the value put out when the toggle is not set.
label	is the global label of all instances of the corresponding interactor stand-in. An interactors instance's local label (set from the Control Panel) overrides a global label. If not specified, the global label is set by the user interface.

Example Visual Programs

`UsingSwitchAndRoute.net`

See Also

Selector

Trace

Category

Debugging

Function

Enable or disables tracing options.

Syntax

`Trace(`**`what, how`**`);`

Inputs

Name	Type	Default	Description
what	string	none	the object to be traced
how	integer	1	0: set tracing off 1: set tracing on

Functional Details

what is (1) a keyword specifying the object to be traced *or* (2) a set of letters specifying a group of debug message classes, as used by the **DXEnableDebug()** library function (see *IBM Visualization Data Explorer Programmer's Reference*).

At present, the only keyword value allowed for this parameter is **"time"**: **Trace("time", 1);** will begin the trace, and **Trace("time", 0);** will print a record of time use since the tracing was started. The output appears in the Message window.
If **what** is a string of lowercase letters other than **time**, the module calls the **DXEnableDebug** library function. (The uppercase letters A–Z and the numbers 0–9 are reserved for system use).

how specifies whether the trace option is activated or not.

Note: If the executive is started with the flag `-timing` on, the command `Trace("time", 0)` causes all module entry and exit times to be printed. In the VPE window, this command can be enabled using the Debug Tracing button in the **Commands** menu of the Message window (see 8.2, “Using the Message Window” on page 174 in *IBM Visualization Data Explorer User's Guide*).

Script Language Example

In this example, the module traces the use of time by Isosurface.

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
electrondensity = Partition(electrondensity);
camera = AutoCamera(electrondensity, width=5);
Trace("time", 1);
isosurface = Isosurface(electrondensity, 0.3);
Trace("time", 0);
Display(isosurface, \camera);
```

See Also

Usage

Transform

Category

Rendering

Function

Performs a generalized transform of an object.

Syntax

```
output = Transform(input, transform);
```

Inputs

Name	Type	Default	Description
input	object	none	object to be transformed
transform	tensor	none	*3×3* or *3×4* transformation matrix

Outputs

Name	Type	Description
output	object	object marked for transformation

Functional Details

This module prepares a specified object for being moved, rotated, and resized. A Transform object containing the specified transformation matrix is inserted at the root of the object. This transform is applied during rendering.

The module is more primitive than Translate, Rotate, and Scale, but it allows direct entry into a *3×3* or *3×4* matrix (for combining several transformations or for a skew transform).

Each $[x\ y\ z]$ point in the object is transformed to the new point $[x'\ y'\ z']$ by:

$$[x'\ y'\ z'] = [x\ y\ z]\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} + [j\ k\ l]$$

Nine numbers are interpreted as *a* to *i* with no translation; twelve numbers are *a* to *l*. No translation occurs for the default value of:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + [0\ 0\ 0]$$

Components

All input components are propagated to the output.

See Also

Rotate, Scale, Translate

Translate

Category

Rendering

Function

Translates an object.

Syntax

```
output = Translate(input, translation);
```

Inputs

Name	Type	Default	Description
input	object	none	object to translate
translation	vector	[0 0 0]	amount of translation along *x*, *y*, and *z* axes

Outputs

Name	Type	Description
output	object	object marked to be translated

Functional Details

This module prepares a specified object for being moved (translated) a specified amount along the *x*, *y*, and *z* axes.

Note: A Transform object containing the specified transformation matrix is inserted at the root of the object. This transform is applied during rendering.

input specifies the object to be translated.

translation is the translation vector. No translation occurs for the default value of [0 0 0]. A specification of [0 1 0] (to take one example) will move the specified object one unit in the y-direction.

Components

All input components are propagated to the output.

Example Visual Program

`Imide_potential.net`

See Also

Rotate, Scale

Transmitter

Category

Special

Function

Transmits an object to a Receiver.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
`object`	object	none	object transmitted

Functional Details

To maintain the modularity and readability of large programs, Data Explorer provides two tools that allow connections between input and output tabs of separate modules without the use of a visible connecting line. These tools, Transmitter and Receiver, allow you to separate visual programs into logical blocks. For example, the output of several logical blocks can be transmitted to another block that receives them, collects them, and produces an image.

Receivers and Transmitters can also be used to communicate information between pages in the Visual Program Editor (see “Creating pages in the VPE” on page 115 in *IBM Visualization Data Explorer User’s Guide*). Pages are a valuable way of structuring complex visual programs into logical blocks.

Note: Macros provide another way of structuring visual programs in logical blocks (see 7.2, “Creating and Using Macros” on page 149 in *IBM Visualization Data Explorer User’s Guide*).

To remotely connect input and output tabs:

1. Select a Transmitter tool (in the Special category in the tool palette) and place it near the output tab of the module that is to be “connected.”
2. Connect the module’s output tab to the Transmitter’s input tab.
3. Select a Receiver tool (also in the Special category), and place it near the input tab of the other module that is to be “connected.”
4. Connect the Receiver’s output tab to the second module’s input tab.

   The Transmitter and Receiver are now connected.

The Receiver automatically assumes the same name as the Transmitter. More than one Receiver can be connected to a single Transmitter and they assume the same name until a new Transmitter is placed on the VPE canvas.

Notes:

1. To change the name of a Transmitter and Receiver, use the **`Notation`** field of the appropriate configuration dialog box (see in “Entering Values in a Configuration Dialog Box” on page 107 in *IBM Visualization Data Explorer User’s Guide*). Changing the name of a Transmitter changes the name of all the Receivers connected to it. Changing the name of a Receiver affects only that receiver.
2. For more information see “Using Transmitters and Receivers” on page 106 in *IBM Visualization Data Explorer User’s Guide*.

Example Visual Programs

Receivers and transmitters are used by many of the example visual programs, including:

```
AlternateVisualizations.net
Imide_potential.net
```

See Also

Receiver

Transpose

Category

Import and Export

Function

Performs a generalized transpose.

Syntax

`output` `= Transpose(`**`input, dimensions`**`);`

Inputs

Name	Type	Default	Description
`input`	field	none	object to be transposed
`dimensions`	integer list, string list	no transposition	new coordinate list

Outputs

Name	Type	Description
`output`	field	transposed object

Functional Details

This module rearranges the dimensions of a specified input field.

`input`	specifies the field to be transposed.
`dimensions`	specifies the list of transposed coordinates. The *n* dimensions of the object can be specified by number (from 0 to *n* – 1). The first three dimensions can also be specified as *x*, *y*, and *z*, respectively.

The output also has *n* dimensions, and the *i*th output dimension is the same as **`input`**'s **`dimensions`** [*i*]th input dimension.

This module transposes only the "positions" component, leaving other components unaffected. For example, if the "data" component represents a velocity vector, the components of the vector continue to be specified in the original coordinate system. To rearrange the "data" component, use Compute. (If the output of Transpose is later used as input to Slab, Slice, or Stack, the **`dimensions`** parameter must be specified as in the original data before they were transposed.)

Components

All input components are propagated to the output.

Example Visual Program

`GeneralImport2.net`

See Also

Compute, Slice

Tube

Category

Annotation

Function

Changes a specified line into a tube.

Syntax

tube = Tube(**line, diameter, ngon, style**);

Inputs

Name	Type	Default	Description
line	field	none	line to be drawn as a tube
diameter	scalar	input dependent	tube diameter
ngon	integer	8	number of sides to the tube
style	string	"sphere"	style of tube caps

Outputs

Name	Type	Description
tube	field	a renderable tube

Functional Details

This module is intended for use with any module that creates lines (e.g., Streamline).

line	specifies the line that is to be changed into a tube. If a “normals” component is present (as would occur if the input field **curl** were used with Streamline or if the **flag** parameter is set in Streamline), the tube shows a corresponding twist. To show the twist most effectively, use the Ribbon module. Excessive amounts of twist can produce a pinching effect.
diameter	specifies the tube diameter in the same units as those of the original space. If this parameter is not specified, the module provides an appropriate value (1/50 of the diagonal of the boundary box of **line**. This value is attached to the output **tube** as an attribute called “Tube diameter,” which can be extracted with the Attribute module.
ngon	specifies the number of sides of the cross-sectional polygon of the tube.
style	in the current version of Data Explorer has no effect.

Components

Creates new "positions," "connections," and "normals" components. All other components are propagated to the output.

Example Visual Programs

```
Imide_potential.net
Interop.net
PlotLine2.net
RubberTube.net
```

See Also

FaceNormals, Ribbon, Streakline, Streamline

Unmark

Category

Structuring

Function

Unmarks a marked component.

Syntax

```
output = Unmark(input, name);
```

Inputs

Name	Type	Default	Description
input	field	none	field with a marked component
name	string	input dependent	the component to be unmarked

Outputs

Name	Type	Description
output	field	the field with the named component not marked

Functional Details

This module undoes the action of the Mark module by creating an **output** field with the “data” component of the **input** field returned to the **name** component. If a “saved data” component exists, it is copied into the “data” component.

If the **name** parameter is not specified, the module copies the “data” component into the component originally marked with the Mark module (that component name is carried as a “marked component” attribute on **input**). Specifying the parameter explicitly overrides this default behavior.

Components

Copies the “data” component into the **name** component, and the “saved data” component (if it exists) into the “data” component. All other input components are propagated to the output.

Example Visual Programs

```
MakeLineMacro.net
PlotLine.net
PlotTwoLines.net
Sealevel.net
WarpingPositions.net
SIMPLE/MarkUnmark.net
```

See Also

Compute, Mark, Rename

UpdateCamera

Category

Rendering

Function

Alters an existing camera.

Syntax

```
camera = UpdateCamera(camera, to, from, width, resolution, aspect, up,
                      perspective, angle, background);
```

Inputs

Name	Type	Default	Description
camera	camera	none	camera to be altered
to	vector or object	[0 0 0]	look-to point
from	vector or object	[0 0 1]	position of camera
width	scalar or object	100	width of field of view (for orthographic projection)
resolution	integer	640	horizontal resolution of image (in pixels)
aspect	scalar	0.75	height/width
up	vector	[0 1 0]	up direction
perspective	flag	0	0: orthographic projection 1: perspective projection
angle	scalar	30.0	view angle (in degrees) (for perspective projection)
background	vector or string	"black"	image background color

Outputs

Name	Type	Description
camera	camera	altered camera

Functional Details

The altered camera produced by this module is identical to the input **camera** except for the specified camera parameters (**to**, **from**, etc.). For a description of these parameters, see “Camera” on page 49.

See Also

AutoCamera, Camera, Color, Direction, Render

Usage

Category

Debugging

Function

Prints information about current use of resources.

Syntax

```
Usage(what, how);
```

Inputs

Name	Type	Default	Description
what	string	none	the string to be printed
how	integer	0	level of detail

Functional Details

The output of this module appears in the Message window of the user interface. Since, the module traverses the entire memory, it can be used to check for corruption of the memory area.

what specifies what resource information is to be printed. Currently, the only allowed value for this parameter is "memory."

how specifies the level of detail of the printout.

0 Prints out a summary of the total current use of memory. A typical printout might look like:

```
0: 58720256 bytes total : 1379008 in use, 57341248 free
```

1 Prints out a summary of current use of memory, both in small and in large arenas. A typical printout might look like:

```
0: small: 4194304 = hdr 16472 + used 486864 +
                    free 3920 + pool 3687048 (limit 4194304)

0: large: 2097152 = hdr 16472 + used 494656 +
                    free 29704 + pool 1558120 (limit 54525952)
```

where

small is the total number of bytes currently managed by the memory manager for the small arena.

large is the total number of bytes currently managed by the memory manager for the large arena.

hdr is the amount of memory space used by internal data structures.

used is the amount of memory space allocated for use.

free is the amount of memory previously used and available for reuse.

pool is the amount of memory space allocated to Data Explorer but not yet used.

limit is the largest amount of memory that can be managed by the memory manager.

2 lists the number of blocks on each free list

3 lists the number of blocks on each free list plus the actual blocks on each free list

4 lists addresses of all allocated blocks (warning: very long)

5 lists addresses of all allocated and all freed blocks (warning: very long)

Note: You can easily specify Usage("memory", 0) by using the Show Memory Use button in the **Commands** menu of the Message window. The information provided by this module applies to memory use at the time of execution and should be interpreted with that restriction in mind. See "DXPrintAlloc" on page 317 in *IBM Visualization Data Explorer Programmer's Reference* for more information.

Script Language Example

This example shows the use of memory before and after the Isosurface module is printed. (For this example, Data Explorer should be started with the `-readahead off` option.)

```
electrondensity = Import("/usr/lpp/dx/samples/data/watermolecule");
camera = AutoCamera(electrondensity,width=5);
Usage("memory", 1);
isosurface = Isosurface(electrondensity,0.3);
Usage("memory", 1);
Display(isosurface,camera);
```

See Also

Trace

Value

Category

Interactor

Function

Generates a value.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`output`	value	interactor output

Functional Details

This interactor generates a value (i.e. an integer, scalar, vector, or tensor) as output. For information on its use, see “Value Interactor” on page 144 in *IBM Visualization Data Explorer User’s Guide*.

Note: The Value interactor cannot be data driven.

See Also

FileSelector, Integer, IntegerList, Scalar, ScalarList, String, StringList, ValueList, Vector, VectorList

ValueList

Category

Interactor

Function

Generates a value list.

Syntax

Available only through the user interface.

Outputs

Name	Type	Description
`output`	value list	interactor output

Functional Details

The ValueList interactor generates a list of values (i.e., a list of integers scalars, vectors, or tensors) as output. For more information on its use, see “List Interactors” on page 145 in *IBM Visualization Data Explorer User’s Guide.*

Note: The ValueList interactor cannot be data driven.

See Also

FileSelector, Integer, IntegerList, Scalar, ScalarList, String, StringList, Value, Vector, VectorList

Vector

Category

Interactor

Function

Generates a vector.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
data	object	no default	object from which interactor attributes can be derived
refresh	flag	0	reset the interactor
min	scalar, vector	minimum data value	minimum output value
max	scalar, vector	maximum data value	maximum output value
delta	scalar, vector	input dependent	increment between successive scalar outputs
method	string	input dependent	defines interpretation of delta input
decimals	integer	input dependent	number of decimal places to be displayed in output values
label	string	"Vector"	global name applied to interactor stand-ins

Outputs

Name	Type	Description
output	vector	interactor output

Functional Details

This interactor allows the user to interactively change vector values. Through inputs to the module (outputs from other tools or values set in its configuration dialog box) the interactor can be “data driven.”

If the interactor is not data-driven, its attributes (e.g., **stringdata** or **valuelist**) are taken from its **Set Attributes...** dialog box (accessed from the **Edit** pull-down menu in the Control Panel).

Note: The module’s control panel is invoked by double clicking on its icon in the VPE window. Its configuration dialog box is accessed from the **Edit** pull-down menu in the same window.

data	is the object from which the interactor can derive any or all of the minimum, maximum, and delta attributes when their corresponding input uses the default value (tab up). Initially, all inputs are default values.
refresh	resets the interactor so that the output is computed from the current input. If **refresh** = 0 (the default), the output is recomputed only if the current output does not lie within the range of the current **data**. The default for the output of the interactor is, for each component of the vector, the midpoint of the corresponding component of **min** and **max**.
min and **max**	specify the minimum and maximum values of the interactor's vector output. If set, these values override those implied by **data**. Each component of the vector values corresponds to a component of the interactor output. A specified value of [10 20 30] means that the minimum (or maximum) of the first component is 10, of the second is 20, and of the third is 30. When the specified value is scalar, that value is used for all components. If neither **min** nor **data** is specified, the interactor uses the minimum set in the **Set Attributes...** dialog box. If neither **max** nor **data** is specified, the interactor uses the maximum in the **Set Attributes...** dialog box.
delta	specifies a scalar value as a factor for calculating the increment between successive outputs over the specified range. The actual value depends on the interpretation specified by **method** (see below).
method	specifies the interpretation of **delta**: • "rounded": the increment (**max** – **min**) × **delta** is rounded to a "nice" number. The spacing between successive values will approximate the interval specified by **delta**. (For example, the default value of 0.01 specifies an interval of 1/100 of the specified range.) • "relative": the interpretation is the same as for "rounded," but the increment is *not* rounded. • "absolute": **delta** is the absolute value of the interval. (If **delta** has not been specified, its default is 1.) The default value for **method** depends on other input. The default is: – "rounded" if **data** is specified *or* if both **min** and **max** are specified. – "absolute" in all other cases.
decimals	specifies the number of decimal places displayed in the interactor. If neither **data** nor **delta** is specified, the interactor uses the value in its own **Set Attributes...** dialog box.
label	is the global label of all instances of the corresponding interactor stand-in. An interactor instance's local label (set from the Control Panel) overrides a global label. If not specified, the global label is set by the user interface.

Example Visual Programs

```
PlotTwoLines.net
UsingClipPlane.net
UsingCompute.net
```

An example that uses a data-driven vector interactor is:

```
MultipleDataSets.net
```

See Also

Integer, IntegerList, Scalar, ScalarList, VectorList

VectorList

Category

Interactor

Function

Generates a list of vectors.

Syntax

Available only through the user interface.

Inputs

Name	Type	Default	Description
data	object	no default	object from which interactor attributes can be derived
refresh	flag	0	reset the interactor
min	scalar	minimum data value	minimum output value
max	scalar	maximum data value	maximum output value
delta	scalar	input dependent	increment between successive scalar outputs
method	string	input dependent	defines interpretation of delta input
decimals	integer	input dependent	number of decimal places to be displayed in output values
nitems	integer	11	number of items in the initial list
label	string	"ScalarList"	global name applied to interactor stand-ins

Outputs

Name	Type	Description
output	vector list	interactor output

Functional Details

This interactor allows the user to interactively change a list of vector values. Through inputs to the module (outputs from other tools or values set in its configuration dialog box) the interactor can be “data driven.”

If the interactor is not data-driven, its attributes (e.g., **min**, **max**, and **delta**) are taken from its **Set Attributes...** dialog box (accessed from the **Edit** pull-down menu in the Control Panel).

Note: The module’s control panel is invoked by double clicking on its icon in the VPE window. Its configuration dialog box is accessed from the **Edit** pull-down menu in the same window.

data is the object from which the interactor can derive any or all of the minimum, maximum, and delta attributes when their corresponding input uses the default value (tab up). Initially, all inputs are default values.

refresh resets the interactor so that the output is computed from the current input. If **refresh** = 0 (the default), the output is recomputed only if the current output does not lie within the range of the current **data**.

min and max specify the minimum and maximum values of the interactor's vector output. If set, these values override those implied by **data**. Each component of the vector values corresponds to a component of the interactor output.

A specified value of [10 20 30] means that the minimum (or maximum) of the first component is 10, of the second is 20, and of the third is 30. When the specified value is scalar, that value is used for all components.

If neither **min** nor **data** is specified, the interactor uses the minimum set in the **Set Attributes...** dialog box.

If neither **max** nor **data** is specified, the interactor uses the maximum in the **Set Attributes...** dialog box.

delta specifies a scalar value as a factor for calculating the increment between successive outputs over the specified range. The actual value depends on the interpretation specified by **method** (see below).

method specifies the interpretation of **delta**:

- "rounded": the increment (**max** – **min**) × **delta** is rounded to a "nice" number. The spacing between successive values will approximate the interval specified by **delta**. (For example, the default value of 0.01 specifies an interval of 1/100 of the specified range.)
- "relative": the interpretation is the same as for "rounded," but the increment is *not* rounded.
- "absolute": **delta** is the absolute value of the interval. (If **delta** has not been specified, its default is 1.)

  The default value for **method** depends on other input. The default is:
  - "rounded" if **data** is specified *or* if both **min** and **max** are specified.
  - "absolute" in all other cases.

decimals specifies the number of decimal places displayed in the interactor. If neither **data** nor **delta** is specified, the interactor uses the value in its own **Set Attributes...** dialog box.

nitems specifies the number of items in the interactor list. These are evenly spaced between the minimum and maximum values (see above). For example, if this parameter is given a value of 3, and the range is [0 0 0] to [100 100 100], the output list will be {[0 0 0], [50 50 50], [100 100 100]}

Note: If **nitems** changes, a new list is computed.

`label` is the global label of all instances of the corresponding interactor stand-in. An interactor instance's local label (set from the Control Panel) overrides a global label. If not specified, the global label is set by the user interface.

See Also

Integer, IntegerList, Scalar, ScalarList, Vector

Verify

Category

Debugging

Function

Checks an object for internal consistency.

Syntax

```
output = Verify(input, how);
```

Inputs

Name	Type	Default	Description
`input`	object	none	object to be checked
`how`	string	no default	level of verification

Outputs

Name	Type	Description
`output`	object	same object

Functional Details

The Verify module checks an object **`input`** for internal consistency.

- For Fields and Groups, it checks that each component is an array.
- If there are connections, it checks that there are also positions.
- It checks that the positions, if any, are type float, category real, and rank 1.
- It checks that the connections, if any, are type integer, category real, rank 1, and that they possess an “element type” attribute.
- For the various element types, it checks that the shape of the connections array is consistent.
- It checks that the ordering of points in triangles or tetrahedra is consistent.
- For each component in each field, it checks that components that depend on another component have the same number of items as the other component, and that components that reference another component only reference items that exist in the other component.

The **`how`** parameter is currently not used.

Example Visual Program

`SIMPLE/Verify.net`

VisualObject

Category

Debugging

Function

Creates a renderable representation of the hierarchy of a specified object.

Syntax

```
output = VisualObject(input, options);
```

Inputs

Name	Type	Default	Description
`input`	object	none	object to see
`options`	string	"v"	orientation: vertical (“v”) or horizontal (“h”)

Outputs

Name	Type	Description
`output`	field	visual representation of input

Functional Details

The hierarchy of the input object is represented by a tree-like structure that can be passed directly to Image.

`input`	is the object to be represented.
`options`	specifies whether the root of the tree is to be placed at the top of the image (vertical orientation) or at the left (horizontal orientation).

Example Visual Program

`VisualObject.net`

See Also

Image, Describe

WriteImage

Category

Import and Export

Function

Writes an image to a file.

Syntax

```
WriteImage(image, name, format, frame);
```

Inputs

Name	Type	Default	Description
`image`	image or image series	none	the image to be written
`name`	string	"image"	file name
`format`	string	"rgb" or input dependent	format of file
`frame`	integer	format dependent	frame to be written

Functional Details

This module writes an image or series of images to disk.

Note: If you are using the Image tool, the functionality of this module is available in the **`Save Image...`** option of the Image window's **`File`** pull-down menu (see "Saving an Image" on page 94 in *IBM Visualization Data Explorer User's Guide*).

`image` specifies the image to be written to a file on disk.

`name` specifies the name of the file to be written to.

`format` specifies the format in which the image is to be written. The image file format can be specified by the file extension in **`name`** or by **`format`**. If these specifications conflict, **`format`** takes precedence. If the format is one of RGB, R+G+B, YUV, or MIFF, and the given file exists, the image(s) are appended to the file. Otherwise, a new file is always created. The **`format`** parameter allows specification of gamma for all format types. The default gamma is 2. (This is a change from previous releases for which gamma was always 1.) The **`format`** parameter also allows specification that a "delayed colors" image should be written, for all formats other than RGB, R+G+B, and YUV. See PostScript on page 376 for a description of how to specify these two options.

`frame` specifies a frame number in the file on disk. The first frame of a disk file or series is frame number 0 (zero). Its interpretation varies with the format being used.

If the format is one of RGB, R+G+B, or YUV, then **`frame`** indicates the starting image frame in the disk file at which the given image or image series should be written. If the **frame** does not exist in the given file (this is the same as the file not existing), then the file is extended to contain the frames just before the indicated frame number. The contents of the frames that were created to extend the file are undefined. If **frame** is not provided, then the input image(s) are appended to an existing file. If the file does not exist it is created with the given image(s).

For the TIFF and PostScript formats, if **frame** is specified, it is used to modify the output file name. For example, if **frame** = *n*, the name is modified from *name*.tiff to *name.n*.**tiff**. If the image is a series, all frames will be written to this file.

File Formats: See Table 9 for the recognized formats. PostScript** formats may include additional modifiers, separated from the format name by white space. See PostScript on page 376 for the recognized modifiers. The following example sets the format to PostScript and specifies a page size of 4x5 and gamma=1.

```
WriteImage(y,name,"ps page=4x5 gamma=1");
```

The RGB (“**rgb**” and “**r+g+b**”) and YUV formats allow an existing file to be modified, either by overwriting existing frames or by extending the number of frames in the file. The TIFF and PostScript formats do not support this capability. MIFF allows appending of images (but not overwriting of images).

Table 9 summarizes the characteristics of each format. Descriptive sections follow the table.

Table 9. Format Characteristics for WriteImage

File Type	Format Specifier	Resulting File Extension(s)	Multiframe/ Series Data	Modify an Existing File
gif	"gif"	.gif	No	No
rgb	"rgb" “r+g+b”	.rgb and .size .r, .g, .b, and .size	Yes	Yes
TIFF	"tiff"	.tiff	Yes	No
yuv	"yuv"	.yuv	Yes	Yes
Color PostScript	"ps color" “ps”	.ps	Yes	No
Gray PostScript	"ps gray" “ps grey”	.ps	Yes	No
Color PostScript (Encapsulated)	"eps" “eps color”	.epsf	No	No
Gray PostScript (Encapsulated)	"eps gray" “eps grey”	.epsf	No	No
MIFF	"miff"	.miff	Yes	Yes

gif Graphics Interchange Format** adheres to the 87A convention. It includes LZW compression. Images are saved with 24-bit colors, with a maximum palette of 256 colors.

rgb	rgb file format consists of two files, a binary *name*.rgb file that contains the image pixel values and a *name*.size file used to specify the image dimensions and the number of images contained in the *name*.rgb file. The *name*.size file contains a single line of ASCII text in the format "*w*×*h*×*f*," where *w* and *h* give the dimensions of each image and *f* indicates the number of images in the *name*.rgb file. The *name*.rgb file contains (red, green, blue) binary pixel values with 8 bits per color (24 bits/pixel). Pixels are read and written from the image in left-to-right, top-to-bottom order.
r+g+b	r+g+b format is similar to rgb format, except that the *name*.rgb file is replaced by three binary files, one for each color. The *name*.r file contains the red color values for all images, the *name*.g file contains the green color values for all images and the *name*.b file contains the blue color values for all images. Again, all color values are 8 bits (24 bits/pixel).
TIFF	TIFF (tag image file format) files are binary files that contain 24 bits/pixel color resolution. For more information on TIFF files see *Tag Image File Format Specification*, Revision 5.0, available from Aldus Corporation or Microsoft Corporation.
yuv	This format adheres to Abekas YUV format, which can be directly imported to a variety of public domain MPEG encoders.
PostScript	PostScript format files are written out using PostScript's image (or color image) operator and require only PostScript Level 1 interpreter support. WriteImage supports four different varieties of PostScript. Images can be written out in either color or gray-scale and either "Encapsulated" or not. Color images are written out at 24 bits/pixel and gray-scale at 8 bits/pixel. Encapsulated PostScript is a format intended to be used when incorporating images into other documents. For this reason, the Encapsulated PostScript formats do not support series image input (i.e., only one image per file is acceptable). All PostScript output is run-length encoded to reduce the file size. (For more information on PostScript see *PostScript Language Reference Manual*, Second Edition; Addison-Wesley Publishing Company, 1990.)

By default, the image will be scaled and oriented to fill the current page size within the specified **margin** of the edge, while preserving the original aspect ratio of the image. That is, the image will be made as large as possible while maintaining the specified margin on at least one of the two dimensions. If **dpi** or **width** is specified, these specifications will override the autoscaling feature. It is typically only necessary to use the **page** and **margin** modifiers.

Note: The ReadImage module does not support the PostScript formats.

PostScript supports the following format modifiers:

page = w × h	Sets the page size in inches (width × height). The width is the width of the printer (typically, the dimension perpendicular to page motion). The default is 8.5×11.
dpi = *n*	Sets the number of dots (pixels) per inch in the hardcopy image. Note that **dpi** does **not** correspond to the dpi of the printer.

`orient = landscape orient = portrait orient = auto`	Indicates the orientation of the image on the page. **`landscape`** means that the image's "up vector" (bottom to top) runs across the width of the page. In **`portrait`** mode, the up vector runs up the length of the page. The default is **`auto`**.
`width = w`	Specifies the size in inches of the "across the screen" dimension of the image as it appears on the page. If specified, it overrides the **`dpi =`** modifier.
`margin = m`	Sets the desired margin around the image on the page. The default is .5 inch.
`height = h`	Specifies the size in inches of the "top to bottom of the screen" dimension of the image as it appears on the page. If specified, it overrides the **`margin =`** modifier.
`gamma = g`	Sets the gamma correction factor for the output image. The default is 2.0. This modifier is available for all format types.
`delayed = 1`	Specifies that a "delayed colors" image should be written, that is, an image-with-colormap. This modifier is available for PostScript, TIFF, and MIFF formats. In addition, it is set by default for GIF format (as all GIF images are in image-with-colormap format).

For example:

```
WriteImage(y, name, "ps color page=4x5 margin=0.4 orient=landscape");
```

If only **`width`** or **`height`** is specified (but not both), the original aspect ratio of the image is maintained. If both are specified, the image may be stretched.

MIFF is a run-length-encoded format that supports image sequences. Writing an image to an existing miff file name causes the image to be appended to the file. To start a new sequence you must delete the .miff file.

Note: If you are writing out images for use later from within Data Explorer, you will probably want to set gamma correction to 1 (the default is 2). Otherwise, images will be doubly gamma-corrected. (Gamma correction at display time can also be controlled using the `DXGAMMA` environment variable; see C.1, "Environment Variables" on page 292 in *IBM Visualization Data Explorer User's Guide*). For printing or viewing images in another package, use gamma appropriate for that device.

See Also

ReadImage, QuantizeImage

Glossary

Some of the definitions in this glossary are taken from the *IBM Dictionary of Computing*, SC20-1699.

A

array. In Data Explorer, an array structure containing an ordered list of data items of the same type along with additional descriptive information. Arrays are either compact or irregular. See *compact array, irregular array*.

assembly. An object representing a collection of objects.

attribute. A characteristic of an object. Objects can have attributes that are indexed by a string name and have a value that is an object. See also *component attribute*.

C

cell-centered data. Connection-dependent data.

clipping plane. A plane that divides a three-dimensional object into a rendered and an unrendered region, making the object's interior visible.

colormap. A map that relates colors to data values. The colors are carried in the map's "data" component and the data values to which each color applies in its "positions" component.

colormap editor. A special tool for mapping precise colors to specified data values, the results of which are displayed in a visual image.

compact array. Any of five types of compact encoding of array data:

constant array
mesh array
path array
product array
regular array

component. A basic part of a field (such as "positions," "data," or "colors"); each component is indexed by a string (e.g., "positions"), and its value is typically an array object (e.g., the list of position values). See also *component attribute*.

component attribute. A characteristic of a component. Components of a field can have attributes that are indexed by a string name and have a value that is an object.

composite field. A grouping of like fields for processing a single spatial entity. See also *partitioned field*.

connection. Component of an IBM Data Explorer data field that specifies how a set of points are joined together. Also controls interpolation.

connection-dependent data. Cell-centered data. The data value is interpreted as constant throughout the connection element.

contour. On a surface, a line that connects points having the same data value (e.g., pressure, depth, temperature).

cube. A volumetric connection element that connects eight positions in a data field.

cutting plane. An arbitrary plane, in three-dimensional space, onto which data are mapped.

D

data-driven interactors. Interactors whose attributes (such as minimum and maximum) are set by an input data field.

Data Prompter. An interface that enables a user to describe the format of the data in a file. The prompter creates a General Array Format header file that is used by the Import module to import the data.

dependence. A component attribute. One component is said to be dependent ("dep") on another if the items in their component arrays are in one-to-one correspondence to each other.

E

element. Connection item.

element type. An attribute that describes the type of connection element, for example, "cubes", "tetrahedra", or "lines".

executive. The component of the Data Explorer system that manages the execution of specified modules. The term often refers to the entire server portion of the Data Explorer client-server model, including the executive, modules, and data-management components.

F

face. (1) Any planar surface that bounds a three-dimensional object. (2) A polygon.

field. A self-contained collection of data items. A Data Explorer field typically consists of the data itself (the “data” component), a set of sample points (the “positions” component), a set of interpolation elements (the “connections” component), and other information as needed.

flat shading. A shading model in which each face of an object is shaded with a single intensity value. Contrast with *Gouraud shading*.

fork. An operation that causes a program to branch into two or more parallel concurrent paths.

fork-join parallelism. A programming mechanism that supports parallel processing: The fork statement splits a single computation into multiple independent computations. The join statement recombines two or more concurrent computations into one.

G

general array format. A data-importing method that uses a header file to describe the data format of a data file. This “format” makes it possible to import data in a variety of formats.

glyph. A graphical figure used to represent values of a particular variable. The length, angle, or other attribute of the glyph is some function of the value of that variable. Each occurrence of a glyph represents a single value of the variable.

Gouraud shading. Also called intensity interpolation shading. A shading model in which the intensity of values of incident illumination on a polygon are interpolated from intensity values at the vertices of the polygon. Contrast with *flat shading*.

I

icon. A displayed symbol that a user can point to with a device such as a mouse to select a particular operation or software application.

image window. IBM Data Explorer window that displays the image generated by a visual program. Associated with the Image window are special interactors for 3-D viewing.

interactor. A Data Explorer device used to manipulate data in order to change the visual image produced by a program. See also *data-driven interactor, interactor stand-in*.

interactor stand-in. An icon used in the VPE window to represent an interactor. Stand-ins are named after the type of data they generate:

- integer
- scalar
- selector (outputs a value and a string)
- string
- value
- vector

interpolation element. An item in the connections component array. Each interpolation element provides a means for interpolating data values at locations other than the specified set of sample points. See *positions component*.

invalid. A classification of an array item (typically positions or connections). An invalid item is not to be rendered or realized.

irregular array. In contrast to a compact array, an array in which the data is stored explicitly.

isosurface. A surface in three-dimensional space that connects all the points in a data set that have the same value.

isovalue. The single value that characterizes each and every point constituting an isosurface. By default, this value is the average of all the data values in the set being visualized.

J

join. An operation that merges two or more computation paths.

L

line. An element that connects two positions in a field.

M

macro. In IBM Data Explorer, a sequence of modules that acts as a functional unit and is displayed as a single icon. Macros can also be defined in the Data Explorer scripting language.

member. An individual unit or object in a group. A collection of members makes a group.

menu bar. In windows, a horizontal bar that displays the names of one or more menus (or tasks). When the user selects a menu, a pull-down list of options for that menu is displayed.

mesh array. A compact array that encodes multidimensional regularity of connections. It is a product of path arrays. In a mesh array, which positions are connected to one another is implicitly rather than explicitly defined.

N

navigate. To move the camera (changing the "to" and "from" points) around the image scene, using the mouse.

netCDF. Network Common Data Form.

network. In Data Explorer the set of tool modules, interactor stand-ins, and connections that constitute a visual program. In the VPE window, a network appears as a set of icons connected by arcs.

Network Common Data Form (netCDF). A data format that stores and retrieves scientific data in self-describing, multidimensional blocks (netCDF is not a database management system, however). netCDF is accessible with C and FORTRAN.

normal. (1) Perpendicular to a surface. (2) In IBM Data Explorer, a vector that is perpendicular to a face or surface of an object. A normal may depend on connections or positions. A connection-dependent normal results in flat shading; a position-dependent normal results in Gouraud shading.

O

object. In IBM Data Explorer, any discrete and identifiable entity; specifically, a region of global memory that contains its own type-identification and other type-specific information.

opacity. The capacity of matter to prevent the transmission of light. For a surface, an opacity of 1 means that it is completely opaque; an opacity of 0, that it is completely transparent. For volume, opacity is defined as the amount of attenuation (of light) per unit distance.

P

partitioned field. A composite field, created by partitioning a single field into a collection of separate fields; used for parallel processing and data-management purposes.

path array. A compact array that encodes linear regularity of connections. It is a set of n–1 line segments, where the ith line segment joins points i and $i + 1$.

polygon. (1) Any multi-sided planar figure. (2) A face of a three-dimensional object.

position-dependent data. Data that are in one-to-one correspondence with positions.

positions component. A component that consists of a set of dimensional points in a field.

probe. A list of one or more vectors that represent points in a graphical image. Probes can be used with Data Explorer tools that accept vectors as input (such as ClipPlane and Streamline) or to control the view of an image.

product array. A compact array that encodes multidimensional positional regularity. It is the set of points obtained by summing one point from each of the terms in all possible combinations. In the simplest case, each term is a regular array.

Q

quad. An element that connects four positions in a field.

R

realization. A description of how raw data is to be represented in terms of boundaries, surfaces, transparency, color, and other graphical, image, and geometric characteristics.

reference. A component attribute. One component is said to *refer to* another ("ref") if the items in the first array are integer indices into the second array. The connections component references the positions component.

regular array. A compact array that is a set of n points lying on a line, with constant spacing between them, which can represent one-dimensional regular positions.

rendering. The generation of an image from some representation of an object, such as a surface, or from volumetric information.

ribbon. A figure derived from lines (e.g., from streamlines and streaklines). Ribbons may twist to indicate vorticity.

S

sample point. A point that represents user data. Data is interpolated between sample points by interpolation elements (connections).

scalar. A non-vector value characterized by a single, real number.

scatter data. A collection of sample points without connections.

screen. An illuminated display surface (e.g., the display surface of a CRT or plasma panel).

scripting language. The IBM Data Explorer command language. Used for writing visual programs, to manage the execution of modules, and to invoke visualization functions.

sequencer. An IBM Data Explorer tool for creating "animated" sequences of images.

series. In IBM Data Explorer, used to represent a single field sampled across some parameter (e.g., a simulation of a CMOS device across a temperature range). Members of a series have a position. A copy of the position is found in the "series position" attribute.

shared. A term used to indicate the availability of a resource for use by more than one program at the same time.

specular reflection. A reflection from a shiny object.

stand-in. See *interactor stand-in*.

streaklines. Lines that represent the path of particles in a changing vector field. Also called rakes.

streamlines. Lines that represent the path of particles in a vector field at a particular time. Also called flow lines.

T

tetrahedron. A volumetric connection element that connects four positions in a field.

tool. In IBM Data Explorer, a general term for any icon used to build a visual program (specifically, module, macro, or interactor stand-in).

triangle. A connection element that connects three points in a field.

tube. A surface centered on a deriving line (e.g., a streamline or streakline). Tubes may twist to indicate vorticity. See also *user display station*.

V

value. An instance of an attribute (for example, "blue" as the value of the attribute "color").

vector. A quantity characterized by more than one component.

visual program. A user-specified interconnected set of Data Explorer modules that performs a sequence of operations on data and typically produces an image as output.

vertex. One of the positions that define a connection element.

volume. The amount of three-dimensional space occupied by an object or substance (measured in cubic units). To be distinguished from an object's surface, which is a mathematical abstraction.

volume rendering. A technique for using color and opacity to visualize all the data in a 3-dimensional data set. The internal details visualized may be physical (such as the structure of a machine part) or they may be other characteristics (such as fluid flow, temperature, or stress).

vorticity. Mathematically defined as the curl of a velocity field. A particle in a velocity field with nonzero vorticity will rotate.

W

wireframe. Connected lines that represent a surface.

Index

Readers' Comments — We'd Like to Hear from You

IBM Visualization Data Explorer
User’s Reference
Version 3 Release 1 Modification 4

Publication No. SC38-0486-03

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC38-0486-03

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
Thomas J. Watson Research Center/Hawthorne
Data Explorer Development
P.O. Box 704
YORKTOWN HEIGHTS, NY
USA 10598-0704

Fold and Tape **Please do not staple** Fold and Tape

SC38-0486-03

Cut or Fold
Along Line

Printed in U.S.A.

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber

SC38-0486-03