AVS MODULE REFERENCE

Release 5
February, 1993

Advanced Visual Systems Inc.

Part Number: 320-0014-03, Rev A

NOTICE

Advanced Visual Systems Inc.
300 Fifth Ave.
Waltham, MA 02154

NAME

AVS modules – introduction to manual pages for AVS modules

DESCRIPTION

This man page summarizes all modules available with the AVS distribution in alphabetical order. The individual module man pages follow in alphabetical order.

The manual pages are also available on-line. You can view them within AVS by clicking on the small square "dimple" in any module icon with the middle or right mouse button to open its Module Editor window. Then click the **Show Module Documentation** button to view the complete manual page for the module. They may also be seen using the regular help browser, in the following directory:

$AVS_PATH / runtime / help / modules

MODULE LIBRARIES

Modules are organized into module libraries for easy interactive access. By default, these module libraries appear when you first start AVS:

Supported
Imaging
Volume
UCD
FiniteDiff
Animation (if present on your system)
Chemistry
Unsupported

All modules are in the Supported module library except Animation, Chemistry, and the Unsupported modules.

Any one module can be in multiple module libraries. At the top of each module's man page there is an "Availability" line that lists which module libraries the module can be found in, in addition to the default Supported library.

Each module library is described on its own man page.

INPUT/OUTPUT DATA TYPE NOTATION

The data-types that AVS modules operate on are described in the "Importing Data into AVS" chapter of the *AVS User's Guide*, and in the *AVS Developer's Guide* in the chapter, "AVS Data Types". Throughout the manual pages for AVS modules, a number of terms are used to describe these data types. It is important to understand these terms, as they specify what inputs a given module can receive, and what outputs it will generate.

any-dimension:

when a module accepts fields of *any-dimension*, this means that it can process fields that are 1D, 2D, 3D, and in some cases 4D; but never more than this.

n-vector:

if a field has one value at each location, it is a *scalar* field. When a module accepts *n-vector* fields, it can receive fields with an indeterminate number of values at each location.

any-data:

if a module accepts *any-data*, this means it can receive byte, short, integer, float, or double data. If it is more restrictive, this will be declared.

any-coordinates:

if a module accepts data of *any-coordinates*, this means that it can operate on fields which have uniform, rectilinear, or irregular coordinates. If a module cannot operate on one of these types of field, this will be declared.

PLATFORM DEPENDENCE

Some mapper modules required specialized graphics rendering support such as 3D texture mapping (**brick**, **excavate brick**, etc.) and object transparency (**alpha blend**, **volume render**, etc.). This specialized rendering support can be provided in software (via the software renderer), or by hardware. However, not all hardware rendering platforms support all specialized rendering features. The hardware rendering features available on your platform should be defined in a table in the release notes that accompany the AVS software product for that platform. The software renderer supports most specialized rendering features except vertex transparency.

Each module with specialized rendering requirements has an "Availability" notation near the top of its module man page that defines the support needed. If your renderer does not support the function, the picture will not appear as documented. For example, a texture-mapped object will appear as an uncolored, featureless object. Transparent objects will be opaque, or not drawn at all. You can almost always acquire the specialized rendering support by switching on the **Software Renderer** option on the Geometry Viewer's **Cameras** submenu. If no such selection appears on the **Cameras** submenu, it means that the software renderer is probably the only renderer available and is already performing rendering functions in the AVS Geometry Viewer.

MODULE LISTING

The modules included in this release of AVS are:

AVS modules	introduction to manual pages for AVS modules
Finite Difference Module Library	a list of supported modules that are also in the FiniteDiff module library
Imaging Module Library	a list of supported modules that are also in the Imaging module library
UCD Module Library	a list of supported modules that are also in the UCD module library
Unsupported Module Library	a list of unsupported modules
Volume Module Library	a list of supported modules that are also in the Volume module library
AVS module groups	Types of AVS modules
avs	Starting the Application Visualization System. Describes AVS command line options, *.avsrc* startup file keywords, and environment variables.
alpha blend	generate 2D image from 3D colored data (unsupported library)
animated float	send a sequence of floating point numbers to a module's parameter port
animated integer	send a sequence of integers to a module's parameter port

animate lines	animate stream lines for a vector field
antialias	antialias an image
arbitrary slicer	map 3D scalar field to 3D mesh
average down	downsize a field in X, Y, or Z by averaging
AVS Animator	keyframe animation module (Animation library)
background	create a shaded backdrop image
blend colormaps	interpolate between two colormaps in HSVA space
boolean	send a user-entered boolean value to one or more module(s) boolean parameter port(s)
brick	show uniform volume as a solid (requires 3D texture mapping support)
bubbleviz	generate spheres to represent values of 3D field
calc warp coeffs	calculate warp coefficients for ip warp module
cfd values	calculate values for a field containing read plot3D data (unsupported library)
character string	send a user-entered string to one or more module(s) string parameter port(s)
clamp	restrict values in data field
clip geom	specify arbitrary clipping planes for geometric objects (requires arbitrary clipping plane support)
color legend	display color-to-data value mappings in geometry viewer window
color range	store minimum and maximum field values in an AVS colormap
colorize geom	assign vertex colors, vertex transparency, and/or UVW values to verticies of a geometry using field and colormap (requires vertex transparency and/or 3D texture mapping support)
colorizer	convert field of data values to color values
colormap manager	share colormaps among subnetworks (unsupported library)
combine scalars	combine scalar fields into a vector field
compare field	compare two AVS fields, display and write data difference
composite	blend two images using alpha transparency
compute gradient	compute gradient vectors for 2D or 3D data set
compute shade	combined colorizer/compute gradient/gradient shade module
contour to geom	create geometry of 2D or 3D scalar field contour slices
contrast	perform linear transformation on range of field values
convolve	apply a signal processing filter to 2D field
create geom	interactively create and manipulate geometry objects such as polylines, arcs, and surfaces

crop	extract subset of elements from a field
cube	perform ray-traced volumetric rendering on volume data
data dictionary	read external data file using a form specification
Data Viewer	run the Data Viewer application
dialog box	use a long dialog box to create a long string
display image	show image in a display window
display pixmap	show pixmap in a display window (unsupported library)
display tracker	display and directly manipulate the tracer module's output
dot surface	generate points that define an isosurface (unsupported library)
downsize	reduce size of data set by sampling
draw grid	draw a grid on top of an image
edit substances	create a substance table for the cube module
euler transformation	send object transformation matrix to other modules
excavate	remove an octant from a 3D uniform field, revealing interior features
excavate brick	show uniform volume with orthogonal slices (requires 3D texture mapping support)
extract graph	extract and display a graph of a 1D slice from a 2D data set
extract scalar	extract a scalar field from a vector field
extract vector	subset of field vector elements as new field
field legend	select value from scalar field using color legend
field math	perform math operations between fields
field to byte	transform any field to an byte-valued field
field to double	transform any field to a field of double-precision floating point values
field to float	transform any field to a field of single-precision floating point values
field to int	transform any field to an integer-valued field
field to mesh	transform a 2D scalar field to a surface in 3D space
field to short	transform any field to a field of short values
field to ucd	convert AVS field to unstructured cell data format
file browser	send a filename to one or more module(s) filename parameter port(s)
file descriptor	create a data form specification to read an external data file
flip normal	change direction of each vertex normal for a geometry object
float	send a floating point number to one or more module(s) floating point parameter port(s)
generate axes	generate 3D geometric axes

generate colormap	output AVS colormap
generate filters	generate 2D filters for image processing
generate grid	creates grids on XY, XZ, and YZ coordinate planes
generate histogram	plot distribution of data values in a scalar field
geometry viewer	display and manipulate collections of 3D objects (Geometry Viewer subsystem)
gradient shade	apply lighting and shading to colored data set
graph viewer	create XY and contour plots of data (Graph Viewer subsystem)
hedgehog	show vectors in a 3D 3-vector field
histogram stretch	balance the histogram of a data set
image compare	display two images together
image manager	share images among subnetworks (unsupported library)
image measure	measure distance between two image pixels
image probe	report data values at selected pixel location
image to cgm	convert image to CGM and store in file
image to pixmap	convert image to pixmap (unsupported library)
image to postscript	convert image to gray-scale or color PostScript and store in file
image viewer	display and manipulate collections of images (Image Viewer subsystem)
integer	send a user-entered integer to the integer parameter port of one or more module(s)
interpolate	compute intermediate values to change the size of a field
ip absolute	absolute value of a field
ip arithmetic	arithmetic operations on fields
ip blend	alpha or compositing blend of two fields
ip compare	compare two fields
ip contour	draw iso-level contours
ip convolve	convolve with image float kernel
ip dilate	dilate a field
ip edge	enhance edges in a field
ip erode	erode a field
ip extrema	find data value extrema
ip fft	Fourier transform a field
ip fft display	calculate magnitude an phase of packed FFT field
ip fft multiply	multiply two packed complex fields
ip fft pack	fold conjugate symmetric FFT representation

ip fft unpack	unfold conjugate symmetric FFT representation
ip float math	floating point operations on a field
ip histogram	field histogram
ip ifft	inverse Fourier transform for conjugate data sets
ip lincomb	inter-band linear combination
ip linremap	linearly remap a field
ip logical	bitwise logical operations
ip lookup	pass field through lookup table
ip median	median field filter
ip merge	merge two fields
ip morph	morphological operation
ip read kernel	read a convolution kernel from a file into a field
ip read line	read line of data between two image pixels
ip read mtable	read a morphology table from a file into a field
ip read sel	read a structuring element from a file into a field
ip read vff	import a SunVision *.vff*-format image file into an AVS field
ip reflect	rotate or transpose field
ip register	determine maximum correlation position
ip rescale	rescale a field
ip rotate	rotate a field
ip statistics	find field mean and variance
ip threshold	threshold field against a float value
ip translate	field translation
ip twarp	arbitrary field warp using warp data from table
ip warp	polynomial image warp
ip write vff	save an AVS image-format field as a SunVision *.vff*-format image file
ip zoom	zoom field with interpolation
isosurface	generate an isosurface for a volume of data
label	creates a title for flexible geometry viewer annotation
local area ops	image processing based on pixel neighborhoods
luminance	compute the luminance of an image
minmax	set min and max values of a selected vector in an AVS field
mirror	reverse array indices in a 2D or 3D data set
Module Generator	create skeletal C or FORTRAN module source code from menu description
offset	deform, or "blow up" a geometry object based on vector values at each node

oneshot	send a oneshot value to one or more module(s) "oneshot" parameter port(s)
orthogonal slicer	slice through 3D or 2D field with plane perpendicular to coordinate axis
output postscript	convert pixmap to PostScript™ and store in file (unsupported library)
particle advector	release grid of particles into velocity field
pdb to geom	create molecule geometry from Protein Data Bank(PDB) file (unsupported library)
pixmap to image	transform AVS pixmap to AVS image (unsupported library)
print field	create an ASCII printable / readable version of an AVS field
probe	interactively show numeric data values in a geometry rendered field
read field	read AVS field from a disk file, or import data files into AVS field format
read geom	reads a data file containing an AVS ´geometry´
read image	read image file from disk into a field
read plot3d	read a PLOT3D format file into an AVS field (unsupported library)
read ucd	read UCD structure from disk file
read volume	read volume file from disk into a field
render geometry	manipulate collections of 3D objects (unsupported library)
render manager	share geometries among subnetworks (unsupported library)
replace alpha	replace the alpha channel (transparency) in an image
ribbons	generate ribbon representation of streamlines
samplers	extract a subset of locations from a 3-vector 3D field
scatter dots	generate spheres at points in 3D space
scatter to ucd	convert a scatter field to a tetrahedral UCD structure
set view	view objects in geometry viewer from fixed orthogonal orientations
shrink	make polygons of a geometry object smaller
sketch roi	create a region-of-interest field
sobel	apply an edge detecting filter to 2D field
statistics	display statistics on AVS field contents
stream lines	generate stream lines for a vector field
3D bar chart	3d bar chart with average statistics and annotation
threshold	restrict values in data field
thresholded slicer	slice through volume data with high / low values invisible
time sampler	extract 3D time slices from 4D time series field with interpolation

tracer	perform ray-traced volumetric rendering on volume data
track ball	send object transformation matrix to other modules
transform pixmap	perform 3D transformation on pixmap (unsupported library; requires 2D texture mapping support)
transpose	exchange dimensions in a 2D or 3D data set
tristate	send a tristate value to one or more module(s) tristate parameter port(s)
tube	convert lines to cylindrical tubes
ucd anno	show data values of cells or nodes of a UCD structure
ucd cell	convert ucd cell-based data into node data
ucd cell color	color ucd structure based on cell or material id values
ucd contour	generate list of color values associated with unstructured cell data
ucd crop	subset UCD structure data using slice plane or box
ucd curl	compute the curl of a vector UCD structure
ucd div	compute the divergence of a vecor UCD structure
ucd extract	extract single node component from a UCD structure
ucd extract scalars	extract scalar node components from scalar and vector components of a UCD structure
ucd extract vector	extract single vector node component from scalar components of a UCD structure
ucd grad	compute the vector gradient of a UCD structure
ucd hex to tet	convert a UCD structure from hexahedral cells to tetrahedral cells
ucd hog	show UCD node vector values as line segments in 3D space
ucd iso	generate an isosurface for a UCD structure with scalar node data
ucd isolines	generate isolines on the exterior boundary of a UCD structure
ucd legend	creates a color legend relating UCD data to a color scale
ucd math	perform math operations between UCD structures
ucd minmax	set min and max values of a component in a UCD structure
ucd offset	deform a UCD structure based on vector values at each node
ucd plot	create a field to graph a linear sample through a UCD structure
ucd print	create a readable format of a UCD structure
ucd probe	interactively show numeric data values in a geometry rendered UCD structure
ucd reverse cell	repair topology of imported UCD structures; reverse cell normals

ucd rslice	slice away portions of a UCD structure
ucd rubber sheet	map values as a 3D surface with height proportionate to value
ucd slice2D	extract 2D slice from a UCD structure
ucd streamline	generate stream lines for a UCD structure with vector node data
ucd threshold	restrict values in a UCD structure
ucd to geom	convert a UCD structure into an AVS geometry
ucd tracer	perform ray-traced volumetric rendering on a UCD structure
ucd vector mag	compute the magnitude of a vector ucd
ucd vol integral	calculate the volume of a UCD structure, and the volume integral of a scalar data component
vector curl	compute the curl of a vector field
vector div	compute the divergence of a vector field
vector grad	compute the vector gradient of a scalar field
vector mag	compute the magnitude of a vector field
vector norm	normalize a vector field
volume bounds	generate bounding box of 3D 3-vector field
volume manager	share volumes among subnetworks (unsupported library)
volume render	volume render a uniform volume with geometry (requires 3D texture mapping with alpha transparency and volume rendering support)
wireframe	convert object from surface to wireframe representation
write field	write a field description to disk
write image	store image data in a file
write ucd	write unstructured cell data to disk
write volume	write volume data to a file
x-ray	perform simple orthographic volume visualization

Finite Difference Module Library

NAME

FiniteDiff Module Library – modules suited to finite difference networks

DESCRIPTION

The FiniteDiff module library is a subset of the supported AVS modules that are suited to finite difference applications.

This man page lists the modules in two ways: alphabetically, and classified by their type (Data Input, Filters, Mappers, Data Output). See the individual module man pages for specific information on each module.

ALPHABETIC LIST

3D bar chart
animated float
animated integer
animate lines
arbitrary slicer
average down
boolean
brick
bubbleviz
character string
clamp
clip geom
color legend
color range
colorize geom
colorizer
combine scalars
compare field
compute gradient
compute shade
contour to geom
contrast
create geom
crop
cube
display image
display tracker
downsize
edit substances
euler transformation
excavate
excavate brick
extract graph
extract scalar
extract vector
field legend
field math
field to byte
field to double
field to float
field to int
field to mesh
field to short
file browser
file descriptor
flip normal
float
generate axes
generate colormap
generate grid
generate histogram
geometry viewer
gradient shade
graph viewer
hedgehog
histogram stretch
image to cgm
image to postscript
image viewer
integer
interpolate
isosurface
label
minmax
mirror
oneshot
orthogonal slicer
particle advector
print field
probe
read field
read geom
read volume
ribbons
samplers
scatter dots
set view
shrink
statistics
stream lines
threshold
thresholded slicer
time sampler
tracer
track ball
transpose
tube
vector curl
vector div
vector grad
vector mag
vector norm
volume bounds
volume render
wireframe
write field
write image
write volume
x-ray

DATA INPUT MODULES

animated float
animated integer
boolean
character string
clip geom
color range
create geom
edit substances
euler transformation
file browser
file descriptor
float
generate axes
generate colormap
generate grid
integer
label
oneshot
read field
read geom
read volume
samplers
set view
track ball

FILTERS

animate lines
average down
clamp
colorize geom
colorizer
combine scalars
compute gradient
compute shade
contrast
crop
downsize
excavate
extract graph
extract scalar
extract vector
field math
field to byte, double, float, int, short
flip normal
generate histogram
gradient shade
histogram stretch
interpolate
minmax
mirror
ribbons
shrink
threshold
time sampler
transpose
tube
vector curl
vector div
vector grad
vector mag
vector norm
wireframe
x-ray

MAPPERS

3D bar chart
arbitrary slicer
brick
bubbleviz
color legend
contour to geom
cube
excavate brick
field legend
field to mesh
hedgehog
isosurface
orthogonal slicer
particle advector
probe
scatter dots
stream lines
thresholded slicer
tracer
volume bounds
volume render

DATA OUTPUT

compare field
display image
display tracker
geometry viewer
graph viewer
image to cgm
image to postscript
image viewer
print field
statistics
write field
write image
write volume

Imaging Module Library

NAME

Imaging Module Library – modules suited to Imaging networks

DESCRIPTION

The Imaging module library is a subset of the supported AVS modules that are suited to imaging applications.

This man page lists the modules in two ways: alphabetically, and classified by their type (Data Input, Filters, Mappers, Data Output). See the individual module man pages for specific information on each module.

VECTOR LENGTHS

Many of the **ip** image processing modules are described as accepting *n-vector* input. In fact, the maximum number of vector elements (or "channels", or "bands") that these modules accept is 12.

ALPHABETIC LIST

3D bar chart
animated float
animated integer
antialias
average down
background
boolean
calc warp coeffs
character string
clamp
color legend
color range
colorizer
combine scalars
compare field
composite
compute gradient
compute shade
contour to geom
contrast
convolve
crop
data dictionary
display image
downsize
draw grid
extract graph
extract scalar
extract vector
field legend
field math
field to byte
field to double
field to float
field to int
field to mesh
field to short
file browser
file descriptor
generate filters
generate grid
generate histogram
geometry viewer
gradient shade
graph viewer
histogram stretch
image compare
image measure
image probe
image to cgm
image to postscript
image viewer
integer
interpolate
ip absolute
ip arithmetic
ip blend
ip compare
ip contour
ip convolve
ip dilate
ip edge
ip erode
ip extrema
ip fft
ip fft display
ip fft multiply
ip fft pack
ip fft unpack
ip float math
ip histogram
ip ifft
ip lincomb
ip linremap
ip logical
ip lookup
ip median
ip merge
ip read mtable
ip read sel
ip read vff
ip reflect
ip register
ip rescale
ip rotate
ip statistics
ip threshold
ip translate
ip twarp
ip warp
ip write vff
ip zoom
label
local area ops
luminance
minmax
mirror
oneshot
orthogonal slicer
print field
read field
read image
replace alpha
set view
sketch roi
sobel
statistics
threshold
transpose
write field
write image

float	ip morph	
generate axes	ip read kernel	
generate colormap	ip read line	

DATA INPUT MODULES

animated float	float	label
animated integer	generate axes	oneshot
background	generate colormap	read field
boolean	generate filters	read image
calc warp coeffs	generate grid	set view
character string	integer	sketch roi
color range	ip read kernel	
data dictionary	ip read mtable	
file browser	ip read sel	
file descriptor	ip read vff	

FILTERS

antialias	histogram stretch	ip median
average down	image compare	ip merge
clamp	interpolate	ip morph
colorizer	ip absolute	ip reflect
combine scalars	ip arithmetic	ip rescale
composite	ip blend	ip rotate
compute gradient	ip contour	ip threshold
compute shade	ip convolve	ip translate
contrast	ip dilate	ip twarp
convolve	ip edge	ip warp
crop	ip erode	ip zoom
downsize	ip fft	local area ops
draw grid	ip fft display	luminance
extract graph	ip fft multiply	minmax
extract scalar	ip fft pack	mirror
extract vector	ip fft unpack	replace alpha
field math	ip float math	sobel
field to byte,	ip ifft	threshold
double, float,	ip lincomb	transpose
int, short	ip linremap	
generate histogram	ip logical	
gradient shade	ip lookup	

MAPPERS

3D bar chart	field to mesh	ip histogram
color legend	image measure	ip read line
contour to geom	image probe	orthogonal slicer
field legend		

DATA OUTPUT

compare field	image viewer	print field
display image	ip compare	statistics
geometry viewer	ip extrema	write field
graph viewer	ip register	write image
image to cgm	ip statistics	
image to postscript	ip write vff	

UCD Module Library

NAME

UCD Module Library – modules suited to UCD and finite element analysis networks

DESCRIPTION

The UCD module library is a subset of the supported AVS modules that are suited to UCD and finite element analysis applications.

This man page lists the modules in two ways: alphabetically, and classified by their type (Data Input, Filters, Mappers, Data Output). See the individual module man pages for specific information on each module.

ALPHABETIC LIST

animated float	read ucd	ucd isolines
animated integer	samplers	ucd legend
blend colormaps	scatter to ucd	ucd math
character string	set view	ucd minmax
clip geom	tube	ucd offset
create geom	ucd anno	ucd plot
data dictionary	ucd cell to node	ucd print
field to ucd	ucd cell color	ucd probe
file browser	ucd contour	ucd reverse cell
file descriptor	ucd crop	ucd rslice
flip normal	ucd curl	ucd rubber sheet
float	ucd div	ucd slice 2D
generate axes	ucd extract	ucd streamline
generate colormap	ucd extract scalars	ucd threshold
generate grid	ucd extract vector	ucd to geom
geometry viewer	ucd grad	ucd tracer
graph viewer	ucd hex to tet	ucd vecmag
integer	ucd hog	ucd vol integral
oneshot	ucd iso	write ucd
read field		

DATA INPUT MODULES

animated float	file browser	integer
animated integer	file descriptor	oneshot
character string	float	read field
clip geom	generate axes	read ucd
create geom	generate colormap	samplers
data dictionary	generate grid	set view

FILTERS

blend colormaps	ucd curl	ucd math
field to ucd	ucd div	ucd minmax
flip normal	ucd extract	ucd offset
scatter to ucd	ucd extract scalars	ucd reverse cell
tube	ucd extract vector	ucd threshold
ucd cell to node	ucd grad	ucd vecmag
ucd crop	ucd hex to tet	

MAPPERS

ucd anno	ucd isolines	ucd rubber sheet
ucd cell color	ucd legend	ucd slice 2D
ucd contour	ucd plot	ucd streamline
ucd hog	ucd probe	ucd to geom
ucd iso	ucd rslice	ucd tracer

DATA OUTPUT

geometry viewer	ucd print	write ucd
graph viewer	ucd vol integral	

Unsupported Module Library

NAME

Unsupported Library – unsupported AVS modules

DESCRIPTION

The Unsupported module library contains modules distributed with AVS, but which are unsupported. They may be unsupported for a variety of reasons. Often, the modules are obsolete and are being staged to unsupported before being removed from AVS altogether.

This man page lists the modules in two ways: alphabetically, and classified by their type (Data Input, Filters, Mappers, Data Output). See the individual module man pages for specific information on each module.

ALPHABETIC LIST

alpha blend
cfd values
colormap manager
display pixmap
dot surface
image manager
image to pixmap
luminence
output postscript
pdb to geom
pixmap to image
read plot3D
render geometry
render manager
transform pixmap
volume manager

DATA INPUT MODULES

colormap manager
image manager
pdb to geom
read plot3d
volume manager

FILTERS

cfd values
dot surface
luminence

MAPPERS

image to pixmap
pixmap to image

DATA OUTPUT

alpha blend
display pixmap
output postscript
render geometry
render manager
transform pixmap

NAME

Volume Module Library – modules suited to volume visualization networks

DESCRIPTION

The Volume module library is a subset of the supported AVS modules that are suited to volume visualization applications.

This man page lists the modules in two ways: alphabetically, and classified by their type (Data Input, Filters, Mappers, Data Output). See the individual module man pages for specific information on each module.

ALPHABETIC LIST

3D bar chart
animated float
animated integer
arbitrary slicer
average down
boolean
brick
bubbleviz
character string
clamp
clip geom
color legend
color range
colorize geom
colorizer
combine scalars
compare field
compute gradient
compute shade
contour to geom
contrast
crop
cube
data dictionary
display image
display tracker
downsize
edit substances
euler transformation
excavate
excavate brick
extract graph
extract scalar
extract vector
field legend
field math
field to byte
field to double
field to float
field to int
field to mesh
field to short
file browser
file descriptor
flip normal
float
generate axes
generate colormap
generate grid
generate histogram
geometry viewer
gradient shade
graph viewer
histogram stretch
image to CGM
image to postscript
image viewer
integer
interpolate
isosurface
label
minmax
mirror
oneshot
orthogonal slicer
print field
probe
read field
read volume
scatter dots
set view
statistics
threshold
thresholded slicer
time sampler
tracer
track ball
transpose
volume bounds
volume render
wireframe
write field
write image
write volume
x-ray

DATA INPUT MODULES

animated float
animated integer
boolean
character string
clip geom
color range
data dictionary
edit substances
euler transformation
file browser
file descriptor
float
generate axes
generate colormap
generate grid
integer
label
oneshot
read field
read volume
set view
track ball

Volume Module Library

FILTERS

average down
clamp
colorize geom
colorizer
combine scalars
compute gradient
compute shade
contrast
crop
downsize
excavate
extract graph
extract scalar
extract vector
field math
field to byte, double, float, integer, short
flip normal
generate histogram
gradient shade
histogram stretch
interpolate
minmax
mirror
threshold
time sampler
transpose
wireframe
x-ray

MAPPERS

3D bar chart
arbitrary slicer
brick
bubbleviz
color legend
contour to geom
cube
excavate brick
field legend
field to mesh
isosurface
orthogonal slicer
probe
scatter dots
thresholded slicer
tracer
volume bounds
volume render

DATA OUTPUT

compare field
display image
display tracker
geometry viewer
graph viewer
image to cgm
image to postscript
image viewer
print field
statistics
write field
write image
write volume

NAME

AVS module groups – Types of AVS modules

DESCRIPTION

The AVS modules can be grouped according to the type of data they operate on, and the operations they perform on that data. This can be helpful, for instance, when you need to find out which modules take fields and convert them to geometries, or which modules save data to disk. The following is a possible division of AVS modules by data type and function.

MODULE GROUPS

READING DATA

read field	read image	read ucd
read geom	read volume	pdb to geom
read plot3D	file descriptor	data dictionary
ip read vff	time sampler	

DISPLAYING DATA

display image	display pixmap	image viewer
geometry viewer	graph viewer	display tracker
print field	compare field	

SAVING/PRINTING DATA

image to postscript	output postscript	write field
write image	write volume	write ucd
print field	ucd print	image to cgm
ip write vff		

COLORING DATA

colorizer	colorize geom	color range
generate colormap	field legend	ucd contour
ucd legend		

GENERATING VALUES TO PARAMETER PORTS

animated float	animated integer	boolean
character string	generate colormap	integer
float	file browser	float
oneshot	tristate	generate filters
samplers	field legend	euler transformation
ucd legend	minmax	ucd minmax
dialog box		

FIELD CONVERSION

field to byte	field to double	field to float
field to int	extract scalar	combine scalars
extract vector	field to mesh	field to ucd
field to short		

FIELD PROCESSING AND FILTERING

clamp	crop	downsize
threshold	histogram stretch	interpolate
mirror	offset	transpose
extract scalar	extract vector	combine scalars
cfd values	excavate	blend colormaps
minmax		

CONVERTING FIELDS TO GEOMETRIES

clip geom	bubbleviz	excavate brick
field to mesh	isosurface	contour to geom
hedgehog	probe	stream lines
volume bounds	thresholded slicer	arbitrary slicer
scatter dots	brick	particle advector
volume render	3D bar chart	

CONVERTING FIELDS/UCD TO GRAPHS

orthogonal slicer	generate histogram	
extract graph	ip read line	ucd plot
3D bar chart		

VECTOR PROCESSING

hedgehog	particle advector	stream lines
extract scalar	combine scalars	extract vector
compute gradient	vector div	vector grad
vector mag	vector norm	vector curl
samplers	compute shade	ribbons

CONVERTING VOLUMES TO IMAGES

tracer	orthogonal slicer	display tracker
cube	x-ray	edit substances
euler transformation		track ball

CONVERTING FIELD TO UCD

field to ucd
scatter to ucd

UCD UTILITIES

ucd anno	ucd extract	ucd hex to tet
ucd cell to node	ucd extract scalars	ucd extract vector
ucd contour	ucd legend	write ucd
ucd vecmag	ucd print	ucd rslice
ucd rubber sheet	ucd curl	ucd div
ucd grad	ucd math	ucd cell color
ucd minmax	ucd reverse cell	ucd vol integral

UCD MAPPING

ucd crop	ucd hog	ucd isosurface
ucd isolines	ucd offset	ucd probe
ucd slice2D	ucd streamlines	ucd threshold
ucd tracer		

CONVERTING UCD STRUCTURES TO GEOMETRIES

ucd to geom

CONVERTING GEOMETRIES TO IMAGES

geometry viewer

CONVERTING PIXMAPS AND IMAGES

pixmap to image	image to pixmap	transform pixmap

IMAGE PROCESSING—IMAGE ANALYSIS

ip contour	ip dilate	ip erode
ip extrema	ip histogram	ip lincomb
image probe	image measure	ip read line
ip linremap	ip merge	ip morph
ip rescale	ip threshold	ip statistics
ip blend	ip register	ip read mtable
ip read kernel	ip read sel	
contrast	crop	mirror
generate filters	convolve	luminance
background	sobel	interpolate
threshold	clamp	antialias
composite	image compare	local area ops
transpose	replace alpha	

IMAGE PROCESSING—IMAGE ARITHMETIC

ip absolute	ip float math	ip logical
ip compare	ip arithmetic	

IMAGE PROCESSING—DRAWING AND EDITING

ip lookup	draw grid	sketch roi

IMAGE PROCESSING—FILTERING

ip convolve	ip edge	ip median

IMAGE PROCESSING—GEOMETRIC OPERATIONS

ip reflect	ip rotate	ip twarp
ip warp	calc warp coeffs	ip zoom
ip translate	mirror	

IMAGE PROCESSING—TRANSFORMATION

ip fft	ip fft display	ip fft multiply
ip fft pack	ip fft unpack	ip ifft

IMAGE PROCESSING—INPUT/OUTPUT

ip read vff	ip write vff	ip read mtable
ip read kernel	ip read sel	

GEOMETRY UTILITIES

flip normal	offset	shrink
tube	wireframe	set view
generate axes	generate grid	create geom
color legend	dialog box	

PRESENTATION MODULES

color legend	generate axes	3D bar chart
label	image to cgm	image to postscript

NAME

avs – Application Visualization System

SYNOPSIS

avs *option(s)*

DESCRIPTION

The Application Visualization System (AVS) is an interactive tool for scientific visualization. It includes the following subsystems:

- **Image Viewer.** A high-level tool for manipulating and viewing images.
- **Graph Viewer.** A high-level tool for graphing data.
- **Geometry Viewer.** Allows you to compose "scenes" that contain geometrically-defined objects. The objects must have been created by programs or AVS modules that use AVS's GEOM programming library. You can transform the objects themselves (move, rotate, scale); you can change the viewing parameters (e.g. move the eye point, perspective view, etc.); and you can control the way in which the graphical images are rendered (lighting and shading, Z-buffering, etc.).
- **Network Editor.** A visual programming interface for connecting computational modules together into networks that perform visualization functions.

AVS also includes a sample application, the **AVS Data Viewer**. The Data Viewer provides a simplified, pulldown menu interface for building visualization networks. It is a useful tool for the novice user learning basic scientific visualization techniques.

STARTING AVS

AVS may be located anywhere on your system. To find AVS, you should:

1. Add the AVS binary directory to your default path. For example, if AVS were located in */users/me/avs*, then *csh* users would add a line like the following to one of their startup files, usually *.cshrc* or *.login*:

   ```
   set path=($path /users/me/avs/bin)
   ```

   while *sh* or *ksh* users would add a line like the following to their startup file, usually *.profile*:

   ```
   PATH=$PATH:/users/me/avs/bin
   ```

2. Define a **Path** for AVS by one of the following means. **Path defaults to** */usr/avs* **until you define it otherwise.** The examples are listed in their order or precedence. In these examples, AVS is located in */users/me/avs*:

   - Start AVS with the **-path** option:

     ```
     avs -path /users/me/avs
     ```

   - Have the following line in your personal *.avsrc* file:

     ```
     Path  /users/me/avs
     ```

   - Define the environment variable **AVS_PATH**:

     ```
     csh:         setenv AVS_PATH /users/me/avs
     sh or ksh:   AVS_PATH=/users/me/avs; export AVS_PATH
     ```

     You should define **AVS_PATH** in any event in one of your startup files.

Use the *avs* command to start AVS when your terminal or workstation is directly-connected to the system that will run AVS.

```
avs
```

When running AVS as a remote X client on a different hardware platform that does not support remote hardware rendering (few do) or when you are displaying on an "X terminal" you should use the *avs* command together with the **-nohw** option or **NoHW 1** startup file keyword. For example:

```
avs -nohw
```

AVS runs as an X Window System client, and thus requires that the DISPLAY environment variable be set correctly. These are usually the only options necessary to start an AVS session. However, see the AVS release notes for your platform for additional platform-specific information on which options, such as **VisualType**, may be required to start AVS correctly on your workstation.

CONTROLLING AVS STARTUP

Three entities can affect how AVS starts. They are listed in their order of precedence:

1. Command line options.
2. The *.avsrc* startup file. The startup file contains keyword-value pairs. AVS always reads the system default startup file in *$AVS_PATH/runtime/avsrc* first. Users may override or supplement these system default options with a personal *.avsrc* file. AVS will look for a personal startup file in *./.avsrc* (in the current directory), then *$HOME/.avsrc* (in your HOME directory). It uses the first *.avsrc* that it finds.
3. Environment variables.

OPTIONS

All optional keywords begin with a hyphen (e.g. **–data**). In many cases, the keyword is followed by an additional word (e.g. a directory name). You must separate the keyword and the additional word with whitespace (SPACE and / or TAB characters).

All options keywords can be abbreviated, as long as there is no ambiguity. For example, **–data** can be abbreviated to **–da**. But you cannot abbreviate it to **–d**, since this might indicate either **–data** or **–display**.

In many cases, you can use an entry in the AVS startup file (*.avsrc*) as an alternative to a command line option. For example, a **DataDirectory** entry in the startup file is equivalent to a **–data** option. See the next section for details on the startup file.

–class *string*

(startup file equivalent: none) This is the command line option equivalent of the DISPLAYCLASS environment variable. You can use it to make AVS behave in different ways when it is started from different types of display hardware. **-class** has two effects:

1. An *Xdefaults* file specifies the "look" of the AVS interface; what shades of grey are used for command buttons, what fonts to use, whether the background is "stippled" or a flat color, etc. When **-class** *string* is given, AVS does not use the default *$AVS_PATH/runtime/avs.Xdefaults* file. Instead, it looks for an Xdefaults.*string* file in the *$AVS_PATH/avs/runtime* directory and uses it. At present, the only alternate X defaults file supplied is *Xdefaults.X*.
2. If such a file is present, it will use an alternate startup file, *$AVS_PATH/runtime/avsrc.string*. Otherwise, it uses *$AVS_PATH/runtime/avsrc*. It will also look for a .avsrc.*string* file in the current, then HOME directory and use it instead of your usual *.avsrc* file.

   -class is used when running AVS from an "X terminal." See the full discussion in the "AVS on Color X Servers" appendix to the *AVS User's Guide*.

–cli (startup file equivalent: none) Run AVS with the Command Language Interpreter functioning in the terminal emulator window from which AVS was invoked. This takes an optional argument, which is a CLI command string, to be executed after AVS starts up. See the chapter on the "Command Language Interpreter" in the *AVS Developer's Guide* for details.

-compile_library *source_filespec compiled_filespec*

(startup file equivalent: none) This is a utility for maintaining module libraries whose component modules are changing. It follows a "source module library" vs "compiled module library" paradigm. Specifically, **-compile_library** takes the *source_filespec* to be an AVS module library file containing a list of **file** commands followed by the name of a module binary file. It executes each module listed in order to extract the module description information. From this, it generates *compiled_filespec* as an AVS module library file containing the description information necessary to load the module into the Network Editor's Palette quickly without actually executing the module binary. This option does not start a full AVS session.

See the "Constructing a Module Library" discussion in the "Advanced Network Editor" chapter of the *AVS User's Guide* for more information.

–data *directory*

(startup file equivalent: **DataDirectory**) Specifies the directory in which all subsystem data input file browsers, including the Image Viewer, the Graph Viewer, the Geometry Viewer, and the data input modules in the Network Editor, will initially look for data files (files used an input to computational modules). This is the major tool for redirecting AVS's default data input focus off the sample data files provided in *$AVS_PATH/data* and onto your own data files.

The default data directory is *$AVS_PATH/data*. If an AVS Path is not defined, it defaults to */usr/avs*.

–dials *devicefilespec*

(startup file equivalent: **DialDevice**) Specifies the serial communications port to which a dialbox device is attached (e.g. */dev/tty2*). If **-dials** is present, AVS automatically connects the dialbox dials to the Geometry Viewer's rotation, translation, and scaling transformations. You must know which serial communications port your dialbox is connected to. This argument also corresponds to the environment variable DIALS. Dialboxes are not supported on all platforms.

–display *display-name*

(startup file equivalent: none) Specifies the X Window System display on which AVS is to display. This overrides the current setting of the DISPLAY environment variable.

–gamma *number*

(startup file equivalent: **Gamma**) Controls the brightness of the display for all AVS windows except Geometry Viewer output windows produced with a hardware renderer. The default varies from platform to platform. Values between 1.7 to 2.2 are good starting points for experimentation. Higher real values produce a lighter display.

–geometry *[geom-option(s)]*

(startup file equivalent: none) Automatically invokes the Geometry Viewer subsystem at startup. There will be no **Data Viewers** button to access other subsystems. If you use this option, it must be the *last* option on the command line, followed only by the options listed below that are specific to this subsystem. All other options that follow *-geometry* will be ignored.

–scene *scene-file.scene* or *geomcli-file.scr*

(startup file equivalent: none) This option executes the Geometry Viewer's **Read Scene** function, using the file *scene-file.scene* or *geomcli-file.scr*, depending upon the setting of the **AVS_GEOM_WRITE_V30** environment variable.

–filter *pathname*

Specifies *pathname* as the directory to search for geometry conversion utilities, named *..._to_geom*. See the "Importing Data Into AVS" chapter of the *User's Guide*.

The default directory for these programs is *$AVS_PATH/bin*.

–defaults *filename*

Specifies a Geometry Viewer defaults file. The format of this file is described in the "Geometry Viewer Script Language" appendix.

–geometry *Xgeometry*

Specifies an X Window System geometry (e.g. **500x500-5-5**) for the initial window created by the Geometry Viewer.

-noroll Turns off track rolling. Track rolling occurs when you perform a transformation and release the mouse button while the mouse is still moving. This "flings" the transformable, causing it to continue in motion.

–usage Displays a list of Geometry Viewer startup options.

–graph Automatically invokes the AVS Graph Viewer at system startup. There will be no **Data Viewers** button to access other subsystems.

–image Automatically invokes the AVS Image Viewer at system startup. There will be no **Data Viewers** button to access other subsystems.

–library *filespec*

(startup file equivalent: **ModuleLibraries**) Specifies which AVS module library file to load into the Network Editor at system startup. Module library files are ASCII files describing sets of modules. *$AVS_PATH/avs_library/Supported* is an example. This is the major tool that allows you to load your own sets of modules—either modules you've written yourself or subsets of the supplied modules that you have customized to your needs—instead of always relying on the system default module libraries specified in the *$AVS_PATH/runtime/avsrc* file.

To load more than one module library, use multiple **-library** *filespec* option pairs.

It is equivalent to using the Network Editor's **Read Module Library** function.

–modules *directory* or *filename*

(startup file equivalent: none) Specifies the directory or file in which the AVS Network Editor subsystem initially will look for executable modules. All executable files in a directory are examined to determine whether they contain one or more modules.

-modules differs from **-library** above in that it loads *binary* module files, not ASCII module *library* files. It is slower to load modules as binary files rather than libraries.

You can use more than one **–modules** options to specify multiple individual module binaries, or to have AVS search through multiple directories for modules. This is the main tool for loading individual modules (perhaps modules that you are debugging) that you have not yet formalized into a module library. It is equivalent to the Network Editor's **Read Module(s)** function. It cannot be used to read remote modules.

The default modules directory is *$AVS_PATH/avs_library*. If an AVS Path is not defined, it defaults to */usr/avs*.

–name *string*

(startup file equivalent: **Name**) Causes the specified name to appear in window manager window title bars instead of "AVS". Names containing blanks or special characters should be enclosed in double quotes ("").

Widget windows under control of the Layout Editor will be named with the specified string followed by their corresponding module's designation (for example, **-name MyAVS** causes boolean parameter widget windows to appear as "MyAVS boolean.user.0"). If these names are too long, you can force truncation back to the simple string by appending the ! character to the string (for example, **-name** "**MyAVS!**"). Note that a ! requires surrounding double quotes.

–netdir *directory*

(startup file equivalent: **NetworkDirectory**) Specifies the directory in which the AVS Network Editor subsystem initially will look for network files (**Read Network** and **Write Network** functions). This is the tool to use to redirect AVS's default network focus away from the samples provided in *$AVS_PATH/networks* and onto your own network files.

The default network directory is *$AVS_PATH/networks*.

–network *network-file*

(startup file equivalent: none) Starts AVS and brings up the Network Editor's module control panel with the controls for the network displayed. The full Network Editor subsystem is not displayed or accessible. This is one way to make an individual production network available to a user.

-nodmc (startup file equivalent: **DirectModuleCommunication 0**) Turns off the default direct module-to-module communication. This is useful if you want to perform timing tests to compare network execution speed with / without direct module-to-module communication.

-nohw (startup file equivalent: **NoHW 1**) Tells the AVS Geometry Viewer to not initialize any hardware renderers. Without a hardware renderer, the AVS Geometry Viewer will use a software renderer to create its 3D scenes instead of the platform's native graphics facilities.

-nohw must be used when you are running AVS as a remote X client on a different hardware platform that does not support remote hardware rendering (few do) or when you are using an "X terminal." The software renderer creates an X image rendering of the 3D scene and ships only the image to the local X server for display rather than a stream rendering commands that may not be understood by the local system.

-nomenu (startup file equivalent: **NoMenu**) Prevents the main AVS control panel from appearing. This is intended to be used by application developers who need to hide the fact that AVS underlies the application. Their application would issue it as part of the command it uses to start AVS.

-parallel *n* (startup file equivalent: none) Sets the maximum number of module processes that will attempt to execute in parallel at any one time. The default is 1 (no parallelization.) You should set this figure intelligently for the system(s) that you are running on. If two processors are available (a two-processor system, or a local and a remote system) then this figure can reasonably be set to 2. If you give a value that exceeds the number of processors available, the underlying operating systems will serialize the processes. There is no inherent upper limit to the *n* parameter.

Modules must be in separate processes to execute in parallel. Most modules supplied with AVS are combined into a single executable that runs as a single process. Thus, they will not run in parallel unless they are divided into separate processes. This may be done wholesale with the **-separate** option, or precisely using the Network Editor's module group editing facility. See the discussion on parallel module execution in the "Advanced Network Editor" chapter of the AVS *User's Guide* for more information.

–path *directory*

(startup file equivalent: **Path**) Specifies the directory tree in which AVS itself is installed.

In the absence of this command line option, or a **Path** specification in your personal *.avsrc* keyword file, or the **AVS_PATH** environment variable being defined, path defaults to */usr/avs*.

If you specify another path, then the default data directory and network directory are modified accordingly. For example:

If:	path	= */usr/local/avs*
Then:	data directory	= */usr/local/avs/data*
	network directory	= */usr/local/avs/networks*

This option is also useful to switch between multiple versions of AVS (for example, a test release and a production release).

–reindex (startup file equivalent: none) This option creates AVS help system *.topics* files. It does not start an AVS session. It is useful if you are creating help files for applications that you want to be accessible through the AVS help system. See the appendix on creating help files in the *AVS Developer's Guide* for more information.

-renderer *"string"*

(startup file equivalent: **Renderer**) Specifies which renderer will be the default selected in the Geometry Viewer when a camera window is first created. *"string"* is the literal name found on the renderer buttons under the Geometry Viewer's **Cameras** menu, usually either "Software Renderer" or "Hardware Renderer", though other strings are possible. It must match exactly, in spelling, case, and spacing. The double quote marks must be present. Where there is a hardware renderer available, **-renderer** defaults to "Hardware Renderer". If the user specified **-nohw**, then only one renderer is available, the software renderer, and this option is ignored.

–separate (startup file equivalent: none) This option disables AVS's multiple modules in one process feature. It forces each module to execute as a separate process, whether or not it is combined in an executable with other modules. The option is primarily useful for debugging, or when parallel module execution is desired. (In this last case, it is better to not use **-separate**, since it usually increases memory utilization. Instead, individually divide modules into different executables using the Network Editor's module process group editing facility.) See the section on "Multiple Modules in a Single Process" in the *AVS Developer's Guide*.

–server (startup file equivalent: none) This option opens a connection that an external process can use to connect to AVS and exchange with it a stream of Command Language Interpreter (CLI) commands and their output. See the chapter on the CLI in the *AVS User's Guide* for details.

–shm/noshm

(startup file equivalent: **SharedMemory** *on/off*) This turns the AVS shared memory option on and off. When shared memory is on, AVS keeps only one copy of AVS field and UCD data that all modules in a network share. (GEOM-format data and pixmaps do not use shared memory.) This improves performance by saving memory and processor time. **-noshm** can disable shared memory if, for example, AVS's use of the finite shared memory area is interfering with other applications. On most systems shared memory is on by default.

–size *XDIM*x*YDIM*

(startup file equivalent: **ScreenSize**) Specifies size, in pixels, to use for AVS's virtual display screen size. AVS will automatically resize its interface to fit into the virtual screen. You could use this to confine AVS to run within one section of your screen instead of across the whole screen.

–spaceball *devicefilespec*

(startup file equivalent: **SpaceballDevice**) Specifies the serial communications port to which a Spaceball device is attached (e.g. */dev/tty2*). If **-spaceball** is present, AVS automatically connects the Spaceball device to the Geometry Viewer's rotation, translation, and scaling transformations. You must know which serial communications port your spaceball is connected to. This entry also corresponds to the environment variable SPACEBALL. Spaceballs may not be supported on all platforms.

–timer (startup file equivalent: none) Writes Geometry Viewer performance data to *stderr*. This should be used in conjunction with the **Object Info** panel to display the number of polygons being rendered. To get the measurement, use track rolling to set the object in continuous motion (middle mouse button to rotate, release mouse button while mouse is still moving, thereby "flinging" the object into continuous motion). Wait several seconds (the longer, the more accurate), then press any mouse button in the window to stop the object. Minimize mouse movements while the measurement is being taken. The measurement looks like:

73 frames in 6.632989 seconds for 11.005596 FPS

FPS stands for "frames per second." By convention, the "standard unit" is *$AVS_PATH/data/geometry/teapot.geom*, in the default-sized window, with no additional rendering options (color, shading, etc.). In this case, FPS can be referred to as TPS ("teapots per second").

–version Displays the AVS version number. (Does not start an AVS session.)

–usage Displays a usage message for AVS. No AVS session is started.

AVS STARTUP FILE

When it begins execution, AVS uses a *startup file*, which specifies such things as where AVS is located, which module libraries to load, the locations of various directories, where to look for Help files, how big to make the AVS interface, etc.

AVS always first reads the system default startup file in *$AVS_PATH/runtime/avsrc*. If an AVS Path is not defined on the command line, in your personal *.avsrc* file, or by means of the **AVS_PATH** environment variable, it defaults to */usr/avs/runtime/avsrc*.

Users may override or supplement the options in the system startup file with a personal *.avsrc* file. AVS looks for user *.avsrc* files in the order listed, using the first that it finds:

```
./.avsrc              (current directory)
$HOME/.avsrc          (home directory)
```

You can copy the system default *$AVS_PATH/runtime/avsrc* file to your HOME or other directory, modify it according to your needs and preferences, and rename it with the "." prefix.

If you give the **-class X** command option, or set the DISPLAYCLASS X environment variable, AVS will use a different startup file: *$AVS_PATH/runtime/avsrc.X*. In the same manner as the regular startup file, AVS will look for personal *.avsrc.X* file in the current directory, then your HOME directory. This file is used to customize AVS when you are running it from an "X terminal."

.avsrc Startup File Format

Each line of the AVS startup file consists of keyword-value pair, with whitespace separating the keyword and the value. For example:

```
Path                 /users/me/avs
ModuleLibraries      $Path/avs_library/Supported \
                     /usr/johnp/avs/modules/MyModlib
NetworkWindow        867x567+407+2
NetworkDirectory     /usr/johnp/avs/nets
DataDirectory        /usr/johnp/avs/data
DialDevice           /dev/tty02
```

Use the \ character to continue specifications across line boundaries.

Often, the keyword corresponds to one of the command line options described in the preceding section. If you use a command line option, it overrides the specification, if any, in the startup file.

Startup File Keywords

The AVS startup file keywords are listed below.

NOTE: Where startup file keywords have command line equivalents, see the command line description above for the most complete discussion of the feature.

Applications *filespec*

(command line equivalent: none) Causes AVS to use a file other than *$AVS_PATH/runtime/AVS.applns* to build the large Applications menu. This is how a user would create their own set of application networks and have them accessible from AVS's Applications menu without modifying the central system file. If a simple filename is given rather than an absolute file and pathname, AVS will look for the file in the directory defined by Path on the command line, in the *.avsrc* file, or by the **AVS_PATH** environment variable. If no AVS Path has been defined, Path defaults to */usr/avs*.

BoundingBox *switch*

(command line equivalent: none) If **BoundingBox on** is set, then the AVS Image Viewer and Geometry Viewer will come up with their **Bounding Box** control already turned on. A "bounding box" is a less compute-intensive style of moving geometric objects and Image Viewer subimages. Instead of moving the object "real time," it only moves a wirebox representation of the object. Only when you release the mouse button is the object/subimage rendered at its new location. **BoundingBox** is most useful when you are using AVS on lower performance graphics systems, with the software renderer, or from an "X terminal." **Bounding Box** is usually off by default.

Colors *r g b gray*

(command line equivalent: none) This option controls how many cells of a *system* colormap AVS will attempt to allocate to itself when it starts. *r g b g* represent numbers for red, green, blue, and gray. This is primarily intended for people who are using AVS from an "X terminal" or PseudoColor workstation that objects to the number of colormap cells that AVS tries to allocate for itself. See the discussion on "AVS on Color X Servers" in the AVS *User's Guide*.

DataDirectory *directory*

(command line equivalent: **–data**) Specifies the directory in which the various AVS data input file browsers used in the subsystems (Image Viewer, Graph Viewer, and Geometry Viewer) and Network Editor modules ‘‘read data’’ modules (**read field**, **read geometry**, etc.) initially will look for data files. This is the main tool to refocus AVS's data input attention off the sample data files in *$AVS_PATH/data* and onto your own data files. If no AVS Path has been defined on the command line, in the *.avsrc* file, or by the **AVS_PATH** environment variable, Path defaults to */usr/avs*.

DialDevice *devicefilespec*

(command line equivalent: **-dials**) Specifies *devicefilespec* as the serial communications port to which a dialbox device is attached (e.g. */dev/tty1*). If **DialDevice** is specified, AVS automatically connects the dialbox dials to the Geometry Viewer's rotate, translate, and scale transformations.

This entry also corresponds to the environment variable DIALS. Dialboxes may not be supported on all platforms.

DirectModuleCommunication *switch*

(command line equivalent: **-nodmc**) Turns direct module-to-module communication on and off. This is useful if you want to perform timing tests to compare network execution speed with/without direct module-to-module communication. Direct module-to-module communication is on by default.

DisplayPixmapWindow *Xgeometry*
(command line equivalent: none) Controls the default X Window System geometry of the **display pixmap** module's window.

Gamma *number*
(command line equivalent: **-gamma**) Controls the brightness of the display for all AVS windows except Geometry Viewer output windows produced with a hardware renderer. The default varies from platform to platform. Values between 1.7 to 2.2 are good starting points for experimentation. Higher real values produce a lighter display.

GridSize *n* (command line equivalent: none) Controls the size in pixels of the Layout Editor's alignment squares when **Snap to Grid** is switched on. The default is 10.

HelpPath *directory ...*
(command line equivalent: none) Expands the list of directories that AVS will search to find a module's documentation when you click **Show Module Documentation** in the module's Module Editor window. This is useful when you are using modules other than the set provided with AVS. For the format of the "Help" path, see Appendix D of the *AVS Developer's Guide*, concerning "On-Line Help".

Hosts *fullfilespec*
(command line equivalent: none) Gives the name of a "Hosts" file that lists machines, access methods, and directories of remote modules. It provides a personal override to the system default *$AVS_PATH/runtime/hosts* file when you click on the Network Editor's **Read Remote Module(s)** button under **Module Tools**. See the "Running Remote Modules" section in the *AVS User's Guide* "Advanced Network Editor" chapter for details.

ImageAutomagnify *switch*
(command line equivalent: none) In AVS 3 and later releases, the display image window will not rescale an image when the window is resized. Turning this option "on" will restore the AVS2 behavior of automatically magnifying the image.

ImageScrollbars *switch*
(command line equivalent: none) If set to the value **off**, suppresses the adding of scrollbars to display windows that are too small for the image they are currently displaying. (You can always see more of the image simply by dragging it with the mouse.)

ModuleLibraries *filespec filespec ...*
(command line equivalent: **-library**) Specifies which libraries of modules will be loaded into the Network Editor's module palette. The *last* module library listed will be the "default" library showing in the module Palette when you enter the Network Editor. The other module libraries listed can be called up by clicking on their iconic representation at the top of the Network Editor's main panel. To continue the list of module libraries to a new line, use the \ *.avsrc* continuation character.

ModulePanelHeight *integer*
(command line equivalent: none) Controls the proportion of the Network Construction window devoted to the module Palette as opposed

to the Workspace.

Name *string*

(command line equivalent: **-name**) Causes the specified name to appear in window manager window title bars instead of "AVS". Names containing blanks or special characters should be enclosed in double quotes ("").

Widget windows under control of the Layout Editor will be named with the specified string follows by their corresponding module's designation (for example, **Name MyAVS** causes boolean parameter widget windows to appear as "MyAVS boolean.user.0"). If these names are too long, you can force truncation back to the simple string by appending the ! character to the string (for example, **Name** "**MyAVS!**"). Note that a ! requires surrounding double quotes.

NetworkDirectory *directory*

(command line equivalent: **–netdir**) Specifies the directory in which the AVS Network Editor subsystem initially will look for network files (**Read Network** and **Write Network** functions).

NetworkWindow *Xgeometry*

(command line equivalent: none) Specifies the X Window system geometry of the Network Construction Window, which includes the Network Editor menu, the Module Palette, and the Workspace in which you construct networks of modules. You may need this if your display is substantially smaller than the usual 1280x1024 pixels.

NoHW *switch*

(command line equivalent: **-nohw**) **NoHW 1** tells the AVS Geometry Viewer to not initialize any hardware renderer. Without a hardware renderer, the AVS Geometry Viewer will use a software renderer to create its 3D scenes instead of the platform's native graphics facilities.

NoHW 1 must be used when you are running AVS as a remote X client on a different hardware platform that does not support remote hardware rendering (few do) or when you are using an "X terminal." The software renderer creates an X image rendering of the 3D scene and ships only the image to the local X server for display rather than a stream of rendering commands that the local display may not understand. The default is **NoHW 0** (do initialize hardware renderers) on systems that support a hardware renderer.

NetWriteAllParams *switch*

(command line equivalent: none) AVS saves only parameters that have been modified out to a network file. Setting this option to **on**, will enable saving all parameters, as was the default in AVS 2. The default is **off**.

NoMenu (command line equivalent: **-nomenu**) Prevents the main AVS control panel from appearing. This is intended to be used by application developers who need to hide the fact that AVS underlies the application.

Path (command line equivalent: **–path**) Specifies the directory tree in which AVS itself is installed. For example, if AVS is installed in */user/me/avs*, you would define **Path** in your *.avsrc* as follows:

```
Path      /users/me/avs
```

Other lines that refer to the same directory can then be abbreviated with the symbol **$Path**, e.g.:

```
ModuleLibraries   $Path/avs_library/Supported
DataDirectory     $Path/data
```

PrintNetwork *command*

(command line equivalent: none) The Network Editor's **Print Network** button normally sends output to your default printer. This lets you specify an alternate print command to execute. The command should be a regular shell command, such as:

```
lpr -Plw2
```

ReadOnlySharedMemory *switch*

(command line equivalent: none) Shared memory is normally "read only." Occasionally, the system developer might wish to keep shared memory turned on, but allow it to be written into. Setting **ReadOnlySharedMemory 0** accomplishes this. The default is **1**.

Renderer "*string*"

(command line equivalent: **-renderer** "*string*") Specifies which renderer will be the default selected in the Geometry Viewer when the first camera window is created. *"string"* is the literal name found on the renderer buttons under the Geometry Viewer's **Cameras** menu, usually either "Software Renderer" or "Hardware Renderer", though other strings are possible. It must match exactly, in spelling, case, and spacing. The double quote marks must be present. Where there is a hardware renderer available, **Renderer** defaults to "Hardware Renderer". If the user specified **NoHW 1**, then only one renderer is available, the software renderer, and this option is ignored.

SaveMessageLog *switch*

(command line equivalent: none) If set to the value **on**, causes the AVS message log to be preserved when the AVS session ends normally. By default, the message log (*/tmp/avs_message.log_**XXX***, where ***XXX*** is the AVS process number) is deleted automatically. The log file is always preserved if AVS exits abnormally (e.g. **Ctrl-C** interrupt, system crash).

ScreenSize *XDIM*x*YDIM*

(command line equivalent: **size**) Specifies the size of AVS's virtual display in pixels, confining AVS to run within this area. AVS scales its interface to fit the virtual screen.

SharedMemory *switch*

(command line equivalent: **shm/noshm**) Specifying **SharedMemory off** turns off AVS's shared memory feature.

SpaceballDevice *devicefilespec*

(command line equivalent: **-spaceball**) Indicates the serial communications port to which a Spaceball device is attached (e.g. */dev/tty1*). If **Spaceball** is specified, AVS automatically connects the Spaceball to the Geometry Viewer's rotate, translate, and scale transformations.

This entry also corresponds to the environment variable SPACEBALL. Spaceballs may not be supported on all platforms.

StackSelector *option*

(command line equivalent: none) People who build very large networks sometimes find that the Network Editor's control panel "overflows," making some of the module buttons difficult to access, because the radio buttons take up too much of the control panel. Setting **StackSelector**

choice_browser displays the module names as a scrolling list similar to the file browsers instead of as the default **radio_buttons**.

VisualType *visualtype*

(command line equivalent: none) This command may be necessary when you are seeing less color rendition than you know your display is capable of.

AVS normally uses the X server's default visual. Occasionally, this is the wrong visual to use. For example, the default may be set to PseudoColor when there actually is a TrueColor visual available. (The standard X Window System command to list which X visuals are available and which is being used as the default is *xdpyinfo*. This command may not be available on all platforms.)

VisualType lets you specify a *visualtype*, either **PseudoColor**, **TrueColor**, or **DirectColor**. AVS will then search the X server's visual list until it finds the first visual with the given visual type and use it.

You can also specify an explicit visual using the string **VisualID** followed by a number *n* that is the decimal equivalent of the X server's hexadecimal visual id for the visual you want to use. For example:

```
VisualType  VisualID  41
```

This option may also be useful to people using AVS from "X terminals."

Note: Poor color rendition may also be caused because your display is using double buffering. It may be using its 24 planes as two double-buffered 12 planes (or 12/6, or 8/4). Turning off double buffering on the Geometry Viewer's **Cameras** submenu will fix this, but you will see the object being drawn.

WindowMgr *mgr*

(command line equivalent: none) This option ensures that the Network Editor's Layout Editor and the X Window System window manager that you are using work correctly together. The default for this parameter is specified in the *$AVS_PATH/runtime/avs.Xdefaults* file. The currently recognized values are: **awm**, **mwm** (Motif-style window managers), **twm**, **uwm**, **olwm**(Open Look), and **dxwm**(Dec XVI).

XWarpPtr *on*

(command line equivalent: none) Causes the mouse cursor to be automatically moved ("warped") into typein panels when they appear. **XWarpPtr** is off by default.

AVS ENVIRONMENT VARIABLES

AVS uses the following environment variables. Only DISPLAY must be set correctly before AVS will work.

AVS_ADAPT_TABLE *switch*

A block table is a data structure that maps field points' I, J, K indicies in an irregular field within a "block" of X, Y, Z world space. Modules such as **arbitrary slicer** and **probe** use the block table to interpolate values at points "on" their sampling surface, determining which need to be mapped as colored polygons.

AVS normally builds a regular, evenly-dimensioned block table. Where data points are fairly uniformly spaced within the field, such a block table provides efficient access to the I, J, K values in each block of the grid—each block has approximately the same number of points. However, where data values are concentrated in some areas of the field, but sparse elsewhere (e.g., the wing surface of the *bluntfin.fld* dataset) search times in the dense blocks become much longer.

An adaptive block table creates the block table as an octree. Where data values are dense, the block grid is divided and subdivided again until each block contains only a short list of I, J, K values to search through, improving performance.

Adaptive block tables are slower to construct, but execute more rapidly in the areas with dense grids. People with irregular datasets where the distribution of data points is uneven should try setting AVS_ADAPT_TABLE 1 to see if it improves the performance of the **arbitrary slicer**, **threshold slicer**, **streamline**, **particle advector**, **hedgehog**, **probe**, and **color geom** modules. AVS_ADAPT_TABLE is 0 (off) by default.

AVS_GEOM_WRITE_V30 *switch*

A 1 value causes the Geometry Viewer's **Save Scene** and **Save Object** functions to save scenes and objects as Geometry Viewer Script Language *.scene* and *.obj* files, as occurred in AVS Release 3.0 and earlier, rather than in a single CLI *.scr* file. It is provided for backward compatibility. It is 0 (off) by default.

AVS_HELP_PATH

Specifies one or more locations in the file system for AVS to use when searching for on-line help files. See Appendix D of the *AVS Developer's Guide* for more on this variable.

AVS_MEM_CHECK *switch*

AVS_MEM_HISTORY *switch*

AVS_MEM_VERBOSE *integer*

These three environment variables are all used by the alternate memory allocation routines invoked with the include file *$AVS_PATH/include/mem_defs.h*. These routines replace the UNIX standard memory allocation utilities such as *malloc* with AVS utilities that perform extensive dynamic memory allocation/deallocation bug checking. See the "Memory Allocation Debugging" section in the "Advanced Topics" chapter of the AVS *Developer's Guide* for more information on these utilities.

AVS_MG_TROFF *switch*

Causes the AVS Module Generator to generate its module man page documentation templates in *troff* format rather than the default pre-formatted text man page using tabs and blanks. This option is 0 (off) by default.

DIALS *devicefilespec*

Indicates the serial communications port to which a dialbox device is attached. Dialboxes may not be supported on all platforms.

DISPLAY *host:server.screen*

(required) Used by the X Window System to indicate the display screen at which you're working.

DISPLAYCLASS *string*

string is used to specify an alternate *$AVS_PATH/runtime/Xdefaults* file, such as the supplied *$AVS_PATH/runtime/Xdefaults.X*. Also causes AVS to use alternate *.avsrc.string* startup files, both the default in the *$AVS_PATH/runtime* directory (no such alternative is supplied with the release), and user *.avsrc* files. Both may be customized to make AVS behave differently on different types of display hardware, such as an X terminal. **-class** is the command line equivalent.

EDITOR The AVS Module Generator will use this common UNIX environment variable's value as the default text editor that it will start when you press the Module Generator's **Edit** function.

SPACEBALL *devicefilespec*

Indicates the serial communications port to which a Spaceball device is attached. Spaceballs may not be supported on all platforms.

NAME

alpha blend – generate 2D image from 3D colored data

SUMMARY

Name	alpha blend				
Availability	requires alpha blending support in hardware				
Unsupported	this module is in the unsupported library				
Type	data output				
Inputs	field 3D 4-vector byte uniform				
Outputs	pixmap				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	X-Rot	float	0.0	*none*	*none*
	Y-Rot	float	0.0	*none*	*none*

DESCRIPTION

The **alpha blend** module generates an image (2D grid of pixels) from a 3D block of voxels. (*Voxels* are the 3D analogue of *pixels*.) The *alpha blending* technique treats the voxel block as a set of 2-dimensional images, stacked on top of one another. For each line of sight, you can see though layers that contain semi-transparent voxels, up to the nearest layer with an opaque voxel.

The voxel color values are blended from back to front, using each voxel's opacity value:

auxiliary	red	green	blue
this field interpreted as voxel's opacity value	these three fields make up voxel's color value		

This produces cloud-like images, with the densities of the clouds controlled by the **Opacity** ramp of the colormap that assigned the color values.

AVAILABILITY

This module requires alpha blend support in hardware. Alpha blend is supported on only a few hardware renderers (see the release note information that accompanies AVS on your platform). The software renderer does not support alpha blend. See the newer, faster **tracer** as an alternative. **alpha blend** will only appear in the unsupported module palette on systems where it is available.

INPUTS

Data Field (required; field 3D 4-vector byte uniform)
The input data must be a 3D block of voxels. That is, the data at each point of the 3D field must be a 4-vector of bytes in the alpha-red-green-blue format used in images.

PARAMETERS

By default, the "front" from which the block is viewed is the direction of the positive Z-axis. You can change the direction by rotating the block about the X-axis and/or Y-axis, using these parameters:

X-Rot A floating point value that simulates rotating the data set around the X-axis (horizontal).

Y-Rot A floating point value that simulates rotating the data set around the Y-axis (vertical).

OUTPUTS

Pixmap The output data is in the form of an AVS pixmap.

EXAMPLE 1

The following network shows how 3D data can be colored using the **colorizer** module, then blended into a 2D image using the **alpha blend** module:

```
GENERATE COLORMAP           READ VOLUME
            |                   |
            |      |------------|
            |      |
         COLORIZER
               |
               |
        ALPHA BLEND
               |
               |
     TRANSFORM PIXMAP
               |
               |
       DISPLAY PIXMAP
```

Note that this network uses the **transform pixmap** module to allow the user to resize the image with the window manager. Otherwise, the generated image will be a fixed size, determined by the size of the original data set. For instance, a 64x64x64 data set would produce a fixed-size 128x128 pixel image. (The extra pixels accommodate rotation of the data, which produces a larger image.)

EXAMPLE 2

Another interesting technique is to apply a light source to the data. In order, to do this, the gradient of the data (which approximates the "surface normal") must be computed. A network for doing this "gradient shading" is:

```
GENERATE COLORMAP        READ VOLUME
         |                    |
         |--------------|     |----------------|
                        |     |                |
                      COLORIZER       COMPUTE GRADIENT
                           |                   |
                           |      |------------|
                           |      |
                      GRADIENT SHADE
                              |
                              |
                       ALPHA BLEND
                              |
                              |
                     TRANSFORM PIXMAP
                              |
                              |
                      DISPLAY PIXMAP
```

LIMITATIONS

Because of the shearing technique used to simulate axis rotations, there are certain X- and Y-axis angles for which the image breaks up and eventually disappears completely. Complete rotations around one axis only (zero rotation around the other axis) always work correctly.

RELATED MODULES

Modules that could provide the **Data Field** input:

colorizer
gradient shade

Modules that could be used in place of **alpha blend**:

tracer
cube
x-ray

Modules that can process **alpha blend** output:

transform pixmap
display pixmap

animated float

NAME

animated float – send a sequence of floating point numbers to a module's parameter port

SUMMARY

Name	animated float
Availability	Imaging, UCD, Volume, FiniteDiff module libraries
Type	data coroutine
Inputs	*none*
Outputs	float

Parameters

Name	*Type*	*Default*	*Min*	*Max*
min value	float typein	0.0	*unbounded*	*unbounded*
max value	float typein	0.0	*unbounded*	*unbounded*
steps	int typein	10	2	*unbounded*
sleep	switch	on		
mode	choice	one time		

DESCRIPTION

The **animated float** module automatically modifies floating point parameters. It is used to create simple animations or to drive user simulation code. You plug **animated float** into another module's floating point parameter port (color-coded dark purple), type in minimum and maximum floating point values, and a number of steps (default 10). When you turn off sleep, **animated float** calculates the delta value ((max-min)/step), starts at the minimum value, and begins to send a continuous sequence of evenly-spaced floating point numbers down the connection to the receiving module. Because **animated float** is a coroutine, the AVS flow executive passes one floating point parameter value down the network at a time until the network has fully executed, then signals **animated float** to send the next floating point parameter value. **animated float** can be set to either "One-time" (e.g., 1 2 3 4 5), "Continuous" (e.g., 1 2 3 4 5 1 2 3 4 5) or "Bounce" (e.g., 1 2 3 4 5 4 3 2 1) when it reaches the maximum value. In the last two cases, **animated float** continues to execute until you again toggle "sleep."

For example, you could connect **animated float** to the **isosurface** module's "level" parameter port. By setting minimum, maximum, and step values, you could watch a series of output pixmaps that show the different isosurfaces for each value.

It is often useful to set the minimum and maximum values relative to the range of your data. The **statistics** module can be used to determine reasonable value for these parameters.

The "frame rate" (speed) of the animation depends upon how compute-intensive the downstream modules are. With a compute-bound module like **tracer**, the animation will be quite slow. With simple modules, it will more closely resemble continuous motion. There is no direct way to regulate the speed at which **animated float** executes.

Before you can connect **animated float** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor appears, click any mouse button on its "Port Visible" switch. A purple parameter port should appear on the module icon. Connect this parameter port to the **animated float** module icon in the usual way.

If you bring up the receiving module's control panel, you can watch the parameter values change.

animated float can be connected to multiple modules.

You can save an animation created with **animated float**. Use the **image viewer** module's Action submenu to save a "flipbook" cycle of images (See Example 1).

PARAMETERS

minimum value
A typein to specify the lowest value in the floating point number sequence. It is typed in as a real number (e.g., 1.25 or -.005). There are no upper or lower bound restrictions. The default is 0.0.

maximum value
A typein to specify the maximum value in the floating point number sequence. It is typed-in as a real number (e.g., 5.5 or .003). If the maximum value is less than the minimum value, the delta calculated will be negative and the animation will run backwards. There are no upper or lower bound restrictions. The default is 0.0.

steps An integer typein specifying how many steps the interval between minimum and maximum should be divided into. It cannot be less than two. The default is 10.

sleep A toggle switch that turns **animated float** on and off. It is off by default. When you turn off the stream of floating point numbers by pressing sleep, some number of additional values may continue to flow through the network before **animated float** actually goes to sleep.

mode A set of choices which determine what **animated float** does when it reaches its maximum value. The default is "One-time".

One-time
With "One-time" on (the default), the values are sent only once (e.g., 1 2 3 4 5), and **animated float** sleeps once the values are sent.

Continuous
When "Continuous" is selected, the values being sent wrap around continuously from highest to lowest (e.g., 1 2 3 4 5 1 2 3 4 5 ...).

Bounce
When "Bounce" is selected, the values count up and then count down again repeatedly (e.g., 1 2 3 4 5 4 3 2 1 ...).

OUTPUTS

Floating Point Number (parameter)
A floating point number intended to be input into a floating point parameter port of another module.

EXAMPLE 1

The following network animates the Offset parameter of the **brick** module. The output is sent to two places: to the usual **geometry viewer** module, and to the **image viewer** module through the **geometry viewer**'s image output port. The animation can be saved using the **image viewer**'s Action submenu.

```
                 READ VOLUME
                     |
ANIMATED FLOAT       |
      |              |
      |----------|   |
                 |   |
                 BRICK
                   |
             GEOMETRY VIEWER
                   |
             IMAGE VIEWER
```

EXAMPLE 2

The following network animates the alpha value (transparency) of a volume that has been gradient-shaded, then rendered with **tracer**. Note that **display tracker** sends an upstream transform to the **tracer** module.

```
GENERATE COLORMAP       READ VOLUME
       |                    |
       |--------------------|
  COLORIZER                 |
       |             COMPUTE GRADIENT
       |                    |
       |---------------|    |
                      GRADIENT SHADE
                            |
  ANIMATED FLOAT            |
             |------------| |
                          | |
              |---------| | |
              |         TRACER
              |            |
              |      DISPLAY TRACKER
              |------------|
```

RELATED MODULES

Modules that can process **animated float** output:

any module with a floating point parameter

SEE ALSO

animated integer, which behaves exactly like **animated float**, but for integer parameters.

The example script ANIMATED FLOAT demonstrates the **animate float** module.

NAME

animated integer – send a sequence of integers to a module's parameter port

SUMMARY

Name	animated integer				
Availability	Imaging, UCD, Volume, FiniteDiff module libraries				
Type	data coroutine				
Inputs	*none*				
Outputs	integer				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	min value	int typein	0	*unbounded*	*unbounded*
	max value	int typein	0	*unbounded*	*unbounded*
	steps	int typein	10	2	*unbounded*
	sleep	switch	on		
	mode	choice	one time		

DESCRIPTION

The **animated integer** module automatically modifies integer parameters. This can be used to create simple animations or to drive user simulation code. You plug **animated integer** into another module's integer parameter port (color-coded light purple), type in minimum and maximum integer values, and a number of steps (default 10). When you turn off sleep, **animated integer** calculates the delta value ((max-min)/step), starts at the minimum value, and begins to send a continuous sequence of evenly-spaced integer numbers down the connection to the receiving module. Because **animated integer** is a coroutine, the AVS flow executive passes one parameter value down the network at a time until the network has fully executed, then signals **animated integer** to send the next integer parameter value. **animated float** can be set to either "one time" (e.g., 1 2 3 4 5), "continuous" (e.g., 1 2 3 4 5 1 2 3 4 5) or "bounce" (e.g., 1 2 3 4 5 4 3 2 1) when it reaches the maximum value. In the last two cases, **animated float** continues to execute until you again toggle "sleep."

For example, you could connect **animate integer** to the **orthogonal slicer** module's "slice plane" parameter port. By setting minimum, maximum, and step values, you could watch a series of output pixmaps that show progressive slices through the volume data. Without interrupting **animated integer**, you could change the axis from among I, J, and K and see the animated slice sections from any axis.

It is often useful to set the minimum and maximum values relative to the range of your data. The **statistics** module can be used to determine reasonable value for these parameters.

The "frame rate" (speed of the animation) depends upon how compute-intensive the downstream modules are. With a compute-bound module like **tracer**, the animation will be quite slow. With simple modules it will more closely resemble continuous motion. There is no direct way to regulate the speed at which **animated integer** executes.

Before you can connect **animated integer** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A light purple parameter port should appear on the module icon. Connect this parameter port to the **animated**

integer module icon in the usual way.

If you bring up the receiving module's control panel, you can watch the parameter values change.

animated integer can be connected to multiple modules.

You can save an animation created with **animated integer**. Use the **image viewer** module's Action submenu to save a "flipbook" cycle of images.

PARAMETERS

minimum value
A typein to specify the lowest value in the integer number sequence. It is typed-in as a whole number (e.g., 25 or -170). This parameter has no upper or lower bounds. The default is 0.

maximum value
A typein to specify the maximum value in the integer number sequence. It is typed-in as a whole number (e.g., -255 or 700). If the maximum value is less than the minimum value, the delta calculated will be negative and the animation will run backwards. This parameter is unbounded. The default is 0.

steps An integer typein specifying how many steps the interval between minimum and maximum should be divided into. If the (max-min)/step delta calculation produces real values, each value is rounded down to the nearest whole integer value. Step cannot be less than two. The default is 10.

sleep A toggle switch that turns **animated integer** on and off. It is off by default. When you turn off the stream of integer numbers by pressing sleep, some number of additional values may continue to flow through the network before **animated integer** actually goes to sleep.

mode A set of choices which determine what **animated float** does when it reaches its maximum value. The default is "one time".

one time
With "one time" on (the default), the values are sent only once (e.g., 1 2 3 4 5), and **animated float** sleeps onbce the values are sent.

continuous
When "continuous" is selected, the values being sent wrap around continuously from highest to lowest (e.g., 1 2 3 4 5 1 2 3 4 5 ...).

bounce
When "bounce" is selected, the values count up and then count down again repeatedly (e.g., 1 2 3 4 5 4 3 2 1 ...).

OUTPUTS

Integer Number (parameter)
An integer number intended to be input into an integer parameter port of another module.

EXAMPLE 1

The following network animates slices through a volume:

```
                      READ VOLUME
                          |
GENERATE COLORMAP         |
             |---------|  |
                       |  |
                     COLORIZER
                          |
ANIMATED INTEGER          |
             |------|     |
                    |     |
                   ORTHOGONAL SLICER
                          |
                          |
                     IMAGE VIEWER
```

RELATED MODULES

Modules that can process **animated integer** output:

any module with an integer parameter

SEE ALSO

animated float, which behaves exactly like **animate integer**, but for floating point parameters.

The example script ANIMATED INTEGER demonstrates the **animate integer** module.

animate lines

NAME

animate lines – animate lines for a vector field

SUMMARY

Name	animate lines
Availability	FiniteDiff module library
Type	filter
Inputs	geometry upstream transform
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Objects	text			
Max Length	text			
Length	integer	2	2	16
Animate	oneshot	off		

DESCRIPTION

animate lines takes a set of streamlines output by the **stream lines** module and animates them. **animate lines** outputs successive segments of the streamlines to produce a dynamic representation of them.

Because **animate lines** is a coroutine, the AVS flow executive passes one set of line segments down the network at a time, until the network has fully executed, then signals **animate lines** to send the next set of line segments.

The "frame rate" (speed of the animation) depends upon how many streamlines are passed as input to **animate lines**. With up to an intermediate number of streamlines the animation appears as continuous motion. There is no direct way to regulate the speed at which **animate lines** executes.

INPUTS

Stream Lines (geometry)

A set of disjoint lines generated by the module **stream lines**.

Upstream Transform (optional, invisible, autoconnect)

When the **animate lines** module coexists with **stream lines**, and **geometry viewer** in a network, **geometry viewer** feeds information on how **stream lines**' point, circle or other "sample probe" has been moved back to this input port on the **animate lines** module. **animate lines** then relays the information up the network to **stream lines**. The modules connect automatically, through data pathways that are normally invisible. This gives direct mouse manipulation control over **stream line**'s sample probe.

PARAMETERS

Objects A text window which displays the number of line segments which make up the input streamlines.

Max Length

A text window which displays the maximum length of the input streamlines.

Length An integer dial which controls the length of the line segments that are animated along the path of the streamlines.

Animate (oneshot)

A oneshot button that initiates the animation of the streamlines.

OUTPUTS

Animated Lines (geometry)

successive portions of the input streamlines are output sequentially.

EXAMPLE

The following network reads in a 3D vector field, and calculates streamlines for the field. **animate lines** is used to dynamically represent the output of **stream lines**.

```
READ FIELD
     |
     |-------------------|
     |                   |
STREAM LINES             |
     |             VOLUME BOUNDS
     |                   |
ANIMATE LINES            |
     |                   |
     |-------------------|
     |
GEOMETRY VIEWER
```

RELATED MODULES

hedgehog, particle advector, stream lines

antialias

NAME

antialias – antialias an image

SUMMARY

Name	antialias
Availability	Imaging module library
Type	filter
Inputs	field 2D uniform 4-vector byte *(image)*
Outputs	field 2D uniform 4-vector byte *(image)*
Parameters	none

DESCRIPTION

The **antialias** module downsamples an image using a Gaussian 3x3 convolution filter. This produces an antialiasing effect, reducing jagged edges. The output image is half the size of the input image in each dimension—a 512x512 image becomes a 256x256 image after antialiasing.

INPUTS

Image (required; field 2D uniform 4-vector byte)

The image to be antialiased.

OUTPUTS

Image (field 2D uniform 4-vector byte)

The output antialiased image. This image is half the size of the input image in each dimension.

EXAMPLE 1

The following network reads an image, antialiases it, and displays it through the **image viewer**.

```
 READ IMAGE
     |
 ANTIALIAS
     |
IMAGE VIEWER
```

RELATED MODULES

Modules that could provide the **Image** input:

colorizer
composite
convolve
field math
localops
read image
replace alpha

Modules that can process **antialias** output:

extract scaler
image viewer
display image

See also **downsize**, **interpolate**, **average down**, **ip convolve**, **sobel**

The script ANTIALIAS demonstrates the **antialias** module.

arbitrary slicer

NAME

arbitrary slicer – map 3D scalar field to 3D mesh

SUMMARY

Name arbitrary slicer

Availability Volume, FiniteDiff module libraries

Type mapper

Inputs field 3D scalar *any-data any-coordinates*
colormap (optional)
upstream transform (optional, invisible, autoconnect)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Values*
X Rotation	float	0.0	0.0	360.0	
Y Rotation	float	0.0	0.0	360.0	
Distance	float	0.0	-2.0	2.0	
Mesh Res	integer	36	8	144	
Sampling Style	radio	point			point, trilinear

DESCRIPTION

The **arbitrary slicer** module extracts a 2D slice from a 3D volume of data. The slice plane can be oriented arbitrarily — it need not be parallel to any of the coordinate axes.

The volume of data is represented as a 3D scalar field (which defines a uniform lattice within the volume). The slice plane is represented as a 2D grid, with a parameter-controlled resolution. The intersection of the volume and the grid is a *mesh* of vertices in 3D space.

Each vertex in the mesh is assigned a color that corresponds to one or more values of the 3D scalar field. Since, in general, the mesh vertices do *not* coincide with the original lattice points, an interpolation method can be used — see the *Sampling Style* input parameter below.

By default, the volume is placed at the origin and the slice plane is the X-Y plane. The orientation of the slice plane is controlled by two mechanisms. First, you can control the position of the slice plane using the floating-point dials, X rotation and Y rotation. Second, you can "pick" the slice plane object by clicking on it with the left mouse button. Once it has been "picked" you can orient the slice plane using the same "virtual trackball" paradigm that is used in the Geometry Viewer. Then **arbitrary slicer** receives an upstream transform from the **geometry viewer** module which tells it how the slice plane has beem moved. Using this information **arbitrary slicer** computes a new mesh output. These two mechanisms can be used together to manipulate the slice plane, in which case the dial transformations are applied first, followed by the upstream transform.

You can control the resolution of the mesh using the **mesh res** parameter. At lower resolutions, fewer original data points are used in the computations; at higher resolutions, more points are used.

Note that by default the mesh is displayed with **No Lighting** selected. To override this feature, select the slice plane object in the Geometry Viewer, and change its type from **No Lighting** to **Gouraud**, **lines**, or **flat**.

The optimal way to use this module is to start off with a low resolution mesh, position it as desired, then increase the resolution and turn on trilinear mapping.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)
The input data must be a 3D field, with any type of scalar data value at each location in the field. The field can be uniform, rectilinear, or curvilinear.

Colormap (optional; colormap)
By default, the value computed for each vertex of the mesh is used as the hue in HSV space. If you specify a colormap, the values are used to index into the colormap.

Upstream Transform (optional, invisible, autoconnect)
When the **arbitrary slicer** module coexists with the **geometry viewer** module in a network, and the slice plane object has been "picked", **geometry viewer** feeds information on how the slice plane has been moved back to this input port on the **arbitrary slicer** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **arbitrary slicer**'s slice plane.

PARAMETERS

X Rotation A floating point dial widget that controls the rotation of the slice surface in the X direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction. This controls the orientation of the slice plane in object space.

Y Rotation A floating point dial widget that controls the rotation of the slice surface in the Y direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction. This controls the orientation of the slice plane in object space.

Distance A floating point value between -2.0 and 2.0 which moves the slice plane back and forth in the direction of the normal to the slice plane. This value is scaled by the largest dimension of the input field. Consequently, you can move the slice plane along the normal from -(2 * **max dimension**) to (2 * **max dimension**).

Mesh Res Controls the resolution of the slice plane mesh. Higher resolution meshes result in higher quality representations, but take longer to compute and render. The default mesh is 8x8.

Sampling Style
(radio buttons) Controls the way in which each vertex of the output mesh is assigned a color:

- If **point**, a nearest-neighbor algorithm is used. Each mesh vertex is assigned the byte value of the nearest point in the lattice.
- If **trilinear**, a trilinear interpolation is performed. The value at each vertex depends on the byte values at the eight lattice points that are the corners of the "enclosing cube".

The trilinear interpolation method is more accurate but takes longer to compute, particularly with larger meshes.

OUTPUTS

Geometry (geometry)

The output is an AVS *geometry*.

EXAMPLE

This example shows a common usage of the **arbitrary slicer** module. The **volume bounds** modules gives a reference frame for orienting the slice plane.

```
                   READ FIELD
                       |
                       |
GENERATE COLORMAP      |------------------|
       |               |                  |
       |----------|    |                  |
                  |    |                  |
                  |    |                  |
               ARBITRARY SLICER     VOLUME BOUNDS
                       |                  |
                       |------------------|
                       |
                GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the input field:

read field
read volume
Any module that outputs a 3D field.

Modules that can replace **arbitrary slicer**:

brick
orthogonal slicer
thresholded slicer

Modules that can process **arbitrary slicer**'s output:

geometry viewer
render geometry
Any module that inputs a geometry

SEE ALSO

The example script PROBE demonstrates the **arbitrary slicer** module.

NAME

average down – downsize a field in X, Y, or Z by averaging

SUMMARY

Name	average down
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field 2D | 3D uniform scalar byte
Outputs	field *same-dims* uniform scalar byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
X	int dial	4	1	16
Y	int dial	4	1	16
Z	int dial	4	1	16

DESCRIPTION

average down reduces the size of a 2D or 3D scalar uniform byte field in any combination of dimensions. To create the reduction, it averages the values of adjacent data points in a "chunk" whose size is given by the **X**, **Y**, and/or **Z** parameters. (Z is only present if the input is 3D.)

For example, if you have a 2D 7x5 field and you want to average down by 3 in the X dimension, and 2 in the Y dimension, the result will be a 2D 3x3 field whose values are based on averages composed of the following chunks of cells:

```
               0      1      2       3     4     5        6    (old i's)
            -------------------  -------------------   -----
         0  | X      X      X |  |  X     X     X  |   | X |      0
         1  | X      X      X |  |  X     X     X  |   | X |
            -------------------  -------------------   -----
    oldj 2  | X      X      X |  |  X     X     X  |   | X |      1  (new j's)
         3  | X      X      X |  |  X     X     X  |   | X |
            -------------------  -------------------   -----
         4  | X      X      X |  |  X     X     X  |   | X |      2
            -------------------  -------------------   -----
                     0                    1               2    (new i's)
```

These smaller fields use less memory and render more quickly. For example, one could use **average down** temporarily while experimenting with other modules' parameter changes until a satisfactory output is achieved, and then remove the **average down** module to produce a full resolution rendering.

average down differs from the similar **downsize** and **interpolate** modules in several ways:

- Where **downsize** simply selects one of the adjacent data values and discards the others, **average down** averages among the adjacent data values.
- **downsize** and **interpolate** downsize all three dimensions uniformly. **average down**'s dial parameters let you select any combination of X, Y, and Z for the reduction. This is useful, for example, in medical datasets where the X and Y dimensions are high resolution images (for example, 256 x 256), while the Z dimension is small (for example, 16 slices). With **average down** you can downsize the high resolution image planes while retaining the same number of slices.
- **average down** is to be preferred over **interpolate** for downsizing data. At .5 reduction, the two are the same. However, at .25 reduction in X and Y (**X** and **Y** dial parameter values set to 4 in **average down**, or a parameter dial setting on

interpolate), **average down** will have averaged 16 data values, while **interpolate** has averaged just the four "corner" data values.

- **average down** is more restrictive on the type of input field it will accept.

INPUTS

Data Field (required; field 2D | 3D uniform scalar byte)
The input field must be a 2D or 3D scalar uniform field containing byte data.

PARAMETERS

X An integer dial that establishes the "chunk" size of points in the X dimension that will be averaged together. The minimum is 1 (no reduction); the maximum is 16, and the default is 4.

Y An integer dial that establishes the "chunk" size of points in the Y dimension that will be averaged together. The minimum is 1 (no reduction); the maximum is 16, and the default is 4.

Z An integer dial that establishes the "chunk" size of points in the Z dimension that will be averaged together. The minimum is 1 (no reduction); the maximum is 16, and the default is 4. This control only appears if the input is 3D.

OUTPUT

Data Field (field *same-dims* uniform scalar byte)
The output field has the same dimensionality as the input field, but the number of elements in the specified dimensions is reduced to 1/dial-value. "Remainder" values, for example— a 10x12 field reduced in X by 4 with 2 remaining values—are averaged together.

The **min_val** and **max_val** (minimum and maximum data values) of the input field, if present, are invalidated in the output field since the operation has likely changed these values. The new coordinate data in the output field is used to define the physical extents as being equal to the original data, but at lower resolution.

EXAMPLE

```
                    READ VOLUME
                         |
                         |
                   AVERAGE DOWN
GENERATE COLORMAP        |
           |             |
           |----------|  |
                    TRACER
                         |
                  DISPLAY TRACKER
```

RELATED MODULES

interpolate
downsize

Modules that could provide the field input:

read field
read volume
extract scalar
any module that outputs a 2D or 3D scalar uniform byte field

Modules that can process the output field:

any module that can process a field

SEE ALSO

The example script AVERAGE DOWN demonstrates the **average down** module.

AVS Animator

NAME

AVS Animator – create keyframe animations of data visualizations

SUMMARY

Name	AVS Animator
Availability	vendor dependent
Type	data input
Inputs	*none*
Outputs	integer (frame number) integer (frames / second) float (current time) field 2D scalar float uniform (parameter path)
Parameters	*various, internal use*

DESCRIPTION

The **AVS Animator** is an interface to create keyframe animations of AVS data visualizations. It is the centerpiece of a set of modules that collectively form the AVS Animation Application. To use the **AVS Animator**, simply move its module icon into the Network Editor Workspace. The module is part of the **Animation** module library. It does not need to be connected to other modules. The compact Animator interface panel will appear.

The **AVS Animator** can be used to automatically generate animations of:

- All object manipulations produced by the Geometry Viewer interface including object, camera, and light transformations, object properties and colors. One can thus animate objects rotating or moving in space, or cameras "flying by" objects or zooming in to examine them closely, or objects changing properties such as dissolving from opaque into transparent.
- Changes produced in a Geometry, Image, Graph Viewer, **display image**, or **display tracker** output window produced by modifying parameters on subroutine modules in an AVS network. One could animate multiple slice planes marching through volumes, or image processing filters acting on an image.

The AVS Animator is a keyframe animator. In typical use, the user sets up an initial scene in a Geometry Viewer window. The contents of the scene window may have been read directly into the Geometry Viewer using **Read Object** or **Read Scene**, or it may have been produced by an AVS network. The user then presses a button that establishes this as a "keyframe." The Animator records the current settings of all Geometry Viewer options and network module parameters. Next, the user introduces some change into the scene window: either using the Geometry Viewer interface to move the object(s) or the camera, or manipulating the parameter controls of the modules in the network that produced the output geometry. Again, pressing a key establishes this as a new keyframe. The Animator records those Geometry Viewer and module parameter settings that have changed since the previous keyframe.

To play back the animation, press one of the playback buttons. The Animator uses the values of the keyframes and frames per second to automatically generate "inbetween" values for all Geometry Viewer and module parameter settings that are being animated, producing a smooth, interpolated animation in the output window.

The user can change keyframe positions, the number of interpolation steps, the type of interpolation used, the direction and manner of playback (keyframes only, forward, backward, circular, bounce), edit individual keyframe values, and gradually build up a full animation by recording and playing back multiple individual animation tracks (just object rotation, then just camera movement, then just module parameter value changes). A *.animrc* file can be used to instruct the Animator to ignore parameter changes from listed modules.

Animations are saved as compact ASCII scripts that contain the instructions for recreating animations. Other modules in the Animation Application can save the animations as actual frames, preprocess the frames for video output, and write the output to video devices.

Because the Animator automatically generates inbetween frames, it differs from existing AVS "flipbook" animation facilities in the Image Viewer, Geometry Viewer, and **display image** and **display pixmap** modules, which require the user to manually create and record all frames that make up an animation sequence. Animations, unlike flipbooks, are easily edited. Animator animation scripts are much more compact to store than flipbook frames.

OUTPUTS

frame number

An integer that contains the current frame number, as reported at the top left of the Animator control panel. This port can be used to generate a synchronization signal for coroutine modules that have a synchronous input port option such as the **particle advector** module.

frames/second

An integer that represents the current playback interpolation rate. The default value is 30 frames/second, which corresponds to NTSC video rates. (PAL is 25 frames/second and film is usually 24 frames/second.) This output can be used for video output modules that need to know the video rate.

current time

A real number that represents the current time in seconds (e.g. 62.25). This value could be fed into a module that generates a time stamp label in the geometry viewer.

parameter path (field 2D scalar real uniform)

A field structure containing keyframe setting information for an individual parameter in the animation.

EXAMPLE 1

This network shows the **AVS Animator** recording the output of a typical visualization network. Note that the Animator is not connected to the main network.

```
                          READ FIELD
                              |
 GENERATE COLORMAP            |---------------|
        |                     |               |
        |              ORTHOGONAL SLICER      |
        |                     |               |
        |--------------|      |               |
                        FIELD TO MESH   VOLUME BOUNDS
                              |               |
                              |---------------|
                              |
                     GEOMETRY VIEWER
                                                    AVS ANIMATOR
```

EXAMPLE 2

This network shows the **AVS Animator** recording the output of the **particle advector** module. The Animator does not normally work with coroutine modules. However, **particle advector** has been modified to include a synchronous execution option port. The **AVS Animator**'s rightmost **frame number** output port acts as a "fire once" signal to **particle advector**'s leftmost input port, causing it to simulate one advection step each time the Animator playback increments the frame number.

```
                        READ FIELD
                            |
                            |-------------------------|
                            |                         |
 AVS ANIMATOR               |                   VOLUME BOUNDS
           |                |                         |
           |-------|        |                         |
                 PARTICLE ADVECTOR                    |
                        |   |                         |
                        |---|-------------------------|
                            |
                    GEOMETRY VIEWER
```

SEE ALSO

write frame seq
read frame seq
output ImageNode
prepare video
output VideoCreate

The **AVS Animator** and its associated modules are fully described in the *Animating AVS Data Visualizations* document.

AVAILABILITY

The **AVS Animator** and its associated modules may be available only under separate license from your AVS vendor. If present, the modules may be kept in a separate module library in the *$AVS_PATH* directory that must be loaded manually, or by including it in a personal *.avsrc* file. See your platform's release notes for specific information.

LIMITATIONS

The **AVS Animator** records changes that are made to the following entities, but does not interpolate between their old and new values. Animations with such changes will suddenly "jump" to the new renditions.

- Object rendering modes (such as wireframe dissolving into gouraud), in the Geometry Viewer.
- Data values, such as fields, UCD structures, and geometries. The AVS Animator records and interpolates changes occuring through the AVS interface that are detectable via CLI commands; it does not interpolate between data values such as might be found in two fields containing data on the same grid, but at different times. Such an animation could be achieved by writing an "interpolate field" module.
- Coroutine modules, such as simulations, that act asychronously with an AVS network. To be animated, coroutine modules would need to be rewritten with a synchronous input port option. The **particle advector** module is so modified.
- Image and Graph Viewer control panel manipulations. (However, the images and graphs apearing in these viewers are animated.)

The **AVS Animator** neither records nor interpolates changes made to a colormap through the **generate colormap** or **ucd contour** modules. The Animator also does not record or interpolate Geometry Viewer label manipulations if the label is a title. If the label is attached to an object, the label moves in conjunction with the object

background

NAME

background – create a shaded backdrop image

SUMMARY

Name	background
Availability	Imaging module library
Type	data
Inputs	field 2D 4-vector byte uniform *(image)* (OPTIONAL)
Outputs	field 2D 4-vector byte uniform*(image)*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Upper Left Hue	Dial float	0.67	0.0	1.0
Upper Right Hue	Dial float	0.67	0.0	1.0
Lower Left Hue	Dial float	0.0	0.0	1.0
Lower Right Hue	Dial float	0.0	0.0	1.0
Upper Left Sat	Slider float	1.0	0.0	1.0
Upper Left Value	Slider float	1.0	0.0	1.0
Upper Right Sat	Slider float	1.0	0.0	1.0
Upper Right Value	Slider float	1.0	0.0	1.0
Lower Left Sat	Slider float	1.0	0.0	1.0
Lower Left Value	Slider float	0.0	0.0	1.0
Lower Right Sat	Slider float	1.0	0.0	1.0
Lower Right Value	Slider float	0.0	0.0	1.0
X Resolution	Typein int	128	0	1024
Y Resolution	Typein int	128	0	1024
Dither	Switch	off		

DESCRIPTION

background generates a linearly-shaded image that is typically used as a background for other renderings. You specify the color of each corner with a separate Hue dial. You then use sliders to specify the saturation and value of the color, again individually for each corner. **background** takes the hue-saturation-value of each corner and evenly blends them toward the center of the image.

The results of **background** can be used with the **replace alpha** and **composite** modules to create the effect of a semi-transparent tinted film overlaid upon a regular image. For example, you could create a grey overcast on the image of a sunny sky. When doing this, connect the image to **background**'s input port—this will create a background image the same size as the input image.

The default output image is a 128x128 pixels, shaded blue-to-black image.

INPUTS

Image (optional; field 2D 4-vector byte uniform)

The input image automatically sets the **X Dimension** and **Y Dimension** of the output image. It has no other effect.

PARAMETERS

Upper Left Hue
Upper Right Hue
Lower Left Hue
Lower Right Hue

Floating point dials to select the hue (color) of each corner. The defaults for the upper left and right are .67 (blue); the defaults for the lower left and right are 0.0.

Note:

0.000 = black   0.320 = green   0.670 = blue   1.000 = red
0.167 = yellow   0.500 = cyan   0.833 = magenta

Upper Left Sat
Upper Left Value
Upper Right Sat
Upper Right Value
Lower Left Sat
Lower Left Value
Lower Right Sat
Lower Right Value

Floating point slider bars to select the saturation (how much "white" is mixed in with the hue (1.0=none) and value (how much "black" is mixed in with the hue (1.0=none). All parameters default to 1.0 (fully saturated with no black) except both lower values. These are set to 0.0, making the default lower part of the image all-black.

X Resolution
Y Resolution

An integer typein specifying the size, in pixels, of the output image. The default is 128x128. These parameters will not be visible if there is an optional input image.

Dither A close examination of the **background** image would reveal contour bands of color as the corners shade off if interpolating over a small range of colors over a large screen distance. **Dither** adds a bit of noise in the lower bits of the color value to smooth out this contouring effect. This is a boolean switch that is off by default.

OUTPUTS

Image (field 2D 4-vector byte uniform)
The shaded output image.

EXAMPLE 1

The following network creates a shaded image and writes the image to disk:

```
  BACKGROUND
      |
      |--------------------|
 IMAGE VIEWER          WRITE IMAGE
```

EXAMPLE 2

The following network takes an image, computes the luminance, uses that to create an alpha mask, renders a shaded background, and composites the rendered image over the shaded background:

```
                           READ IMAGE
                               |
          _____________________|
          |           |        |
      BACKGROUND  LUMINANCE    |
          |           |        |
          |        CONTRAST    |
          |           |------| |
          |                REPLACE ALPHA
          |                    |
          |----------------------| |
                           COMPOSITE
                               |
                          DISPLAY IMAGE
```

EXAMPLE 3

This network takes a geometry, displays it on the screen, then converts the screen pixmap to an image, computes its luminance, uses that to create an alpha mask, renders a shaded background and composites the rendered image over the shaded background.

In the **contrast** module, you typically want contrast_in_minimum and contrast_in_maximum to both equal 1 to get any non-zero pixel to overlay the background.

```
                           READ GEOM
                               |
                        RENDER GEOMETRY
                               |-----------------------|
                        PIXMAP TO IMAGE                |
          _____________________|                DISPLAY PIXMAP
          |           |        |
      BACKGROUND  LUMINANCE    |
          |           |        |
          |         CONTRAST   |
          |           |------| |
          |                REPLACE ALPHA
          |                    |
          |----------------------| |
                           COMPOSITE
                               |
                          DISPLAY IMAGE
```

RELATED MODULES

Modules that could provide the **Image** input:

read image, pixmap to image

Modules that can process **background** output:

any module that takes an Image as an input...
image viewer
composite

SEE ALSO

Two BACKGROUND example scripts demonstrate the **background** module.

blend colormaps

NAME

blend colormaps – interpolate between two colormaps in HSVA space

SUMMARY

Name	blend colormaps				
Availability	UCD module library				
Type	filter				
Inputs	cmap1 (first colormap) cmap2 (second colormap)				
Outputs	colormap				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	scale	float dial	0.00	0.00	1.00

DESCRIPTION

blend colormaps interpolates linearly between two colormaps in HSVA space. This is useful when using the **AVS Animator** module, which does not interpolate between colormaps. The Animator will interpolate the **scale** parameter, which governs the proportionate value of **cmap1** to **cmap2**. It can also be used with **animated float**, etc.

Every value of every band (hue, saturation, value, and opacity (alpha)) is evaluated separately. Generally, it is best to confine the differences between the two input colormaps to one variable, such as transparency, or the results can be non-intuitive. Note that interpolation between hue values (such as 0.00 for red and 0.66 for blue) will produce the intermediate yellow, green, and cyan shades, not a "dissolve" from red to blue. The module assumes colormaps that are 256 entries long.

INPUTS

cmap1 (required; colormap)

The first colormap. This colormap can be created by the **generate colormap** module, which can also save the colormap to a file.

cmap2 (required; colormap)

The second colormap. This colormap can be created by the **generate colormap** module, which can also save the colormap to a file.

PARAMETERS

scale

The dial **scale** controls the blending between the colormaps. When **scale** = 0.0 the output is entirely cmap1. When scale = 1.0, the output is entirely cmap2. In the middle, the output is:

```
out = (cmap1 * (1.0 - scale)) + (cmap2 * scale)
```

OUTPUTS

cmap out (colormap)

The output colormap.

EXAMPLE 1

This network uses the **AVS Animator** to rotate a volume rendering and change the colormap's transparency simultaneously.

One could also use the **animated float** module instead of the **AVS Animator** to animate the colormap blending alone. **animated float**'s output would feed into **blend colormap**'s **scale** parameter. To make the parameter port visible, click on the module's dimple to bring up the Module Editor, then click on the **scale** parameter to bring up the Parameter Editor. Then toggle **Port Visible**.

```
GENERATE COLORMAP       GENERATE COLORMAP      READ FIELD
        |________     __________|                   |
               |    |                               |
            BLEND COLORMAPS                         |
                 |____________      _______________|
                             |      |
EULER TRANSFORMATION         |      |
        |____________________|___   |          AVS ANIMATOR
                             |  |   |
                              TRACER
                                 |
                         DISPLAY TRACKER
```

RELATED MODULES

generate colormap

SEE ALSO

The example script BLEND COLORMAPS demonstrates the **blend colormaps** module.

boolean

NAME

boolean– send a user-entered boolean value to one or more module(s) boolean parameter port(s)

SUMMARY

Name	boolean		
Availability	Imaging, Volume, FiniteDiff module libraries		
Type	data		
Inputs	none		
Outputs	boolean		
Parameters	*Name*	*Type*	*Default*
	Boolean Value	*choice*	*off*

DESCRIPTION

The **boolean** module sends a single user-specified boolean value to one or more boolean-type parameter ports on one or more receiving modules. Its purpose is to make it possible for you to simultaneously control boolean parameter input to more than one module using only a single input widget.

Before you can connect **boolean** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **boolean** module icon in the usual way.

PARAMETERS

Boolean Value (boolean)

The single user-supplied boolean value, either on or off, to be sent to the receiving module(s) boolean parameter port(s). The default value is off.

OUTPUTS

Boolean (boolean)

The boolean value is sent to all modules with boolean-type parameter ports that are connected to the **boolean** module.

EXAMPLE 1

In the following network, the **boolean** module has been connected to **isosurface**'s "Flip Normal" parameter:

```
BOOLEAN           READ VOLUME
    |                   |
    |--------|    |------|
             |    |
          ISOSURFACE
               |
               |
       GEOMETRY VIEWER
```

RELATED MODULES

Modules that can process **boolean** output:

all modules with boolean-type parameter ports

brick

NAME

brick – show uniform volume as a solid

SUMMARY

Name	brick				
Availability	Volume, FiniteDiff module libraries requires 3D texture mapping support				
Type	mapper				
Inputs	field 3D uniform *n-vector any-data* upstream transform (optional, invisible, auto-connect)				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	X Rotation	float dial	0.0	*unbounded*	*unbounded*
	Y Rotation	float dial	0.0	*unbounded*	*unbounded*
	Offset	float dial	0.0	*unbounded*	*unbounded*
	Sides	boolean	on		

DESCRIPTION

The **brick** module is another way of visualizing 3D uniform volume data. The **arbitrary slice** module displays a slice plane through a volume of data. Outside the slice plane, everything is clear "empty air." **brick** displays the volume as a *solid*— you see the six outside surfaces of an otherwise opaque volume (hence the name "brick"). You can use the **X Rotation**, **Y Rotation**, and **Offset** parameters to slice a chunk off the brick to reveal the data inside, as one might lop off part of a fruitcake. If you turn off the **Sides** switch, you will see just the slice plane. The effect is similar to the output of **arbitrary slicer**. Only one of the six surfaces of the volume is a moveable slice plane.

brick creates its picture of the volume data using 3D texture mapping (**arbitrary slicer** uses sampling). In this method, the boundary of the volume has three values, *u, v, w*, associated with each of its vertices. When **brick**'s slice plane intersects this volume, *u, v, w* values are computed for the vertices of the resulting solid. These values are attached to the vertices of the geometry object which **brick** produces, and are used by **geometry viewer** to perform 3D texture mapping.

Texture mapping is much faster than the sampling technique used by arbitrary slicer, particularly for large datasets. The point sampling is always done at the resolution of the data; thus differences in data values within a small area are not obscured as they can be with **arbitrary slicer**.

The 3D texture map is created with a combination of the **generate colormap**, **colorizer**, and possibly **color range** modules. Their output is connected to the **geometry viewer** module's center texture map port (see example below).

brick has the invisible "upstream transform" input port. This means that "brick" shows up as an object in the Geometry Viewer's object hierarchy. If you select the "brick" object and rotate, scale, or translate it with the mouse, the **geometry viewer** module informs the **brick** module of the new orientation of the slice plane, and **brick** remaps the volume data accordingly. The effect is that you have direct mouse manipulation control over the shape of the brick.

AVAILABILITY

This module requires 3D texture mapping support. 3D texture mapping is supported on only a few hardware renderers (see the release note information that accompanies AVS on your platform). If a renderer does not support 3D texture mapping, then the volume will appear, but the geometry object will appear as a featureless white solid.

Where there are multiple renderers available, you can select **Software Renderer** on the Geometry Viewer's **Cameras** submenu to switch renderers. Otherwise, the software renderer is the only renderer present. After changing to the software renderer, you may have to change one of the **brick** module's dials to get the proper results.

INPUTS

Data Field (required; field 3D uniform n-vector any-data)
The input field is a 3D uniform volume. The data can be of any type.

Upstream Transform (optional, invisible, autoconnect)
When the **brick** module coexists with the **geometry viewer** module in a network, **geometry viewer** feeds information on how the "brick" object has been moved in the Geometry Viewer back to this input port on the **brick** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **brick**'s slice plane.

PARAMETERS

X Rotation A floating point dial widget that controls the rotation of the slice surface in the X direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction.

Y Rotation A floating point dial widget that controls the rotation of the slice surface in the Y direction. The center of rotation is mid-way through the slice plane, like a revolving door, as opposed to at the edge of the slice plane, like a swinging door. The initial rotation is 0.0 (no rotation). The dial is unbounded and may be rotated more than 360 degrees in either the positive or negative direction.

Offset A floating point dial widget that controls the movement of the slice surface in the Z direction. The 0.0 initial value is defined to be *midway* through the volume. Hence, a volume with a Z dimension of 64 has 0.0 in the middle, with +32.0 and -32.0 in either direction. The dial itself is unbounded. If you enter a value outside the actual volume, the slice surface stops at the actual bounds.

Sides A boolean switch that controls whether all six surfaces of the volume are displayed (on), or only the slice surface (off). **Sides** is on by default.

OUTPUTS

Geometry (geometry)
The output geometry is the solid version of the volume.

EXAMPLE 1

The following network reads a byte volume. The volume is fed to **colorizer** to paint the byte values as colors, to **brick** to map the surfaces, and to **volume bounds** to draw a box around the limits of the volume. The **generate colormap**, **colorizer**, and **geometry viewer** parts of the network are vital; they create the 3D texturemap. All in turn feed into **geometry viewer**.

brick

```
                      READ VOLUME
                          |
GENERATE COLORMAP         |
        |                 |
        |  |--------------|---------------|
        |  |              |               |
     COLORIZER          BRICK      VOLUME BOUNDS
        |                 |               |
        |                 |               |
        |----------|      |---------------|
                   |      |
                  GEOMETRY VIEWER
```

EXAMPLE 2

The following network is the same as the previous example in basic structure. The difference is that the uniform volume data is a 3D field of real values, not bytes. The **vector mag** module is used to convert the vector field into a scalar float field. The addition of the **color range** module scales the color values in the colormap to match the range of the data. It should be included whenever the data is not of type byte.

```
                      READ FIELD
                          |
                          |
                      VECTOR MAG
GENERATE COLORMAP         |
        |                 |
        |   |-------------|---------------|
        |   |             |               |
   COLOR RANGE          BRICK      VOLUME BOUNDS
        |                 |               |
    COLORIZER             |               |
        |------------|    |---------------|
                     |    |
              GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
read field
Any module that outputs a 3D uniform field

Modules that could be used in place of **brick**:

excavate brick
volume render
arbitrary slicer
orthogonal slicer
thresholded slicer

Modules that can process **brick** output:

geometry viewer

SEE ALSO

Two BRICK example scripts demonstrate the **brick** module.

NAME

bubbleviz – generate spheres to represent values of 3D field

SUMMARY

Name	bubbleviz
Availability	Volume, FiniteDiff module libraries
Type	mapper
Inputs	field 1D / 2D / 3D scalar *any-data any-coordinates* colormap
Outputs	field 1D 3-coord 4-vector real

Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Radius	float	0.0	0.0	100.0

DESCRIPTION

The **bubbleviz** module generates spheres of various radii and colors at the element locations of a 1D, 2D or 3D field. This is a "cuberille" style of volume visualization, except that it uses spheres rather than cubes.

The colors and radii of the spheres are calculated by mapping the input field values to the color and opacity values in the colormap. This means that you can change the color of spheres by editing the hue, saturation and brightness panels of the colormap widget. The radii of the spheres is taken from the opacity data (last field) of the input colormap. To change the radii of an entire group of spheres, simply edit the **generate colormap**'s opacity panel.

This module can be used for non-uniform input fields (rectilinear or irregular).

Note that systems which do not have hardware support for sphere rendering have an additional Geometry Viewer control that lets you specify the number of polygons used to render spheres. The control's slider is located at the bottom of the Geometry Viewer control panel, and is titled "subdivision". The subdivision value ranges from 1 to 8; using a low value, e.g. 2, can improve the performance of **bubbleviz** considerably. In addition, overall system performance can be improved by shrinking the dataset size using the **downsize** module.

INPUTS

Data Field (required; field 1D / 2D / 3D scalar *any-data any-coordinates*)

The principal input data for the **bubbleviz** module is a 1D, 2D or 3D field. The data at each point of the field can be byte, integer, float or double. The values will be interpreted as numbers in the range 0..255.

Color Map (colormap)

The colormap may be of any size. Since each input datum is a byte, the natural size for the colormap is 256. If you specify a larger colormap, its entries beyond the 256th are unused.

A zero value in the opacity field of the colormap suppresses the generation of a sphere for the input datum.

PARAMETERS

Radius A multiplier factor for the sphere radii. This is particularly useful for irregular fields, for which the computational-to-physical mapping often makes the default spheres too small. The value of Radius is used to scale the opacity element in the input colormap.

The default **Radius** is zero; this causes spheres to be rendered as points (individual pixels).

OUTPUTS

Data Field (field 1D 3-coord 4-vector real)

The output is a list of points in 3D space, with a 4-vector of reals at each point:

- The first element is interpreted as the sphere's radius. If the radius value is 0.0, no sphere is generated as output. If the radius value is 1.0, the sphere's radius will equal the current value of the **Radius** parameter.
- The 2nd–4th elements of the lookup value specify the red-green-blue components of the sphere's color (0.0 = no color; 1.0 = maximum color).

EXAMPLE

A typical network using this module looks like this:

```
GENERATE COLORMAP           READ VOLUME
        |                        |
        |  |---------------------|----------------|
        |  |                     |                |
   COLOR RANGE                   |                |
           |                     |                |
           |------------------|  |                |
                              |  |                |
                         BUBBLEVIZ         VOLUME BOUNDS
                               |                  |
                               |                  |
                         SCATTER DOTS             |
                               |                  |
                               |------------------|
                               |
                        GEOMETRY VIEWER
```

Note that the list of points generated by the **bubbleviz** module is converted to a geometry by the **scatter dots** module.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
read field

Modules that could be used in place of **bubbleviz**:

colorizer
gradient shade
dot mapper

Modules that can process **bubbleviz** output:

scatter dots

LIMITATIONS

The **bubbleviz** module can generate extremely large databases (one sphere per voxel for volume data). Use 0.0 values in the last field of the input colormap ("opacity" field) to eliminate unnecessary data.

SEE ALSO

The example script BUBBLEVIZ demonstrates the **bubbleviz** module.

calc warp coeffs

NAME

calc warp coeffs – calculate warp coefficients for ip warp

SUMMARY

Name	calc warp coeffs
Availability	Imaging module library
Type	data
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 1D uniform 2-vector float (*optional, tiepoints*) image viewer id structure (*invisible, autoconnect*) mouse info structure (*invisible, autoconnect*)
Outputs	field 1D 2-vector float (*warp coefficients*) image draw structure

Parameters

Name	*Type*	*Default*	*Min*	*Max*
choice	choice	linear		
N tiepoints	int dial	3 | 4 | 6 | 9	3	*unbounded*
set pick mode	oneshot			
Pick Status	string	*none*		

DESCRIPTION

calc warp coeffs calculates the warp coefficients required by the **ip warp** module. **calc warp coeffs** calculates the XY warp coefficients from the given tiepoint list using matrix inversion.

The warp coefficients can be created in two ways:

- If you already have a set of XY tiepoints, you can input it as a field through the optional input port.
- You can interactively select the tiepoints through the **image viewer** using upstream picking.

When used interactively, designating the tiepoints involves an interaction between **calc warp coeff** and the **image viewer** module. **calc warp coeff** must be receiving the same image input as the **image viewer** module. **calc warp coeffs**'s left **image draw structure** output must be connected to the **image viewer** module's leftmost **image draw structure** input. **calc warp coeffs**'s right warp coefficient output is connected to **ip warp**'s center coefficient input. (See "Example 1" below).

To select the tiepoints in the Image Viewer window:

1. The **calc warp coeffs** module must have control of the left mouse button in the Image Viewer window. When **calc warp coeffs** is first connected and data first passes through it, it should have control of the left mouse button.
2. Specify the type of warp, and any changes to the default **N tiepoints** (3-**linear**, 4-**bilinear**, 6-**quadratic**, 9-**biquadratic**).
3. Select the first "source" tiepoint by clicking with the left mouse button. **calc warp coeffs**'s message box will prompt you for the corresponding "destination" tiepoint. Each source/destination pair will be connected with a line. The prompting will continue until **N tiepoints** have been selected. Then, the module will fire.

If there are multiple images in the Image Viewer window, and/or multiple sketching modules, then some other module or the Image Viewer itself may have control of the left mouse button. To get control back to **calc warp coeffs**,

1. Make the image the current image (use shift-left mouse button or left mouse button).
2. Press **set pick mode** on **calc warp coeffs**'s control panel.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)
The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, the coefficient calculation is performed just once.

Data Field (optional; field 1D uniform 2-vector float)
This input port will receive a 1D 2-vector float field that is the list of tiepoints. Tiepoints come in pairs—there is a source tiepoint and a destination tiepoint. Each tiepoint is a 2-vector float. The first vector is the X coordinate; the second vector is the Y coordinate. Thus, the first element of the 1D field is a source tiepoint (2-vector float), the second element is a destination tiepoint (2-vector float), and so on through the field.

The module must know how many tiepoint pairs are in the field. By convention, this value (# of tiepoints * 2) should be stored in the field's maximum X extent value. A module that is creating the warp field would set this value with the **AVSfield_set_extent** routine. There is no interactive way to set this value. This input is optional.

image viewer id structure (optional; invisible, autoconnect)
This input port is invisible by default. It is used when interactively selecting tiepoints. It connects automatically to the **image viewer** module's **image viewer id structure** output. The two modules communicate the **image viewer** module's scene id on this connection. Normally, you can ignore its existance.

mouse info structure (optional; invisible, autoconnect)
This input port is invisible by default. It is used when interactively selecting tiepoints. It connects automatically to the **image viewer** module's **mouse info structure** output. The two modules communicate image name, mouse pointer location and button up/down information on this connection. Normally, you can ignore its existance.

PARAMETERS

choice A set of radio buttons that determines the order and type of the warp. If the number of input tiepoints is equal to the minimum stated below, a warp using the returned coefficients perfectly match the tiepoints. If the number of input tiepoints exceeds the minimum, a set of coefficients which best fit the tiepoints is returned.

linear produces a separable set of coefficients for X and Y; that is, the xy term is zero. This warp type requires at least three tiepoints and limits the subsequent warp to rotation, scaling, reflection, and skewing.

bilinear allows non-zero XY coefficients and requires at least four tiepoints.

quadratic requires six input tiepoints and returns a separable set of quadratic coefficients.

biquadratic
requires nine tiepoints and returns a non-separable quadratic set of coefficients.

N tiepoints

An integer dial that specifies the number of tiepoint pairs to generate. This is used when there is no tiepoint input field, and tiepoints are being generated interactively. The default is 3, 4, 6, or 9, depending upon the type of warp. The maximum is unbounded.

set pick mode

A oneshot that sets the **image viewer**'s upstream mouse picking focus to this module. It is used when interactively generating points.

Pick status A string "prompter" that guides the user through interactively selecting warp initial XY tiepoint and destination XY tiepoint pairs. The number of prompts depends upon the **N tiepoints** value. The center input contains the polynomial warp coefficients.

OUTPUTS

Data Field (required; field 1D uniform 2-vector float)

The field containing the polynomial warp coefficients. It is a 1D uniform 2-vector float field. The first vector element contains the X polynomial warp coefficients; the second vector element contains the Y polynomial warp coefficients. This output is meant to be fed to **ip warp**'s center input port.

image draw structure (optional)

The left output port contains the **image draw structure** that connects to the **image viewer** module's leftmost input port. It is optional if you input a field of tiepoints. It is required when you interactively select tiepoints.

EXAMPLE 1

This network shows the connections necessary to interactively generate tiepoints. (The invisible, automatically created upstream connections between **image viewer** and **calc warp coeffs** are not shown.)

```
                    READ IMAGE
                          |
                          |
            |-------------|----|
            |             |    |
CALC WARP COEFFS          |    |
        |   |             |    |
        |   |---------|   |    |
        |             |   |    |
        |            IP WARP   |
        |                 |    |
        |---------|       |    |
                  |       |----|
                IMAGE VIEWER
```

EXAMPLE 2

This network shows the tiepoints provided through a field.

```
    READ FIELD      READ IMAGE
        |                 |
        |                 |
        |   |-------------|
        |   |             |
CALC WARP COEFFS          |
            |             |
            |---------|   |
                      |   |
                   IP WARP
                          |
                          |
                 IMAGE VIEWER
```

RELATED MODULES

ip warp

SEE ALSO

The example script Imaging / CALC WARP COEFFS demonstrates this module.

cfd values

NAME

cfd values - calculate values for a field containing read plot3D data

SUMMARY

Name	cfd values
Availability	Unsupported module library
Type	filter
Inputs	field 1D, 2D, or 3D irregular 5-vector float
Outputs	field 1- to 12-vector irregular same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Gamma	float	1.4	1	5
Gas Const	float	1	0	5
Value	choice	*all*		
vector length	integer	12	1	12

DESCRIPTION

cfd values takes the 5 vector irregular field, which **read plot3D** outputs, and derives 7 additional values for each point in the field. Thus, **cfd values** outputs a field of the same type as its input field, but with a vector of up to 12 values at each field location. Note that the input field must have a 5-vector at each location.

The field that **cfd values** receives from **read plot3D** has the following 5 values: density, X momentum, Y momentum, Z momentum, and stagnation.

From the these 5 values **cfd values** computes 7 new values: energy, pressure, enthalpy, mach number, temperature, total pressure, total temp. The gamma constant(γ) and the gas constant (R) are user controllable parameters, and the following variables are defined:

$U_1 = density$

$U_2 = x\ momentum$

$U_3 = y\ momentum$

$U_4 = z\ momentum$

$U_5 = stagnation$

The equations used to derive the new values are as follows:

energy (E) = $\frac{U_5}{U_1}$

pressure (p) = $(\gamma - 1)\left[U_5 - \frac{1}{2}\frac{(U_2^2 + U_3^2 + U_4^2)}{U_1}\right]$

enthalpy = $\frac{p}{U_1}$

mach number (M) = $\frac{(U_2^2 + U_3^2 + U_4^2)^{1/2}}{cU_1}$

temperature (T) = $\frac{p}{(U_1 R)}$

total pressure $(p_0) = p\left(1 + \frac{\gamma - 1}{2}M^2\right)^{\frac{\gamma}{\gamma - 1}}$

total temp $(T_0) = T(1 + \frac{\gamma - 1}{2} M^2)$

Note that, in calculating the 7 derived quantities, **cfd values** uses the same assumptions about the non-dimensionality, or normalization, of data that the National Aeronautics and Space Administration's PLOT3D, and the **read plot3D** module themselves use.

cfd values displays a set of buttons for specifying which values to include in its output field. To specify the number of values in the output field, first select the desired number of values using the "vector length" parameter. Then, pick which values to include; **cfd values** will output when you have chosen vector length elements. Note that, **cfd values**, actually only computes the values required by your selections.

INPUTS

Data Field (required; field 1D, 2D, or 3D irregular 5-vector float)

cfd values receives its input field from the module **read plot3d**. This is a 1D, 2D, or 3D irregular field, with a vector of 3 to 5 values at each field location.

PARAMETERS

Gamma A floating point value between 1 and 5, which determines the value of the (γ) constant. The formulas assume an ideal gas with a constant ratio of specific heats, (γ). The default value is 1.4.

Gas Constant

A floating point value between 0 and 5, which determines the value of the gas constant. The default value is 1.

Value A list of 12 buttons, displaying the names of the values that **cfd values** computes. To specify that a specific value should be included in **cfd values**'s output field, click on the value's button. The field output by **cfd values** can have between 1 and 12 values at each field location.

vector length

An integer dial, which specifies the number of data values at each location in the field **cfd values** outputs.

OUTPUTS

Output Field (field 1- to 12-vector irregular same type as input)

The output field is the same type as the input data field. However, the **cfd values** module computes up to 7 new values for each field location. Thus, the output may have a vector of between 1 and 12 values at every point in the field.

EXAMPLE

The following example shows how **cfd values** and **read plot3d** can be used. The **extract scalar** on the right extracts one value from the 12-vector that **cfd values** outputs. **isosurface** computes the isosurface for this scalar output, and **volume bounds** is used to draw a bounding box for the data. The left hand **extract scalar** module extracts another value from **cfd values** output. This second scalar field is used to color the isosurface. The **color range** module is used to scale the colormap to the range of the extracted cfd value. This network will allow you, for example, to generate an isosurface of the density in a field, and then color this isosurface based on the temperature values at each point on the isosurface.

```
                               READ PLOT3D
                                    |
                                    |
                               CFD VALUES
                                    |
GENERATE COLORMAP         |---------|--------|
        |                 |                  |
        |          EXTRACT SCALAR     EXTRACT SCALAR
        |                 |                  |
        |-----|   |-------|----|             |
              |   |            |   |---------|---------|
             COLOR RANGE       |   |                   |
                  |            |   |             VOLUME BOUNDS
                  |---------|  |   |                   |
                            |  |   |                   |
                           ISOSURFACE                  |
                                |                      |
                                |----------------------|
                                |
                         GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read plot3D

Modules that can process **cfd values**'s output:

isosurface
orthogonal slicer
hedgehog
bubbleviz
tracer

REFERENCES

Pieter Buening, **PLOT3D Reference Manual**.

SEE ALSO

The example scripts READ PLOT3D and CFD VALUES demonstrate the **cfd values** module.

NAME

character string - send a user-entered string to one or more module(s) string parameter port(s)

SUMMARY

Name	character string		
Availability	Imaging, UCD, Volume, FiniteDiff module libraries		
Type	data		
Inputs	none		
Outputs	string		
Parameters	*Name*	*Type*	*Default*
	character	typein	off

DESCRIPTION

The **character string** module sends a single user-specified string to one or more string parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control string parameter input to more than one module using only a single string input widget.

Before you can connect **character string** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A blue-green (teal) parameter port should appear on the module icon. Connect this parameter port to the **character string** module icon in the usual way one connects modules.

Note that the module **file browser** is functionally equivalent to **character string**. They both allow you to send strings to one or more other modules. Conceptually, however, the strings sent by **file browser** will tend to be filenames. While those sent by **character string** can be filenames, they are not limited to these.

PARAMETERS

character string (string)

The single string, specified through a string typein widget, to be sent to the receiving module(s) filename string parameter port(s). The default value is NULL.

OUTPUTS

string (string)

The string value is sent to all modules with string-type parameter ports that are connected to the **character string** module.

EXAMPLE 1

The following network shows (a somewhat contrived) example of how the **character string** module can be used to send a string constant to two different modules:

```
         CHARACTER STRING
             |
             |-----------------------|
             |                       |
         READ IMAGE              READ IMAGE
             |                       |
             |                       |
       IMAGE VIEWER              CONTRAST
                                     |
                                     |
                                 IMAGE VIEWER
```

RELATED MODULES

Modules that can process **character string**'s output:

all modules with string-type parameter ports

SEE ALSO

The DEMO script cli.scr demonstrates the **character string** module.

NAME

clamp – restrict values in data field to user-specified range

SUMMARY

Name	clamp
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	field of same type as input

Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	clamp_min	float	0.0	none	none
	clamp_max	float	255.0	none	none

DESCRIPTION

The **clamp** module transforms the values of a field as follows:

- Any value less than the value of the **clamp_min** parameter is set to **clamp_min**.
- Any value greater than the value of the **clamp_max** parameter is set to **clamp_max**.
- All values within the **clamp_min**-to-**clamp_max** range are not changed.

After being **clamp**'ed, a data set's values are all in this range:

clamp_min ≤ *value* ≤ **clamp_max**

If appropriate, **clamp** also changes the values of the **min_val** and **max_val** attributes of the output field in accordance with the **clamp_min** and **clamp_max** values. **clamp** works with uniform, rectilinear and irregular fields, whether they are vector or scalar.

The **statistics** module can be used to determine the **min_val** and **max_val** of the input field, so you can know what range is reasonable to clamp to.

Note the difference between the **clamp** and **threshold** modules:

- **threshold** sets values outside the specified range to be zero.
- **clamp** sets values outside the specified range to be the range's minimum and maximum values.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field. It may be uniform, rectilinear or irregular; and either vector or scalar.

PARAMETERS

clamp_min
A floating-point number that specifies the minimum output value.

clamp_max
A floating-point number that specifies the maximum output value.

OUTPUTS

Data Field (field *same-dimension same-vectorsame-data same-coordinates*)
The output field has the same dimensionality and type as the input field.

EXAMPLE

The following network reads in an AVS field. The **statistics** module is used to display the field contents with and without clamping:

```
READ FIELD
     |
     |--------------------|
     |                    |
   CLAMP              STATISTICS
     |
     |
 STATISTICS
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
any other filter module

Modules that could be used in place of **clamp**:

threshold

Modules that can process **clamp** output:

colorizer
any other filter module

Modules that tell you the range of data in the field:

statistics
print field
generate histogram

SEE ALSO

The example script CLAMP demonstrates the **clamp** module.

NAME

clip geom – specify arbitrary clipping planes for geometric objects

SUMMARY

Name	clip geom
Availability	UCD, Volume, FiniteDiff module libraries requires arbitrary clipping plane support
Type	data
Inputs	*none*
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Choices*
clip plane	choice	Red Plane	Red, Green, Blue, Cyan
Inside	oneshot		
Outside	oneshot		
Don't Clip	oneshot		
Inherit	oneshot		
Reparent	oneshot		
Show Outline	oneshot		
Hide Outline	oneshot		

DESCRIPTION

The **clip geom** module allows the user to specify four clipping planes to the **geometry viewer** module. Each clipping plane can have an arbitrary orientation and position. When an object is clipped by a plane, only the geometry that lies on one side of the clipping plane will be drawn.

The four clipping planes are named: **Red Plane**, **Green Plane**, **Blue Plane** and **Cyan Plane**. Each clipping plane is defined as a normal geom object that is created the first time that the clip plane is manipulated from the module. Clip planes initially appear at 0,0,0 as the Y=0 plane. A graphical depiction of the clipping plane object can be displayed using the **Show Outline** button and hidden using the **Hide Outline** button. The color of the clipping plane icon is red for the **Red Plane**, green for the **Green Plane**, etc.

In order to cause an object to be clipped by the Geometry Viewer, you should first make sure that the appropriate clip plane is selected in the **clip plane** choice menu (e.g. Red Plane), then select the object whose clipping state you wish to modify using the Geometry Viewer. Now you can modify the clipping state of the object by choosing one of the four functions. The **Inside** function causes the clip plane to clip the current object to one side of the plane. **Outside** causes the clip plane to clip the current object to the other side of the plane. (At the clip plane's initial Y=0 position, **Inside** means that only the parts of objects with positive Y components are drawn while **Outside** draws only the parts of objects with negative Y components.) The **Don't Clip** function says that the clip plane should not clip the object at all. The **Inherit** function causes the clip plane to inherit the clip state for this clip plane from its parent object.

For example, if the top-level object is the current object and you pick the **Inside** button, all objects will be clipped to the inside of the current clip plane. You might then choose a child of top and select **Don't Clip**. Now all objects will be clipped (because they inherit the clip state of top) except for the child you chose, which will not be clipped by the object.

Additionally, the current clip plane can be reparented to the current object by selecting the **Reparent** oneshot. This has the affect of concatenating the clip plane's transformation after the new parent's transformation. This makes it possible to manipulate the orientation of the clip plane either by transforming the parent object (in which case the clip plane will move with the parent object) or by selecting the clip plane directly (in which case it will move independently of the parent object).

The scale of the clipping plane object affects the size of the graphical representation of the clipping plane only. It does not affect the way in which objects are clipped.

AVAILABILITY

clip geom requires that the underlying graphics renderer support arbitrary clipping planes. Not all hardware renderers support arbitrary clipping planes (see the release note information that accompanies AVS on your platform). The AVS software renderer does support arbitrary clipping planes. If a renderer does not support arbitrary clipping planes, then the clipping planes will appear, and you can manipulate them as described above, but the geometry objects will not actually be clipped. To get the clipped objects on multi-renderer platforms, you can turn on the **Software Renderer** button under the Geometry Viewer's **Cameras** submenu.

PARAMETERS

clip plane The clip plane parameter specifies the current clip plane. All other functions affect how the current clip plane interacts with the currently selected geometric object. Available choices for this parameter are: **Red Plane**, **Green Plane**, **Blue Plane**, **Cyan Plane**.

Inside This parameter causes the current clip plane to clip the current object to the inside. At the clip plane's initial Y=0 position, this draws only the parts of the object(s) with positive Y components.

Outside This parameter causes the current clip plane to clip the current object to the outside. At the clip plane's initial Y=0 position, this drawns only the parts of object(s) with negative Y components.

Inherit This parameter causes the current object to inherit the clip state for this object from its parent object rather than assigning the clip state to itself. The default clip state for each object is **Don't Clip**.

Don't Clip This parameter causes the current object not to be clipped by the current clip plane.

Reparent This causes the current clip plane object to be reparented to the currently selected object in the Geometry Viewer.

Show Outline
This button causes a graphical depiction of the current clip plane to be displayed in the Geometry Viewer.

Hide Outline
This button causes the graphical display of the current clip plane to be removed from the Geometry Viewer.

OUTPUTS

Geometry (geometry)
The output contains the clip plane specification information.

EXAMPLE 1

The following example will clip an object read into the Geometry Viewer through its **Read Object** function.

```
                      CLIP GEOM
                          |
                          |
                   GEOMETRY VIEWER
```

EXAMPLE 2

The following example will clip a geometry entering the Geometry Viewer from an upstream module.

```
              any module that outputs a geometry
                          |
      CLIP GEOM           |
          |               |
          |---------------|
                          |
                          |
                   GEOMETRY VIEWER
```

RELATED MODULES

geometry viewer

LIMITATIONS

The current clipping state is not displayed on the menu panel when a clip object is selected.

Clip plane state is not saved / restored when a network is saved and restored.

color legend

NAME

color legend – display color-to-data value mappings in geometry viewer window

SUMMARY

Name	color legend
Availability	Imaging, Volume, FiniteDiff module libraries
Type	mapper
Inputs	colormap
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Legend Control				
position	choice	vertical		
Reverse Colors	boolean	off		
Legend Outline	boolean	off		
Outline Gray Scale	int slider	255	0	255
Label Controls				
Labels	boolean	on		
Ticks	boolean	off		
Number of Ticks	int slider	2	2	20
Label Height	float slider	0.05	0.01	1.0
Decimal Precision	int slider	1	0	10
Label Gray Scale	int slider	255	0	255
Label Font	int slider	0	0	20
Legend Position				
X Position	float slider	-.78	-1.0	1.0
Y Position	float slider	-.45	-1.0	1.0
Z Position	float slider	.99	-1.0	1.0
Thickness	float slider	.05	.01	2.0
Length	float slider	1.0	.01	2.0

DESCRIPTION

color legend shows the colormap-to-data value mapping in the **geometry viewer** display window. It makes it easy to quickly identify which colors correspond to which numeric values.

color legend creates a colored bar in the **geometry viewer** output window. The colored bar shows the current composite colormap. The color legend can be overlaid with tick marks and labels that show the data values that correspond to the colors. The color legend can be vertical or horizontal, positioned within the geometry window, and made wider and/or longer.

INPUTS

position A pair of radio buttons that select the **vertical** or **horizontal** orientation of the color legend. **vertical** is the default.

Reverse Colors

A boolean switch. If **off**, lower numbers are to the left/bottom of the scale. If **on**, lower numbers are to the right/top of the scale. The default is **off**.

Legend Outline
A boolean switch that surrounds the color legend with a grayscale box for appearance purposes. The default is **off**.

Outline Gray Scale
An integer slider that establishes the grayscale color of the color legend outline, and the grayscale color of tick marks, if present. The range is 0 to 255. The default is 255 (white).

Labels A boolean switch. If **on**, the color legend is labeled with data values. The labels are taken from the lower and upper bound values found in the input colormap. These lower and upper bound values are established by the **hi value** and **lo value** dials in the **generate colormap** module (default 255 and 0), or—more typically—with the **color range** module. (**color range** copies the field's minimum and maximum data values to the colormap, if present, or calculates the minimum and maximum. Thus, it scales the colormap to the data range.) The default is **on**.

Ticks A boolean switch. If **on**, tick marks are placed on the color legend above each label. **off** is the default.

Number of Ticks
An integer slider that establishes how many labels and tick marks will appear on the color legend. The color legend is divided into *n*-1 intervals. The default is 2. The range is 2 to 20.

Label Height
A float slider that controls the size of the labels. Note that most systems support a limited number of font sizes. **Label Height** selects the closest actual font size available. The default is 0.05. The range is 0.01 to 1.0.

Decimal Precision
An integer slider. *n* is the number of places to right of the decimal point to display in labels. The default is 1. The range is 0 (whole numbers only) to 10.

Label Gray Scale
An integer slider that sets the grayscale color of the labels. The default is 255 (white). The range is 0 to 255.

Label Font An integer slider that picks the label font. The number to actual font correspondence varies from platform to platform. The default is 0. The hypothetical range is 0 to 20.

X Position
Y Position
Z Position Floating sliders that control the position of the color legend within the geometry window in screen coordinates. X,Y=0 is the center of the window. **X Position** and **Y Position** define the left edge (if vertical) or bottom (if horizontal) of the color legend. **X Position** defaults to -.78. **Y Position** defaults to -.45. Their range is -1.0 to 1.0.

Z Position defines whether the color legend is in front of or behind objects in screen coordinates. The default is .99 (in front). The range is -1.0 to 1.0.

Thickness Floating slider to set the width of the color legend. The range is 0.01 to 2.0. The default is 0.05.

color legend

Length Sets the length of the color legend. The default 1.0 is half the size of the geometry window. The range is .01 to 2.0.

OUTPUTS

geometry (geometry)

The output is a geometry representing the color legend. This geometry cannot be transformed using the **geometry viewer**.

EXAMPLE

This network places a color legend in the **geometry viewer**'s display window. Note the use of **color range** to establish the correct data value range in the colormap.

```
                       READ FIELD
                           |
GENERATE COLORMAP    EXTRACT SCALAR
            |              |
            |  |-----------|----------------|
       COLOR RANGE         |                |
               |           |         VOLUME BOUNDS
         |-----|-------|   |                |
         |          ARBITRARY SLICER        |
         |                 |                |
 COLOR LEGEND              |                |
           |               |                |
           |---------------|----------------|
                           |
                GEOMETRY VIEWER
```

RELATED MODULES

generate colormap
color range
field legend

SEE ALSO

The example script COLOR LEGEND demonstrates this module.

LIMITATIONS

color legend can only be automatically used with field data. The UCD module colorizing apparatus does not store into the colormap's upper and lower bound areas. (They default to 0 to 255.) You can still use **color legend** to annotate UCD data if you manually set the **lo value** and **hi value** dials on **generate colormap**'s control panel.

NAME

color range – scale AVS colormap to the range of data in a field

SUMMARY

Name	color range
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data
Inputs	field *any-dimension* scalar *any-data any-coordinates* colormap MODIFIES INPUT
Outputs	colormap
Parameters	*none*

DESCRIPTION

color range adjusts the minimum and maximum values of a colormap to those of an AVS field, thus normalizing the colormap to the range of the data in the field. To do this, **color range** examines a scalar AVS field to see if the minimum and maximum data values are specified in the field's data structure. If they are not, it calculates the minimum and maximum values and stores them in the field's data structure. In both cases, **color range** also stores the minimum and maximum data values into its output AVS colormap data structure.

Use **color range** whenever you have data that you want represented as colors, but that data's range of values is either not evenly distributed between 0 and 255, or much of the data values lie outside the 0 to 255 range.

For example, your input field contains floating point values between the range 0 and 1. If you were to give this range of data values to one of the modules that produces colors from numbers (e.g., **arbitrary slicer** or **field to mesh**) all of the numbers would map to the same color. Because data coloring is done by using a byte value 0-255 to index into the AVS colormap, all of these floating point values would map to the number 1, and hence to the same color. In the default colormap this is the same blue.

Similarly, if you have data that lies in the range -55 to +500, all values outside the range 0-255 will be "clamped" to the two boundary values and visual information about the data's true character will be lost.

Applying **color range** between the output of the **generate colormap** module and a scalar version of your data field stores the range of your data values into the color-map data structure. Modules downstream can use these minimum and maximum values to scale their index into the colormap intelligently. A narrow range of data values will be made to "fan out" across the whole colormap. A wide range of data values will be scaled to fit within the 0-255 range without clipping outlying values. Note, however, that this desirable effect does *not* occur just because **color range** is in the network; it occurs because the downstream modules that receive the modified colormap data structure have been written to make intelligent use of the new minimum / maximum values **color range** generates.

INPUTS

Data Field (required; field *any-dimension* scalar *any-data any-coordinates*)

This is the AVS field whose field data structure will be scanned to see if it already contains minimum and maximum data values. If it does, these data values will be stored into the output colormap data structure. If it does not, **color range** calculates the minimum and maximum values and stores them into both the original AVS field's data structure and the output colormap. Because **color range** can modify the original AVS field, data passing through this module is not shared.

Color Map (required; colormap)

This is the original AVS colormap. Any minimum or maximum values that may have been set in the input colormap are ignored.

OUTPUTS

Color Map (colormap)

The output from **color range** is a new colormap containing the calculated (or transferred from the input field data structure) minimum / maximum data values.

EXAMPLE

The following network reads in a 3-vector field, i.e. every field location has 3 values associated with it. The **extract scalar** module selects one of the fields values. **color range** stores the field’s min and max values so that the colormap can be scaled to the range of data in the field:

```
                  READ FIELD
                      |
GENERATE COLORMAP     |
        |             |
        |       EXTRACT SCALAR
        |             |
        |             |-------------|
        |---------|   |             |
                  |   |             |
                COLOR RANGE    ORTHOGONAL SLICE
                      |             |
                      |    |--------|
                      |    |
                 FIELD TO MESH
                      |
                GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read field
extract scalar (for fields with vectors)

Modules that could provide the **Color Map** input:

generate colormap

Modules that can process **color range** output:

arbitrary slicer
bubbleviz
colorize
field legend
field to mesh
isosurface
probe

Modules that can be used instead of **color range**:

minmax

SEE ALSO

The example script COLOR RANGE demonstrates the **color range** module.

NAME

colorize geom – assign vertex colors, vertex transparency and vertex UVW's (for 3D texture mapping) to vertices of a geometric object using a field and colormap.

SUMMARY

Name	colorize geom		
Availability	Volume, FiniteDiff module libraries requires vertex transparency and / or 3D texture mapping support		
Type	filter		
Inputs	geometry field 3D scalar *any-coordinates any-data* colormap (optional) upstream transform (*optional, invisible, autoconnect*)		
Outputs	geometry upstream transform (*optional, invisible, autoconnect*)		
Parameters	*Name*	*Type*	*Default*
	Vertex Colors	boolean	on
	Vertex Trans	boolean	off
	Vertex UVW	boolean	off

DESCRIPTION

The **colorize geom** module assigns vertex colors and / or transparency and / or UVW information to the vertices of a geometry that is passed as an input using an input field and a colormap.

For vertex colors and transparency, the exact method for doing this is as follows: 1) find where in the field the vertex lies (the **points** array in the field determines the coordinate system of the field), 2) interpolate adjacent field values to determine the value of the field at the vertex, 3) use that value as an index into the colormap to obtain the color / transparency of the vertex. This method works for uniform, rectilinear and irregular data.

For the UVW's required for 3D texture mapping, the module finds the location of the vertex in the field and uses this to determine a value of between 0 and 1 for each of U, V and W. If the vertex lies at the 0,0,0 corner of the field, it will be assigned a UVW value of 0,0,0. If it is at the maximum of the three dimensions, it gets a UVW value of 1,1,1. All other values are interpolated inbetween. **Note:** this technique only produces correct values with *uniform* fields; the values and colors generated for rectilinear or irregular fields will not be accurate. Once UVW's have been associated with a geometric object, it can be used with 3D texture mapping. The generation of UVW's does not require a colormap connected to the colormap input port.

If the **Vertex Colors** parameter is on, the vertex colors are used for the object. If the object already has vertex colors, the new vertex colors replace them.

If the **Vertex Trans** parameter is on, the "opacity" channel of the colormap is used to determine the transparency of each vertex in the object. This can be adjusted using the **generate colormap** module's colormap editor **opacity** controls.

If the **Vertex UVW** parameter is on, the extent of the field is used to determine UVW values at each vertex.

One notable use of this module is to combine viewing of multiple related scalar values in the same view. For example, streamlines of velocity can be assigned vertex colors based on pressure. Another example is a slice plane of temperature that is displayed with vertex transparency based on pressure.

Another use of vertex transparency is to cull out the rendering of data that is not interesting to the visualization. With this module you could remove all parts of a slice plane that have temperature less than a threshold value. In this way, this module has a role similar to the **thresholded slicer** module but that it can apply to any mapping technique and a continuous drop-off can be achieved rather than a simple binary classification (which will tend to introduce artifacts).

Vertex UVW's can be used to map a 3D texture map onto a geometric object. 3D texture mapping is an alternative to using vertex colors for sampling within a 3D uniform volume. The main advantage of using 3D texture mapping over vertex colors for this application is that texture mapping does not require a high-resolution mesh to represent a high-resolution data set. As each polygon is drawn, the 3D texture mapping algorithm chooses the closest color in the field for that pixel. Very high-resolution data sets can be represented with low-resolution polygonal objects.

AVAILABILITY

There are two techniques in this module that require underlying graphics renderer support: vertex transparency and 3D texture mapping. Vertex transparency and 3D texture mapping are supported on only a few hardware renderers (see the release note information that accompanies AVS on your platform). The software renderer does support 3D texture mapping; it does *not* support vertex transparency. Where a rendering function is not present, you can still use the other visualization options the **colorize geom** module provides.

On renderers without vertex transparency, the opacity channel on the colormap editor will have no effect on the transparency / opacity of verticies when **Vertex Trans** is selected—all will be opaque. On platforms without 3D texture mapping, the object will appear white rather than colored if **Vertex UVW** is selected.

Where there are multiple renderers available, you can select **Software Renderer** on the Geometry Viewer's **Cameras** submenu to switch renderers. Otherwise, the software renderer is the only renderer present.

INPUTS

Geometry (required; geom)

The geometry input provides the geometry on which the colorization process operates. All attributes contained in the geometry structure are passed through unmodified.

Data Field (required; 3D scalar field *any-data any-coordinates*)

The field data for the **colorize geom** module is used to determine the value to index into the colormap to obtain the color / opacity to color the vertex by. The **points** array in the field is used to determine the physical coordinate system in which to correlate the vertices. This is true regardless of which type of field is used (uniform, rectilinear and curvilinear).

Color Map (optional; colormap)

The colormap may be of any size, but any entries beyond the 256th are unused. If the colormap port is left empty a default grey-scale ramp is used to generate vertex colors, and a default 0-1 opacity ramp is used to generate vertex transparency.

upstream transform (optional, invisible; struct upstream_transform)

If any data changes on this input port, it will be passed on to the producing module. This port is generally invisible and is connected automatically when a compatible module is connected to the geometry output port. Through this port, the module receives the information from the **geometry viewer** module necessary for direct mouse manipulation control of sampling objects. It will "forward" this information back up the network to a mapper module that produces the sampling object through its upstream transform output port (below).

OUTPUTS

Geometry (geom)

The geometry output port contains the geometry that has been colorized and / or given vertex transparency.

upstream transform (invisible, struct upstream_transform)

This port is generally connected automatically when a compatible module is connected to the geometry input port. It passes along any upstream transform information that is received on the input port directly.

EXAMPLE

The following example network can be used to assign vertex transparency to the vertices in the **arbitrary slicer**. Since **arbitrary slicer** already assigns vertex colors, it is redundant to use the **Vertex Colors** parameter in the **colorize geom** module so we turn that parameter off and turn on the **Vertex Trans** parameter.

```
                            READ VOLUME
                                 |
GENERATE COLORMAP                |----------|
      |--------------------------|------|   |
      |                          |   ARBITRARY SLICER
      |                          |          |
      |---------------------|    |  |-------|
                            |    |  |
                          COLORIZE GEOM
                               |
                               |
                         GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

- read volume
- read field
- any module that produces a 3D scalar field

Modules that could provide the **geometry** input:

- arbitrary slicer
- hedgehog
- isosurface
- streamlines
- contour to geom
- field to mesh
- scatter dots
- threshold slicer

Modules that could provide the **Color Map** input:

- generate colormap
- color range

Modules that can process **colorize geom** output:

- geometry viewer
- render geometry

colorizer

NAME

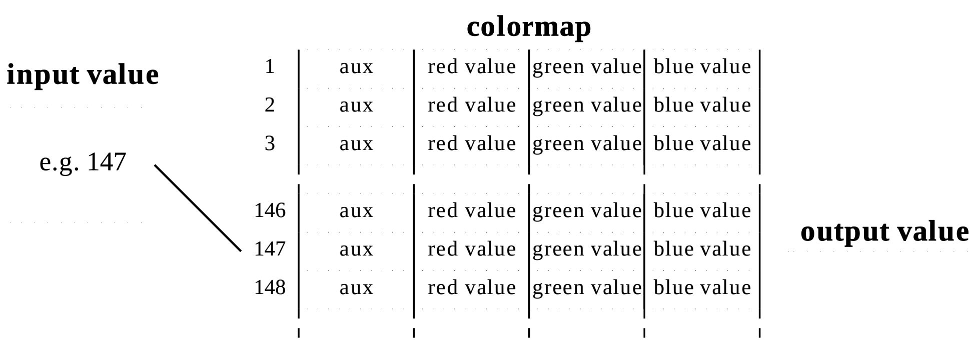
colorizer – convert field of data values to color values

SUMMARY

Name	colorizer
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension* scalar *any-data any-coordinates* colormap
Outputs	field *any-dimension* 4-vector byte *any-coordinates*
Parameters	*none*

DESCRIPTION

The **colorizer** module converts the data at each point of a scalar field from the input value (which can be any data type) to a *color* (4-vector of bytes). The conversion is accomplished by using the input value as an index into a *colormap*:



colorizer accepts field of any type (byte, integer, real, double). However, the field of colors output by **colorizer** contains only byte data.

INPUTS

Data Field (required; field *any-dimension* scalar *any-coordinates*)

The principal input data for the **colorizer** module is a field, which can be of any dimensionality. The data at each point of the field may be of any data type.

Color Map (optional; colormap)

The optional colormap may be of any size, but any entries beyond the 256th are unused. **Default:** If this input is omitted, a gray-scale colormap is used (lo-value = black; hi-value = white).

OUTPUTS

Field of Colors (field *any-dimension* 4-vector byte *any-coordinates*)

Each input value is transformed into a color value, which is structured as four bytes, as illustrated above. The red, green, and blue bytes specify a true-color pixel value. The *auxiliary* byte is typically used to specify an opacity value (lo-value = completely transparent; hi-value = completely opaque).

The dimensionality of the output field is the same as that of the input field. For byte input, the output field is four times as large as the input field, since each byte (8 bits) is converted to a color value (32 bits).

The **min_val** and **max_val** attributes of the output field are invalidated. The dimensions of the 4-vector output data are assigned the labels

"Alpha", "Red", "Green", and "Blue".

EXAMPLE

The following network reads in an AVS image, which is a 2D field of 4-vector bytes. **extract scalar** takes one of the bytes, generating a 2D field with a single byte at each location. These bytes are then translated back into colors by **colorizer**:

```
                             READ IMAGE
                                  |
                                  |
GENERATE COLORMAP           EXTRACT SCALAR
          |                       |
          |---------------------| |
                                | |
                             COLORIZER
                                 |
                                 |
                            DISPLAY IMAGE
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
field to byte

Modules that could provide the **Color Map** input:

generate colormap

Modules that could be used in place of **colorizer**:

arbitrary slicer

Modules that can process **colorizer** output:

alpha blend
gradient shade
display image
tracer

SEE ALSO

Many of the AVS example scripts demonstrate the **colorizer** module.

colormap manager

NAME

colormap manager – share colormaps among subnetworks

SUMMARY

Name	colormap manager	
Unsupported	this module is in the unsupported library	
Type	data	
Inputs	none	
Outputs	colormap	
Parameters	*Name*	*Type*
	Colormap Manager	colormap
	Colormap Choices	choice

DESCRIPTION

The **colormap manager** module produces an AVS *colormap* data structure, for use by modules that transform input data into color values. These modules include:

colorizer
arbitrary slicer
bubbleviz
field to mesh
isosurface

colormap manager works exactly like **generate colormap**, with one exception: separate active subnetworks, each with its own **colormap manager** module, share a single "pool" of colormaps.

A menu of all the active colormaps appears in a choice menu below each *colormap manager*'s editing widget. All the menus have the same entries — different maps can be selected in different managers.

PARAMETERS

Colormap Manager

A colormap generator widget. See the **generate colormap** manual page for details on using this widget.

Colormap Choices

A set of choices, listing each of the currently active colormaps.

OUTPUTS

colormap The output is an AVS colormap.

EXAMPLE

Suppose the following two subnetworks are active, created to slice through two different databases:

```
                 READ VOLUME                          READ VOLUME
                      |                                    |
                      |                                    |
 COLORMAP MANAGER     |              COLORMAP MANAGER      |
                |     |                              |     |
            COLORIZER                            COLORIZER
                |                                    |
         ORTHOGONAL SLICER                    ORTHOGONAL SLICER
                |                                    |
           DISPLAY IMAGE                        DISPLAY IMAGE
```

Each **colormap manager** module has its own *colormap editor* control widget. Below the two colormap editors are two choice menus:

```
+-------------------------+    +-------------------------+
| Active Colormaps        |    | Active Colormaps        |
+-------------------------+    +-------------------------+
| * colormap 0            |    |   colormap 0            |
+-------------------------+    +-------------------------+
|   colormap 1            |    | * colormap 1            |
+-------------------------+    +-------------------------+
```

The same "pool" of colormaps is shown in each menu, but a different colormap is currently selected for each subnetwork.

By default, each new **colormap manager** that is instantiated from the module Palette has it's own unique colormap editor. You can click on the **colormap 0** button for the second subnetwork in order to have both subnetworks use the same colormap:

```
+-------------------------+    +-------------------------+
| Active Colormaps        |    | Active Colormaps        |
+-------------------------+    +-------------------------+
| * colormap 0            |    | * colormap 0            |
+-------------------------+    +-------------------------+
|   colormap 1            |    |   colormap 1            |
+-------------------------+    +-------------------------+
```

Now, editing the colormap in *either* **colormap manager** module is reflected in both subnetworks.

You can extend the sharing of colormaps to any number of currently active subnetworks. Each must have its own **colormap manager** module.

NOTE

colormap manager modules are used in both the AVS2 *Image Viewer* and AVS2 *Volume Viewer* subsystems. However, these subsystems are no longer a supported part of the AVS release.

combine scalars

NAME

combine scalars – combine scalar fields into a vector field

SUMMARY

Name	combine scalars
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension* scalar *any-data any-coordinates* (channel 0 — optional) field *any-dimension* scalar *any-data any-coordinates* (channel 1 — optional) field *any-dimension* scalar *any-data any-coordinates* (channel 2 — optional) field *any-dimension* scalar *any-data any-coordinates* (channel 3 — optional)
Outputs	field *same-dimension* 1D–4D *same-data*

Parameters

Name	Type	Default	Min	Max
Vector Len	Dial	4	1	4

DESCRIPTION

The **combine scalars** module combines up to four fields with scalar data values into a field whose data values are vectors. The input field must be of like dimension and the scalar values must be of the same type.

This module is generally most useful for constructing images or gradient fields from separately computed components.

The inputs ports on this module's Network Editor icon are processed right-to-left: the rightmost port contributes a value to the first element (lowest memory location) of each output vector; the leftmost port contributes a value to the last element (highest memory location) of each output vector.

If the selected scalars have labels and / or units associated with them, those labels will be carried over to the newly constructed vector.

INPUTS

None of the input fields is absolutely required, but at least one of them must be provided. If an input field is omitted, zero values may be output in the corresponding element of each output vector, depending on the vector dimension set by **Vector Length**.

Channel 0 (optional; field *any-dimension* scalar *any-data any-coordinates*)
The rightmost input port. A set of values to be output in the *first* dimension of the output vectors.

Channel 1 (optional; field *any-dimension* scalar *any-data any-coordinates*)
A set of values to be output in the *second* dimension of the output vectors.

Channel 2 (optional; field *any-dimension* scalar *any-data any-coordinates*)
A set of values to be output in the *third* dimension of the output vectors.

Channel 3 (optional; field *any-dimension* scalar *any-data any-coordinates*)
The leftmost input port. A set of values to be output in the *fourth* dimension of the output vectors.

PARAMETERS

Vector Length
Specifies the dimension of the output vectors—1 – 4.

OUTPUTS

Field (field *same-dimension* 1D–4D *same-data*)

The scalar input streams are assembled into a single output stream consisting of vectors, whose dimension is specified by **Vector Length**. The coordinate type (e.g. uniform, rectilinear, or irregular) of the output field is the same as the leftmost, nonempty input field. The field's **min_val**, **max_val**, **veclen**, **label**, and **unit** are updated.

EXAMPLE 1

The following network performs contrast stretching on only the *red* band of an image.

```
                        READ IMAGE
                            |
           ____________________|____________________
           |                   |                   |
  EXTRACT SCALAR[red]  EXTRACT SCALAR[green]   EXTRACT SCALAR[blue]
           |                   |                   |
        CONTRAST               |                   |
           |________________   |   ________________|
                           |   |   |
                     COMBINE SCALARS (channel 0 not used)
                               |
                          DISPLAY IMAGE
```

EXAMPLE 2

The following network swaps the *green* and *blue* bands of an image:

```
                        READ IMAGE
                            |
           ____________________|____________________
           |                   |                   |
  EXTRACT SCALAR[red]   EXTRACT SCALAR[green]   EXTRACT SCALAR[blue]
           |______________     |______     _________|
                         |          |     |
                         |     /---- | ----'
                         |     |     |
                       COMBINE SCALARS
                               |
                          DISPLAY IMAGE
```

RELATED MODULES

extract scalar

SEE ALSO

The example script CONTRAST demonstrates the **combine scalars** module.

compare field

NAME

compare field – compare two AVS fields, display and write data difference

SUMMARY

Name	compare field
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field *any-dimension n-vector any-data any-coordinates* field *same-dimension same-vector same-data same-coordinates*
Outputs	none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Do Compare	oneshot	off		
Max Elements	integer	100	1	1000
Output File	typein	*/tmp/cfield_...*		

DESCRIPTION

The **compare field** module compares **any** two identically-structured AVS fields. It will print out differences between the headers if they are different. If the headers are the same, it will proceed to do a comparison of the data contents of the two fields. If the fields are not identical in their data components, **compare field** will print the message, "fields are DIFFERENT", to standard output.

The output of the compare is a list of up to **Max Elements** data *differences*. The results of the compare are both displayed in an **Output Browser** widget in the control panel and written to a file.

The Output Browser in which **compare field** displays its output can be resized, like any other widget, using the AVS Layout Editor. For a detailed description of how to do this, see the section titled "Layout Editor," in the chapter "Advanced Network Editor" of the *AVS User's Guide*.

compare field was originally written to make sure that two identical modules, one written in C and one written in Fortran, produced the same results. It could also be useful to compare the contents of a field before and after an operation has been performed on it.

INPUT

Input Field 1 (required; field *any-dimension n-vector any-data any-coordinates*)

The input AVS field can be 1, 2, 3, or 4 dimensional; it can be vector or scalar, can contain byte, int, float or double data, and can have uniform, rectilinear, or irregular coordinates.

Input Field 2 (required; field *any-dimension n-vector any-data any-coordinates*)

The second AVS input field must match the first in the number of dimensions (Ndim), the size of each dimension (Dims), the number of coordinate dimensions (Nspace), the vector length (Veclen), the data type (byte, float, double, etc.), and the type of coordinate system (uniform, rectilinear, curvilinear), if a comparison of the two fields' data is to be done.

PARAMETERS

Do Compare

A oneshot "do it now" switch that triggers the actual comparison after both input fields exist.

Max Elements

An integer dial that controls how many of the data *differences* to display in the **Output Browser** and write to the output file. The allowable range is -1 (none) to 1000. The default is 100. **compare field** compares the entire fields, until this limit is reached.

Output File

An ASCII typein for specifying the output file. By default, **compare field** writes to a file in the */tmp* directory called *cfield_nnnn* (where nnn is the process id of the **compare field** module. The **Output File** is rewritten whenever any of the other parameters or input files change. Since the Output Browser is limited in size, this output file can be useful to examine directly, using a conventional text editor.

EXAMPLE 1

The following network reads an image into an AVS field. One version of the image goes directly to **compare field**, the other is passed through a **contrast** filter. The "before" and "after" images are compared and the different alpha, red, green, blue values at each pixel are listed.

```
              read image
                   |
                   |
      -------------|
      |            |
   contrast        |
      |            |
      --------|    |
              |    |
           compare field
```

RELATED MODULES

ip compare
print field

LIMITATIONS

compare field writes to */tmp* by default. This can cause problems if: (1) there is no */tmp* mounted on your system, or (2) the */tmp* directory does not have very much room in it or has inaccessible protections.

SEE ALSO

The example script COMPARE FIELD demonstrates the **compare field** module.

composite

NAME

composite – blend two images using alpha transparency

SUMMARY

Name	composite
Availability	Imaging module library
Type	filter
Inputs	field 2D uniform 4-vector byte *(foreground image)* field 2D uniform 4-vector byte *(background image)*
Outputs	field 2D uniform 4-vector byte *(blended image)*
Parameters	none

DESCRIPTION

The **composite** module takes the contents of the foreground image's alpha channel (the image's opacity) and uses it to blend the foreground image over the background image. The equation for this blending is:

red = (Foreground(red) ∗ ALPHA) + (Background(red) ∗ (1.0 - ALPHA))
green = (Foreground(green) ∗ ALPHA) + (Background(green) ∗ (1.0 - ALPHA)
blue = (Foreground(blue) ∗ ALPHA) + (Background(blue) ∗ (1.0 - ALPHA)

where ALPHA is the foreground image's alpha channel byte value. If the two inputs are reversed, the alpha of the new foreground image will be used.

INPUTS

Image (required; field 2D uniform 4-vector byte)
The right input port on the module receives the foreground image.

Image (required; field 2D uniform 4-vector byte)
The left input port on the module receives the background image. The size of the background image must be identical to the size of the input image.

OUTPUTS

Image (field 2D uniform 4-vector byte)
The blended image of the two input images.

EXAMPLE 1

The following network reads an image, computes its luminance, (gray scale intensities) uses that to create an alpha mask, generates a shaded background, and composites the rendered image over the shaded background image.

```
                       READ IMAGE
                           |
  _________________________|
  |              |         |
BACKGROUND   LUMINANCE     |
  |              |         |
  |          CONTRAST      |
  |              |------|  |
  |                 REPLACE ALPHA
  |                        |
  |---------------------|  |
                      COMPOSITE
                           |
                    DISPLAY IMAGE
```

RELATED MODULES

Modules that could provide the foreground **Image** input:

read image
replace alpha

Modules that could provide the background **Image** input:

background

Modules that can process **composite** output:

image viewer
display image

See also **background**, **luminance**, **replace alpha**, **contrast**, and **extract scalar**.

SEE ALSO

The two BACKGROUND example scripts demonstrate the **composite** module.

compute gradient

NAME

compute gradient – compute gradient vectors for 2D or 3D data set

SUMMARY

Name	compute gradient
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field 2D / 3D scalar byte *any-coordinates*
Outputs	field *same-dimension* 3-vector real *same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
2D Height	float	0.5	0.0	1.0
Flip		toggle on	off	on

DESCRIPTION

The **compute gradient** module computes the gradient vector at each point in a 2D or 3D field of data. The gradient is can be used (e.g. by **gradient shade**) as a "pseudo surface normal" at each point.

A "nearest neighbor" approach is used to compute the gradient: in each direction, the component of the gradient vector is the difference of the *next* data and the *previous* data. In two dimensions, this can be pictured as follows:

x,y-1

positive Y direction — *x-1,y* *x,y* *x+1,y*

x,y+1

positive X direction

$\Delta x_{x,y} = data_{x-1,y} - data_{x+1,y}$

$\Delta y_{x,y} = data_{x,y-1} - data_{x,y+1}$

$\Delta z_{x,y,z} = data_{x,y,z-1} - data_{x,y,z+1}$ (<— for 3D data)

$\Delta z_{x,y} = 2D\ height$ (<— for 2D data)

This is backwards from the standard definition of a gradient which usually subtracts the previous value from the next. This was done because the standard defintion yields gradients in which the Z componant will typically point in the negative direction. While the standard definition is better known, the definition of "gradient" as used by this module produces more useful images since the Z componant of the gradient now points towards the eye instead of away from it. However, for the purists, there is a button called **Flip** (on by default) which lets you disable this "feature" and produce a typical gradient.

This module is slightly different from the **vector grad** module in a second respect. Since the intent of this module is to produce gradients useful to lighting calculations, the vectors are automatically normalized.

INPUTS

Data Field (required; field 2D / 3D scalar byte *any-coordinates*)

The input field may be either 2D or 3D. The data at each point of the field must be a single byte. The byte values will be interpreted as integers in the range 0..255.

PARAMETERS

2D Height (appears for 2D data only) Supplies the Z-coordinate of the gradient. It can be used the change the apparent height of the surface. A value of 1.0 is generally a very "rough" or "noisy" surface, whereas values approaching 0.0 will show little effect for shading.

Flip This toggle (on by default) causes the "correct" gradients to be flipped so that the Z axis generally points towards the eye, making gradients which are more useful for computing lighting calculations. If the "real" gradient is desired, then this button can be turned off and the gradients will not be flipped.

OUTPUTS

Data Field (field *same-dimension* 3-vector real *same-coordinates*)

The output field has the same dimensionality as the input field. For each element, the output data is a 3D vector of reals, representing the 3D gradient.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE 1

The following network shades a 2D image:

```
READ IMAGE
     |--------------
EXTRACT SCALAR     |    (choose 1 (= red))
     |             |
COMPUTE GRADIENT   |
     |    _________|
     |    |
GRADIENT SHADE
     |
DISPLAY IMAGE
```

EXAMPLE 2

The following network fragment shows how to get the same results as *compute gradient* using other modules:

```
READ FIELD
     |
FIELD TO FLOAT
     |
VECTOR GRAD
     |
FIELD TO MATH (multiply by -1.0)
     |
VECTOR NORM
```

EXAMPLE 3

The following network shades a 3D image:

compute gradient

```
                    READ VOLUME                GENERATE COLORMAP
                         |----------------|           |
                         |                |--------|  |
                    COMPUTE GRADIENT            COLORIZER
                         |   .______________________|
          |---------|    |   |
          |         GRADIENT SHADE
          |                |
          |--------------| |
          |              TRACER
          |                |
          |                |
          |         DISPLAY TRACKER
          |-----------|
```

RELATED MODULES

compute shade	
gradient shade	
display image	(for two-dimensional data)
alpha blend	(for three-dimensional data)
extract scalar	(to get a single scalar height field from an image)
vector grad	(to compute non-normalized true gradients)
vector norm	(to normalize vector fields)

LIMITATIONS

There may be algorithms better than "nearest-neighbor" for computing the gradient.

This module produces 12 bytes per pixel (voxel). For example, a 128 x 128 x 128 byte volume is about 2.1 MB before the gradient is computed. The **compute gradient** module produces a 25.2 MB internal data set from this data. This will have an adverse performance effect on systems whose physical memory is limited and may even exceed the available swap space.

SEE ALSO

The example scripts ANIMATED FLOAT and HEDGEHOG demonstrate the **compute gradient** module.

NAME

compute shade – combined colorizer / compute gradient / gradient shade module

SUMMMARY

Name	compute shade
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field 2D | 3D scalar byte *any-coordinates* colormap (*optional*) field 2D scalar float uniform (*optional, transform, autoconnect*)
Outputs	field *same-dims* 4-vector byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
ambient	float dial	0.10	0.00	1.00
diffuse	float dial	0.80	0.00	1.00
specular	float dial	0.00	0.00	1.00
gloss	float dial	20.00	0.00	50.00
lt theta	float dial	0.00	*unbounded*	*unbounded*
lt off_ctr	float dial	0.00	-90.00	90.00
2D height	float dial	0.5	0.00	1.00

DESCRIPTION

This module combines the functions of the **colorizer**, **compute gradient**, and **gradient shade** modules into a single, memory efficient module. These modules are used primarily to make shaded, ray traced images. The problem is that they are highly inefficient in terms of memory allocation:

- **colorizer** takes in 1 byte per voxel and outputs 4 bytes per voxel.
- **compute gradient** takes in 1 byte per voxel and outputs 12 bytes (3 floats).
- **gradient shade** outputs 4 bytes per voxel.

These three modules together produce 20 bytes for every input data set byte. It is for this reason that some people have experienced problems trying to render ray castings of large data sets. The tracing code itself is fairly computationally efficient; most of the system resources go to swapping data, rather than computing the image.

The **compute shade** module does gradient computation, colorizing, and shading on a per slice basis. It takes less time than running the original three modules in sequence.

compute shade is useful for extremely large data sets (> 100 ∗ 100 ∗ 100 voxels) that consume a system's memory.

INPUTS

Data Field (required; field 2D | 3D scalar byte *any-coordinates*)
The input data set to be shaded.

Colormap (optional; colormap)
The colormap input is optional. However, without it the image is grey scale with a linear opacity map

Data Field (optional; field 2D scalar float uniform)
This is a 4x4 transformation matrix that normally comes from either **display tracker**'s upstream data or **euler transformation** or **track ball**. Without this input, the light source is calculated as coming from the (object's) positive Z direction. This input port will connect automatically if the module immediately downstream outputs this same

transformation.

PARAMETERS

ambient The contribution of ambient (uniform background) lighting to the color. When this is set to 0.0, all surfaces facing away from the light source are black. When this is set to 1.0, surfaces appear in their own colors, with no shading information present. The range is 0.0 to 1.0; the default is 0.10.

diffuse The contribution of diffuse (directional) lighting to the color. The range is 0.0 to 1.00; the default is 0.80.

specular The contribution of specular lighting to the color. The range is 0.0 to 1.0; the default is 0.0.

gloss The sharpness of the specular highlight. The larger this value, the smaller and sharper the specular highlights. The range is 0.0 to 50.0; the default is 20.0.

lt off-ctr The angle between the light source and the positive Z axis. (The positive Z axis is perpendicular to the plane of the screen.) The range is 0.0 to 1.0; the default is 0.0.

lt theta The angle between (1) the projection of the light source on the XY plane and (2) the positive Y axis. This value measures how much an off-center light source "swings around" the Z-axis. The range is unbounded; the default is 0.0.

With **lt theta** = 0.0 and **lt off-ctr** = 0.0, the light source is coming straight from the eye perpendicular to the data. A positive **lt off-ctr** value moves the light source up (in the positive Y direction); a negative value moves it down.

2D height (appears for 2D data only). Supplies the Z-coordinate of the gradient. It can be used to change the apparent height of the surface. A value of 1.0 is generally a very rough or "noisy" surface, whereas values approaching 0.0 will show little effect for shading.

The equation for calculating the intensity of light reflected by a spot of surface is:

$$(int_{amb} * ambient) + (int_{diff} * diffuse * \cos(phi)) + (int_{diff} * specular * \cos^{gloss}(lt\ off\text{-}ctr))$$

In performing this computation, **compute shade**:

- Assumes that int_{amb} and int_{diff} are both maximal (1.0).
- Uses *lt theta* and *lt off-ctr* to compute *phi*, the angle between the surface normal (gradient vector) and the light source. The quantity **cos**(*phi*) is the attenuation (reduction) factor for the directional (diffuse) light.
- Computes the quantity $\cos^{gloss}(lt\ off\text{-}ctr)$, the attenuation factor for the specular highlight.

OUTPUTS

Data Field (field *same-dims* 4-vector byte)
Each voxel becomes a colorized, shaded voxel. The output has the same dimensions as the input. 2D output can be sent to **image viewer**. 3D output can be sent to the **tracer** or **cube** modules.

EXAMPLE 1

This is the fastest way to generate a lighted color image from a uniform byte field. Note the upstream transform connections from **display tracker** to **tracer**, relayed up to **compute shade**. These connections occur automatically.

```
GENERATE COLORMAP        READ VOLUME
          |_______      _______|
                  |   |
        |------|  |   |
        |    COMPUTE SHADE
        |           |
        |---------| |
        |       TRACER
        |           |
        |    DISPLAY TRACKER
        |-------|
```

EXAMPLE 2

This is a good network for making a ray traced animation where the volume rotates, and the light source stays fixed relative to the eye. The **animated float** module controls one of the axis parameters for **euler transformation** (this gives the rotation). The **image viewer**'s **Action** menu is used to store the frames of the flipbook animation.

This may take a while for large data sets since, for every angle, the **compute shade** module will refire. To avoid this, disconnect the **euler transformation** module from **compute shade**. The disadvantage to this is that the light source stays fixed relative to the object, not the eye.

```
                  GENERATE COLORMAP       READ FIELD
ANIMATED FLOAT            |                   |
      |                   |   ________________|
EULER TRANSFORMATION      |   |
      |_______________    |   |
      |              |    |   |
      |            COMPUTE SHADE
      |__________________ |
                        | |
                      TRACER
                        |
                   IMAGE VIEWER
```

RELATED MODULES

colorizer
compute gradient
gradient shade

SEE ALSO

The module man pages for **colorizer**, **compute gradient**, and **gradient shade**.

The example scripts COMPUTE SHADE and TRACER demonstrate the **compute shade** module.

contour to geom

NAME

contour to geom – create geometry of 2D or 3D scalar field contour slices

SUMMARY

Name	contour to geom				
Availability	Imaging, Volume, FiniteDiff module libraries				
Type	mapper				
Inputs	field 2D / 3D scalar *any-data any-coordinates*				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	threshold	float dial	128.0	*unbounded*	*unbounded*

DESCRIPTION

The **contour to geom** module finds and creates contour lines of similar value in a scalar field, then outputs the result as an AVS geometry. The contour lines can be disjoint. The **threshold** parameter controls the contour level. **contour to geom** handles 2D and 3D datasets, and uniform, rectilinear, and irregular grids.

INPUTS

Data Field (required; field 2D / 3D scalar *any-data any-coordinates*)
The input field is 2D or 3D scalar field, containing any data, using any coordinate system.

PARAMETERS

threshold A floating point dial that controls what value the contour lines are created for. The default is 128.0. This parameter is unbounded, with no minimum or maximum.

OUTPUTS

Geometry The contour lines are represented as an AVS geometry.

EXAMPLE 1

The following network finds a contour on the red channel of the mandrill.x image.

```
  READ IMAGE
      |
EXTRACT SCALAR
      |
CONTOUR TO GEOM
      |
GEOMETRY VIEWER
```

EXAMPLE 2

The following network finds the magnitude of a vector field and contours it.

```
                READ FIELD
      ______________|____________
      |                         |
  VECTOR MAG                    |
      |                         |
      |                   VOLUME BOUNDS
      |                         |
CONTOUR TO GEOM                 |
      |-------------|-----------|
                    |
              GEOMETRY VIEWER
```

RELATED MODULES

ip contour

Modules that can process **contour to geom** output:

geometry viewer
render geometry

SEE ALSO

Two CONTOUR GEOMETRY example scripts demonstrate the **contour to geom** module.

contrast

NAME

contrast – perform linear transformation on range of field values

SUMMARY

Name	contrast
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
cont_in_min	float	0.0	*none*	*none*
cont_in_max	float	255.0	*none*	*none*
cont_out_min	float	0.0	*none*	*none*
cont_out_max	float	255.0	*none*	*none*

DESCRIPTION

The **contrast** module transforms all the values in a field. Two different types of transformation take place:

- **Linear transform:** All values that fall within the "input range" specified by the **cont_in_min** and **cont_in_max** parameters are transformed linearly to the "output range" **cont_out_min** .. **cont_out_max**.

$$new\_value = \frac{(cont\_out\_max - cont\_out\_min) * (value - cont\_in\_min)}{cont\_in\_max - cont\_in\_min} + cont\_out\_min$$

  (More precisely, this is an *affine transformation*.) In essence, this transformation "stretches" or "compresses" one specified range of data to fit another specified range.

- All values that fall outside the specified input range are "clamped" to the limit values of the output range.

The **contrast** module typically is used to remove low-level noise from images and volumes, or to increase the contrast in faded images and volumes.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input data may be an AVS field of any dimensionality.

PARAMETERS

cont_in_min
Specifies the bottom of the range of input values that will be transformed linearly.

cont_in_max
Specifies the top of the range of input values that will be transformed linearly.

cont_out_min
Specifies the bottom of the range of output values. All values ≤ **cont_in_min** will be transformed to this value.

cont_out_max
Specifies the top of the range of output values. All values ≥ **cont_in_max** will be transformed to this value.

OUTPUTS

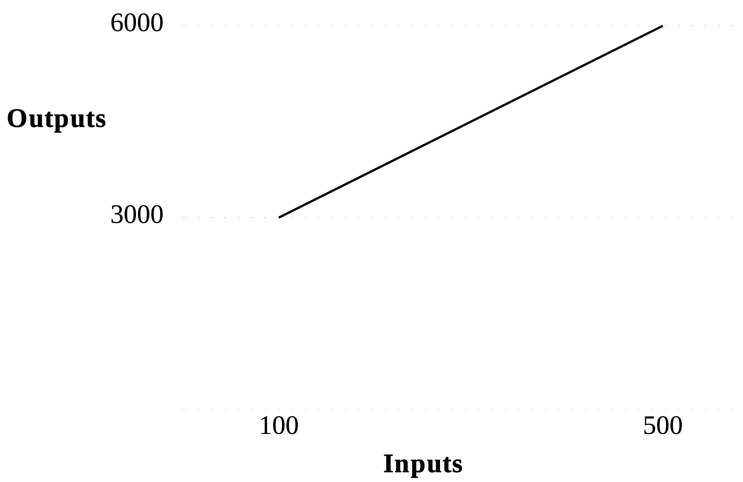
Data Field The output field has the same dimensionality and type as the input field.

If the input field has byte values, appropriate new **min_val** and **max_val** values are written to the output field.

EXAMPLE 1

The following diagram shows how field values are transformed given these parameters:

cont_in_min = 100
cont_in_max = 500
cont_out_min = 3000
cont_out_max = 6000



You can use **contrast** to make a negative out of an image by "flipping" the output values (e.g. **cont_out_min** = 255; **cont_out_max** = 0).

EXAMPLE 2

The following network reads in an image, extracts the red, green and blue channels, contrast stretches only the red channel, and then uses **combine scalars** to pack the seperate channels back into an image.

```
                         READ IMAGE
                             |
         |---------------------|---------------------|
         |                     |                     |
   EXTRACT SCALAR       EXTRACT SCALAR       EXTRACT SCALAR (red)
         |                     |                     |
         |                     |                     |
         |                     |                  CONTRAST
         |                     |                     |
         |---------------------|---------------------|
                             |
                       COMBINE SCALARS
                             |
                             |
                        DISPLAY IMAGE
```

RELATED MODULES

ip linremap

Modules that could provide the **Data Field** input:

read volume

SEE ALSO

The example script CONTRAST demonstrates the **contrast** module.

NAME

convolve - apply a signal processing filter to 2D field

SUMMARY

Name	convolve		
Availability	Imaging module library		
Type	filter		
Inputs	field 2D n-vector *any-data any-coordinates* (image) field 2D scalar float uniform(convolution filter)		
Outputs	field of same type as input		
Parameters	*Name*	*Type*	*Default*
	normalize	boolean	on

DESCRIPTION

convolve takes a signal processing filter and applies it to a source field to produce a destination image. Typically, the source and destination fields will be AVS *images*, but they might also be 2D slices of 3D fields. Filters can be produced by the module **generate filters** or by user-written modules.

Convolution is a frequently used technique in signal and image processing. Applying a "high pass" filter, such as a Laplacian, to an image will emphasize edges in the image. On the other hand, a "low pass" filter, such as a Gaussian, will smooth images. These techniques can be helpful in removing artifacts from images, and in compensating for the inherently discrete nature of digital data.

The filter must be a 2D array of floating-point values. The source field must also be 2D, but it can hold any size vector of any data-type. The field output by **convolve** will be the same type as the source field. The filter must be smaller than the field it is being applied to. **convolve** typically normalizes filters to the range 0 to 1 before applying them to an image.

Filters are applied as follows, taking a typical case in which a small, 10x10 filter is applied to a larger, 256x256 image: One can imagine the filter sitting on top of the source image centered on one pixel in the image, say (45,45). Each of the 100 values in the filter array is multiplied by the value of the pixel *beneath* it. These 100 products are then added together, and their sum becomes the value of the pixel at (45,45) in the destination image. Then the filter is shifted so that it is centered over the next pixel. This process is repeated to produce each element in the output image.

This approach is known as the "sliding window" method. It is an N x M algorithm, where N is the number of elements in the convolution filter and M is the number of elements in the image. As a result, it is recommended that filters be small; larger filters (i.e. above 12x12) require a great deal of computation.

convolve accepts data of any type. In the case of an image, which is a 2D field of vectors each containing 4 bytes, **convolve** disregards the alpha bytes and separates the red, green and blue bytes. Then it applies the filter separately to each color field, before reassembling the bytes into image format. In the case of non-image data, for example a 2D field of 5-vector floats, **convolve** handles one component of the vector at a time. All data-types are converted to floats during computation and then converted back in **convolve**’s output.

To avoid edge effects, a border around the perimeter of the source field is not convolved. The border’s width is half the width of the filter.

convolve

INPUTS

Data Field (required; field 2D n-vector *any-data any-coordinates*)

A 2D AVS field, typically an image, to be convolved. The field is input through the right input port.

Filter Field (required; field 2D scalar float uniform)

A 2D AVS field of floating-point scalar values. Filters can be created by using the module **generate filters**, which produces Gaussian, Laplacian, and other filters. Alternately, you can write your own modules to generate filters. Filters are input through the left input port.

PARAMETERS

normalize (toggle)

If **normalize** is selected, filters are normalized such that the sum over all the elements in the filter equals 1. In other words, each element in the filter is divided by the sum.

OUTPUTS

Output Field

The output field is the same type as the input data field.

EXAMPLE

The following network reads in an image and a filter, convolves the two, and displays the resulting image:

```
GENERATE FILTERS                READ IMAGE
        |                             |
        |-------------|  |------------|
                      |  |
                   CONVOLVE
                       |
                       |
                 IMAGE VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read image
pixmap to image
orthogonal slicer
any other module which outputs a 2D field

Modules that could provide the **Filter** input:

generate filters
any (user written) module which outputs a 2D scalar float field

Modules that can process **convolve** output:

display image
image viewer
any other module which takes a 2D field as input

Modules that could be used instead of **convolve**:

ip convolve

SEE ALSO

The example scripts CONTRAST, GENERATE FILTERS, and SOBEL demonstrate the **convolve** module.

NAME

create geom – generate & manipulate geometry objects such as lines, arcs, surfaces

SUMMARY

Name create geom

Availability UCD, FiniteDiff module libraries

Type data

Inputs upstream geometry (*required, invisible, autoconnect*)
upstream transform (*required, invisible, autoconnect*)

Outputs geometry
field 3D irregular float (sampler field)

Parameters

Name	*Type*	*Default*
Action Menu	choice	ADD
SubAction Menu	choice	DONE
Output Samplers	boolean	false
Output Object	choice	none

DESCRIPTION

The **create geom** module allows the user to interactively create geometry objects such as Points, Polylines, Arcs, Circles, Surfaces, Revolutions, and Extrusions. It also provides a set of operations to modify created objects, such as Insert Vertex into Polyline, Close Polyline, Move Vertex, Move Object, Flip Normals of the Surface, and Delete object.

The objects **create geom** creates can be used for any purpose. One particularly useful application is to use the objects as samplers for the various vector mapping modules. The **create geom** module can output a sampler field that contains verticies of all or only the current geometry object. This field can be used as an optional input to the **hedghog**, **streamlines** and **particle advector** modules.

MODES

The **create geom** module provides three different modes of user interaction with the Geometry Viewer window: *Pick Mode, Select Mode* and *Normal Mode.*

Picking

Some operations, like **ADD** and **MODIFY** require "picking" a location in the Geometry Viewer window. For example, the sequence **ADD, Point** puts the module into the *Pick Mode.* To pick, press (or hold down while moving) the left mouse button. It is important to notice that picking works only on the *existing geometry objects.* This means that to add objects, the user must have some other geometry objects already drawn in the Geometry Viewer window. For example, the **generate grid** module can be used to create coordinate planes.

Selecting

Some operations, like **CONSTRUCT**, **MODIFY**, and **DELETE** require "selecting" an object that they will be applied to. To select an object, set the **Select** button in the SubAction Menu. This puts the module into *Select Mode.* Next, point to an object in the Geometry Viewer window and press (or hold down while moving) the left mouse button. The selected object is colored in red. The selected obect becomes the Current Object.

There are also operations, like **MODIFY, Move Object** that require selecting both an object and a vertex on the object. Pointing to an object while in Select Mode makes the closest vertex of the Current Object become the Current Vertex. The Current

Vertex is marked with the red "+" symbol.

The module mantains the Current Object and Current Vertex when switching between operations in the Action menu and the SubAction menu.

Normal

To "unselect" Current Object and Current Vertex, set **DONE** in the SubAction menu. This puts the module into the *Normal Mode.* In Normal Mode, you can use Geometry Viewer operations to control the objects.

ADD

Selecting **ADD** in the Action Menu brings up the SubAction Menu to create Point, Polyline, Arc 3 point, or Circle 3 point objects.

For example, selecting **Polyline** puts the module into Pick Mode. It expects you to pick a location in the Geometry Viewer window on the existing geometry. The module will interpret this location as the next vertex of the polyline. A sequence of picking in the Geometry Viewer window will produce the segments of the polyline. Note that if the Current Vertex was already selected, the module will use it as the starting vertex of the polyline.

Switching to **Arc 3 point** in the SubAction menu causes the module to add an arc to the current polyline. The first point of the arc will be the end of the current polyline (Current Vertex), and the user must pick two more points.

CONSTRUCT

The **CONSTRUCT** option in the Action menu creates surfaces from existing polylines and arcs. By default, it puts the module into **Select** mode. You have to select an existing polyline or arc (or combination of both). After selecting an object, you use the SubAction menu to choose the type of surface to generate:

Surface will create a surface bounded by the selected polyline.

Revolution

will create a surface of revolution from a selected object about a specified axis.

Extrusion will extrude a selected object in the specified direction with the specified length. It also can scale the cross section of the extrusion with a given scale. The number of cross sections is controled by the **N segment** parameter.

Cap Surface

creates "caps" for extrusions and revolutions. The Current Object should be an extrusion or revolution surface. The Current Vertex defines the location of the cap surface. The cap surface will be made a child object of the parent extrusion or revolution, which means it will be transformed with its parent.

MODIFY

The **MODIFY** option in the Action Menu changes existing objects. By default it puts the module into **Select** mode. You must select an existing object.

Insert Vertex

will create a new vertex at the picked location and insert it before or after the Current Vertex of the Current Object.

Close Polyline

connects the last vertex of the polyline to its first vertex.

Move Object
moves the Current Object from the Current Vertex location to the picked location.

Move Vertex
moves the Current Vertex of the Current Object to the picked location.

Flip Normals
changes the orientation of the Current Object.

DELETE

The **DELETE** option in the Action Menu deletes objects. By default it puts the module into **Select** mode. You must select the object you want to delete. Then, choose **Delete Object** in the SubAction menu.

REDRAW

The REDRAW option just redisplays all the existing objects.

INPUTS

Upstream geometry (required, invisible, autoconnect)
A data structure from the **geometry viewer** module that supplies the left mouse button picking information **create geom** needs. Note that this is required, and that the connection will be made automatically and invisibly with the **geometry viewer** module. The information may be relayed through the vector mapping module.

Upstream transform (required, invisible, autoconnect)
A data structure from the **geometry viewer** module that supplies the object transformation information **create geom** needs. Note that this is required, and that the connection will be made automatically and invisibly with the **geometry viewer** module. The information may be relayed through the vector mapping module.

PARAMETERS

Action Menu
The choice of operations: **ADD, CONSTRUCT, MODIFY, DELETE,** or **REDRAW**.

SubAction Menu
This menu is different for each of the Action menu selections. See the descriptions above.

Output Samplers
A boolean that controls whether or not to output a sampler field. The default is **off**.

Output Object
Chooses between **Current Object** and **All Objects** for output of a sampler field.

axis Chooses which axis, **X**, **Y**, or **Z**, that **Extrusion** will use for direction, or **Revolution** will use for rotation.

Tolerance A float dial used by **Arc**, **Circle**, and **Revolution**. Specifies the maximum deviation of the arc segment from the line segment with which it is approximated.

Length Specifies the length of an **Extrusion**.

Scale Specifies the scale factor for the last cross section **Extrusion**.

Nsegement
Specifies the number of the intermediate cross sections of an **Extrusion**.

OUTPUTS

Geometry (geom)
The output is a geometry containing the created objects.

Sampler Field (field 3D Irregular)
The output is a sampler field containing the locations of the verticies of the geometry objects. This field can be used as an optional input to the **hedghog**, **streamlines**, and **particle advector** modules. **Output Samplers** must be selected to produce this field.

EXAMPLE 1

The following network is used to draw a simple geometry. The resulting geometries are saved permanently using the **geometry viewer**'s **Save Object** button. The **generate grid** module is necessary to create some geometry in the output window (in this case verticies and lines) that can be picked, so that **create geom** can position its objects in space. This person is also using **set view** to quickly line up their view of the grid on the X, Y, and Z axis.

```
CREATE GEOM      GENERATE GRID
      |                |
      |----------|-----|    SET VIEW
                 |
       GEOMETRY VIEWER
```

EXAMPLE 2

The following example uses the **create geom** module to generate sample points that are located on an isosurface. These sample points are used as input to the **stream lines** module. Thus, one generates streamlines upon an isosurface.

```
CREATE GEOM               READ FIELD
      |                       |
      |              ----------------------
      |              |                    |
      |       EXTRACT VECTOR       EXTRACT SCALAR
      |              |                    |
      |----------|   |               ISOSURFACE
                 |   |                    |
            STREAM LINES                  |
                 |                        |
                 |----------------------|
                           |
                           |
                   GEOMETRY VIEWER
```

RELATED MODULES

Modules that can process **create geom**'s output:

tube
geometry viewer

Modules that can be used with **create geom**:

generate grid

set view
streamlines
particle advector
hedgehog

crop

NAME

crop – extract subset of elements from a field

SUMMARY

Name	crop
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field 2D / 3D *n-vector any-data any-coordinates*
Outputs	field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
min x	int	1st indx	1st indx	last indx
max x	int	last indx	1st indx	last indx
min y	int	1st indx	1st indx	last indx
max y	int	last indx	1st indx	last indx
min z	int	1st indx	1st indx	last indx
max z	int	last indx	1st indx	last indx
size to fit	toggle	off	off	on

DESCRIPTION

The **crop** module changes the size of a field by extracting the data within a specified range of elements. This process is analogous to "cropping" a photographic image.

This module is useful for subsampling the data without changing it (e.g. by interpolation). It preserves the resolution of the data, but may change its aspect ratio. Typical uses are to eliminate uninteresting portions of the data and to increase processing speed by reducing the amount of data.

Once a field is input to **crop**, the module's min and max dials are set to the min and max indices of the field's data array. From then on the dials cannot be turned lower than the min index, or higher than the max index (min cannot equal max, and min must be less that or equal to max-1). If you use the Dial Editor to change these values they would "snap back" to their original values. This makes sense, because you can only take a subset from the field within the field's array indices.

When the minx and max dials are set to the same value, **crop** first tries to set max=max+1. If max is already at its maximum index, then **crop** sets min=min-1.

INPUTS

Data Field (required; field 2D / 3D *n-vector any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

Note that the parameters indicate *positions* of elements in the field — they have nothing to do with the *values* of field elements.

min x Specifies the lower bound array index in the field's first dimension.

max x Specifies the upper bound array index in the field's first dimension.

min y Specifies the lower bound array index in the field's second dimension.

max y Specifies the upper bound array index in the field's second dimension.

min z (Does not appear for 2D input data sets) Specifies the lower bound array index in the field's third dimension.

max z (Does not appear for 2D input data sets) Specifies the upper bound array index in the field's third dimension.

size to fit In the default mode, **crop** does not change the extent information in the output field structure. This is because you may wish to merge cropped interpretations of a data with interpretations of the original data. By retaining the original extent information, the cropped version of the data causes the Geometry Viewer to adjust its extents appropriately. The points array contains the actual cropping information. With uniform fields, when the **size to fit** button is turned on, the points array gets copied to the extents array which has the effect of causing the Geometry Viewer to scale the window to exactly fit the cropped data rather than the extents of the original, uncropped data.

OUTPUTS

Data Field The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced.

The **min_val** and **max_val** attributes of the output field are invalidated.

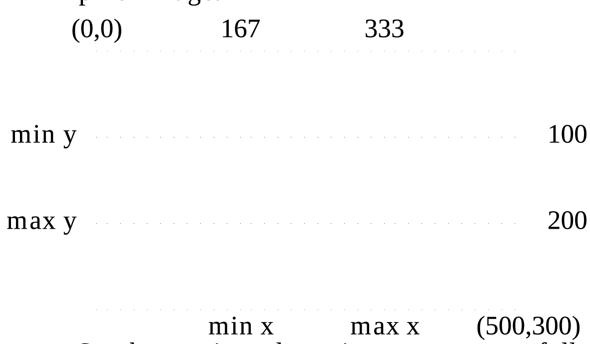
EXAMPLE 1

The following network reads a 2D field (image), crops it and displays the result:

```
  READ IMAGE
       |
       |
     CROP
       |
       |
 DISPLAY IMAGE
```

EXAMPLE 2

Suppose you want to process the middle third of a field that contains an 500x300 pixel image:



Set the x-axis and y-axis parameters as follows:

min x = 167
max x = 333
min y = 100
max y = 200

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
filter modules

Modules that could be used in place of **crop**:

downsize
interpolate
average down

Modules that can process **crop** output:

colorizer
gradient shade
arbitrary slicer
orthogonal slicer
any other filter module

LIMITATIONS

crop works for 2D and 3D data sets only.

SEE ALSO

The example script CROP demonstrates the **crop** module.

NAME

cube – perform ray-traced volumetric rendering on volume data

SUMMARY

Name	cube
Availability	Volume, FiniteDiff module libraries
Type	mapper
Inputs	field uniform 3D byte scalar struct substances (substance table, optional) field 2D scalar float (transformation matrix, optional, autoconnect) field 2D scalar float (light source transformation matrix, optional)
Outputs	field 2D uniform 4-vector byte (image)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Mode	choice	texture		
width	int typein	64		
height	int typein	64		
outline	toggle	on		
shaded	toggle	on		
trilinear	toggle	off		
xrot	float	0.0	-180.0	180.0
yrot	float	0.0	-180.0	180.0
zrot	float	0.0	-180.0	180.0
distance	float	0.0	0.0	100.0

DESCRIPTION

cube belongs to a family of modules (along with **x-ray** and **tracer**) that render volume data. **cube** takes a volume, which can be visualized as a block of cubic "voxels" (volume elements), and generates a 2D image using ray tracing. Each voxel in the volume has color and opacity values associated with it. This module is an AVS module version of the SunVoxel tool called 'cube' found in the SunVision visualization package.

There are four modes of rendering with **cube**: *texture, maximum, ray cast,* and *create surfaces*. *texture* mode is similar to the AVS module **brick** in that it shows only the texture-mapped exterior surfaces of the volume. *maximum* mode is similar to the maximum option of the **x-ray** module except that **cube** allows for off-axis rotations. *ray cast* and *create surfaces* mode are ray casting algorithms for rendering surfaces at different density levels. The surfaces are classified as substances by their value. Substances are specified by using the **edit substances** module.

The ray casting method is as follows. For each pixel in the output image a ray is "shot" into the volume. A substance table, supplied by the **edit substances** module, is used to define the voxel intensity levels to which the intersecting rays are sensitive. Each voxel the ray passes through is evaluated to see if the intensity level has left one substance classification and entered another. If this is determined to have happened, then a surface is assumed to exist at that point and is rendered according to the surface properties defined in the substance table.

This renderer is most effective when used on data which is readily classified into distinct material types. In medical imaging, these types might correspond to "skin", "muscle", and "bone". In non-destructive evaluation, the types might be described for "air", "engine wall", "engine interior". If the data is more continuous, such as temperature in a room, then the **tracer** module may be more appropriate since it deals better with continuous, rather than discrete data.

Volumetric rendering allows you to penetrate beneath the surface of 3D data, and see depths surrounded by "translucent" outer layers. The degree of opacity and color for each substance can be controlled by changing their values in the substance table.

Another feature of **cube** is an optional oblique slicing plane. The plane's position can be controlled with three sliders (one for each cardinal axis) for orientation and one slider for distance into the volume. All the rendering modes are affected by this slice plane. Typically, you go into the fast *texture* mode to set the position of the slice plane and then switch over to one of the more expensive modes for a clearer picture.

INPUTS

Data field (required; field 3D byte scalar)

The input data must be a scalar 3D uniform byte field. Data from other formats may be converted using the **extract scalar** module (for N-vector data), or the **field to byte** module (for data which is not initially in byte format).

Substance Table (optional; struct substance)

This is a user defined data type (specified in *$AVS_PATH/include/substances.h*) which contains the substance table information necessary for the ray-cast and create_surfaces renderering modes. Although you can supply your own substance table, it is easier to use the table provided by the **edit substances** module.

Transformation matrix (optional; field 2D scalar float, autoconnect)

The center port on **cube** can receive a 4x4 transformation matrix describing rotations and translations to apply to the volume data. This matrix (field 2D scalar float) can come from an appropriate downstream module such as **display tracker**, or from the **euler transformation** or **track ball** modules. These mechanisms allow you to rotate the volume in 3-space.

For example, when the **cube** module is connected to the **display tracker** module in a network, **display tracker** sends a transformation matrix back to this port on **cube**. This allows you to directly manipulate the volume by moving the mouse in **display tracker**'s window, using the "virtual spaceball" paradigm. For a more detailed description of direct manipulation see the section titled "Transforming Objects" in the "Geometry Viewer" chapter of the *AVS User's Guide.*

Light source transformation matrix (optional; field 2D scalar float)

The leftmost port on **cube** can receive a 4x4 transformation matrix describing rotations and translations to apply to the light source. This matrix (field 2D scalar float) can come from an appropriate downstream module such as **display tracker**, or from the **euler transformation** or **track ball** modules. These mechanisms allow you to rotate the light source around in 3-space. The light source is only used when the *shaded* option is selected and is never used when rendering in *maximum* mode.

PARAMETERS

Rendering Mode (choice: texture, maximum, ray cast, create surfaces)

These are the four rendering modes produced by this module.

texture texture maps the data onto the exterior surfaces of the volume. This is similar to the AVS **brick** module.

maximum mode is similar to the maximum option of the **x-ray** module except that **cube** allows for off-axis rotations. It selects the maximum value encountered for each ray as it passes through the volume.

ray cast and *create surfaces* mode are ray casting algorithms for rendering surfaces at different density levels. The use the Substance Editor to define what levels, colors, and opacities those surfaces are at.

create surfaces mode takes longer to render initially because it is storing the list of surfaces encountered by each ray. It can then use this information in subsequent renderings to allow you to rapidly change surface opacities and colors without "re-rendering" the entire scene. If you change orientation, add new surfaces, or change the intensity level for any of the existing surfaces, then it does the initial (longer to compute) set up again.

width (integer typein)

Value which determines the width in pixels of the output image. Another way of thinking of this is the width determines the number of rays that will be projected into the volume along the x direction. This changes the shape of the window through which you view the volume.

Note: Downstream modules such as **display tracker** have controls that will enlarge the image in the output window without computing at higher resolution.

height (integer typein)

Value which determines the height in pixels of the output image. Another way of thinking of this is the height determines the number of rays that will be projected into the volume along the y direction. This changes the shape of the window through which you view the volume.

outline (toggle)

Allows you to draw a white wireframe box around the exterior of the volume. This is on by default.

shaded (toggle)

Toggles between performing shading computations against the derived surfaces or just using the assigned surface color. This is on by default. There is little computational overhead involved with performing these shading computations.

trilinear (toggle)

Allows you to select between sampling the volume using a fast, nearest neighbor (point) sampling technique (the default) or choosing a more accurate trilinear sampling. When on, it takes roughly four times longer to compute an image. This method produces a more accurate rendering of the volume.

xrot (float point slider)

This slider controls the rotation of an oblique slice plane through the data set. In particular, *xrot* controls the rotation of the slice plane around the x-axis (the horizontal one).

yrot (float point slider)

This slider controls the rotation of an oblique slice plane through the data set. In particular, *yrot* controls the rotation of the slice plane around the y-axis (the vertical one).

zrot (float point slider)

This slider controls the rotation of an oblique slice plane through the data set. In particular, *zrot* controls the rotation of the slice plane around the z-axis (the clockwise one facing the screen).

dist (float point slider)

This slider control the distance into the volume that the oblique slice plane passes. This dial can be used in combination with the xrot, yrot, and zrot dials.

OUTPUTS

Data Field (field 2D uniform 4-vector byte)

The output field is an AVS image.

EXAMPLE 1

The following network reads a scalar 3D uniform byte field (a volume) and ray traces it. The module **euler transformation** allows you to rotate the volume to produce views from any angle. If the input was not originally byte values, it could be converted with the **field to byte** module. Note: Because the **edit substances** module is not present, the ray-cast and create-surfaces mode cannot be used.

```
                                           READ VOLUME
EULER TRANSFORMATION                            |
                |                               |
                |----------------------------|  |
                                             |  |
                                            CUBE
                                              |
                                        DISPLAY IMAGE
```

EXAMPLE 2

The following network is similar to the previous, except that this network uses the module **display tracker**, which allows you to directly manipulate the volume being viewed by moving the mouse. **display tracker** feeds information on the mouse's movements back to **cube** through its center input port through an invisible upstream transformation connection.

Also, the **edit substances** module is now being used to create a substance table so that the ray-cast and create-surfaces rendering modes can be used. The output from **edit substances** can also be fed into **field legend** so that the substance table can be viewed relative to the voxel values they represent.

```
                          READ FIELD
                               |
       EDIT SUBSTANCES  _______|
                |_______|____  |
                |   ____|    | |
                |  |        CUBE
                |  |         |
         FIELD LEGEND   DISPLAY TRACKER
```

RELATED MODULES

Modules that could be used in place of **cube**:

brick
excavate brick
x-ray
tracer

Modules that could provide the **Substance Table** input:

edit substance

Modules that could provide the **Data Field** input:

read volume

read field
any other module which outputs a 3D byte scalar field.

Modules that could provide the **Transformation Matrix** input:

euler transformation
track ball
display tracker (using upstream data)

Modules that can process **cube**'s output:

display tracker
display image
image viewer
image to postscript
any other module which takes an AVS image as input.

SEE ALSO

$AVS_PATH/include/substances.h contains the substance table user defined data definition.

The example script CUBE demonstrates the **cube** module.

data dictionary

NAME

data dictionary – read external data file using a form specification

SUMMARY

Name	data dictionary		
Availability	Imaging, UCD, Volume module libraries		
Type	data		
Inputs	none		
Outputs	field		
Parameters	*Name*	*Type*	*Default*
	Select Data File	Browser	
	read form	toggle	false
	header information	oneshot	false
	send data	oneshot	false
	Browser for File *n*	toggle	false

DESCRIPTION

Using a data form specification created with the **file descriptor** module, the **data dictionary** module reads in an external format data file and converts it into an AVS field.

The general order of the operations is:

1. Press the **read form** button. This attaches the file browser to the **read form** function.
2. Use the **Select Data File** browser to specify a data form file. Upon selecting or typing in a filename, the data form will be read.
3. Data forms require one or more input files. For example, there may be one input file containing data, and another input file containing coordinate information. The number of input files required is shown by the number of **Browser for File** *n* buttons.

   For each input file required, press **Browser for File** *n* and then use the **Select Data File** browser to establish which actual file corresponds to file *n*. Work down the list establishing these logical file to real file correspondences. No data will be read yet.
4. If you wish, examine the contents of the data form with the **header information** function. If the data form specifies that part of the input parsing instructions will come from the input file itself (e.g., the dimensions of the data), then the input file(s) will be read in at this point according to the correspondences established in step 3.
5. When all logical file to real file correspondences have been defined, press the **send data** button to actually read the input data file(s) and convert it to an AVS field using the rules in the data form.

PARAMETERS

Select Data File

A file browser widget. This file browser is shared among the **read form** and **Browser for File** *n* parameters. The correct order to select these options is: specify which other parameter the file browser will represent by pressing one of **read form** or the various **Browser for File** *n* parameters. Then, select a file using this file browser widget.

read form A toggle button that sets the current state of the data file browser. After this is selected, use the **Select Data File** browser to specify a form file to read. It will be read immediately upon specification. You must read in a form before you can logically specify a **Browser for File**, because the data form may contain definitions for multiple input files.

header information
A oneshot button that displays a scrolling list with the field header information of the file being read in.

send data A oneshot button that causes the data to be read from the external file(s) and converted into a field. This field is then output on the module's output port.

Browser for File *n*
A set of buttons that set the current state of the data file browser. First press one of these **Browser for File** *n* buttons, then use the **Select Data File** browser to define which real file will be used as file *n*. Specify a logical file to real file correspondence for each required input file.

OUTPUTS

Data Field (field)
The output is the field containing data held by the external data file being read.

EXAMPLE

There are example forms for the **data dictionary** module in the directory *$AVS_PATH/data/adia*. The example form *dat_format* can be used to read in AVS .dat format files. The example form *x_format* can be used to read in AVS .x format files.

This simple example displays an image.

```
DATA DICTIONARY
      |
 DISPLAY IMAGE
```

RELATED MODULES

file descriptor

SEE ALSO

The "AVS Data Interchange Application" discussion in the *AVS Applications Guide* describes using **file descriptor** and **data dictionary** to import external format data files into AVS.

Data Viewer

NAME

Data Viewer – simplified pulldown menu interface to build AVS networks

SUMMARY

Name	Data Viewer
Type	data output
Inputs	*none*
Outputs	*none*
Parameters	*none*

DESCRIPTION

The **Data Viewer** is a simplified user interface to the Application Visualization System's most commonly-used scientific visualization techniques.

You normally construct visualization networks with the Network Editor. The individual modules required to perform the visualization are selected from the Network Editor's Palette, dragged one-by-one into the Workspace, then connected together.

The Data Viewer takes an alternate approach to network construction. Rather than building networks manually, the Data Viewer provides a pulldown menu interface from which you select input, filtering, mapping, and data output techniques. Each of these choices represents a predefined subnetwork. Behind the scenes, the Data Viewer automatically selects the corresponding modules and constructs the visualization network.

This approach preserves a large measure of the flexibility and dyanamics of the Network Editor, while eliminating much of the detail knowledge of network structure, data types, and mouse button mechanics required to use it.

The Data Viewer's predefined visualization techniques can manipulate uniform and curvilinear, scalar and vector field data, as well as unstructured cell data (UCD). It is primarily useful to the new AVS user learning visualization techniques, terminology, and the AVS interface.

The **Data Viewer** module does not connect to other modules in a network through standard data flow connections Rather, it performs its functions by sending CLI commands to the AVS kernel.

The Data Viewer can also be invoked as an application from the main AVS Applications menu.

SEE ALSO

The Data Viewer is fully described in the *AVS Applications Guide.*

RELATED FILES

The Data Viewer uses networks found in *$AVS_PATH/networks/dv*. The menus are defined in *$AVS_PATH/networks/dv/data_viewer.men*.

NAME

dialog box – use a long dialog box to create a long string

SUMMARY

Name	dialog box		
Type	data (coroutine)		
Inputs	none		
Outputs	string		
Parameters	*Name*	*Type*	*Default*
	Edit	boolean	off

DESCRIPTION

The **dialog box** module creates long strings that are sent as parameters to downstream modules that accept string parameters. These strings are useful as long expressions, lists of integers, etc., where the normal string typein widget provided by the downstream module is too short. (Typein widgets do not scroll.)

To connect the string output, you will need to make the downstream module's string parameter port visible. The module must be instantiated. Click on the module icon's dimple to raise its Module Editor. Click on the string parameter in question to raise the Parameter Editor. Lastly, click on **Port Visible**.

PARAMETERS

Edit Press the **Edit** switch to raise a dialog box on the screen. Place the cursor in the dialog box and enter the string. **Ctrl-U** deletes the entire string. **Enter** or clicking OK in the dialog box sends the string to the downstream module and takes down the dialog box.

OUTPUTS

string The output data is a string.

RELATED MODULES

character string
float
integer
boolean

EXAMPLE

This network creates a long title in the **geometry viewer**.

```
                READ GEOM
                    |
  DIALOG BOX        |
       |            |
     LABEL          |
       |            |
       |------------|
                    |
          GEOMETRY VIEWER
```

SEE ALSO

The example script DIALOG BOX demonstrates this module.

display image

NAME

display image – show image in a display window

SUMMARY

Name	display image				
Availability	Imaging, Volume, FiniteDiff module libraries				
Type	data output				
Inputs	field 2D 4-vector byte uniform				
Outputs	*none*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Magnification	choice	x1	x1	x16
	Automag_Size *(internal)*	integer	256	50	1024
	Max Image Dimension *(internal)*	integer	1280	100	4096
	Dither	choice	dither		

DESCRIPTION

The **display image** modules takes an input image and displays it in a display window. This window has a pulldown menu, accessed via the small square in the window's title bar. The menu allows you to control image magnification, window resizing, and other options relating to the display window.

When the image is larger than the display window, you can scroll it with the mouse, either by "dragging" the image itself or by using horizontal and vertical scrollbars.

You can resize the display window manually, using the X Window System window manager. You can also have the window resize itself automatically, in response to a change in the image contents or a magnification selected from the display window's pulldown menu.

Note that when running avs as a remote client on a pseudocolor X terminal, **display image** has an additional choice parameter for selecting the "dithering" method. For details about running avs on an X server, and dithering colors on pseudocolor machines, see the discussion on Color X Servers in the *AVS User's Guide*.

Note that the **display image** window can be reparented to page and stack widgets using the AVS Layout Editor.

INPUTS

Data Field (required; field 2D 4-vector byte uniform)

The input field must be in the AVS *image* format.

PARAMETERS

Magnification

A choice to specify a power of 2 (1,2,4,8,16) by which to multiply each dimension of the image.

Automag_Size

(for internal use only) This is used as a communications port to handle resizing of the image. Do not change this parameter.

Maximum Image Dimension

(for internal use only) This parameter is no longer used in AVS4. It has been kept solely for the purpose of backward compatibility. See description below.

Dither (only appears when running avs on pseudocolor X terminals)

A choice of five dithering methods. These improve the appearance of color graphics displayed on pseudocolor terminals.

- **dither** uses an internal dither mask to simulate colors that are "between" the colors actually available on a pseudocolor terminal.
- **floyd steinberg** generates better pictures than an ordered dither, but it is slower.
- **random** uses an randomly generated dither mask to simulate colors that are "between" the colors actually available on a pseudocolor terminal.
- **monochrome** computes the luminance of the colors in the input image, by combining the red, green, and blue values for each point, according to a linear relation. The luminance values are then used to find a greyscale equivalent for each pixel. Selecting **monochrome** converts the color image into a monochrome image, resembling a black and white photograph.
- **none** each color in the input image is approximated by the closest color in the spectrum of colors actually available on a pseudocolor terminal.

MAGNIFICATION

You can magnify an image for closer examination, although the magnified image will provide no new detail. Magnification is implemented by duplicating the pixels in the original image. The result is "blockier" but provides a closer look at the image. There are several magnification levels (x1,x2,x4,x8,x16) in the pulldown menu, with the current magnification marked as (*selected*).

Since **display image** now only requests X window resources for the actual displayed window area, the **Maximum Image Dimension** parameter is no longer used.

RESIZING

The display window can be resized in several ways. You can use the X window manager's *resize* window operation to enlarge or shrink the display window. An approximate image magnification is automatically chosen that makes the image at least as large as the window. (This is now only done if the ImageAutomagnify *.avsrc* option is enabled). For a more detailed description of *.avsrc* options, see the **avs** man page.

The ImageAutomagnify parameter reenables the automatic magnification of the image to at least fill the window when the window is resized, as was the default behavior in AVS2. By default, this is disabled, because the combination of autofit and automagnification can produce unexpected window behavior.

Also see the **image viewer** module, which has continuous scale magnification.

The pulldown menu also provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Resize to Fit Image.** Resizes the window to fit the image exactly at the current magnification. (The maximum size window is the full screen window described above.) As with **Zoom Full Screen**, an embedded display window becomes a top-level window.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen** or a **Resize to Fit Image**. If the window originally was embedded in a page or stack, it will be re-embedded there.

- **AutoFit - Turn On/Off.** This toggle switch controls the automatic fitting of the display window size to its image. When this feature is enabled (the default), **display image** automatically resizes the display window whenever the image size changes. This can occur when you select a new magnification or when an entirely new image is input to **display image**. The new display window size exactly fits the new image size (unless the window is currently embedded in a page or stack).

SCROLLING

Whenever the image is larger than the display window, only a portion of the image is visible. You can "pan" over the entire image in two ways:

- Using the horizontal and vertical scrollbars that automatically appear. These scrollbars work the same way as those on File Browser windows.
- By dragging the image itself. Place the mouse cursor anywhere in the image, click and hold down any mouse button, and drag the mouse. The image moves continuously, and the scrollbars are updated when you release the mouse button. The image automatically stops scrolling when it hits its borders.

The **Scrollbars – Turn On/Off** selection on the pulldown menu allows you to disable or reenable the appearance of scrollbars along the right and bottom edges of the display window. (The "drag-the-image" method is always enabled.) You may want to suppress the scrollbars to reduce distraction or to provide additional viewing space.

The **ImageScrollbars** parameter in the AVS startup file (see Chapter 2) determines whether image windows get scrollbars by default when they contain oversize images. If you do not use this startup parameter, scrollbars are initially enabled.

EXAMPLE

```
  READ IMAGE
      |
   DOWNSIZE (optional)
      |
DISPLAY IMAGE
```

RELATED MODULES

display pixmap image viewer

SEE ALSO

The example scripts ANIMATED INTEGER, FIELD MATH, and GENERATE FILTERS as well as others demonstrate the **display image** module.

NAME

display pixmap – show pixmap in a display window

SUMMARY

Name	display pixmap			
Availability	this module is in the unsupported library			
Type	data output			
Inputs	pixmap			
Outputs	*none*			
Parameters	*Name*	*Type*	*Default*	*Choices*
	Store Frames	toggle		
	Append Frame	oneshot		
	Delete Current	oneshot		
	Replay	choice	Off	Continuous Bounce Off
	Current Frame	integer		
	Max Frames	integer		
	Replace Speed	integer		
	Save Image	string		

DESCRIPTION

Note: The **geometry viewer** module superceded **render geometry** in AVS 4. **geometry viewer** displays directly to the screen. There is thus little need for this older **display pixmap** module. It is retained in the unsupported module library for backward compatibility only.

The **display pixmap** module displays its input pixmap in a display window. It automatically sizes the pixmap to fit the window.

display pixmap is most frequently used in conjunction with the **render geometry** module to display geometry output.

In addition, you can:

- Save the pixmap as an AVS image in a file.
- Create and play back a "flipbook" of consecutive images.

These capabilities are invoked using the module’s input parameters, as described in the sections below.

Note that the **display pixmap** window can be reparented to page and stack widgets using the AVS Layout Editor.

INPUTS

pixmap The input data must be an AVS *pixmap*, typically created by the **render geometry** module.

PARAMETERS

The following parameters control a pixmap-animation capability. Note that this is independent of the animation facility in the Geometry Viewer (**render geometry** module), and works somewhat differently. See the *ANIMATION* section below for more information.

Store Frames

This toggle controls whether all new frames are automatically added to the animation sequence.

AppendFrames
Explicitly adds the currently displayed pixmap to the animation sequence. (Use when **Store Frames** is off.)

Delete Current
Deletes the currently displayed pixmap from the animation sequence.

Replay This choice widget controls how the animation sequence is to be played back: The choices are **Continuous**, **Bounce**, and **Off**.

Current Frame
The number of the current frame in the animation sequence (first frame = 0). This field is a typein — change the number to jump directly to another frame.

Max Frames
A typein field that specifies the ceiling for the number of frames that you can place in an animation sequence.

Replay Speed
Controls the rate at which an animation is played back. The larger the value, the greater the delay between frames.

Save Image
This is a typein field. If you type a filename or pathname into this field, the current pixmap is written to a file when you press **Return**.

RESIZING

display pixmap's pulldown menu, which is accessed by clicking on the "dimple" in the upper lefthand corner of the display window, provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen**. If the window originally was embedded in a page or stack, it will be re-embedded there.

SAVING AN IMAGE

To save an image in a file, type the filename as the value of the **Save Image** parameter. When you press **Return**, the file is created. To save another image under the same name, you can move the mouse cursor to the **Save Image** input area and press **Return** again.

ANIMATION

By changing the input data or by adjusting the parameters of upstream modules (e.g. **transform pixmap**), you can have the **display pixmap** window show a sequence of images. You can create an animation ("flip book") by designating certain images to be "frames". Then, you can play back the images, adjusting the speed with a control widget.

Because each of the images in a flip book takes up a significant amount of system memory, there is a *Max Frames* parameter. Be sure that its value is low enough so that your system can comfortably keep all of the images in memory at the same time. AVS requires roughly 4 bytes of memory per pixel of each your image. The larger the display window, the greater the memory requirements.

There are two ways to create a flip book:

- To save *all* the images that appear in the window (actually, just the last *Max Frames* that are produced — see below), turn on the **Store Frames** toggle. As each image is drawn, it will be appended to the end of the flip book. If *Max Frames* images have already been saved, this new pixmap will replace the oldest pixmap in the cycle.
- If you want to selectively add images to the flip book, modify the image until it is as you want it, then select the one-shot **Append Frame**. This appends the image to the end of the existing flip book. This method allows you to carefully construct a flipbook animation.

The **Replay** parameter controls the way in which the flip book is displayed. It has three selections:

- **Continuous** plays through all of the frames in the animation, wrapping around when it reaches the end.
- **Bounce** plays forward through the last *Max Frames* or fewer frames. When it reaches the end, it plays backwards through those frames.
- **Off** turns off the animation facility

The **Replay Speed** parameter controls the rate at which flip book frames are displayed.

The **Current Frame** parameter allows you to select a particular frame "manually". It is normally updated to display the current frame, but for cases in which such updating would impact animation performance, it is not updated. Note that since only the last *Max Frames* frames are stored, the animation can begin at a frame other than 0.

After you select a particular frame, you can delete it with the one-shot **Delete Frame**.

EXAMPLE 1

The following network reads in an image, converts it to a pixmap and then displays the image using **display pixmap**:

```
   READ IMAGE
        |
IMAGE TO PIXMAP
        |
 DISPLAY PIXMAP
```

EXAMPLE 2

The following network reads in a geometry object, renders it and then displays the rendered object using **display pixmap**:

```
   READ GEOM
       |
       |
RENDER GEOMETRY
       |
       |
DISPLAY PIXMAP
```

RELATED MODULES

transform pixmap, alpha blend, render geometry

LIMITATIONS

There is no way to store the "first *max frames*" frames of an animation loop.

display tracker

NAME

display tracker - display and directly manipulate the tracer module's output

SUMMARY

Name	display tracker
Availability	Volume, FiniteDiff module libraries
Type	data output
Inputs	field 2D 4-vector byte uniform ("image")
Outputs	upstream transform (invisible, optional, autoconnect)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
scale	integer	1	1	16
interpolate	toggle	off		

DESCRIPTION

display tracker is designed specifically to work with the modules **tracer**, and **ucd tracer**. The module **tracer** takes in volume data and performs volumetric rendering on it using ray-tracing. **tracer** outputs a 2D AVS image; **display tracker** displays this image in a window.

In addition to displaying **tracer**'s output, **display tracker** allows you to directly manipulate an image in its window using the mouse. You can rotate or translate a volume being rendered by moving the mouse, employing the "virtual spaceball" paradigm.

When you press the middle mouse button a bounding box appears superimposed around the rendered volume. Moving the mouse causes this bounding box to rotate. When the desired rotation is achieved, release the mouse button. The volume will be rendered again to show it rotated to the new position. The bounding box will disappear once the volume is redrawn. Translations are achieved in a similar way, using the right mouse button. To scale the object, use shift key in combination with the middle mouse button. To reset the object, press the left mouse button. This will reset the volume to its original orientation.

Note that **display tracker** takes AVS images as input. It can receive these images from any module that outputs an image. However, it will allow direct manipulation of images only when the module above it is equipped to receive the upstream transform that **display tracker** outputs.

INPUTS

Data Field (required; field 2D 4-vector byte uniform)

An AVS image, typically output by the module **tracer**.

PARAMETERS

scale (integer)

Multiplies size of input image by selected value. Scaling an image by a large amount will result in slower display times. In combination with the "width" and "height" parameters of **tracer**, you can use scale to create very large images.

interpolate (toggle)

With interpolate off (default) the image is scaled using pixel replication. In other words, pixels are simply copied to increase the size of the image.

With interpolate selected, bilinear interpolation is performed on the image when it is scaled. This results in smoother gradations in the color of pixels in the scaled image.

OUTPUTS

Upstream Transform (optional, invisible, autoconnect)

The output port on the module **display tracker**, which is usually invisible, sends out a 4x4 transformation matrix describing rotations and translations that have been applied to the image through movements of the mouse. This output port will automatically connect with **display tracker**'s invisible lefthand input port. This allows you to directly rotate and translate an image by moving the mouse in **display tracker**'s window.

EXAMPLE

The following network shows how **display tracker** is used to display the output of **tracer**. Note that **display tracker** also sends data "upstream" to **tracer**.

```
GENERATE COLORMAP    READ VOLUME
           |              |
           |   |----------|
           |   |
         COLORIZER
    |--|   |
    |  |   |
    | TRACER
    |      |
    |      |
    |     DISPLAY TRACKER
    |_________|
```

RELATED MODULES

Modules that could provide the **Data Field** input:

- cube
- tracer
- *any other module which outputs an AVS image*

Modules that can receive **display tracker**'s upstream transform:

- cube
- tracer
- gradient shade

SEE ALSO

The example script ANIMATED FLOAT demonstrates the **display tracker** module.

dot surface

NAME

dot surface – generate points that define an isosurface

SUMMARY

Name	dot surface				
Availability	this module is in the unsupported library				
Type	filter				
Inputs	field 3D scalar *any-data any-coordinates*				
Outputs	field 1D scalar (irregular 3-space)				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Stepsize	real	.01	1.0E–5	1.0
	Threshold	real	.02	0	1

DESCRIPTION

The **dot surface** module accepts a 3D scalar field as input and generates a list of points that defines an isosurface. The input field is composed of cells, where each cell is defined as a subvolume composed of six faces. Each cell is processed checking for a possible intersection of the surface. If the cell does contribute to the surface it is then subdivided until the maximum physical dimension of the resulting subcell is ≤ the value of the **Stepsize** parameter. A smooth surface can be generated in this manner, given a sufficiently small **Stepsize** value.

The running time of this module is directly proportional to the number of cells processed and the number of cells that contribute to the surface. It is inversely proportional to the **Stepsize** value.

If the input field is uniform, then a physical grid is generated mapping the data volume into a canonical size. The largest dimension of the volume is mapped into the interval: [-1.0, +1.0]. Other dimensions are scaled accordingly, thus if a uniform volume consisting of 100 nodes in the x direction, 50 in the y direction and 20 in the z direction will have a bounding volume of: x=[-1.0, +1.0], y=[-0.5,+0.5], z=[-0.2,+2.0]. The distance between each node is then approximately equal to 0.02. The Stepsize parameter is relative to this length scale.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)
This module uses a scalar data value for each field element. If the input is a vector-valued field, then the first component of the vector is used as the scalar value.

PARAMETERS

Stepsize A floating-point value that determines the resolution of the isosurface. The smaller this value, the smoother the surface.

Threshold A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the field element's data value equals the **Threshold** value.

OUTPUTS

Point List (field 1D scalar irregular 3-space) The scalar data value for each output field element is unused. The only useful information is the 3D coordinate data.

EXAMPLE 1

```
READ VOLUME
     |
 DOWNSIZE
     |
DOT SURFACE
     |
SCATTER DOTS
     |
GEOMETRY VIEWER
```

LIMITATIONS

The number of points may be inadequate to represent areas of small surface curvature with respect to the cell's local coordinate system.

A maximum of 80,000 points will be generated. Once the module calculates this number of points, it returns leaving all other cells unprocessed. Use **downsize** to avoid this if possible.

RELATED MODULES

Modules that could provide the Data Field input:

read volume
combine scalars

Modules that could be used in place of dot surface:

isosurface
tracer

Modules that can process dot surface output:

scatter dots

SEE ALSO

The example script DOT SURFACE demonstrates the **dot surface** module.

downsize

NAME

downsize – reduce size of data set by sampling

SUMMARY

Name	downsize
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field 2D / 3D *n-vector any-data any-coordinates*
Outputs	field of same type as input

Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	downsize	integer	8	1	16

DESCRIPTION

The **downsize** module changes the size of the input data set by subsampling the data. It extracts every *N*th element of the field along each dimension, where *N* is the value of the **downsize factor** parameter. This technique preserves the aspect ratio of the input data.

This module is useful for operating on a reduced amount of data, in order to adjust other processing parameters interactively, or save memory. After the parameter values have been set, you can remove the **downsize** module, so that the full data set is used for final processing.

Alternatively, retain the **downsize** module in the network, so that you can interactively choose between image quality (**downsize factor** = 1 for highest-resolution data) and execution speed (**downsize factor** > 1 for lower-resolution data).

INPUTS

Data Field (required; field 2D / 3D *n-vector any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

downsize Determines how data elements from the field are sampled. Increasing this parameter causes more elements to be skipped over, thus *decreasing* the size of the output.

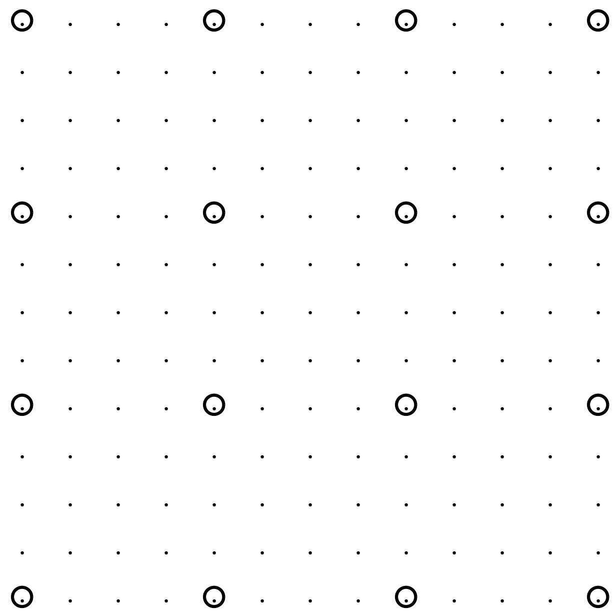
OUTPUTS

Data Field The output field has the same dimensionality as the input field, but the number of elements in each dimension is reduced by the **downsize factor**.

The **min_val** and **max_val** attributes of the output field are invalidated. Note that the extent is unmodified; this module changes the resolution of the data within the physcial space delimited by the extents. It does not alter the physical extents of the data.

EXAMPLE

The following diagram shows how a **downsize factor** of 4 reduces a 2D field. Each element of the field is represented by a dot. Only the larger dots are included in the output field.



LIMITATIONS

downsize works for 2D, and 3D data sets only.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
read field
filter modules

Modules that could be used in place of **downsize**:

interpolate (arbitrary sampling)
crop (subset at high resolution)
average down (average in X, Y, and / or Z, independently)

SEE ALSO

The example scripts FIELD MATH, and GRAPH VIEWER demonstrate the **downsize** module.

draw grid

NAME

draw grid – draw a grid on top of an image

SUMMARY

Name	draw grid
Availability	Imaging module library
Type	filter
Inputs	field 2D | 3D uniform byte | short | float *n-vector*
Outputs	field *same-dims same-vector same-data*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
N Grids	int dial	10	2	*max dim/2*
Square	boolean	off		
red	int dial	255	0	255
green	int dial	255	0	255
blue	int dial	255	0	255

DESCRIPTION

draw grid draws a grid of lines superimposed upon an image.

INPUTS

Data Field (required; field 2D | 3D uniform byte | short | float *n-vector*)
The input is a 2D or 3D uniform field with *n* vectors of either byte, short, or float data. If the field is 3D, lines are created over Z successive XY slices.

PARAMETERS

N Grids An integer dial that specifies how many full-size horizontal and vertical areas to create. The range is 2 to *max dim* / 2, where *max dim* is the larger of the X or Y dimension. The default is 10. To create the grid lines, **draw grid** calculates *xskip* = *max x-dim* / N Grids and *yskip* = *max y-dim* / N Grids using integer arithmetic, then places a grid line every *xskip* and *yskip* pixels.

square A switch that forces the grid marks to be squares. To produce the square grid, **draw grid** employs the grid spacing used by the largest dimension (X width or Y height), and replicates this spacing across the other dimension. For example, if the image is wider than it is tall (X>Y), then the vertical grid line spacing will also be used for the horizontal grid line spacing. The default is off.

red
green
blue Three integer dials that together set the RGB color of the grid lines. The default is 0 (alpha), 255, 255, 255 (white). The lines are created by setting the data values at the line positions equal to these values. Note that if the input is scalar, these dials are ignored and the data value at the line position is set to 0. Similarly, a 2-vector will be set to 0,255, and a 5-vector's last vector element will not be reset at all.

OUTPUTS

Data Field (field *same-dims same-vector same-data*)
The output is a field with the same dimensions, vector length, and data type as the input field.

EXAMPLE 1

```
        READ IMAGE
             |
             |
        DRAW GRID
             |
             |
  IMAGE VIEWER
```

EXAMPLE 2

```
        READ VOLUME
             |
             |
         DRAW GRID
             |
             |
     ORTHOGONAL SLICER
             |
             |
      COLORIZER
             |
             |
   IMAGE VIEWER
```

RELATED MODULES

image viewer

SEE ALSO

The Imaging / DRAW GRID sample script demonstrates the **draw grid** module.

LIMITATIONS

If the image is rescaled in a module such as **image viewer**, some of the grid lines may disappear or become wider due to the rescaling algorithm used.

edit substances

NAME

edit substances – create a substance table for the **cube** module

SUMMARY

Name	edit substances
Availability	Supported, Volume, Finite Differences module libraries
Type	data input
Inputs	None
Outputs	struct substances (substance table) colormap

Parameters

Name	*Type*	*Default*	*Min*	*Max*
filename	file browser	.sub suffix		
read file	onehot			
write file	oneshot			
write colormap	oneshot			
current substance	islider	1	1	32
name	string	"Unused"		
lo threshold	float	0.0	0.0	256.0
hi threshold	float	128.0	0.0	256.0
opacity	float	0.0	0.0	1.0
red	float	0.0	0.0	1.0
green	float	0.0	0.0	1.0
blue	float	0.0	0.0	1.0
skip layers	int	0	0	100

DESCRIPTION

edit substances creates the substance table required by the **cube** module's *ray-cast* and *create-surfaces* rendering modes. It communicates to **cube** via a user defined data structure called *substances*. The definition of this structure is found in *$AVS_PATH/include/substances.h*.

The substance table is a list of 32 substances with seven parameters per substance: a name, the starting intensity of the substance, the substance opacity, the substance color (red, green, and blue), and the number of layers to skip when rendering in **cube**'s *create-surfaces* mode.

In addition, **edit substances** also outputs a conventional AVS colormap (and optionally writes that colormap to disk). This colormap can be used by other volume rendering tools such as **tracer** and **volume render**.

cube is most effective when used on data which is readily classified into distinct material types. In medical imaging, these types might correspond to "skin", "muscle", and "bone". In non-destructive evaluation, the types might be described for "air", "engine wall", "engine interior". **edit substances** creates the substance table which lets you specify the levels at which new substances are defined, the colors and opacities of those substances, and names for each of the substances which make sense for a particular application.

edit substances also can read and write ASCII substance files which contain the substance table information. The substance file typically has a *.sub* suffix to differentiate it from other files. Each substance file must contain 32 lines with eight entries per line:

```
number  name      threshold  opacity  red      green    blue     skip_layers
(int)   (string)  (float)    (float)  (float)  (float)  (float)  (int)
```

Here is the substance file for the default substance table:

```
0 Unused 0.000000 0.000000 0.000000 0.000000 0.000000 0        (black)
1 Unused 128.000000 1.000000 1.000000 1.000000 1.000000 0      (white)
2 Unused 256.000000 1.000000 1.000000 0.000000 0.000000 0      (red)
3 Unused 256.000000 1.000000 1.000000 1.000000 0.000000 0      (yellow)
4 Unused 256.000000 1.000000 0.000000 1.000000 0.000000 0      (green)
5 Unused 256.000000 1.000000 0.000000 1.000000 1.000000 0      (cyan)
6 Unused 256.000000 1.000000 0.000000 0.000000 1.000000 0      (blue)
7 Unused 256.000000 1.000000 1.000000 0.000000 1.000000 0      (magenta)
8 Unused 256.000000 1.000000 0.800000 1.000000 0.000000 0      (yellow-green)
9 Unused 256.000000 1.000000 0.600000 1.000000 0.200000 0
10 Unused 256.000000 1.000000 0.400000 1.000000 0.400000 0
 ...          ...          ...          ...          ...
31 Unused 256.000000 1.000000 0.000000 0.200000 0.500000 0
```

This default substance table has several interesting features: there is only one visible substance and it is white, opaque, and starts at value 128.0. The other substances (which are assigned default colors that follow the spectrum) are "turned off" by having them start at value 256.0 which is beyond the possible range of byte data. To turn on other substances, you need to set their thresholds to be in the 0-255 range.

Substances are defined sequentially—in order for substance 5 to be used, substances 1-4 must be valid. Furthermore, their starting thresholds must be in increasing order. For instance:

Substance	starts at	ends at
1	0	64
2	64	128
3	128	255

is a valid collection of substances while:

Substance	starts at	ends at
1	0	64
2	32	128
3	255	2

is not and will result in an error.

PARAMETERS

filename (file browser: sensitive to *.sub* suffixes)

This is a multi-purpose file browser which lets you specify the names of the ASCII substance files that can be read or written as well as the name of an AVS colormap file to write. Note: you cannot read colormap files because they can contain more than 32 entries and the substance table is limited to 32 substances. The number of substances available is a limitation of the **cube** module.

read file (oneshot)

When you have a valid **filename**, hitting this button causes the ASCII substance file to be read in, replacing the internal substance table with that contained in the file.

write file (oneshot)

When you have a valid **filename**, hitting this button causes the internal substance table to be written (in ASCII) to the specified file.

write colormap (oneshot)

When you have a valid **filename**, hitting this button causes the internal substance table to be translated into AVS colormp format and an AVS colormap file to be written. Note: you should follow the convention of naming colormap files with a *.cmap* suffix.

current substance (islider)

Although there are 32 substances in the table, you can only change one at a time. This slider lets you select which substance you are editing.

name (string)

Each substance can be assigned a name of up to 80 characters. Typical names may include entries like: "air", "skin", "bone", "engine wall". These are used for identification purposes only and have no effect on rendering.

lo threshold (float, typein)
hi threshold (float, typein)

Each substance is defined as being those voxels whose value is greater than **lo threshold** and less than or equal to **hi threshold**. Internally, only the **lo threshold** is stored (and transmitted) per substance—the **hi threshold** is derived as being the **lo threshold** of the next substance. This is reflected in the user interface for this module; when you edit substance N's **hi threshold**, you are also changing substance N+1's **lo threshold** parameter. This is a convenience which makes it easier to adjust the range of a particular substance without bouncing around between substances.

opacity (float, typein)

The opacity of the current substance ranging from 0.0 (transparent) to 1.0 (fully opaque). Each substance can have a different opacity.

red (float, typein)
green (float, typein)
blue (float, typein)

The color of the current substance ranging from 0.0 (black) to 1.0 (fully on). Each substance can have a different color.

skip layers (int, typein)

This is only valid in the *create surfaces* mode of **cube**. When in the *create surfaces* mode, each pixel in the image is stored with the surface intersections for the pixel's ray. It is possible to ignore a given number of these intersections using the *skip layers* feature. For instance, a perfect sphere would usually have two ray intersections per pixel; one entering the sphere and the other leaving it. Normally, you would only see the front side of the sphere, and if it were opaque, nothing else. With the *skip layers* feature, you can instruct the ray caster to ignore the first intersection (the front of the sphere) but to render all the rest of them.

One practical application of this feature is in medical imaging. Say the skin and the brain of a MRI head scan are the same value. To image the brain requires looking "through" the skin, yet you don't want to make all voxels in the "skin-brain" range transparent because this would hide the brain as well! Using the *skip layers* feature, it is possible to ignore the first several intersections but to render the rest.

OUTPUTS

Substance Table (struct substances)

This is a user defined data structure described in *avs/include/substances.h* which is used to transmit the substance table between **edit substances** and **cube**.

Colormap (AVS colormap)

This is a standard AVS colormap version of the substance table. This can be used by other volume rendering tools such as **tracer** or **volume render**. It can also be fed into **field legend** for real-time viewing of the colors and ranges of the substances being defined.

EXAMPLE

The following network shows one way to use **edit substances** with the **cube** module. The output from **edit substances** is also fed into **field legend** so that the substance table can be viewed relative to the voxel values they represent.

```
                        READ FIELD
                             |
  EDIT SUBSTANCES  _______|
           |_______|____  |
           |   ____|    | |
           |  |        CUBE
           |  |          |
    FIELD LEGEND   DISPLAY TRACKER
```

RELATED MODULES

Modules that can process **edit substances**’s substance table output:

cube

Modules that can process **edit substances**’s colormap output:

field legend
tracer
volume render
colorizer
generate colormap
any other module which takes an AVS colormap as input.

SEE ALSO

The example script CUBE demonstrates the **edit substances** module.

LIMITATIONS

edit substances cannot currently read in SunVision *.subs* files.

euler transformation

NAME

euler transformation - send object transformation matrix to other modules

SUMMARY

Name	euler transformation
Availability	Volume, FiniteDiff module libraries
Type	data
Inputs	none
Outputs	field uniform 2D scalar float (transformation matrix)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
theta	float	0	0	360
phi	float	0	0	360
rho	float	0	0	360
scale	float	1	0	10

DESCRIPTION

euler transformation allows you to generate a 4x4 transformation matrix specifying scaling and rotations in x, y and z.

euler transformation is designed to be used with modules that can transform data in object space. This means that rotations and scaling operations are applied to a 3D data "object" before it is rendered and turned into a 2D image. **euler transform** does not supply the full "upstream transform" accepted by such modules as **brick** and **thresholded slicer**. Currently **euler transform** will work only with the modules **gradient shade** and **tracer**.

Using **euler transformation**'s dials you can select a transformation matrix that will scale and/or rotate an object. The order in which rotations are applied is x-y-z. If you rotate an object through a number of angles, it is always the original data that is transformed, i.e., transformations are not remembered and accumulated.

PARAMETERS

theta A floating point dial widget which controls rotation of the object's x axis. The x axis initially runs horizontally from negative on the left to positive on the right.

phi A floating point dial widget which controls rotation of the object's y axis. The y axis initially runs vertically from negative at the bottom to positve at the top.

rho A floating point dial widget which controls rotation of the object's z axis. The z axis initially runs perpendicular to the screen, with the positive z axis coming "out" of the screen, and the negative z axis "behind" the screen.

scale A floating point dial widget which controls the scaling coefficient of the transformation matrix. This makes the data "object" look larger or smaller.

OUTPUTS

Transformation Matrix (field 2D uniform scalar float)

The output is a 4x4 array of floating point values which specifies rotations and scaling operations that can be applied to transform an object around the origin of its own coordinate system.

EXAMPLE 1

The following network performs volumetric ray-tracing using **tracer**. By setting parameters in the module **euler transformation** you can rotate or scale the volume being rendered, so you can see all sides of the volume:

```
           GENERATE COLORMAP       READ VOLUME
                          |       |
                          |       |
                         COLORIZER
  EULER TRANSFORMATION        |
                  |------|    |
                        TRACER
                           |
                           |
                    DISPLAY IMAGE
```

RELATED MODULES

Modules that accept **euler transformation**'s output:

cube
tracer
gradient shade

SEE ALSO

The example script EULER TRANSFORMATION demonstrates the **euler transformation** module.

excavate

NAME

excavate – remove an octant from a 3D uniform field, revealing interior features

SUMMARY

Name	excavate				
Availability	Volume, FiniteDiff module libraries				
Type	filter				
Inputs	field 3D scalar byte				
Outputs	field 3D scalar byte				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	X	integer dial	*x res/2*	*0*	*x res*
	Y	integer dial	*y res/2*	*0*	*y res*
	Z	integer dial	*z res/2*	*0*	*z res*
	flip X	boolean	off		
	flip Y	boolean	off		
	flip Z	boolean	off		

DESCRIPTION

The **excavate** module excavates the selected octant from a 3D scalar byte field, revealing the interior data. It does this by setting the data values to 0 within the specified region. Regions are selected with a combination of dials that specify the location of the slice plane and toggle switches that specify which side of the slice planes' data should be zeroed out.

excavate is especially useful for "looking inside" volumetric data that may be hard to segment—for example, medical imaging data.

INPUTS

Data Field (required; field 3D scalar byte uniform)
The input is a field 3D scalar byte. It can be of any uniform type.

PARAMETERS

X An integer dial indicating on which X location the YZ cutting plane is to be placed. This dial ranges from zero to the X-dimension of the data set. The default value is the middle of the data set.

Y An integer dial indicating on which Y location the XZ cutting plane is to be placed. This dial ranges from zero to the Y-dimension of the data set. The default value is the middle of the data set.

Z An integer dial indicating on which Z location the XY cutting plane is to be placed. This dial ranges from zero to the Z-dimension of the data set. The default value is the middle of the data set.

flip X A toggle that indicates on which side of the YZ cutting plane the data will be zeroed. When off, the data from the cutting plane location to the maximum dimension of the data is zeroed. When on, the data from the cutting plane to the minimum dimension of the data is zeroed.

flip Y A toggle that indicates on which side of the XZ cutting plane the data will be zeroed. When off, the data from the cutting plane location to the maximum dimension of the data is zeroed. When on, the data from the cutting plane to the minimum dimension of the data is zeroed.

flip Z A toggle that indicates on which side of the XY cutting plane the data will be zeroed. When off, the data from the cutting plane location to the maximum dimension of the data is zeroed. When on, the data from the cutting plane to the minimum dimension of the data is zeroed.

OUTPUTS

Data Field The output field is a field 3D scalar byte field that is the same shape and size as the input field. The only difference is that some of the data in the output field has been zeroed out.

EXAMPLE 1

The following example shows how **excavate** is used in a network with the **tracer** module:

```
       READ VOLUME
            |
         EXCAVATE
            |
        COLORIZER
            |
          TRACER
            |
     DISPLAY TRACKER
```

EXAMPLE 2

The following example shows how **excavate** is used in a network with the **isosurface** module. In this example, the surface will be cropped at the excavated boundaries and the slice planes along the cropping will be colored by the interior of the data set

```
GENERATE COLORMAP       READ VOLUME
         |            _______|
         |            |      |
         |_________   |    EXCAVATE
                  |   |   ____|
                  |   |   |
                 ISOSURFACE
                      |
                GEOMETRY VIEWER
```

RELATED MODULES
SEE ALSO

The example script EXCAVATE demonstrates EXAMPLE 2 above.

excavate brick

NAME

excavate brick – show uniform volume with orthogonal slices

SUMMARY

Name	excavate brick				
Availability	Volume, FiniteDiff module libraries requires 3D texture mapping support				
Type	mapper				
Inputs	field 3D uniform *n-vector any-data*				
Outputs	geometry				
Parameters	Name	Type	Default	Min	Max
	X	integer dial	0	0	*x res*
	Y	integer dial	0	0	*y res*
	Z	integer dial	0	0	*z res*
	Flip_X	boolean	off		
	Flip_Y	boolean	off		
	Flip_Z	boolean	off		
	Draw_Sides	boolean	on		

DESCRIPTION

The **excavate brick** module is another way of visualizing 3D uniform volume data. The volume is displayed using multiple orthogonal slice texture mapped slice planes. The slice planes are in the form of a cube with a cubical shaped "chunk" removed on one corner. The size of the chunk can be controlled using the **X**, **Y**, and **Z** parameter controls. The selected corner that is to be removed is specified **Flip_X**, **Flip_Y** and **Flip_Z** controls. The sides of the cube will be draw only if the **Draw_Sides** parameter is set.

excavate brick creates its picture of the volume data using 3D texture mapping (**arbitrary slicer** uses sampling). In this method, the boundary of the volume has three values, *u, v, w*, associated with each of its vertices. When **excavate brick**'s slice plane intersects this volume, *u, v, w* values are computed for the vertices of the resulting solid. These values are attached to the vertices of the geometry object which **excavate brick** produces, and are used by **geometry viewer** to perform 3D texture mapping.

Texture mapping is much faster than the sampling technique used by arbitrary slicer, particularly for large datasets. The point sampling done by the texture mapping technique is always done at the resolution of the data; thus differences in data values within a small area are not obscured as they can be with **arbitrary slicer**.

The 3D texture map is created with a combination of the **generate colormap**, **colorizer**, and possibly **color range** modules. Their output is connected to the **geometry viewer** module's center texture map port (see example below).

AVAILABILITY

excavate brick requires that the underlying graphics renderer support 3D texture mapping. Not all hardware renderers support 3D texture mapping (see the release note information that accompanies AVS on your platform). The AVS software renderer does support 3D texture mapping. If a renderer does not support 3D texture mapping, then the volume will appear, and you can manipulate the excavating cube, but the geometry object will appear as a featureless white solid. To get the 3D texture mapping on multi-renderer platforms, you can turn on the **Software Renderer** button under the Geometry Viewer's **Cameras** submenu.

INPUTS

Data Field (required; field 3D uniform n-vector any-data)
The input field is a 3D uniform volume. The data can be of any type.

PARAMETERS

X, Y, and **Z** These three parameters control the position of the excavating cube. The values are specified in terms of the resolution of the data. A value of 0 indicates that the excavate cube has zero dimensions along the X, Y, or Z dimension.

Flip_X, Flip_Y, Flip_Z
These three parameters indicate whether the excavate cube should be positioned on the positive or negative axis for each of the X, Y, and Z dimensions. If the parameter is true, the excavate cube is positioned on the negative axis.

Draw_Sides
A boolean switch that controls whether the sides of the "main" cube are to be drawn. If this boolean is false, only the faces of the excavate cube are drawn.

OUTPUTS

Geometry (geometry)
The output geometry is the solid version of the volume.

EXAMPLE 1

The following network reads a byte volume. The volume is fed to **colorizer** to paint the byte values as colors by producing a 3D 4-vector field of color values from the original data. The volume is sent to **excavate brick** to map the surfaces, and to **volume bounds** to draw a box around the limits of the volume. The **generate colormap** and **colorizer** parts of the network are vital; they create the 3D texture map that feeds into the **geometry viewer** module's left input port. Without the 3D texture map, the volume would appear as a featureless white solid. The geometries from **volume bounds** and **excavate brick** feed into **geometry viewer**'s right input port.

```
                            READ VOLUME
                                 |
      GENERATE COLORMAP          |
               |                 |
               |  |--------------|---------------|
               |  |              |               |
           COLORIZER    EXCAVATE BRICK    VOLUME BOUNDS
               |                 |               |
               |                 |               |
               |----------|      |---------------|
                       GEOMETRY VIEWER
```

EXAMPLE 2

The following network is the same as the previous example in basic structure. The difference is that the uniform volume data is a 3D field of real values, not bytes. The **vector mag** module is used to convert the vector field into a scalar float field. The addition of the **color range** module scales the color values in the colormap to match the range of the data. It should be included whenever the data is not of type byte.

excavate brick

```
                         READ FIELD
                             |
                             |
                         VECTOR MAG
      GENERATE COLORMAP      |
              |              |
              |   |--|-----------|---------------|
              |   |  |           |               |
         COLOR RANGE |     EXCAVATE BRICK  VOLUME BOUNDS
              |      |           |               |
              |      |           |               |
           COLORIZER             |               |
              |-------------|    |---------------|
                    GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

- read volume
- read field
- Any module that outputs a 3D uniform field

Modules that could be used in place of **excavate brick**:

- arbitrary slicer
- brick
- orthogonal slicer
- field to mesh
- thresholded slicer

Modules that can process **excavate brick** output:

- geometry viewer

SEE ALSO

The EXCAVATE BRICK example script demonstrates the **excavate brick** module.

NAME

extract graph – extract and display a 1D slice from a 2D data set

SUMMARY

Name	extract graph			
Availability	Imaging, Volume, FiniteDiff module libraries			
Type	Filter			
Inputs	field 2D scalar *any-data any-coordinates (required)*			
Outputs	field 2D scalar float geometry			
Parameters	*Name*	*Type*	*Default*	*Choices*
	Graph Select	integer dial	0	
	axis	choice	I	I,J
	Abscissa Mapping	choice	Dist	Dist,Index,X,Y,Z

DESCRIPTION

The **extract graph** module is similar to the **orthogonal slice** module in that it takes a one-dimensional slice out of a two-dimensional data set for the purposes of sending the slice to the **graph viewer** module for display. The differences between these modules is that **extract graph** allows different X-axis mappings and it also creates a 3D geometry showing which slice is being extracted. The **Abscissa Mapping** choices only work for **irregular** data sets. They have no effect for **uniform** and **rectilinear** data.

INPUTS

Data Field (required; field 2D scalar *any-data any-coordinates*)

This field is typically derived by taking an othogonal slice through a volumetric (3D) data set. The volume can be of any type (uniform, rectilinear, or irregular), and data size (byte, float, int, double).

PARAMETERS

Graph Select

This is an integer dial indicating which 1D slice from the 2D input field is to be taken. This is similar to the "slice plane" parameter in the **orthogonal slicer** module. This dial starts off going from 0 to 1, but readjusts itself dynamically according to the dimensions of the currently selected slice axis (see below).

axis In a flat two-dimensional image, you can take one-dimensional slices in constant X or constant Y. Since this module works for irregular datasets as well as uniform ones, we rename these directions to include off-axis slices and call them constant I and constant J. The default is **J** which can be interpreted as constant Y for uniform images.

Abscissa Mapping

Since slices from irregular data sets may not correspond to Cartesian axes, there are several ways to graph the data coming from a one-dimensional slice. The **Dist** option plots the distance along the slice as the X-axis. the **Index** option shows the array index (this is equivalent to what you get when you cascade two **orthogonal slicer** modules). The **X, Y,** and **Z** options project the 1D slice to those axes and display those projections as the X-axis of the plot. These choices have no effect on **uniform** and **rectilinear** data.

OUTPUTS

Data Field (field 2D scalar)

This is the 1D slice represented in AVS field format. This is the field which is sent to the **graph viewer's** rightmost port. This is a 2D field whose dimensions are 2 by the dimensions of the chosen axis. Each pair contains the X value (as described by the Abscissa Mapping) and the Y value (which is the actual data value). The Graph Viewer knows to treat this as "Plot as XY Data".

Geometry (geometry)

The **extract graph** module also outputs two geometric lines (one on each side of the slice) which show the location of the extracted slice. This is critical, because otherwise you have no visual indication of where the slice came from.

EXAMPLE

The following network is a typical application using the **extract graph** module:

```
                  READ FIELD
                      |
                ORTHOGONAL SLICE
         _____________|______________
         |                          |
   FIELD TO MESH              EXTRACT GRAPH
         |  ________________________|
         |  |                       |
   GEOMETRY VIEWER            GRAPH VIEWER
```

RELATED MODULES

orthogonal slicer
ip read line
graph viewer

SEE ALSO

The example script EXTRACT GRAPH demonstrates the **extract graph** module.

NAME

extract scalar – extract a scalar field from a vector field

SUMMARY

Name	extract scalar
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates* (*n* = 1..25)
Outputs	field *same-dimension* scalar *same-data same-coordinates*

Parameters

Name	*Type*	*Default*
Channel *n*	radio buttons	Channel 0

DESCRIPTION

The **extract scalar** module inputs a field whose data values are vectors (1D to 25D), and outputs one of the dimensions ("channels") as a scalar-valued field. The output field has the same structure as the input field, except that its data values are scalars (vector length of 1).

This module is useful for performing operations on individual channels of vector fields. It is frequently used with the **combine scalars** module, which composes vector fields from individual scalar fields.

INPUTS

Data Field (required; field *any-dimension n*-vector *any data any-coordinates*)

The input data may be any field whose data values are vectors with 25 or fewer dimensions. Even scalar fields may be used, since their data values are considered to be 1D vectors.

PARAMETERS

Channel *n* Selects the dimension of the input data values to be output. A set of radio buttons appears, showing the labels that are attached to the dimensions of the *n*-vector data.

OUTPUTS

field (*same-dimension* scalar *same-data same-coordinates*)

The output field has the same dimensionality as the input field. The data for each element is reduced from a vector to a scalar. The *veclen*, *min_val*, *max_val*, *label*, and *unit* values in the field are updated.

EXAMPLE 1

This examples displays a slice of the Y-component of the gradient field of a volume:

```
                        READ VOLUME
                                |
                        COMPUTE GRADIENT
                                |
                        EXTRACT SCALAR [1]   (0=X, 1=Y, 2=Z)
                                |
                        ORTHOGONAL SLICER
                                |
GENERATE COLORMAP         FIELD TO BYTE
        |________________       |
                           |    |
                          COLORIZER
                              |
                        DISPLAY IMAGE
```

For additional examples, see the **combine scalars** manual page.

RELATED MODULES

extract vector
combine scalars

SEE ALSO

The example scripts CONTOUR GEOMETRY, CONTRAST, as well as others demonstrate the **extract scalar** module.

NAME

extract vector – subset of field vector elements as new field

SUMMARY

Name	extract vector
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates* (*n* = 1..25)
Outputs	field *same-dimension n-vector same-data same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Vector Length	integer dial	3	1	25
Channel 0	boolean	off		
Channel 1	boolean	off		
Channel 2	boolean	off		
	.			
	.			
	.			
Channel 24	boolean	off		

DESCRIPTION

The **extract vector** module takes a vector field of any dimension, coordinate system, or data type, and extracts a subset of the vector elements at each node. The output field is identical to the input field, but with only the selected vector elements at each node. This is useful, for example, with PLOT3D format data. PLOT3D data normally has seven vector elements at each node. However, only three of these, X-Momentum, Y-Momentum, and Z-Momentum, are useful if you are trying to visualize momentum vectors with the **hedgehog** module. **extract vector** is a convenient way to segregate just the vector elements needed. It is more convenient (and equivalent to) using **extract scalar** modules to extract individual vector elements and then pasting them together again with **combine scalar**.

extract vector can handle up to 25 vector elements. You can extract any subset of the 25 elements.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

An AVS field with a vector of data elements at each node. The field can be any dimension, using any type of coordinate information, and any kind of data.

PARAMETERS

Vector Length

An integer dial that specifies the vector length of the *output* field. The default is 3, the minimum is 1, and the maximum is 25. This must be set to the number of channels selected below.

Channel 0
Channel 1
Channel 2 ...

A series of on/off switches that specify which of the input vector elements to extract into the output field. If the input vector elements have been labelled, then their labels will appear instead of the default "Channel *n*". Only as many switches will appear as there are input vector elements. By default, all of the switches are "off". There is no way to change the order of vector elements; if X preceded Y in the input field, it will do so in the output field (you can change the order of vector elements by using multiple instances of the **extract scalar** module, feeding into one **combine scalars**).

OUTPUTS

Data Field (field *same-dimension n-vector same-data same-coordinates*)
The output field has the same form as the **Data Field** input, except that its vectors are shorter. The *veclen*, *min_val*, *max_val*, *label* and *unit* of the field are updated.

EXAMPLE 1

The following network extracts the x, y and z momentum vector elements from a field dataset, then plots their sum vector using **hedgehog**. The dataset operated on is *bluntfin.fld*, which contains PLOT3D data in field format.

```
   READ FIELD
       |
 EXTRACT VECTOR
       |
    HEDGEHOG
       |
GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

Any module that produces a vector field output

Modules that could be used in place of **extract vector**:

extract scalar
combine scalar

Modules that can process **extract vector** output:

Any module that can process vector fields

NOTE

This **extract vector** module is *not* the same as the **extract vector** module formerly available in the AVS user-contributed module library.

SEE ALSO

The example script STREAMLINES demonstrates the **extract vector** module.

NAME

field legend - select value from scalar field using color legend

SUMMARY

Name	field legend		
Availability	Imaging, Volume, FiniteDiff module libraries		
Type	mapper		
Inputs	field *n-dimensions n-vector any-data any-coordinates*		
Outputs	real		
Parameters	*Name*	*Type*	*Default*
	node data	choice	<data 1>
	value		float

DESCRIPTION

field legend takes a n-vector input field and a colormap and produces a "color legend" widget. The widget displays the range of values associated with one of the field's vector elements, and allows you to pick specific values of interest based on the colors associated with those values. Thus, the colors in the legend will match the colors used to display the field.

field legend displays the current colormap as a horizontal color legend. Beneath this table **field legend** prints a scale representing the range of values of one vector element of the input field. Values along this scale are displayed in scientific notation. The colormap is normalized to map to the range of values present in the input field. **field legend** behaves, in this respect, like the module **color range**. If the selected scalar has some label or unit associated with it (i.e. momentum, m/sec) **field legend** will print these as the title of the color legend.

By moving a "radio tuner" type dial along the color legend you can select specific data values. **field legend** outputs the value selected as a single floating point number.

field legend is designed to work with modules that take fields and allow you to visualize subsets of the data. Such modules include: **isosurface**, **thresholded slicer**, and **contour to geom**. Typically, subsets of data are selected by choosing specific values with a dial widget. For example, using **isosurface** you can select what "level" of data values to display as a surface. Manipulating colored data using **field legend**'s color legend is often more intuitive than using a floating-point parameter widget.

The module **field legend** accepts n-dimensional n-vector fields. Use the **node data** radio buttons to select one scalar element of the field to use for the color legend's range of values. If the input field is scalar to begin with **field legend** provides no buttons.

field legend outputs a single floating-point value. As a result it connects to the floating-point parameter port of another module. Before you can connect **field legend** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A purple parameter port should appear on the module icon. Connect this parameter port to the **field legend** module icon in the usual way one connects modules.

field legend

INPUTS

Data Field (required; field *n-dimensions n-vector any-data any-coordinates*)

An AVS field which supplies the range of values displayed by **field legend**.

Colormap (required; colormap)

An AVS colormap which is used as the legend for selecting values from the data field.

PARAMETERS

node data Selects the scalar element of the input data values to be used as the color legend's range. A set of radio buttons appears, showing the labels that are attached to the dimensions of the *n*-vector data.

value Dial to select the value that is placed on the **field legend** module's output port.

OUTPUTS

Real A single floating-point value selected from the range of values in the field.

EXAMPLE 1

The following network displays isosurfaces of a 3D scalar field. **field legend** allows you to select what "level" of values should be displayed as a surface. Note that **field legend** performs the equivalent of **extract scalar** and **color range**, but these two modules still need to filter the field that **isosurface** receives.

The **extract scalar** module is particularly important when the input field is a 3-vector field. Without the **extract scalar** module, the **field legend** module will display one blank radio button.

Also note that **generate colormap** sends the same colormap to both **field legend** and **color range**

```
GENERATE COLORMAP           READ FIELD
       |                |---|------|--------------|
       |    |-----------|   |      |              |
       |    |               | EXTRACT SCALAR     EXTRACT SCALAR
       |    |               |  |                  |
       |----|------------|  |  |                  |
       |    |            |  |  |                  |
       |    |      COLOR RANGE |                  |
       |    |               |  |                  |
   FIELD LEGEND             |  |  |---------------|
            |-------------| |  |  |
                          | |  |  |
                          ISOSURFACE
                              |
                              |
                        GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read field
any other module which outputs a 3D field

Modules that could provide the **Colormap** input:

generate colormap
color range

Modules that can process **field legend**'s output:

isosurface
thresholded slicer
contour to geom

Modules with similar function:

color legend

SEE ALSO

The example script FIELD LEGEND demonstrates the **field legend** module.

field math

NAME

field math – perform math operations between fields

SUMMARY

Name	field math
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates* field *same-dimension same-vector any-data same-coordinates* (OPTIONAL)
Outputs	field *same-dimension same-vector any-data same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
choice	choice	+		
Normalize	boolean	off		
Constant	float typein	0.0	*unbounded*	*unbounded*

DESCRIPTION

The **field math** module performs unary and binary operations upon fields.

The unary operations are Not, Square, and Sqrt. The binary operations are +, -, *, /, And, Or, Xor, Left-Shift, Right-Shift, and RMS (Root Mean Square). Unary operations are performed against the right port field only. The field that is connected to the left port is ignored. If only one field is provided as an operand for a binary operations, the field must be attached to the right port and the binary operations are performed on the right port field and the **Constant** input parameter.

When two fields are connected to the module, the **Constant** parameter is not displayed and the fields are evaluated against each other.

The input fields must be of the same dimensionality, size, and vector length. When the fields contain different data types, the output field will have the more elaborate data type.

When the fields have different coordinate types, the output field will have the same coordinate type as the right input port field.

Byte data is converted to integer, while short, integer, and float data are converted to double during computation. The result is then converted back to the appropriate output data type and "clamped" to the range:

[0...255]	byte
[-32767...32767]	short
[-2147483647...2147483647]	integer

if **Normalize** is turned off.

With **Normalize** turned on, the result is normalized to between:

[0...255]	byte
[0...32767]	short
[0...2147483647]	integer
[0...1]	float, double

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

The rightmost input field is used as the input to unary operations, or the first operand for binary operations.

Data Field (optional; field *same-dimension same-vector any-data same-coordinates*)
The left field is the second operand in binary operations. It must have the same dimension, size, and vector length as the first input field.

PARAMETERS

+
-
*
/
And (bitwise)
Or (bitwise)
Xor (bitwise)
Not (bitwise)
Left-Shift (bitwise)
Right-Shift (bitwise)
Square
Sqrt
RMS (Root Mean Square)

A choice of operations. For binary operations, if the left port field (field2) is not provided, the **Constant** parameter is used as the second operand. I.e., field2 is replaced by **Constant**.

```
+               field1  +  field2
-               field1  -  field2
*               field1  *  field2
/               field1  /  field2 (result is 0 if field2 is 0)
And             field1 AND field2 |
Or              field1 OR  field2 |
Xor             field1 XOR field2 | not applicable for
Not             NOT field1        | floats and doubles
Left-Shift      field1 << field2  |
Right-Shift     field1 >> field2  |
Square          field1 * field1
Sqrt            sqrt(field1)
RMS             sqrt (field1**2 + field2**2)
```

Normalize Selecting **Normalize** causes the results of the operation to be normalized to between 0 and 1 for floats and doubles, 0 and 255 for bytes, 0 and 32767 for shorts, and 0 and 2147483647 for integers. **Normalize** is off by default.

Constant A floating point typein to specify the constant value to be used as the second operand in binary operations. If two fields are connected to the module, **Constant** is ignored, and disappears from the control panel. The default is 0.0. There is no upper or lower limit.

OUTPUTS

Data Field (field *same-dimension same-vector any-data same-coordinates*)
The output field has the same form as the input fields. If the input fields differed in the data type, the output field will have the more elaborate data type. If the input fields had different coordinate types, the output field will have the same coordinate type as the right input port field.

The **min_val** and **max_val** attributes of the output field are updated and validated.

EXAMPLE 1

The following network inverts (flips the look-up table) an image using the Not function, with Normalize on. The same effect can be achieved by multiplying the image by -1.

```
         READ IMAGE
              |
      FIELD MATH
            |
    DISPLAY IMAGE
```

EXAMPLE 2

This network does a logical AND on a volume against the constant 128 (0x80) which produces a volume with only 0s and 255s based on whether the source voxel was greater or less than 128.

```
       READ VOLUME
            |
   FIELD MATH
            |
   ORTHOGONAL SLICER
            |
       COLORIZER
            |
     DISPLAY IMAGE
```

RELATED MODULES

Modules that could provide the **Data Field** inputs:

Any module that outputs a field

Modules that can process **field math** output:

Any module that inputs a field

Modules that can be used instead of **field math**:

ip fmath
ip arithmetic
ip logical

SEE ALSO

Two FIELD MATH example scripts demonstrate the **field math** module.

NAME

field to byte – transform any field to an byte-valued field

SUMMARY

Name	field_to_byte
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	field *same-dimension same-vector* byte *any-coordinates*

Parameters

Name	*Type*	*Default*	*Choices*
byte normalize	toggle	on	on,off

DESCRIPTION

The **field_to_byte** module takes a field of data (*integer*, *real*, *double*, or *byte*) and converts it to an *byte* field. It can be used in conjunction with volume visualization modules that have a bias towards byte fields (i.e., **compute gradient**).

By default, the input data is normalized to the range 0..255 If the toggle parameter **byte_normalize** is turned off, the data is "clamped" to that range instead. (See below for details.)

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

byte_normalize
This is a toggle parameter:

- **If on:** The data is transformed linearly into the range 0..255:

```
               (value - min) * 255
new_value =   ---------------------
                    max  -  min
```

- **If off:** The data is "clamped" so that no value falls outside the range 0..255:

If *value* < 0	*new_value* = 0
If 0 ≤ *value* ≤ 255	*new_value* = *value*
If *value* > 255	*new_value* = 255

OUTPUTS

Data Field (field *same-dimension same-vector* byte *same-coordinates*
The output field has the same dimensionality as the input field, but each scalar value is forced to be a byte.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume

Modules that could be used in place of **field_to_byte**:

field to short
field to int
field to float
field to double

Modules that can process **field_to_byte** output:

read volume

SEE ALSO

The example scripts FIELD TO BYTE and FIELD TO INTEGER demonstrate the **field to byte** module.

NAME

field to double – transform any field to a field of double-precision floating point values

SUMMARY

Name	field_to_double
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	field *same-dimension same-vector* double *same-coordinates*

Parameters

Name	*Type*	*Default*	*Choices*
double normalize	toggle	off	on,off

DESCRIPTION

The **field_to_double** module takes a field of data (*byte*, *real*, *double*, or *integer*) and converts it to an *double* field. This may be useful for computing fields at greater data resolutions.

By default, the input data is simply cast (re-typed) to be double-precision floating point. If the toggle parameter **double_normalize** is turned on, the data is also normalized to the range 0..1. (See below for details.)

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

The input data may be any AVS field.

PARAMETERS

double_normalize

This is a toggle parameter:

- **If on:** The data is transformed linearly into the range 0..1:

```
                   (value - min)
new_value =   ---------------------
                    max  -  min
```

- **If off:** The data is converted to double-precision floating point format.

OUTPUTS

Data Field (field field *same-dimension same-vector* double *same-coordinates*

The output field has the same dimensionality as the input field, but each scalar value is forced to be a double-precision number.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

read volume
field to byte
field to int
field to float

SEE ALSO

The example script FIELD TO INTEGER demonstrates the **field to double** module.

field to float

NAME

field to float – transform any field to a field of single-precision floating point values

SUMMARY

Name	field to float
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field field *any-dimension n-vector any-data any-coordinates*
Outputs	field *same-dimension same-vector* float *same-coordinates*

Parameters	*Name*	*Type*	*Default*	*Choices*
	float normalize	toggle	off	on,off

DESCRIPTION

The **field to float** module takes a field of data (*byte*, *short*, *real*, *double*, or *integer*) and converts it to an *float* field. It can be used in conjunction with modules that have a bias towards *float* fields (**particle advector**, **samplers**).

By default, the input data is simply cast (re-typed) to be single-precision floating point. If the toggle parameter **float normalize** is turned on, the data is also normalized to the range 0..1. (See below for details.)

INPUTS

Data Field (required; *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field.

PARAMETERS

float normalize
This is a toggle parameter:

If **ON**, the data is transformed linearly into the range 0..1:

```
                (value - min)
new_value =   ---------------------
                 max  -  min
```

If **OFF**, the data is converted to single-precision floating point format.

OUTPUTS

Data Field (field *same-dimension same-vector* float *same-coordinates*
The output field has the same dimensionality as the input field, but each scalar value is forced to be a single-precision number.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

read volume
particle advector
samplers
field to byte
field to short
field to int
field to double

SEE ALSO

The example script FIELD TO INTEGER demonstrates the **field to float** module.

LIMITATIONS

Overflow or underflow may occur when converting a double field to a float field with **float normalize** turned off.

field to int

NAME

field to int – transform any field to an integer-valued field

SUMMARY

Name	field to int
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	field *same-dimension same-vector* integer *same-coordinates*

Parameters

Name	*Type*	*Default*	*Choices*
int normalize	toggle	off	on,off

DESCRIPTION

The **field to int** module takes a field of data (*byte*, *short*, *real*, *double*, or *int*) and converts it to an *int* field. This may be useful for performing integer math with greater precision ($-2^{31}-1$ to $2^{31}-1$, -2147483647...2147483647) than that offered by byte fields (0..255).

By default, the input data is "clamped" to the range $-2^{31}-1...2^{31}-1$. If the toggle parameter **int normalize** is turned on, the data is normalized to $0..2^{31}-1$ instead. (See below for details.)

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

The input data may be any AVS field.

PARAMETERS

int normalize

This is a toggle parameter:

If ON, the data is transformed linearly into the range $0..2^{31}-1$:

```
               (value - min) * 2147483647
new_value =   ----------------------------
                         max  -  min
```

If OFF, the data is "clamped" so that no value falls outside the range -2147483647...2147483647. Values greater than 2147483647 are set to 2147483647. Values less than -2147483647 are set to -2147483647.

OUTPUTS

Data Field (field *same-dimension same-vector* integer *same-coordinat*)

The output field has the same dimensionality as the input field, but each scalar value is forced to be an integer.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

field to byte
field to short
field to float
field to double

SEE ALSO

The example script FIELD TO INTEGER demonstrates the **field to int** module.

NAME

field to mesh – transform a 2D scalar field to a surface in 3D space

SUMMARY

Name	field_to_mesh
Availability	Imaging, Volume, FiniteDiff module libraries
Type	mapper
Inputs	field 2D scalar *any-data any-coordinates* colormap (optional)
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Z scale	float	1.0	*unbounded*	*unbounded*

DESCRIPTION

The **field to mesh** module converts a two-dimensional field into a surface in 3D space, represented as a GEOM-format *mesh*. Each element of the field is mapped to a point in a base plane. The height of the mesh above each point in this plane is proportional to the scalar value of the field.

For irregular fields, the "base plane" need not actually be planar. The 2D grid of field elements is mapped into 3D space using the coordinate array included in the field description.

INPUTS

Data Field (required; field 2D scalar *any-data any-coordinates*)

The input data must be a 2D field with a scalar data value at each element. The data value may be of any primitive type: byte, integer, float, or double, and have uniform, rectilinear, or irregular coordinates.

Colormap (optional)

Colors each vertex of the mesh, according to the data value at that point. If no colormap is supplied, the vertices are colored white.

PARAMETERS

Z scale Determines the height of the mesh.

OUTPUTS

Geometry The output is an AVS *geometry*.

EXAMPLE 1

This example uses the "red band" (red component of the RGB color) of an image as a 2D field. It then converts this field to a mesh, using a colormap, and displays the mesh.

```
  READ IMAGE
      |
 EXTRACT SCALAR         (set dial to '1' for red band)
      |
      |       GENERATE COLORMAP
      |    +----------|
      |    |
 FIELD TO MESH
      |
GEOMETRY VIEWER
```

EXAMPLE 2

This example shows how to take orthographic slices through a curvilinear data set, showing them as <*XYZ*> meshes:

```
 READ FIELD   (read curv.fld)
     |
ORTHOGONAL SLICER
     |
     |     GENERATE COLORMAP
     |   +----------|
     |   |
FIELD TO MESH
     |
GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
read field
color range
generate colormap
extract scalar
orthogonal slicer

Modules that could be used in place of **field to mesh**:

arbitrary slicer

Modules that can process **field to mesh** output:

geometry viewer

LIMITATIONS

This module can output meshes that are too big for the **geometry viewer** module to handle, causing AVS to crash. Use the **downsize** filter module to reduce the size of the input data.

SEE ALSO

The example script COLOR RANGE demonstrates the **field mesh** module.

NAME

field to short – transform any field to a short--valued field

SUMMARY

Name	field to short
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	field *same-dimension same-vector* short *same-coordinates*

Parameters	*Name*	*Type*	*Default*	*Choices*
	short normalize	toggle	off	on,off

DESCRIPTION

The **field to short** module takes a field of data (*byte*, *short*, *real*, *double*, or *int*) and converts it to a *short* field. This may be useful for performing integer math with greater precision (-2^{15-1} to 2^{15-1}, –32767 to 32767) than that offered by byte fields (0..255).

By default, the input data is "clamped" to the range -32767...32767. If the toggle parameter **short normalize** is turned on, the data is normalized to 0..32767 instead. (See below for details.)

INPUTS

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

The input data may be any AVS field.

PARAMETERS

short normalize

This is a toggle parameter:

If ON, the data is transformed linearly into the range 0..32767:

```
               (value - min) * 32767
new_value =    ---------------------
                    max  -  min
```

If OFF, the data is "clamped" so that no value falls outside the range -32767...32767. Values greater than 32767 are set to 32767; values less than -32767 are set to -32767.

OUTPUTS

Data Field (field *same-dimension same-vector* short *same-coordinates*)

The output field has the same dimensionality as the input field, but each scalar value is forced to be a short.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

read volume
field to byte
field to int
field to float
field to double

SEE ALSO

The example script FIELD TO INTEGER demonstrates the **field to short** module.

field to ucd

NAME

field to ucd – convert AVS field to unstructured cell data format

SUMMARY

Name	field to ucd
Availability	UCD module library
Type	filter
Inputs	field 3D *n-vector any-data any-coordinates*
Outputs	ucd structure
Parameters	none

DESCRIPTION

field to ucd converts a 3D AVS field into a UCD structure. The cell connectivity list is generated automatically.

If the input field is scalar, **field to ucd** converts the scalar value at each location in the input field into the value of a node in the UCD structure. If the input field is an n-vector, **field to ucd** converts each element of the vector into a scalar component at each node in the output UCD structure. Note that the cells of the output structure will be hexahedra.

An AVS field is an array with a vector of values at each location. On the other hand unstructured cell data (UCD) has a hierarchical structure, consisting of structure data, cell data, and node data. Both structure data and cell data are optional, i.e., UCD structures may often contain only node data.

Structure data refers to data that holds for the entire structure. For example, in a simulation of forces on an object, the location of loads could be stored as structure data. Cell based data is particular to each cell in the structure.

At the lowest level are the nodes, which are the vertices of the cells. Each node can have several data components associated with it. Furthermore, each of these components may itself be either a vector or a scalar.

field to ucd computes the min and max extents of the structure.

Thus, if the input field has dimensions *width* ∗ *height* ∗ *depth*, there will be *width* ∗ *height* ∗ *depth* nodes in the output structure. The number of cells in the structure output by **field to ucd** would be *(width - 1)* ∗ *(height - 1)* ∗ *(depth -1)*.

If the type of the input field is irregular, the coordinates associated with each field data element become the coordinates of the UCD structure's nodes. If the input field is rectilinear, node coordinates are computed using the field's "points" information. If the input field is uniform, node coordinates are computed based on the implicit organization of the field array.

INPUTS

Data Field (required; field 3D *n-vector any-data any-coordinates*)

The input data must be a 3D field, with an *n-vector* of values at each location in the field. The field can be uniform, rectilinear, or irregular.

OUTPUTS

UCD Structure

The output structure is in AVS unstructured cell data (UCD) format.

EXAMPLE

The following network reads in an AVS field, converts it into a UCD structure, then into a geometry, and renders it:

```
GENERATE COLORMAP     READ FIELD
        |                 |
        |            FIELD TO UCD
        |                 |
        |  |--------------|
        |  |              |
   UCD CONTOUR            |
        |                 |
        |--------------|  |
                     UCD TO GEOM
                          |
                          |
                   GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read field
Any module that outputs a 3D field.

Modules that can process **field to ucd**'s output:

ucd to geom, ucd crop, ucd threshold, ucd extract, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d, ucd legend, ucd probe, ucd streamline, write ucd.

Modules that can be used instead of **field to ucd**:

scatter to ucd

SEE ALSO

The example script FIELD TO UCD demonstrates the **field to ucd** module.

file browser

NAME

file browser – send a filename to one or more module(s) filename parameter port(s)

SUMMARY

Name	file browser		
Availability	Imaging, UCD, Volume, FiniteDiff module libraries		
Type	data		
Inputs	none		
Outputs	string		
Parameters	*Name*	*Type*	*Default*
	File Browser	browser	NULL

DESCRIPTION

The **file browser** module sends a single user-specified filename string to one or more string parameter ports on one or more receiving modules. Its purpose is to allow you to simultaneously control filename parameter input to more than one module using only a single File Browser input widget.

Before you can connect **file browser** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A light blue parameter port should appear on the module icon. Connect this parameter port to the **file browser** module icon in the usual way.

PARAMETERS

File Browser (string)

The single filename string, specified through a File Browser widget, to be sent to the receiving module(s) filename string parameter port(s). The default value is NULL.

OUTPUTS

filename (string)

The filename string value is sent to all modules with filename string-type parameter ports that are connected to the **file browser** module.

EXAMPLE 1

The following network inputs the same data file simultaneously to two user-written modules.

```
               FILE BROWSER
                    |
                    |
      ______________________________
     |                              |
     |                              |
     |                              |
USER MODULE 1                  USER MODULE 2
```

RELATED MODULES

Modules that can process **file browser** output:

all modules with filename string parameter ports

SEE ALSO

The example script FILE BROWSER demonstrates the **file browser** module.

file descriptor

NAME

file descriptor – create a data form to read external format data files

SUMMARY

Name	file descriptor
Availability	Imaging, UCD, Volume, FiniteDiff module libraries
Type	data
Inputs	none
Outputs	field

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Select Data File	Browser			
read form	toggle	false		
write form	toggle	false		
header information	oneshot	false		
variable list	oneshot	false		
send data	oneshot	false		
Number of Data Files	typein	1	1	5
Logical Name for File *n*	typein	infile *n*		
Browser for File *n*	toggle	false		

DESCRIPTION

The **file descriptor** module is used to create a *data form* that specifies how to read an external format data file and convert it into an AVS field. This data form can be used either by the **file descriptor** module or the **data dictionary** module to read data into AVS.

To construct the data form, **file descriptor** presents an AVS Field Description Form panel. This panel allows the user to describe where in their external data file format the necessary information is located. Once a form has been filled in, the **file descriptor** module can use it to read in and convert the external file(s). The converted data is output as a field on the module's output port.

Alternately, the data form can be written to disk to be used by the **data dictionary** module to repeatedly read in other external data files with the same format.

This man page will not provide sufficient information for the new user to effectively use **file descriptor**. See the reference under "SEE ALSO" below for complete documentation.

PARAMETERS

Select Data File

A file browser widget. This file browser is shared among the **read form**, **write form**, and **Browser for File** *n* parameters. The correct order to select these options is: specify which other parameter the file browser will represent by pressing one of the **read form**, **write form**, or the various **Browser for File** *n* parameters. Then, select a file using this file browser widget.

read form A toggle button that sets the current state of the **Select Data File** browser. After this is selected, use the **Select Data File** browser to specify a form file to read. It will be read immediately upon specification.

write form A toggle button that sets the current state of the **Select Data File** browser. After this is selected, use the **Select Data File** browser to specify a form file to write. It will be written immediately upon specification.

header information
A oneshot button that displays a scrolling list with the field header information of the file being read in.

variable list
A oneshot button that displays a scrolling list with the list of variables that can be used in value typeins.

send data A oneshot button that causes the data to be read from the external file(s) and converted into a field. This field is then output on the module's output port.

Number of Data Files
A typein that determines the number of external data files that need to be read in order to create a field. Maximum value is 5. The value here determines the number of **Logical Name for File** *n* typeins and **Browser for File** *n* buttons that will be created.

Logical Name for File *n*
A set of typeins that determines a logical name for each of the external input data files. These controls only appear after a **Number of Data Files** greater than 1 has been specified.

Browser for File *n*
A set of buttons that set the current state of the **Select Data File** browser. First press one of these **Browser for File** buttons, then use the **Select Data File** browser to define which real file will be used as file *n*. Specify a real file for each **Browser** button, working down the list. No data will actually be read until either **send data** or **header information** is pressed.

OUTPUTS

Data Field (field)
The output is the field containing data held by the external data file being read.

EXAMPLE

This simple example displays an image.

```
FILE DESCRIPTOR
      |
DISPLAY IMAGE
```

RELATED MODULES

data dictionary

SEE ALSO

The "AVS Data Interchange Application" section of the *AVS Application Guide* describes importing data into AVS using **file descriptor**.

flip normal

NAME

flip normal – change direction of each vertex normal for a geometry object

SUMMARY

Name	flip normal
Availability	UCD, Volume, FiniteDiff module libraries
Type	filter
Inputs	geometry
Outputs	geometry
Parameters	none

DESCRIPTION

The **flip normal** module transforms an AVS *geometry* so that all the vertex normals point in the opposite direction. This is most often used to correct normals that have been calculated incorrectly.

When its normals are backwards, a 3D object appears unaffected by light sources; it frequently appears as a grey silhouette.

INPUTS

Geometry The input can be any AVS *geometry*.

OUTPUTS

geometry The output is an AVS *geometry* that represents the same object.

EXAMPLE

```
  READ GEOM
      |
 FLIP NORMAL
      |
GEOMETRY VIEWER
```

RELATED MODULES

read geom, offset, shrink, tube, render geometry, geometry viewer, ucd reverse cell

NOTES

Some filter modules (e.g. **offset**) sometimes produce bad normals, which can be corrected with **flip normal**.

SEE ALSO

The example script FLIP NORMALS demonstrates the **flip normal** module.

NAME

float – send a floating point number to one or more module(s) floating point parameter port(s)

SUMMARY

Name	float				
Type	data				
Inputs	none				
Outputs	float				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Float Value	dial	0.0	*unbounded*	*unbounded*

DESCRIPTION

The **float** module sends a single user-specified floating point value to one or more float-type parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control floating point parameter input to more than one module using only a single input widget (whether the default dial, or a typein).

Before you can connect **float** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A purple parameter port should appear on the module icon. Connect this parameter port to the **float** module icon in the usual way.

PARAMETERS

Float Value (float)

The single user-supplied floating point value to be sent to the module(s) floating point parameter port(s). The default value is 0.0. There is no minimum or maximum restriction on the value. You should be aware of the range of numbers that it is reasonable to send to the receiving modules. The default widget type is a dial. If you change this to a typein widget, then you should type the value as a real number, e.g., .55 or -100.25.

OUTPUTS

Float Output (float)

The floating point value is sent to all modules with floating point-type parameter ports connected to the **float** module.

EXAMPLE 1

The following network reads a field, then produces both a contour and an isosurface for the same floating point value, with both outputs composited in the **geometry viewer** display window.

float

```
                            READ FIELD
                                |
           |--------------------|
           |                    |
           |                    |
FLOAT      |                    |
   |       |                    |
   |-------|--------------|     |
   |       |              |     |
CONTOUR TO GEOM         ISOSURFACE
        |                   |
        |                   |
        |                   |
        |---------|---------|
                  |
            GEOMETRY VIEWER
```

RELATED MODULES

Modules that can process **float** output:

all modules with float-type parameter ports

NAME

generate axes – generate 3D geometric axes

SUMMARY

Name	generate axes				
Availability	Imaging, UCD, Volume, FiniteDiff module libraries				
Type	data				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Regenerate	oneshot			
	Colored Axes	boolean	on		
	center	typein	0 0 0		
	min	typein	-10 -10 -10		
	max	typein	10 10 10		
	axes	choice	All		
	All | X | Y | Z ...				
	Tick Marks	boolean	off		
	Tick Labels	boolean	off		
	Tick Length	float dial	0.5	0.0	*unbounded*
	Label Spacing	float dial	1.0	0.0	*unbounded*
	Tick Spacing	float dial	1.0	0.0	*unbounded*
	Tick Decimal Precision	int slider	0	0	10
	Label Font	int slider	0	0	20
	Label Height	float slider	0.08	0.01	.40
	Tick Label Font	int slider	0	0	20
	Tick Label Height	float slider	0.05	0.01	.40

DESCRIPTION

generate axes produces X, Y, and Z axes. Axes can have tick marks and / or tick mark labels. You can set attributes such as label font, tick spacing, tick length, tick label precision, tick label font, etc., for **All** axes, or you can control them for each individual X, Y, and Z axes.

The range of the axes is the geometric extent of the top level object when either the module is instanced or whenever the **Regenerate** button is pressed. This range can be manually reset with the axes **center**, **min**, and **max** typeins.

PARAMETERS

Regenerate

A oneshot that recalculates the range of the axes to be the geometric extents of the top level object. Where no specific object extent information is available, the axes extend from -10 to +10.

Colored Axes

Controls whether the axes are drawn in color (X is red, Y is green, Z is blue) or in a contrasting single color. This boolean is on by default.

center A floating point typein that sets the origin of the axes within the top level object. The default is 0 0 0.

min A floating point typein that sets the minimum extent of the axes. When no object is present, the default is -10 -10 -10. When an object is present, the default is the object's minimum X, Y, and Z extents.

Note that an object's minimum extents may not always produce axes that intersect at the 0 0 0 origin.

max A floating point typein that sets the maximum extent of the axes. When no object is present, the default is 10 10 10. When an object is present, the default is the object's maximum X, Y, and Z extents.

axes A set of radio buttons that switches among four sets of parameter widgets. The choices are **All**, **X**, **Y**, and **Z**. This gives you control over the appearance of the entire axes, or of an individual X, Y, or Z axis.

When **All** is selected, a set of parameter widget dials, sliders, and buttons is presented that will set values that will be applied to all (X, Y, and Z) axes.

When **X** is pressed, a set of parameter widget dials, sliders, and buttons is presentd that will set values that will be applied to just the X axis, etc.

The default is **All**.

Tick Marks
X Tick Marks
Y Tick Marks
Z Tick Marks

This is a boolean switch. If it is on, **generate axes** will produces hash marks along the axes. The hash marks are spaced according to the **Tick Spacing** parameter.

There are actually four different boolean switches that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

All default to off (no tick marks).

Tick Labels
X Tick Labels
Y Tick Labels
Z Tick Labels

This is a boolean switch. If it is on, **generate axes** produces numeric labels along the axes. The labels are spaced according to the **Label Spacing** parameter.

There are actually four different boolean switches that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

All default to off (no tick labels).

Tick Length
X Tick Length
Y Tick Length
Z Tick Length

A float dial that controls the length of the tick marks. The default is 0.5; the range is 0.0 to unbounded.

There are actually four different dials that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

Label Spacing
X Label Spacing
Y Label Spacing
Z Label Spacing

A float dial that controls the interval at which tick labels are drawn. Beginning at the **center**, this value is successively added and subtracted until **max** and **min** are reached. The default is 1.0; the range is 0.0 to unbounded.

There are actually four different dials that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

Tick Spacing
X Tick Spacing
Y Tick Spacing
Z Tick Spacing

A float dial that controls the interval at which tick marks are drawn. Beginning at the **center** this value is successively added and subtracted until **max** and **min** are reached. The default is 1.0; the range is 0.0 to unbounded.

When this parameter is set to less than 0.0, it snaps back to 0.1.

There are actually four different dials that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

Tick Decimal Precision
X Tick Decimal Precision
Y Tick Decimal Precision
Z Tick Decimal Precision

An integer slider that sets how many values to the right of the decimal point the tick labels will display. The default is 0; the range is 0 to 10.

There are actually four different sliders that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

Label Font
X Label Font
Y Label Font
Z Label Font

An integer slider that sets the font of the axes labels (the "X", "Y", and "Z"). The number-to-actual font correspondence varies from platform to platform. The default is 0. The hypothetical range is 0 to 20.

There are actually four different sliders that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

Label Height
X Label Height
Y Label Height
Z Label Height

A float slider that controls the size of the axes labels. Note that most systems support a limited number of font sizes. **Label Height** selects the closest actual font size. The default is 0.08; the range is 0.01 to .40.

There are actually four different sliders that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

Tick Label Font
X Tick Label Font
Y Tick Label Font
Z Tick Label Font

An integer slider that sets the font of the tick mark labels. The number-to-actual font correspondence varies from platform to platform. The default is 0. The hypothetical range is 0 to 20.

There are actually four different sliders that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

Tick Label Height
X Tick Label Height
Y Tick Label Height
Z Tick Label Height

A float slider that controls the size of the tick mark labels. Note that most systems support a limited number of font sizes. **Tick Label Height** selects the closest actual font size. The default is 0.05; the range is 0.01 to .40.

There are actually four different sliders that control **All** or individual axes. The **axes** radio buttons select which widget is displayed.

OUTPUTS

Geometry (geom)

The output is a geom containing lines and sometimes labels.

EXAMPLE

The following network generates a set of axes corresponding to a data set read in.

```
GENERATE AXES           READ GEOM
      |                     |
      |---------------------|
                 |
                 |
         GEOMETRY VIEWER
```

RELATED MODULES

Modules that can process **generate axes**'s output:

tube
geometry viewer

SEE ALSO

The example script GENERATE AXES demonstrates the **generate axes** module.

NAME

generate colormap – output AVS colormap

SUMMARY

Name	generate colormap
Availability	Imaging, UCD, Volume, FiniteDiff module libraries
Type	data
Inputs	none
Outputs	colormap

Parameters

Name	*Type*	*Default*	*Min*	*Max*
lo value	float	0	none	none
hi value	float	255	none	none
hue				
saturation				
brightness				
opacity				
composite				
edit	popup window			
read				
write				

DESCRIPTION

The **generate colormap** module produces an AVS *colormap* data structure, for use by modules that transform input data into color values. These modules include:

colorizer
arbitrary slicer
bubbleviz
field to mesh
isosurface

Note that when the range of values in the input field is not evenly distributed between 0 and 255, or if much of the data lie outside the 0 to 255 range, you can use the **color range** module to effectively scale the output colormap to the range of your data. For a more detailed description, see the man page for **color range**.

This module bases its output colormap on the state of the *colormap editor* control widget, which is invoked by clicking the **edit** button in the control panel. The colormap editor uses a *hue-saturation-brightness* (HSB) color space model:

hue	0.00 = red 0.16 = yellow 0.33 = green 0.50 = cyan 0.66 = blue 0.83 = magenta
saturation	0.00 = white 1.00 = **hue**
brightness	0.00 = black 1.00 = **hue**

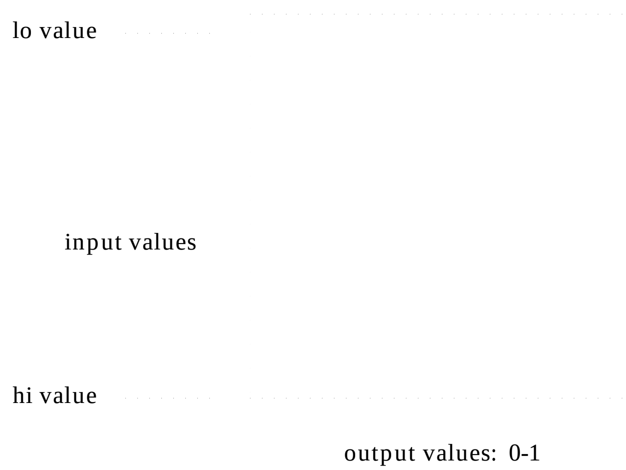
The HSB color space can be thought of as an inverted cone:

- The **hue** axis runs circularly around the cone.

- The **saturation** axis runs from the center of the cone (white) to its perimeter (fully saturated color).
- The **brightness** axis runs from the tip of the cone (black) to the base (white).

You can change an editing panel from its current setting by scribing a curve with the mouse. Place the mouse cursor anywhere within the editing panel, hold down any mouse button, and drag upward or downward.

Each editing panel is organized as follows:



PARAMETERS

The state of the colormap editor control widget specifies the colormap to be generated. This widget is a popup window that includes four *editing panels* and eight buttons. The editing panels are:

hue Raises the **hue** editing panel. The default panel is a linear ramp: 0=blue through 255=red.

saturation Raises the **saturation** editing panel. The default panel has all colors fully saturated: 0–255 = 1.0.

brightness Raises the **brightness** editing panel. The default panel has all colors at full brightness: 0–255 = 1.0.

opacity Raises the **opacity** editing panel. (The **opacity** value is placed in the *auxiliary* field of the colormap.) The default panel is a linear ramp: 0=0.0 through 255=1.0.

The following buttons apply to the editing panel that is currently visible:

composite

This is a toggle — when ON, the editing panel becomes a composite of the hue, saturation, and brightness panels, showing the actual colors that will be used. A line through the composite panel display indicates the status of the currently-selected panel: hue, saturation, brightness, or opacity.

edit

Press this button to pop up an editing window for the current panel. The editing window includes these settings:

Min

Max In the HSB color model, the hue is represented as a circle. By default, the colormap produces hues between 0† and 240† around this circle. This is the hue range from red to blue. The

Min and **Max** parameters allow you to select another hue range.

From / Value
To / Value
do interpolation

These controls provide precise numeric control over the mapping of input values to output colors. This is an alternative to scribing a freehand mapping with the mouse. For example, suppose the input values range from 0 to 175, but the values in the range 160–165 are critical. It would be desirable to have the values in the critical range be mapped to a contrasting hue (or range of hues). To accomplish this, set **From** to 160 and **To** to 165. Set the two **Value** settings to numbers that produce a contrasting hue, e.g. 0.0 (bright red) as the From Value and 0.1 (semi-bright red) as the To Value. Then press the **do interpolation** button to redefine the portion of the colormap specified by the above settings as a linear ramp.

invert Inverts the current editing panel along a horizontal axis. The hue (or saturation, etc.) assigned to the *lo value* becomes assigned to the *hi value*, and vice-versa.

flip Flips the current editing panel along a vertical axis. Each input value is mapped to the complementary output value (e.g. an opacity of 0.667 is becomes 0.333).

cycle Performs a circular shift on the current editing panel. For example, with a **Step** value of 10, pressing the **cycle** button effectively moves the image in the editing panel down by 10 slots (out of 255). Subsequent presses of **cycle** move the image again and again.

ramp Generates a linear ramp on the currently raised editing panel: *lo value* =0.0 through *hi value*=1.0.

smooth Smooths the curves of a hand-scribed editing panel.

read

Reads a colormap from disk storage. Pressing this button pops up a File Browser widget, allowing you to specify a filename. You can also change the working directory.

write

Writes the current colormap to a disk file. Pressing this button pops up a File Browser widget, allowing you to specify a filename. You can also change the working directory.

lo value

(see *LIMITATIONS* below) a floating point dial which specifies the minimum data value that can be used as input to the colormap (the value at the top of the editing panel). The default low value is 0.

hi value

(see *LIMITATIONS* below) a floating point dial which specifies the maximum data value that can be used as input to the colormap (the value at the bottom of the editing panel). The default high value is 255.

OUTPUTS

colormap The output is an AVS *colormap*.

COLORMAP FILE FORMAT

Colormaps are stored on disk as ASCII files, in the following format:

```
number_of_entries
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
low_value high_value
```

The hue, saturation, brightness, and opacity values are normalized to the range 0.0 – 1.0. The default colormap has 255 entries, with the hue, saturation, brightness, and opacity default values as described above.

EXAMPLE

The following network reads in a 3-vector field, i.e. every field location has 3 values associated with it. The **extract scalar** module selects one of the fields values. **color range** stores the field's min and max values so that the colormap can be scaled to the range of data in the field:

```
                       READ FIELD
                                 |
    GENERATE COLORMAP            |
           |                     |
           |               EXTRACT SCALAR
           |                     |
           |                     |---------|
           |---------------|     |         |
                           |     |         |
                         COLOR RANGE       |
                                 |         |
                                 |-----|   |
                                       |   |
                               ARBITRARY SLICER
                                       |
                                       |
                                GEOMETRY VIEWER
```

RELATED MODULES

color range
minmax

LIMITATIONS

The **generate colormap** module can only generate colormaps with 255 entries.

SEE ALSO

The example scripts COLOR RANGE, PROBE, as well as others demonstrate the **generate colormap** module.

NAME

generate filters - generate 2D filters for image processing

SUMMARY

Name	generate filters
Availability	Imaging module library
Type	data
Outputs	field 2D scalar float

Parameters

Name	*Type*	*Default*	*Min*	*Max*
selection	choice	Gaussian		
Size	integer	3	1	65
focus1	float	0.5	0.0	10.0
focus2	float	0.25	0.0	10.0
power	float	1.0	0.0	10.0
angle	float	0.0	0.0	360.0
scale	float	0.5	0.0	1.0

DESCRIPTION

generate filters produces 2D scalar fields of floating point values. These can be used as convolution filters in image processing by feeding them into the **convolve** module.

generate filters outputs the following filters: Gaussian, Laplacian, Power, Ellipse, Line, Random, dx, and dy. All filters, except Laplacian and Random, are normalized to the range 0.0 to 1.0.

PARAMETERS

selection Sets the function used to produce the image processing filter. Each functions has a number of parameter dials associated with it. Only the dials associated with a given function will be visible when you select that function. There are eight options:

Gaussian
Generates filters using a normal-distribution, bell-shaped, function. The Gaussian operator is typically used as a low-pass filter to smooth or blur images.

Laplacian
Generates "mexican hat" shaped function. The Laplacian function produces a high-pass filter. A Laplacian function is produced as the difference between two Gaussian functions. This is why there are two foci for the Laplacian functions: one for each of the two component Gaussians. Laplacian filters are not normalized to the range of 0.0 to 1.0.

Power
Generates an exponential function.

Ellipse
Generates an elliptical function, with two foci.

Line
Generates a filter that has the effect of blurring an image along a given line.

Random
Generates a uniformlly distributed random filter that is not normalized.

dx Generates the x component of the Sobel operator (see **sobel**), which detects changes in the image in the x direction. This can be used to locate vertical edges in images. The **dx** filter is 3x3 and cannot be resized.

dy Generates the y component of the Sobel operator (see **sobel**), which detects changes in an image in the y direction. This can be used to locate horizontal edges in images. The **dy** filter is 3x3 and cannot be resized.

Size Determines the length of the filter's sides. Filters are squares. NOTE: convolving a filter with an image is a N x M operation, where N is the number of elements in the convolution filter and M is the number of elements in the image. Consequently, filters of sizes over 16 require a great deal of computation. The size parameter is used by all of the functions.

focus1 Used in Gaussian, Power, and Line filters to control the width and amplitude of the filter function, which are inversely related. In the Laplacian filter, this controls the width and amplitude of one of the two component Gaussian functions. In the Ellipse filter, this controls the ellipse's first focus.

focus2 In the Laplacian filter, this controls the width and amplitude of the second component Gaussian function. In the Ellipse filter, this controls the ellipse's second focus.

power Value between 0.0 and 10.0, used in the Power filter to set the exponent of the function.

angle Value between 0.0 and 360.0, used in the Line filter to set the angle of the line relative to the horizontal.

scale Value between 0.0 and 1.0, used with the Laplacian or random filters to reduce the range of the function's values.

OUTPUTS

Filter The output is a 2D field of scalar floats, i.e. a grid where every location contains one floating point value.

EXAMPLE 1

The following network generates a filter, convolves it with an image, then displays the result:

```
GENERATE FILTERS          READ IMAGE
        |                      |
        |-----|     |-----------|
              |     |
            CONVOLVE
               |
               |
          IMAGE VIEWER
```

EXAMPLE 2

The following network shows what the convolution filters produced by **generate filters** look like, both as an image, and as an x-y graph. The module **colorizer** makes an AVS *image* out of the filter and colors it with a colormap output by **generate colormap** (NOTE: the colormap's max value must be changed to some small number, such as 0.03, using the Dial Editor). At the same time, **orthogonal slicer** generates a cross section through the filter, which can then be displayed as a histogram using the **graph viewer** module. (NOTE: set **orthogonal slicer** to slice through the middle of the filter.)

```
GENERATE COLORMAP       GENERATE FILTERS
         |                     |
         |--------|   |--------|----------------|
                  |   |                         |
               COLORIZER               ORTHOGONAL SLICER
                    |                           |
                    |                           |
             DISPLAY IMAGE               GRAPH VIEWER
```

EXAMPLE 3

The following network shows how you can combine the **dx** and **dy** filters into the equivalent of a "sobel" edge detecting operator:

```
GENERATE FILTERS(dx)        READ IMAGE     GENERATE FILTERS(dy)
        |                       |                   |
        |   |-------------------|---------------|   |
        |   |                                   |   |
     CONVOLVE                                CONVOLVE
          |                                       |
          |--------------------| |----------------|
                               | |
                          FIELD MATH (RMS)
                                |
                                |
                         DISPLAY IMAGE
```

RELATED MODULES

Modules that can process **generate filter**’s output:

convolve
colorizer
orthogonal slicer

Modules that can be used instead of **generate filters**:

ip read kernel
ip convolve

SEE ALSO

The example script GENERATE FILTERS demonstrates the **generate filter** module.

generate grid

NAME

generate grid – create grids on XY, XZ and YZ coordinate planes

SUMMARY

Name	generate grid		
Availability	Imaging, UCD, Volume, FiniteDiff module libraries		
Type	data		
Inputs	*none*		
Outputs	geometry		
Parameters	*Name*	*Type*	*Default*
	width	float typein	11
	height	float typein	11
	depth	float typein	11
	NX	int typein	11
	NY	int typein	11
	NZ	int typein	11
	XY	boolean	true
	XZ	boolean	true
	YZ	boolean	true
	x-offset	float dial	0
	y-offset	float dial	0
	z-offset	float dial	0

DESCRIPTION

The **generate grid** module creates a geometry representation of the coordinate planes XY, XZ and YZ in the form of grid. The user can control the size of the grids, number of grid lines, and initial position for each plane. The size of the grids and their initial position is determined by the extents of the top level object in the Geometry Viewer when the module is dragged into the Workspace.

PARAMETERS

width Specifies the size of XY and XZ grid plane in X direction.

height Specifies the size of XY and YZ grid plane in Y direction.

depth Specifies the size of XZ and YZ grid plane in Z direction.

NX Specifies the number of grid lines in X direction. The extents are divided by NX.

NY Specifies the number of grid lines in Y direction. The extents are divided by NY.

NZ Specifies the number of grid lines in Z direction. The extents are divided by NZ.

XY Controls whether the X-Y plane is drawn.

XZ Controls whether the X-Z plane is drawn.

YZ Controls whether the Y-Z plane is drawn.

x-offset Specifies the distance in the X direction from the minimum X extent of the top level object's coordinate system to the origin of the grid coordinate system.

y-offset Specifies the distance in the Y direction from the mimimum Y extent of the top level object's coordinate system to the origin of the grid coordinate system.

z-offset Specifies the distance in the Z direction from the mimimum Z extent of the top level object's coordinate system to the origin of the grid coordinate system.

OUTPUTS

Geometry (geometry)
The output is a geometry of grid lines.

EXAMPLE

The following generates a set of grids corresponding to the data set read in.

```
GENERATE GRID          READ GEOM
      |                      |
      |----------------------|
               |
               |
         GEOMETRY VIEWER
```

RELATED MODULES

Modules that can process **generate grid**'s output:

tube
geometry viewer

Modules that can be used with **generate grid**:

create geometry
generate axes

generate histogram

NAME

generate histogram – plot distribution of data values in a scalar field

SUMMARY

Name	generate histogram
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension* scalar *any-data any-coordinates*
Outputs	field 2D scalar float

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Number of Bins	integer dial	256	1	1024
Min Bin	float dial	0.0	*unbounded*	*unbounded*
Max Bin	float dial	255.0	*unbounded*	*unbounded*
Choice	choice	histogram		
Normalize	boolean	on		

DESCRIPTION

The **generate histogram** creates an output field that characterizes the distribution of data values in a scalar field. This output field is intended to be plugged into the **graph viewer** module to be plotted, either as a curve or a bar graph.

Picture an "empty" bar graph. The **Min Bin** and **Max Bin** dial settings determine the range of data values that will be counted. **Number of Bins** determines how many discrete chunks ("bins") the whole range of data values in the input field will be divided into. (**Max Bin - Min Bin**) / **Number of Bins** determines the range of each chunk.

generate histogram reads the input field and examines each value. It decides which sub-data range bin the value would fit in, and increments the integer count for that bin by one. If the value is below **Min Bin** or above **Max Bin**, it is discarded.

generate histogram produces a 2D output: a 2 by **Number of Bins** array where each bin has a data pair: the bin range, and an integer count of the number of original data values that fell into that range. The **graph viewer** uses the bin counts to construct the Y-axis, and the range values to construct and label the X-axis with the value of the bin range. The **graph viewer** knows to interpret this as "Plot as XY" data.

Alternatively, if **cumulative** was selected instead of **histogram**, each bin count reflects its own count *plus* the count of all previous bins.

In either case, the output field should be connected to the **graph viewer** module's rightmost "linear plot" port.

INPUTS

Data Field (required; field *any-dimension* scalar *any-data any-coordinates*)

A scalar AVS field whose distribution of data values is to be counted.

PARAMETERS

Number of Bins

An integer dial that determines how many chunks the range of data values is to be divided into. The default is 256. The minimum allowable is 1, the maximum is 1024.

Min Bin
Max Bin

Two floating point dials that set the endpoints of the range of data values to count. If **Normalize** (default) has been selected, the **Min Bin** and **Max Bin** dials will be initially set to the actual minimum and maximum data values in the input data. Without **Normalize Min Bin** is initially set to 0.0, and **Max Bin** to 255.0. This parameter is unbounded.

Normalize The **Normalize** switch determines whether the **Min Bin** and **Max Bin** dials will be automatically set to the actual minimum and maximum data values in the field. Without **Normalize**, you would need to have some idea of the real data value range in the input field so that you could set the dials in a way that would not inadvertently discard data. With **Normalize** on, **generate histogram** examines the input field's data structure to see if minimum and maximum values have been specified. If they are present, it uses them. If they are not present, it calculates the actual minimum and maximum in order to set the dials.

When **Normalize** is on the **Min Bin** and **Max Bin** dials can not be used; if they are moved, they will "snap back" to their original values. **Normalize** is on by default.

histogram
cumulative

A choice that decides how the data values are counted. If **histogram** (the default) is chosen, each bin contains a count of the number of data values that fell into its sub-range. If **cumulative** is selected, each bin contains a count of the number of data values that fell into its sub-range, *plus* the total of all bins preceding it.

OUTPUTS

Data Field (field 2D scalar float)

The output field is a 2D field, **Number of Bins** long by 2 wide, with each element pair a count of the number of data values that fell into its range and the range itself. It is used as "Plot as XY Data" input to the **graph viewer** module's rightmost input port.

EXAMPLE 1

The following network reads in a volume (byte data in the range 0 to 256), calculates the distribution of values, and graphs the result:

```
     READ VOLUME
          |
          |
  GENERATE HISTOGRAM
          |
          |
     GRAPH VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** inputs:

Any module that outputs a field

Modules that can process **generate histogram** output:

graph viewer

See also **statistics**, **ucd plot**, **ip read line**

SEE ALSO

The example scripts GENERATE HISTOGRAM and GRAPH VIEWER demonstrate the **generate histogram** module.

geometry viewer

NAME

geometry viewer – render and display geometry

SUMMARY

Name	geometry viewer
Availability	Imaging, UCD, Volume, FiniteDiff module libraries
Type	data output
Inputs	geometry (optional, multiple) field 2D / 3D uniform byte, scalar or 4-vector (*texture map, optional*) colormap (*optional*)
Outputs	field 2D 4-vector byte (*image*) upstream transform (*optional, invisible, autoconnect*) upstream geometry (*optional, invisible, autoconnect*) field 2D scalar float pixmap (*invisible*) integer (*invisible, for synchronization*)

Parameters

Name	*Type*	*Default*
Update Always	boolean	on
Update Image	oneshot	

DESCRIPTION

The **geometry viewer** module provides access within an AVS network to the complete Geometry Viewer subsystem. Many different modules can supply input geometries. That is, many *geometry*-format outputs can be connected to **geometry viewer**'s geometry input port. All the objects will be combined into a single scene. Each module providing input to **geometry viewer** can define attributes and geometries for any number of objects. Each of these modules can also define a hierarchical relationship among its objects.

You can also invoke **geometry viewer** with no inputs, so that the "scene" is initially empty. Objects can be added to a scene either by upstream modules or by the **Read Object** selection on the Geometry Viewer control panel. Geometries and descriptions sent by upstream modules can be saved to files using the **Save Object** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Object**.

INPUTS

Geometry (optional, multiple; geometry)

The input data can be any AVS *geometry*. More than one geometry can be input to this port. All the geometries will be combined into the same "scene".

Texture (optional; field 2D / 3D uniform byte, scalar or 4-vector)

The optional input provides one way to perform dynamic texture mapping. The AVS 2D or 3D uniform byte field input to this port is available as a dynamic texture.

An upstream module such as **brick** can bind this texture with an object. If no upstream module does this, then you must make the binding manually by pressing **Set Dynamic Texture** on the **Edit Texture** panel under the **Objects** submenu.

Modules such as **brick**, **excavate brick**, **colorize geom**, and **volume render** use this input port.

Not all hardware renderers support 2D and 3D texture mapping; the

Software Renderer supports both.

Colormap (optional, colormap)

This port is used to create colorized texture maps. An upstream module that wants to produce a colorized, texture mapped geometry has two choices: it can create a geometry with texture mapping data *and* color values specified; or it can create a geometry with texture mapping data, but no color values specified. If it produces this second kind of geometry, then the **geometry viewer** will use the colormap provided on this input port to colorize the object's texture map. If no colormap is provided, **geometry viewer** uses a grayscale colormap.

Most AVS modules that produce texture mapped objects (**brick**, **excavate brick**, **colorize geom**, **volume render**) produce a colorized texture mapped geometry, and thus do not need this port.

This port is only effective with the Software Renderer, and those hardware renderers that support 2D and 3D texture mapping.

PARAMETERS

Update Always

This switch can be used to improve performance on hardware renderers. It is only effective when a module is connected to the **geometry viewer**'s image or Z buffer output port. It is invisible by default.

When this switch is on, every time the scene changes the **geometry viewer** module translates the contents of the frame buffer into an AVS image and sends it to the image output port. If this switch is off, the **geometry viewer** will only translate the frame buffer when the **Update Image** oneshot is pressed. Similarly, Z buffer information is produced or not produced. The default is on.

To use this parameter, first use the Module Editor's (middle or right mouse button on the module dimple) Parameter Editor to make the **Port Visible**. Then, you can either connect the **boolean** module to the new parameter port, or you can create a module control panel for the **geometry viewer** with an **Update Always** button on it by setting **toggle** on the Parameter Editor.

Update Image

A oneshot switch that causes the **geometry viewer** to translate the contents of the frame buffer into an AVS image and send it to the image output port. **Update Image** works no matter how **Update Always** is set.

This parameter is invisible by default. To use it, make it visible in the same way as described for **Update Always**. Then, either connect the **oneshot** module to the parameter port, or set **oneshot** with the Parameter Editor to create a module control panel with an **Update Image** button on it.

OUTPUTS

Image This output is an image containing a *scene* that includes all the input objects. Note that it is not necessary to connect anything to this port for normal operations. This port gives other modules access to the image output by the renderer. One use of this port would be to produce a printable PostScript file with the **image to postscript** module.

Upstream Transform

This port outputs an upstream transformation structure. This structure contains object transformation information that can be used by a module that is connected to **geometry viewer**'s geom input port to create changes in the geometry it outputs to match direct mouse manipulation transformations performed by the user in the ometry viewer's window. Upstream transformations are discussed in the "Advanced Topics" chapter of the *AVS Developer's Guide*.

This port is normally invisible. It is optional. The upstream connection will be made automatically if a module immediately upstream of **geometry viewer** has a matching upstream transformation input port.

Upstream Geometry

This port outputs an upstream geometry structure. This structure contains object picking information that can be used by a module that is connected to **geometry viewer**'s geom input port to create changes in the geometry it outputs to match direct mouse manipulation selections performed by the user in the **geometry viewer**'s window. Upstream geometries are discussed in the "Advanced Topics" chapter of the *AVS Developer's Guide*.

This port is normally invisible. It is optional. The upstream connection will be made automatically if a module immediately upstream of **geometry viewer** has a matching upstream geometry input port.

Z Buffer This output is a field containing the depth information in the scene. It is implemented in support of future functionality. On some systems, connecting a module to this port will slow the rendering process.

pixmap This output is an AVS pixmap (see "AVS Data Types" chapter in the *AVS Developer's Guide*). It is invisible by default. It is provided for those people who had previously used the pixmap output field of the **render geometry** module to obtain the X window id of the window into which the **geometry viewer** draws.

integer This port outputs an integer. It is invisible by default. This integer is merely a signal generated each time the **geometry viewer** finishes re-rendering. It is used to synchronize **geometry viewer** output with a module that might control a video camera or other device. Use this output port instead of the image output port since acquiring the image for output can affect the module's efficiency.

SPECIAL CONSIDERATIONS

This module is special: instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level application menu of the Geometry Viewer subsystem. Thus, when you add the **geometry viewer** icon to a network, no page is added to the Network Control Panel. There are two ways to access the Geometry Viewer menu:

- Click the small square in **geometry viewer** icon with the left mouse button.
- Press and hold down the **Data Viewers** button located at the top of the each subsystem's left control panel. This brings up a pulldown menu of subsystems. Roll down the list and select **Geometry Viewer**.

Note: If the **Update Always** and / or **Update Image** parameters have been made into toggle and oneshot buttons—thus creating a **geometry viewer** module control panel—then the only way to access the main Geometry Viewer control panel is with the **Data Viewers** button.

In some circumstances, it is useful to be able to access both the Geometry Viewer control panel and the Network Control Panel simultaneously. They both occupy the same screen position, along the left edge of the screen. In these cases, use the X Window System window manager to move the one of these menu windows out of the way.

The **geometry viewer**'s control panel also differs from that of other modules in these ways:

- The Network Editor's **Layout Editor** cannot be used to rearrange Geometry Viewer controls.
- If a network includes more than one instance of **geometry viewer**, AVS does *not* create a separate control panel for each instance. Each **geometry viewer** sends its output to a different window, but the same Geometry Viewer application menu controls all the windows. The module whose output window is highlighted in red is the one being controlled. (Current windows that are displayed on remote heads are not highlighted in red.) To switch the focus to another **geometry viewer** output window, just click in it with any mouse button.

GEOMETRY VIEWER VS RENDER GEOMETRY MODULES

In AVS4 and later releases, the **geometry viewer** module takes the place of the older **render geometry/display pixmap** module pair. (**render geometry** and **display pixmap** are retained in the Unsupported module library for backward compatibiity, and still appear in many sample networks.) The **geometry viewer** module is similar in function to **render geometry/display pixmap**, with one major exception: it outputs an AVS image format field (2D 4-vector uniform byte) rather than a pixmap. This has the following advantages:

- Various output modules including the **image to postscript** module and the Animation Application's post-processing modules (e.g., **write frame sequence**) all use AVS image format field data for their input ports. You will not need to insert a **pixmap to image** module between **geometry viewer** and the output modules to convert the data format as you need to do with **render geometry** module.
- Systems that support less than 24-plane true color (such as an 8-plane pseudocolor system) use X images to display their output on the screen. These images are dithered down to the limitations of the X server visual. (For example, on an 8-plane system, 16,777,216 possible color values must become one of 216 possible color values.) If you generate output files from the output of a **render geometry** module (through **pixmap to image**) on such a system, you never get back the full 24-bit true color fidelity the visualization possessed before it was dithered for screen display.

  If you use the software renderer option, the **geometry viewer** module's image output port will produce a full 24-plane true color representation of the display data, even on systems with more limited X server display capabilities.

The **geometry viewer** module should be used instead of **render geometry/display pixmap** in AVS networks.

EXAMPLE 1

This network creates a tube version of an object:

```
                        READ GEOM
                            |
                        WIREFRAME
                            |
                          TUBE
                            |
                  GEOMETRY VIEWER
```

EXAMPLE 2

This network shows a configuration that will input an image that can be used as a 2D texture map on an object into the the **geometry viewer**'s center port. Once the image is read, toggle **Set Dynamic Texture** on the Geometry Viewer's **Edit Texture** panel.

```
              READ IMAGE      READ GEOM
                  |               |
                  |-----|   |-----|
                        |   |
               GEOMETRY VIEWER
```

EXAMPLE 3

The following network shows how **geometry viewer**'s center input port is used to perform 2D/3D texture mapping using the **brick** module. The network reads a byte volume which is sent **colorizer** to paint the byte values as colors, to **brick** to map the surfaces, and to **volume bounds** to draw a box around the limits of the volume. The **generate colormap**, and **colorizer** create the 3D texture map, which is fed to **geometry viewer** through the left input port.

```
                            READ VOLUME
                                  |
      GENERATE COLORMAP           |
              |                   |
              |      |----------|---------------|
              |      |          |               |
          COLORIZER        BRICK      VOLUME BOUNDS
                 |              |               |
                 |--------|     |---------------|
                          |     |
                   GEOMETRY VIEWER
```

EXAMPLE 4

This network shows **geometry viewer** producing a colorized texture map from a geometry, a 3D uniform byte field, and a colormap. The 3D uniform byte field is a vector field, thus one channel must be extracted. The texture map is associated with a particular geometry by selecting **Set Dynamic Texture** under **Object**'s **Edit Texture** panel.

```
                      READ FIELD    READ GEOM
                          |             |
GENERATE COLORMAP         |             |
              |     EXTRACT SCALAR      |
              |           |             |
              |           |             |
              |------|    |      |------|
                     |    |      |
                   GEOMETRY VIEWER
```

RELATED MODULES

read geom

SEE ALSO

The *Geometry Viewer* chapter of the *AVS User's Guide*.

The example scripts BRICK, FLIP NORMALS, PDB TO GEOM, as well as others demostrate the **geometry viewer** module.

gradient shade

NAME

gradient shade – apply lighting and shading to colored data set

SUMMARY

Name gradient shade

Availability Imaging, Volume, FiniteDiff module libraries

Type filter

Inputs field 2D / 3D 4-vector byte uniform *(colorized data)*
field 2D / 3D 3-vector real uniform *(gradient supplied by* **compute gradient***)*
field 2D scalar float (transformation matrix) (optional)

Outputs field *same-dimension* 4-vector byte uniform *(shaded version of colorized data)*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
ambient		0.1	0.0	1.0
diffuse		0.8	0.0	1.0
specular	float	0.0	0.0	1.0
gloss		20.0	0.0	50.0
lt theta	float	0.0	none	none
lt off-ctr	float	0.0	none	none

DESCRIPTION

The **gradient shade** module accepts a colored 2D or 3D data set, along with its gradients (supplied by the **compute gradient** module). It applies a single light source to the colored data, then shades it.

The gradient at each location in the data field substitutes for the *surface normal*, which is used in traditional algorithms for lighting and shading surfaces. (A surface normal at a particular point on a surface is a vector perpendicular to the surface.)

Various shading styles are achievable using the lighting controls (see *PARAMETERS* below). These include creating shiny and matte surfaces, and controlling the location of the light source.

INPUTS

Data Field (required; field 2D / 3D 4-vector byte uniform)
The input field is an image (2D pixel array) or a block of voxels (3D pixel array).

Gradient (required; field 3D 3-vector real uniform)
This field is the gradient of the **Data Field**.

Transformation Matrix (optional, field 2D scalar float)
The transformation matrix is applied to **gradient shade**'s light source, and is used to control the location of the light. This input has the same effect as the **lt theta** and **lt off-ctr** parameters.

PARAMETERS

The way in which all the following parameters determine the coloring of an object is described below.

ambient The contribution of ambient (uniform background) lighting to the color. When this is set to 0.0, all surfaces facing away from the light source are black. When this is set to 1.0, surfaces appear in their own colors, with no shading information present.

diffuse The contribution of diffuse (directional) lighting to the color.

specular The contribution of specular lighting to the color.

gloss The sharpness of the specular highlight. The larger this value, the smaller and sharper the specular highlights.

lt off-ctr The angle between the light source and the positive Z axis (which comes out of the screen at a right angle).

lt theta The angle between (1) the projection of the light source on the X-Y plane and (2) the positive Y axis. This value measures how much an off-center light source "swings around" the Z-axis.

With *lt theta* = 0.0 and *lt off-ctr* = 0.0, the light source is coming straight from the eye perpendicular to the data. A positive *off-ctr* value moves the light source "up" (in the positive Y direction); a negative value moves it "down".

The equation for calculating the intensity of light reflected by a spot of surface is:

$$(int_{amb} * ambient) + (int_{diff} * diffuse * \cos(phi)) + (int_{diff} * specular * \cos^{gloss}(lt\ off\text{-}ctr))$$

In performing this computation, **gradient shade**:

- Assumes that int_{amb} and int_{diff} are both maximal (1.0).
- Uses *lt theta* and *lt off-ctr* to compute *phi*, the angle between the surface normal (gradient vector) and the light source. The quantity **cos**(*phi*) is the attenuation (reduction) factor for the directional (diffuse) light.
- Computes the quantity $\cos^{gloss}(\alpha)$, the attenuation factor for the specular highlight.

OUTPUTS

Data Field (field *same-dimension* 4-vector byte uniform)
The output field has the same form as the **Data Field** input.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE 1

The following network shades a 2D image:

```
READ IMAGE
     |--------------
EXTRACT SCALAR     |    (choose 1 (= red))
     |             |
COMPUTE GRADIENT   |
     |    _________|
     |    |
GRADIENT SHADE
     |
DISPLAY IMAGE
```

EXAMPLE 2

The following network shades a 3D image:

gradient shade

```
                    READ VOLUME              GENERATE COLORMAP
                         |----------------|          |
                         |                |--------|  |
                    COMPUTE GRADIENT            COLORIZER
                         |   ._____________________|
          |---------|    |   |
          |         GRADIENT SHADE
          |                |
          |-------------| |
          |              TRACER
          |                |
          |                |
          |         DISPLAY TRACKER
          |-----------|
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume

Modules that could provide the **Gradient** input:

compute gradient

Modules that could be used in place of **gradient shade**:

compute shade
colorizer

Modules that can process **gradient shade** output:

display image (2D data)

Modules that can supply transformation matrices:

display tracker
euler transformation

See also **extract scalar**, which gets a single scalar height field from an image.

SEE ALSO

The example script ANIMATED FLOAT demonstrates the **gradient shade** module.

NAME

graph viewer – create XY and contour plots of data (Graph Viewer)

SUMMARY

Name	graph viewer
Availability	Imaging, UCD, Volume, FiniteDiff module libraries
Type	data output
Inputs	field *any-dimension* scalar *any-data any-coordinates* (*linear data, optional*) field *any-dimension* scalar *any-data any-coordinates* (*contour data, optional*) field 2D 4-vector byte uniform (*background image, optional*)
Outputs	geometry field 2D 4-vector byte uniform *(image)*
Parameters	none

DESCRIPTION

The **graph viewer** module provides access within an AVS network to the complete Graph Viewer subsystem. Many different modules can supply input data. That is, many *field*-format outputs can be connected to **graph viewer**'s input ports. Depending upon how **graph viewer** is set up, successive sets of incoming data will either replace an existing graph, be added to the graph, or be drawn in a new graph window.

You can also invoke **graph viewer** with no inputs, so that the graph is initially empty. Plots can be added to a graph either by upstream modules or by the various **Read Data** selections on the **graph viewer** control panel. Data sent by upstream modules can be saved to files in a variety of forms using the **Write ASCII XY Data**, **Write AVS Plot Data**, or **Write AVS Geometry Data** selections. In this way, you can save data plots and retrieve them later with **Read Data** selections. In addition, a grayscale PostScript image of the plot can be saved with the **Write PostScript** selection, or a color Postscript image saved by connecting the **graph viewer** module's left output port to the **image to postscript** module.

Note that the **graph viewer** window can be reparented to page and stack widgets using the AVS Layout Editor.

SPECIAL CONSIDERATIONS

This module is the module representation of the Graph Viewer subsystem. Instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level menu of the Graph Viewer subsystem. Thus, when you add the **graph viewer** icon to a network, no page is added to the Network Control Panel. There are two ways to access the Graph Viewer menu:

- Click the "dimple" in the **graph viewer** icon with the left mouse button.
- With the cursor positioned over the **Data Viewers** button located at the top of the Network Control Panel, press and *hold down* any mouse button. When the AVS Data Viewers pop-up menu appears, roll the mouse down to **Graph Viewer** and release the mouse button. This **Data Viewers** button is always visible, even when there is no active network.

In some circumstances, it is useful to be able to access both the Graph Viewer control panel and the Network Control Panel simultaneously. They both occupy the same screen position, along the left edge of the screen. In these cases, use the X Window System window manager to move one of these menu windows out of the way.

The **graph viewer**'s control panel also differs from that of other modules in these ways:

- The Network Editor's **Layout Editor** cannot be used to rearrange Graph Viewer controls.
- If a network includes more than one instance of **graph viewer**, AVS does *not* create a separate control panel for each instance. Each **graph viewer** sends its output to a different window, but the same Graph Viewer application menu controls all the windows. The module whose output window is currently highlighted in red is the one being controlled. To switch the *focus* to another **graph viewer** output window, just click in it with any mouse button.

RESIZING

The **graph viewer**'s pulldown menu, which is accessed by clicking on the "dimple" in the upper lefthand corner of the display window, provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen**. If the window originally was embedded in a page or stack, it will be re-embedded there.

INPUTS

Data Field (optional, field *any-dimension* scalar *any-data any-coordinates*)

The rightmost input port is for *linear* data that is to be made into an XY plot. If the input field is 1D, the values are taken to be Graph Viewer "plot as Y" data, meaning that they are interpreted as Y values that will be graphed against an evenly-spaced set of X values. If the input field is 2D, the values are taken to be Graph Viewer "plot as XY" data, meaning that they are interpreted as X and Y values. Although the **graph viewer** will accept fields of more than 2D, it will only graph the first two dimensions and ignore the rest. Many modules can create 2D subsets of fields (**orthogonal slicer** is an example). If such a module is used twice in succession (Example 2 below) a 1D subset of the field is created. Note that the values at each point must be scalar. If you have a vector field, you must use **extract scalar** or a module with similar effect to produce a scalar version of the field.

Data Field (optional, field *any-dimension* scalar *any-data any-coordinates*)

The center input port is for *contour* data that is to be made into a contour plot. If the input field is 2D, the values are taken to be Graph Viewer "plot as contour" data that is interpreted as X and Y values. There is no size limit on the input file, but if it is large you will get a warning message. The real limit is the size of available memory. Note that the values at each point must be scalar. If you have a vector field, you must use **extract scalar** or a module with similar effect to produce a scalar version of the field.

Image (optional, field 2D 4-vector byte uniform)

The leftmost input port accepts an AVS image. **graph viewer** normally plots its graphs against a black background. If you send an image into this port, it will be used as the background instead, and the plot window will be resized to match the image size.

OUTPUTS

Geometry (optional; geometry)

graph viewer can produce PostScript file versions of plots for hardcopy printing with its **Write Postscript** selection. If you want to create output that will print or display correctly on a different device, this output port leaves the option open for a module that converts AVS geometry-format files to the format of another type of device.

The **graph viewer** normally only creates this output when a new dataset enters it. At other times, use the **Output Geometry** button on the **Write Data** submenu, or the graph_output_geom CLI command.

Image (optional; field 2D 4-vector byte uniform)

graph viewer can produce PostScript file versions of plots for hardcopy printing with its **Write Postscript** selection. These PostScript plots are monochrome and do not contain the plot's background image. If you want to create output that will be in color and/or include the background image, the output port leaves the option open using the **image to postscript** module. Because the conversion of a plot into an image is a computationally intensive operation, the Graph Viewer does not update the image output port every time the current plot is changed. In order to get an image sent out through the **graph viewer** module's image output port, you must select the **Output Image** button in the **Write Data** menu.

EXAMPLE 1

This network reads a volume, then uses **orthogonal slicer** to section out a 2D slice of the volume for plotting as X and Y data. Note that if **graph viewer** is set up to *add* each additional set of data to an existing plot, one could then manipulate the **orthogonal slicer**'s **slice plane** dial to get a single graph with multiple plot lines showing successive slices through the volume.

```
   READ VOLUME
        |
        |
ORTHOGONAL SLICER
        |
        |
   GRAPH VIEWER
```

EXAMPLE 2

This network reads a volume, then uses the **orthogonal slicer** module *twice* to extract a 1D slice through the volume data:

```
   READ VOLUME
        |
        |
ORTHOGONAL SLICER
        |
ORTHOGONAL SLICER
        |
        |
  GRAPH VIEWER
```

EXAMPLE 3

This network reads an image, downsizes the image to a reasonable resolution for graphing, then extracts the "red" data channel from the 4-vector image representation. This data is fed to **graph viewer**'s middle (contour) input data port, and a contour plot of the reds in the image is displayed.

```
                READ IMAGE
                    |
                 DOWNSIZE
                    |
              EXTRACT SCALAR
                    |
               GRAPH VIEWER
```

EXAMPLE 4

This network does the same as above, but displays the contour plot on top of the *mandrill.x* image it is a contour of. As with the network above, downsize the image to some reasonable size, and extract either the red, green, or blue bytes from it. (NOTE: the image of the mandrill will be upside down. This is because 0,0 for an image is located in the upper left corner, while 0,0 for a graph is located in the lower left corner.) The contour data is fed to **graph viewer**'s middle (contour) input data port, and the image is fed in **graph viewer**'s leftmost (image) input data port.

```
                READ IMAGE
                    |
     |---------------|
     |               |
     |           DOWNSIZE
     |               |
     |         EXTRACT SCALAR
     |               |
     ----------|     |
            GRAPH VIEWER
```

EXAMPLE 5

This network plots a section through the Gaussian image-processing filter produced by **generate filters**:

```
                   GENERATE FILTERS
                          |
          |------------|------------|
          |                         |
   FIELD TO BYTE             ORTHOGONAL SLICE
          |                         |
      COLORIZER                     |
          |                   GRAPH VIEWER
    DISPLAY IMAGE
```

RELATED MODULES

generate histogram extract graph

SEE ALSO

The "Graph Viewer" chapter of the *AVS User's Guide*.

Two example GRAPH VIEWER scripts demonstrate the **graph viewer** module.

NAME

hedgehog – show vectors in a 3D 3-vector field

SUMMARY

Name	hedgehog
Availability	FiniteDiff module library
Type	mapper
Inputs	field 3D 3-vector float field irregular 3-space *(optional, from samplers module)* upstream transform *(optional, invisible, autoconnect)* field 3D scalar *(optional, for coloring arrows)* colormap *(optional, for coloring arrow)*
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Vector Scale	float dial	1.0	0.0	10.0
N segments	integer dial	16	2	64
Method	radio	point		
Sample	radio	point		
arrow heads	toggle	on	off	on
Show Bounds	toggle	on	off	on

DESCRIPTION

The **hedgehog** module takes as input a 3D uniform field whose values are 3-vectors of any primitive data type. That is, the data represents a volume of lattice points, each point having a 3D vector of *float* values. This 3D-vector value can be visualized as a small line segment with a particular length and direction.

The **hedgehog** module takes an arbitrarily-oriented (user-controlled) sample of locations within the volume. The sample object can be moved like any other geometry object. To select it, click on it with the left mouse button, or enter the Geometry Viewer and make it the current object. You can choose this sample to be:

- A single point
- A set of points on a line segment
- A set of points on a circle
- A set of points on a plane
- A volume of points
- All nodes (sampling object is ignored)

A bounding diagram is generated to show you the region in which the samples are generated. For the **point** sample, this bounds is represented as a 3-dimensional cross-hair. For other representations, it is represented as a line, a circle, a rectangle, and a retangular prism, depending on which sampling option is chosen. This bounding hull is generated by default, but may be turned off using the **Show Bounds** button.

The module outputs the line segment(s) representing the values of the vector field at the sample location(s). The lines optionally arrows at their ends, showing the direction of the vectors. Often, this collection of line segments resembles the coat of a hedgehog — hence the module's name.

Since arbitrarily oriented sample locations (all samplings except **nodes**) do not, in general, coincide with the lattice points in the data volume, an interpolation method is used to determine a field value based on the values of one or more nearby lattice points.

hedgehog can optionally receive input from the **samplers** module. **samplers** outputs a list of points in space, and these points become the starting location for advecting particles. When **hedgehog** receive input from the **samplers** module, the **N Segment** dial, and the **Sample** buttons disappear from the **hedgehog**'s control panel.

hedgehog generally generates white arrows, but if a second, topologically identical, scalar field and a corresponding colormap are supplied through the optional input ports, then the arrows can be colored by the second scalar field. The first (vector) field is sampled to produce the arrows and the second (scalar) field is sampled to produce the colors for the arrows. If a either the colormap or the optional scalar field are supplied, then the other must be supplied as well.

INPUTS

Volume Data (required; field 3D 3-vector float)

The input data must be a 3D field, representing a volume of points. The data value for each point must be a 3D vector of *float*s.

Sample Input (field irregular 3-space)

This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **hedgehog** will take these locations and display the data values associated with them. This input can be used instead of **hedgehog**'s **Sample** parameter.

Upstream Transform (optional, invisible, autoconnect)

When the **hedgehog** and **geometry viewer** modules coexist in a network, they communicate through a normally-invisible data port. "Hedgehog" shows up as an object in the Geometry Viewer. When you select the hedgehog object and move it, **geometry viewer** informs the **hedgehog** module what the sample's new location is, and the **hedgehog** module recalculates the location and data it is displaying accordingly. This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **hedgehog** module's sample of locations.

Scalar Field (optional)

This port works with the **Colormap** port to color the arrows by a second, scalar field. This field must be topologically identical to the required vector field (i.e. it must have the same dimensions, n-space, etc.). If this port is used, then a colormap must be supplied as well.

Colormap (optional)

If a scalar field is provided to color the arrows with, then a colormap must also be provided to act as a mapping from data space to color space. In order for this to happen, it is important that the range of the colormap be related to the range of the scalar data. This is most easily accomplished by using the **color range** module which adjusts the effective range of the colormap to the field.

PARAMETERS

Vector Scale

The lengths of the line segments output by this module are proportional to this value.

N segments

An integer value which determines the number of points sampled by the line, circle, plane, or space sampling probe. This controls the density of line segments output by **hedgehog**.

Method (radio buttons) Controls the way in which the field value is determined at each sample location. These options are ignored for **nodes**, which does not interpolate.

- If **point**, a nearest-neighbor algorithm is used. Each mesh vertex is assigned the value of the nearest point in the lattice.
- If **trilinear**, a trilinear interpolation is performed. The value at each vertex depends on the values at the eight lattice points that are the corners of the "enclosing cube". The trilinear interpolation method is more accurate but takes longer to compute, particularly at higher resolutions.

Sample (radio buttons) Specifies the type of sample taken from the vector field: **point**, **line**, **circle**, **plane**, **space**, or **nodes**. The default is **point**.

nodes produces a vector at each node rather than **N Segment**s along a sampling space. When it is selected **N Segment**, **Show Bounds**, and **Sampling Style** are ignored. **nodes** can be faster than the other techniques. However, it can create so many vector arrows that the resulting figure is unintelligible and slow to render. It is recommended that you use the **downsize** module before **hedgehog** if you select **nodes**.

arrow heads

Arrows are typically produced with arrow heads so that you can distinguish the source and direction of the vectors. This can be disabled with the **arrow heads** toggle. When on (the default mode), this option causes arrow heads to be generated. When off, no arrow heads are generated.

Show Bounds

A bounding hull for the sample points is typically produced so that you can easily see the extent of the sample positions. This can be disabled with the **Show Bounds** toggle. When on (the default mode), this option causes the bounding hull to be generated as a wireframe geometry. When off, no hull is generated.

OUTPUTS

Hedgehog (geometry)

The output geometry is a collection of line segments that represent the 3D-vector values at the sample locations. The line segments have arrows at their ends, indicating the direction of the vectors.

EXAMPLE 1

The following network visualizes the vector output of the **compute gradient** module as a **hedgehog**.

```
              READ FIELD
                  |
                  |------------------|
                  |                  |
           COMPUTE GRADIENT    VOLUME BOUNDS
                  |                  |
             HEDGEHOG                |
                  |   |--------------|
                  |   |
             GEOMETRY VIEWER
```

EXAMPLE 2

The following network visualizes the output of a PLOT3D data set coloring the hedgehogs with one of the scalar fields:

```
GENERATE COLORMAP                 READ PLOT3D
        |                |-------------|
        |         EXTRACT SCALAR       |
        |_____    ________|            |
             |   |        |        EXTRACT VECTOR
          COLOR RANGE     |            |
               |_______   |   _________|
                      |   |  |         |
                   HEDGEHOG    VOLUME BOUNDS
                         |______________|
                                |
                                |
                         GEOMETRY VIEWER
```

RELATED MODULES

Data input:

read volume, volume manager

Gradient computation:

compute gradient

Vector operations:

vector curl, vector div, vector grad, vector mag, vector norm

Additional geometries:

volume bounds, isosurface

Geometric rendering:

geometry viewer

Sample Input:

samplers

SEE ALSO

The example script HEDGEHOG demonstrates the **hedgehog** module.

NAME

histogram stretch – balance the histogram of a data set

SUMMARY

Name	histogram stretch				
Availability	Imaging, Volume, FiniteDiff module libraries				
Type	filter				
Inputs	field *any-dimension* scalar byte *any-coordinates*				
Outputs	field of same type as input				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	histr_min	int	0	0	255
	histr_max	int	255	0	255

DESCRIPTION

histogram stretch is an image/volume processing module that balances the "histogram" of a data set between specified values. This operation combines histogram balancing (also called "histogram normalization" or "histogram equalization") and contrast stretching.

Finding the *histogram* of an image (or volume) consists of tallying the number of pixels (voxels) of each value into "bins". Byte data typically generates 256 bins (1 bin for each possible data value).

The *histogram equalization* process consists of trying to establish the same number of pixels (voxels) per bin by translating the pixel (voxel) values, using a well-chosen lookup table. This has the effect of creating an even distribution of values throughout the data set. It typically used to enhance low-contrast images (volumes) or images in which the data is "bunched up" at one end of the spectrum.

Equalization is applied only to values within the range specified by the parameters **histr_min** and **histr_max**. Data outside this range is not included in the histogram generation, and is eliminated.

INPUTS

Data Field (required; field *any-dimension* scalar byte *any-coordinates*)

The input data may be an AVS field of any dimensionality, each of whose values is a scalar *byte*.

PARAMETERS

histr_min Specifies the bottom of the range of input values that will be histogrammed, then transformed.

histr_max Specifies the top of the range of input values that will be histogrammed, then transformed.

OUTPUTS

Data Field The output field has the same form as the input field.

Appropriate new **min_val** and **max_val** values are written to the output field.

LIMITATIONS

This module works for *byte* fields only. (For other data types, there is no general way to determine the "right" number of bins to generate.) To apply this module to non-*byte* data, use the **field_to_byte** module to pre-process the data.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
field to byte

Modules that could be used in place of **histogram stretch**:

contrast
ip contrast
ip linremap

Modules that can process **histogram stretch** output:

field to integer
field to float
field to double
any other filter module

SEE ALSO

The example script HISTOGRAM STRETCH demonstrates the **histogram stretch** module.

NAME

image compare – display two images together

SUMMARY

Name	image compare				
Availability	Imaging module library				
Type	filter				
Inputs	field 2D uniform 4-vector byte *(image)* field 2D uniform 4-vector byte *(image)*				
Outputs	field 2D uniform 4-vector byte *(image)*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Select	choice	vert_slice		
	Switch	toggle	off		
	valuator	float	0.5	0.0	1.0

DESCRIPTION

The **image compare** module lets you visually compare two images by displaying portions of those images together in one rectangular area in eight different ways—e.g. as two vertical slices, as two horizontal slices, in a checker pattern, etc. The main intent is to let you see "before" and "after" versions of the same image. One image is designated the "primary image," the other the "secondary image". You can flip back and forth between the dominant and secondary image using the **switch** parameter. In most cases, the **valuator** parameter controls the ratio of image 1 to image 2 appearing in the rectangle.

Both input images must have the same dimensions.

INPUTS

Image (required; field 2D uniform 4-vector byte)
One of the two images to compare.

Image (required; field 2D uniform 4-vector byte)
The other of the two images to compare.

PARAMETERS

selection Sets the way the two images are displayed together in the same rectangle.

vert_slice
vertical bands of the two images are displayed side by side.

horiz_slice
horizontal bands of the two images are displayed, one above the other.

diag_slice
slices from the upper left corner diagonally from one image to the next.

solid
disables the **valuator** dial described below.This lets you flicker between the images using the **switch** toggle described below.

circle
transforms the **valuator** dial to control the radius of a circle centered at the center of the image.

checker
creates a checkerboard pattern between the two images. The smaller the value showing on **valuator**, the more checks in the checkerboard.

venetian
creates alternating horizontal bands of image 1 and image 2.

random
randomly dithers between one image and the other based on the probability assigned by the **valuator** dial.

valuator The **valuator** dial controls the proportion of the rectangle viewing space that each image occupies. Allowable values are from 0.0 to 1.0, with the default 0.5 meaning "show half of one image and half of the other". As you move the dial, one or the other of the images gets more rectangle space.

switch A toggle switch that exchanges the proportions of the screen given to image 1 and image 2.

OUTPUTS

Image (field 2D uniform 4-vector byte)
The output image is the patchwork combination of image 1 and image 2, with the same dimensions.

EXAMPLE 1

The following network compares an image with a contrasted version of itself:

```
          READ IMAGE
    __________|
    |         |
CONTRAST      |
    |_____    |
         |    |
     IMAGE COMPARE
           |
     DISPLAY IMAGE
```

EXAMPLE 2

The following network compares two images and displays the result through the **image viewer**. The images must be the same size.

```
READ IMAGE            READ IMAGE
    |_________     _________|
             |    |
         IMAGE COMPARE
               |
         DISPLAY IMAGE
```

RELATED MODULES

Modules that could provide the **image compare** inputs:

read image
Any module that produces an image as output

Modules that can process **image compare** output:

image viewer

display image
Any module that takes image input

See also **field math**, **constant blend**, ip compare

SEE ALSO

The example scripts IMAGE COMPARE, and IMAGE II demonstrate the **image compare** module.

NAME

image manager – share images among subnetworks

SUMMARY

Name image manager

Unsupported this module is in the unsupported library

Type data

Inputs none

Outputs field 2D 4-vector byte (image)

Parameters

Name	*Type*	*Choices*
IMAGMGR Select	choice	Select, Replace
Image Manager	browser	
Image Choices	choice	

DESCRIPTION

The **image manager** module reads an image file from disk and outputs the image as a "field 2D 4-vector byte". It works like the **read image** module, except that it has both a cacheing mechanism and a way of sharing data among **image manager** modules in separate subnetworks.

See the **read image** manual page for a description of the image format.

PARAMETERS

IMAGMGR Select

A choice that determines how newly-read images will be placed to the list of currently active images:

- If **Select** is chosen, a new image is added to the end of the list.
- If **Replace** is chosen, a new image replaces the currently selected member on this list.

In either case, the change is reflected in *all* the *image manager* modules in all active subnetworks.

Image Manager

A file browser that allows you to select an image file to read.

Image Choices

A set of choices, listing each of the currently active images.

OUTPUTS

Data Field (field 2D 4-vector byte)

The output is an AVS *image*.

EXAMPLE

The following subnetworks might be used to display two images:

```
IMAGE MANAGER          IMAGE MANAGER
      |                      |
      |                      |
      |                      |
DISPLAY IMAGE          DISPLAY IMAGE
```

In this case, both **image manager** managers would contain "select/replace" choice buttons, a file browser, and an area below the browser:

```
+-------------------------+    +-------------------------+
| Active Images           |    | Active Images           |
+-------------------------+    +-------------------------+
| (no images)             |    | (no images)             |
+-------------------------+    +-------------------------+
```

Once a file (e.g. *heart_slice_22*) is selected with the browser in the **image manager** on the left, these buttons would look like this:

```
+-------------------------+    +-------------------------+
| * heart_slice_22        |    |   heart_slice_22        |
+-------------------------+    +-------------------------+
```

If a different file (e.g. *heart_slice_10*) is chosen from the browser in the **image manager** on the right, the buttons would look like this:

```
+-------------------------+    +-------------------------+
| * heart_slice_22        |    |   heart_slice_22        |
|   heart_slice_10        |    | * heart_slice_10        |
+-------------------------+    +-------------------------+
```

By selecting the same active image, you can have both networks display the same image:

```
+-------------------------+    +-------------------------+
| * heart_slice_22        |    | * heart_slice_22        |
|   heart_slice_10        |    |   heart_slice_10        |
+-------------------------+    +-------------------------+
```

Now, if you want to replace this image with a new one, click on the **Replace** buttons above the browser, then select a new file (e.g. *kidney_slice_04*) in just one of the **image manager** browsers. The result is that all **image manager** modules with the old image (*heart_slice_22*) selected as their active image will be automatically updated with the new image (*kidney_slice_04*):

```
+-------------------------+    +-------------------------+
| * kidney_slice_04       |    | * kidney_slice_04       |
|   heart_slice_10        |    |   heart_slice_10        |
+-------------------------+    +-------------------------+
```

RELATED MODULES

Same as for **read image**.

LIMITATIONS

The cached images are not freed until all *image manager* modules are destroyed. This is not the case with **read image** — the old data is freed whenever a new file is read.

image measure

NAME

image measure – measure distance between two image pixels

SYNOPSIS

Name	image measure
Availability	Imaging module library
Type	mapper
Inputs	field 2D uniform [byte | short | float] *n-vector* image viewer id structure (*invisible, autoconnect*) mouse info structure (*invisible, autoconnect*)
Outputs	image draw structure
Parameters	*Name* *Type* Measurements string block set pick mode oneshot

DESCRIPTION

image measure measures the distance between two pixels of an image. The result is reported in pixels.

If the field containing the image has extents information in its coordinate data area that is different from its dimensions (for example, a 512 x 512 image whose coordinate "points" area states that the data is positioned in space from -1000 to 3000 in X and Y) then **image measure** reports both the pixel space and world space measurements.

Performing a measurement involves an interaction between **image measure** and the **image viewer** module. **image measure**'s **image draw structure** output must be connected to the **image viewer** module's leftmost **image draw structure** input. See the "Example" below.

You specify the two pixels to measure interactively in the **image viewer** window as follows:

1. The **image measure** module must have control of the left mouse button in the Image Viewer window. When **image measure** is first connected and data first passes through it, it should have control of the left mouse button.
2. Press and hold down the left mouse button to select the starting pixel.
3. Move the cursor over the image. As you move the cursor, a line follows it anchored at the starting pixel. The distance from the starting pixel is continuously reported in the **Measurements** text widget on **image measure**'s module control panel.
4. To finish the measurement, release the left mouse button. The measurement line disappears. There is now no starting pixel defined.

If there are multiple images in the Image Viewer window, and / or multiple sketching modules, then some other module or the Image Viewer itself may have control of the left mouse button. To get control back to **image measure**:

1. Make the image the current image (use shift-left mouse button or left mouse button).
2. Press **set pick mode** on **image measure**'s control panel.

   This tells the Image Viewer that the left mouse button will be taking image measurements, not picking a current image.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)

The input is a 2D uniform field of type byte, short, or float. It can be any vector length.

Note: Though **image measure** accepts *n-vector* and data type byte, short, or float, the input to **image viewer** can only be byte, 1-vector or 4-vector.

image viewer id structure (required; invisible, autoconnect)

This input port is invisible by default. It connects automatically to the **image viewer** module's **image viewer id structure** output. The two modules communicate the **image viewer** module's scene id on this connection. Normally, you can ignore its existance.

mouse info structure (required; invisible, autoconnect)

This input port is invisible by default. It connects automatically to the **image viewer** module's **mouse info structure** output. The two modules communicate image name, mouse pointer location and button up / down information on this connection. Normally, you can ignore its existance.

PARAMETERS

Measurement

This is a string block. It appears as a text widget on **image measure**'s module control panel. It continuously reports the distance, in pixels, from the starting pixel to the cursor position. When the left mouse button is released, it continues to report the distance of the last cursor position.

set pick mode

A oneshot that sets the **image viewer**'s upstream mouse picking focus to this module. Use it to regain control of the mouse whenever the left mouse button doesn't seem to be working to measure points.

OUTPUTS

image draw structure (required)

The left output port contains the **image draw structure** that connects to the **image viewer** module's leftmost input port. It is required.

EXAMPLE

This example shows a simple network to measure pixel distances. The invisible upstream connections coming from **image viewer** to **image measure** are not shown.

```
                  READ IMAGE
                           |
                           |
          |---------------|
          |                |
     IMAGE MEASURE         |
            |-----|        |
                  |        |
              IMAGE VIEWER
```

RELATED MODULES

image viewer
image probe
sketch roi

SEE ALSO

The example script Imaging / IMAGE MEASURE demonstrates this module.

The upstream feedback mechanism that makes **image measure** work is described in the *AVS 5 Update* document.

NAME

image probe – report data values at selected pixel location

SYNOPSIS

Name	image probe
Availability	Imaging module library
Type	mapper
Inputs	field 2D uniform [byte | short | float] *n-vector* image viewer id structure (*invisible, autoconnect*) mouse info structure (*invisible, autoconnect*)
Outputs	image draw structure
Parameters	*Name* *Type* Values string block set pick mode oneshot

DESCRIPTION

image probe reports the data values present at a pixel location selected in the **image viewer** module's window.

If the field containing the image has extents information in its coordinate data area that is different from its dimensions (for example, a 512 x 512 image whose coordinate "points" area states that the data is positioned in space from -1000 to 3000 in X and Y) then **image probe** reports both the pixel space and world space measurements.

Selecting a pixel involves an interaction between **image probe** and the **image viewer** module. **image probe**'s **image draw structure** output must be connected to the **image viewer** module's leftmost **image draw structure** input. See the "Example" below.

You select a pixel in the **image viewer** window as follows:

1. The **image probe** module must have control of the left mouse button in the Image Viewer window. When **image probe** is first connected and data first passes through it, it should have control of the left mouse button.
2. Press and hold down the left mouse button to select the starting pixel.
3. Move the cursor over the image. As you move the cursor, the data values present at that location are continuously reported in the **Values** text widget on **image probe**'s module control panel.
4. To finish the reporting, release the left mouse button.

If there are multiple images in the Image Viewer window, and / or multiple sketching modules, then some other module or the Image Viewer itself may have control of the left mouse button. To get control back to **image probe**:

1. Make the image the current image (use shift-left mouse button or left mouse button).
2. Press **set pick mode** on **image probe**'s control panel.

   This tells the Image Viewer that the left mouse button will be probing pixels, not picking a current image.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)
The input is a 2D uniform field of type byte, short, or float. It can be any vector length.

Note: Though **image probe** accepts *n-vector* and data type byte, short, or float, the input to **image viewer** can only be byte, 1-vector or 4-vector.

image viewer id structure (required; invisible, autoconnect)
This input port is invisible by default. It connects automatically to the **image viewer** module's **image viewer id structure** output. The two modules communicate the **image viewer** module's scene id on this connection. Normally, you can ignore its existance.

mouse info structure (required; invisible, autoconnect)
This input port is invisible by default. It connects automatically to the **image viewer** module's **mouse info structure** output. The two modules communicate image name, mouse pointer location and button up / down information on this connection. Normally, you can ignore its existance.

PARAMETERS

Values This is a string block. It appears as a text widget on **image probe**'s module control panel. It continuously reports the data values present at the cursor location as it moves over the image. When the left mouse button is released, it continues to report the data values at the last cursor position. All vector elements are reported.

set pick mode
A oneshot that sets the **image viewer**'s upstream mouse picking focus to this module. Use it to regain control of the mouse whenever the left mouse button doesn't seem to be working to probe points.

OUTPUTS

image draw structure (required)
The left output port contains the **image draw structure** that connects to the **image viewer** module's leftmost input port. It is required.

EXAMPLE

This example shows a simple network to report pixel data values. The invisible upstream connections coming from **image viewer** to **image probe** are not shown.

```
                    READ IMAGE
                             |
                             |
           |----------------|
           |                |
    IMAGE PROBE             |
             |-----|        |
                   |        |
              IMAGE VIEWER
```

RELATED MODULES

image viewer
image measure
sketch roi

SEE ALSO

The example script Imaging / IMAGE PROBE demonstrates this module.

The upstream feedback mechanism that makes **image probe** work is described in the *AVS 5 Update* document.

NAME

image to cgm – convert image to CGM and store in file

SUMMARY

Name	image to cgm
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field 2D 4-vector byte (*image*,required)
Outputs	*none*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
CGM File Name	browser			
Encoding	choice	Binary		
Landscape	toggle	off	off	on
Page Width	float typein	8.50	1.00	25.00
Page Height	float typein	11.00	1.00	25.00
Image Width	float dial	7.00	0.00	25.00
Image Height	float dial	9.00	0.00	25.00
Preserve Aspect Ratio	toggle	on		

DESCRIPTION

The **image to cgm** module converts its input image to the Computer Graphics Metafile (CGM) format and stores it in a file. The **geometry viewer** module's right-most output port outputs an image, thus any scene in a **geometry viewer** window can be saved to a CGM file.

After the file is written, the filename is reset to prevent subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

All three types of CGM output are supported:

- **Binary** which is the most compact.
- **Character** which contains only printable characters.
- **Clear Text** which is human readable.

All files are formatted as left-to-right, top-to-bottom scan lines.

By default, the image is centered on the page so that the vertical axis of the image corresponds to the vertical axis of the page. If the **Landscape** option is specified, the vertical axis of the image corresponds to the horizontal axis of the page.

The **Page Width** and **Page Height** parameters control the destination page size of the image. This size is measured in inches.

Because the **image to cgm** module accepts only image data as an input, it cannot draw primitives such as lines, text, polygons and spheres at the resolution of the printer. There is a way to get around this problem: you can increase the resolution of the input image. Using a combination of the **geometry viewer** module with the **Software Renderer** option, you can generate images that are larger than the resolution of the screen.

To avoid problems with color approximation and obscured windows that occur with some devices, it is best to use the **Software Renderer** option when using the **image to cgm** module with the **geometry viewer** module.

image to cgm

INPUTS

Image (field 2D 4-vector byte)
Any AVS image.

PARAMETERS

CGM File Name
A file browser that allows you to specify the name of the CGM file to be created.

Encoding Selects the type of CGM output: **Binary**, **Character**, or **Clear Text**.

Landscape Toggle to rotate image 90 degrees on paper.

Page Width
The horizontal size of the output page in inches.

Page Height
The vertical size of the output page in inches.

Image Width
Width of the printed image in inches.

Image Height
Height of the printed image in inches.

Preserve Aspect Ratio
When selected, the **Image Width** and **Height** are coupled to preserve the aspect ratio of the input image. When not selected, they can be adjusted independently to stretch the image.

EXAMPLE 1

This example converts an image to a CGM file:

```
  READ IMAGE
      |
 IMAGE TO CGM
```

EXAMPLE 2

This example converts the scene in the **geometry viewer** module into a color CGM file, by taking the image from the **geometry viewer** module's rightmost output port.

```
                       READ UCD
                          |
                          |
GENERATE COLORMAP    UCD CELL TO NODE
          |               |
          |     |---------|
         UCD CONTOUR      |
                |         |
                |-----|   |
                      |   |
                     UCD TO GEOM
                          |
                          |
                   GEOMETRY VIEWER
                             |
                          IMAGE TO CGM
```

RELATED MODULES

geometry viewer
image to postscript

SEE ALSO

The example script "Convert AVS image to CGM file for printing" demonstrates this module.

image to pixmap

NAME

image to pixmap – convert image to pixmap

SUMMARY

Name	image to pixmap
Availability	this module is in the unsupported library
Type	mapper
Inputs	field 2D 4-vector byte uniform
Outputs	pixmap

Parameters

Name	*Type*	*Default*	*Choices*
Approximation Technique (Pseudo-color systems only)	choice	none	none, dithering random, monochrome

DESCRIPTION

Note: with AVS 4, the basic internal representation of screen images shifted from a pixmap to an AVS image. For example, the **geometry viewer** module outputs an image, which can be converted to a postscript file with **image to postscript**. There is thus little need for this module. It is retained in the unsupported library for backward compatibility only.

The **image to pixmap** module takes as input an *image* ("field 2D 4-vector byte") and outputs the same image as a *pixmap*. It is useful for converting the output of modules that produce images into modules that require pixmaps.

The *image* and *pixmap* data types differ in these major ways:

- Images are allow for efficient direct manipulation by a module, whereas pixmaps allow for efficient manipulation by the display device.
- Pixmaps are directly usable by a display device (under control of the X server). In X terminology, pixmaps contain "pixel values", images contain "colors". This difference is important only for pseudo-color systems, in which pixmap values are interpreted as indices into the system's color lookup table. An image contains 24-bit color values, which cannot be used on such systems, which have only 12 color planes.
- A pixmap is represented by an X Window System *resource id* (an integer). This means that transferring a pixmap from one module to another is more efficient than transferring all the data that defines an image.

See the **read image** manual page for a description of the AVS image format.

INPUTS

Data Field (required; field 2D 4-vector byte uniform)
The input field must be an AVS *image*.

PARAMETERS

This module has the following parameter only when running on a pseudo-color system.

approximation technique (Pseudo-color systems only)
Controls the conversion of color values to pixel values. There are four approximation techniques:

- **dithering**: uses a dither matrix to approximate each color

- **floyd steinberg**: uses an error diffusion dithering technique
- **random**: uses a random number dither to approximate each color
- **monochrome**: uses the luminance of the color as an index into a greyscale ramp
- **none**: takes the closest approximation for each color

OUTPUTS

pixmap The output is an AVS *pixmap*.

EXAMPLE

This network allows an image to be displayed in an arbitrary-sized window:

```
  READ IMAGE
       |
IMAGE TO PIXMAP
       |
DISPLAY PIXMAP
```

RELATED MODULES

pixmap to image, display pixmap

image to postscript

NAME

image to postscript – convert image to PostScript™ and store in file

SUMMARY

Name	image to postscript
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field 2D 4-vector byte (*image*,required)
Outputs	*none*

Parameters

Name	*Type*	*Default*	*Choices*
filename	typein		
mode	choice	greyscale	greyscale, color
encapsulate	boolean	off	
landscape	boolean	off	
page size x	real	7.5	
page size y	real	10.5	

DESCRIPTION

The **image to postscript** module converts its input image to the PostScript™ page description language and stores it in a file. The **geometry viewer** module's rightmost output port outputs an image, thus any scene in a **geometry viewer** window can be saved to a PostScript file.

After the file is written, the filename is reset to prevent subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

Two types of PostScript output are supported:

- An 8-bit gray scale image suitable for sending to a gray-scale PostScript-compatible laser printer such as a **laserwriter**.
- A 24-bit true color RGB **color** image suitable for sending to a PostScript-compatible laser printer that supports the Level 1 PostScript **colorimage** operator color extensions, or any PostScript Level 2 color printer. The actual format is 3-component (RGB) with 8 bits per component, in *multi* format, with a line of red values, then green values, then blue values for each scan line.

All files are formatted as left-to-right, top-to-bottom scan lines.

If the **encapsulate** boolean is chosen, the PostScript file will be written in **EPSF** (Encapsulated Postscript). Encapsulated PostScript files are designed to be imported by other PostScript processing packages. If you have such a program, you can usually scale, position and combine the image with text or other annotation. Note that some printers do not properly print Encapsulated PostScript files. In this case, deselect **encapsulate**.

By default, the image is scaled, translated, and centered on the page so that the vertical axis of the image corresponds to the vertical axis of the page. If the **landscape** option is specified, the vertical axis of the image corresponds to the horizontal axis of the page. The largest scale of the image that will fit within the page is chosen. The aspect ratio of the image is not altered.

The **page size x** and **page size y** parameters control the destination page size of the image. This size is measured in inches. The default size: 7.5x10.5 allows for a 0.5 inch border surrounding the image. Adjust these parameters to scale the image.

image to postscript's input is an AVS image. The similar **output postscript** module's input is a pixmap. The **output postscript** module does not provide some of the flexibility of the **image to postscript** module.

Because the **image to postscript** module accepts only image data as an input, it cannot draw primitives such as lines, text, polygons and spheres at the resolution of the printer. There are two ways to get around this problem. Firstly, you can increase the resolution of the input image. Using a combination of the **geometry viewer** module with the **Software Renderer** option, you can generate images that are larger than the resolution of the screen.

These images can take a signficant time (and memory) to both generate and print. Another alternative is to use a PostScript output capbility supported by the **geometry viewer** CLI that allows direct postscript output of both text and lines. PostScript does not support primitives that map very well onto shaded surfaces. Images are still the best way to display these on a PostScript device.

To avoid problems with color approximation and obscurred windows that occur with some devices, it is best to use the **Software Renderer** option when using the **image to postscript** module with the **geometry viewer** module.

INPUTS

Image (field 2D 4-vector byte)
Any AVS image.

PARAMETERS

filename A typein that allows you to specify the name of the PostScript file to be created.

Mode Selects the type of PostScript output: **greyscale** or **color**.

encapsulate
Output encapsulated PostScript.

landscape Output image in landscape mode (rotate 90 degrees).

page size x The horizontal size of the output page in inches.

page size y The vertical size of the output page in inches.

EXAMPLE 1

This example converts an image to a PostScript file:

```
   READ IMAGE
       |
IMAGE TO POSTSCRIPT
```

EXAMPLE 2

This example converts the scene in the **geometry viewer** module into a color PostScript file, by taking the image from the **geometry viewer** module's rightmost output port.

image to postscript

```
                         READ UCD
                            |
                            |
GENERATE COLORMAP    UCD CELL TO NODE
          |                 |
          |     |---------|
         UCD CONTOUR        |
                |           |
                |-----|     |
                      |     |
                    UCD TO GEOM
                            |
                            |
                  GEOMETRY VIEWER
                               |
                          IMAGE TO POSTSCRIPT
```

RELATED MODULES

geometry viewer
tracer
output postscript
image to cgm

NAME

image viewer – display and manipulate collections of images (Image Viewer)

SUMMARY

Name	image viewer
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field 2D uniform *any-data*, 1-vector or 4-vector (*image, optional, multiple*) colormap (*optional*) image draw structure (*optional*)
Outputs	field 2D 4-vector byte (*image*) image picking structure (*invisible*) image viewer id structure (*optional, invisible, autoconnect*) mouse info structure (*optional, invisible, autoconnect*)
Parameters	*Name* / *Type* / *Default* Update Always / boolean / on Update Image / oneshot

DESCRIPTION

The **image viewer** module provides access within an AVS network to the complete Image Viewer subsystem. Many different modules can supply the input images. That is, many *image*-format outputs can be connected to the **image viewer**'s image input port. All the images will be combined into a single current scene.

image viewer accepts two kinds of images: 4-vector *any-data* true color images, and 1-vector (scalar) *any-data* images. Non-byte data is converted to byte and normalized to the 0-255 range before display. The scalar byte images will be displayed as grayscale, or can be colorized using the byte values as an index into the optional input colormap.

You can also invoke **image viewer** with no inputs, so that the "scene" is initially empty. Images can be added to a scene either by upstream modules or by the **Read Image** selection on the **image viewer** control panel. Images sent by upstream modules can be saved to files using the **Write Image** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Image**.

The Image Viewer's Action submenu can create simple "flip book" animations. You can send a series of images from upstream modules into the **image viewer** and have it turn them into a simple animation.

Note that the **image viewer** window can be reparented to page and stack widgets using the AVS Layout Editor.

INPUTS

Image (optional, multiple; field 2D uniform *any-data*, 1-vector or 4-vector)

The input data is a 2D uniform field. It can be *any-data*. Non-byte data is converted to byte and normalized to the 0-255 range before it is displayed. The input data can be a 4-vector true color image, or a scalar "image". Scalar images are displayed in grayscale unless the optional **Colormap** input is used. More than one image can be input to this port. All the images will be combined into the same "scene".

Colormap (optional; colormap)

This optional colormap will be used to colorize scalar byte images. (All non-byte fields are converted to byte before display.) The field's byte values are used as an index into the 256-element colormap. Colormaps are generally supplied by the **generate colormap** module.

image draw structure

User-interaction modules (**sketch roi**, **image measure**, **image probe**, etc.) connect to the **image viewer** through this leftmost input port.

The **image draw structure** is described in the "Image Viewer" section of the *AVS 5 Update* document. This port actually exists solely to cause the AVS flow executive to fire the **image viewer** module when the upstream module needs input.

PARAMETERS

Update Always

This switch can be used to slightly improve performance. It is only effective when a module is connected to the **image viewer**'s image output port. It is invisible by default.

When this switch is on, every time the scene changes the **image viewer** module translates the contents of the output buffer into an AVS image and sends it to the image output port. If this switch is off, the **image viewer** will only translate the output buffer when the **Update Image** oneshot is pressed. The default is on.

To use this parameter, first use the Module Editor's (middle or right mouse button on the module dimple) Parameter Editor to make the **Port Visible**. Then, you can either connect the **boolean** module to the new parameter port, or you can create a module control panel for the **image viewer** with an **Update Always** button on it by setting **toggle** on the Parameter Editor.

Update Image

A oneshot switch that causes the **image viewer** to translate the contents of the output buffer into an AVS image and send it to the image output port. **Update Image** works no matter how **Update Always** is set.

This parameter is invisible by default. To use it, make it visible in the same way as described for **Update Always**. Then, either connect the **oneshot** module to the parameter port, or set **oneshot** with the Parameter Editor to create a module control panel with an **Update Image** button on it.

OUTPUTS

Image This rightmost output is an image containing a *view* that includes all the images. Note that it is not necessary to connect anything to this port for normal operations. This port gives other modules access to the image output by the renderer. One use of this port would be to produce a printable PostScript file with the **image to postscript** module. Another use of this port would be to produce a composite image with the **write image** module.

image picking structure (invisible)

The **image viewer** outputs an optional image picking structure. It is contained on the next-to-rightmost output port. If the user clicks on a position in an image in a scene window with the left mouse button, the image picking structure will report, among other data, the X, Y coordinates of the selected location in the image. Downstream modules can use this information, for example, to retrieve the original data present at that location in the field before it was translated into an alpha, red, green, blue true color image. The image picking structure is described in the "Advanced Topics" chapter of the *AVS Developer's Guide*. This output port is invisible by default.

image viewer id structure (optional, invisible, autoconnect)

This second-from-left output port is involved in the upstream data passing that allows user-interaction modules such as **sketch roi**, **image probe**, and **image measure** to function.

This structure tells the upstream module the scene id of this particular instance of the **image viewer** module. This port is invisible by default. It will autoconnect to the **image viewer id structure** input port of the module connected to the **image draw structure** port.

The structure is described in the "Image Viewer" section of the *AVS 5 Update* document.

mouse info structure (invisible)

This leftmost output port is involved in the upstream data passing that allows user-interaction modules such as **sketch roi**, **image probe**, and **image measure** to function.

This structure passes mouse location and button state information upstream. It is invisible by default. It will autoconnect to the **mouse info structure** input port of the module connected to the **image draw structure** input port.

The structure is described in the "Image Viewer" section of the *AVS 5 Update* document.

RESIZING

The **image viewer**'s pulldown menu, which is accessed by clicking on the "dimple" in the upper lefthand corner of the display window, provides several ways to resize the window to certain fixed sizes:

- **Zoom Full Screen.** Resizes the window to fill the square working area of the screen (approximately 1024 x 1024), and magnifies the image to fit. If the window is embedded in a page or stack (see *Layout Editor* in the Network Editor chapter), it becomes a top-level window that can be freely resized and moved using the X window manager.
- **Unzoom.** Resizes and moves the window to return to its location before a **Zoom Full Screen**. If the window originally was embedded in a page or stack, it will be re-embedded there.

SPECIAL CONSIDERATIONS

This module is special: instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level menu of the Image Viewer subsystem. Thus, when you add the **image viewer** icon to a network, no page is added to the Network Control Panel.

There are two ways to access the Image Viewer menu:

- Click the small square in **image viewer** icon with the left mouse button.
- With the cursor positioned over the **Data Viewers** button located at the top of the Network Control Panel, press and *hold down* any mouse button. When the "AVS Data Viewers" pop-up menu appears, roll the mouse down to "Image Viewer" and release the mouse button. This **Data Viewers** button is always visible, even when there is no active network.

In some circumstances, it is useful to be able to access both the Image Viewer control panel and the Network Control Panel simultaneously. They both occupy the same screen position, along the left edge of the screen. In these cases, use the X Window System window manager to move the one of these menu windows out of the way.

The **image viewer**'s control panel also differs from that of other modules in these ways:

- The Network Editor's **Layout Editor** cannot be used to rearrange Image Viewer controls.
- If a network includes more than one instance of **image viewer**, AVS does *not* create a separate control panel for each instance. Each **image viewer** sends its output to a different window, but the same Image Viewer menu controls all the windows. The module whose output window is currently highlighted in red is the one being controlled. To switch the *focus* to another **image viewer** output window, just click in it with any mouse button.

EXAMPLE 1

This network receives a series of images of what were originally AVS geometry objects, composites them over a background image, and creates a simple animation as the user manipulates the geometry object:

```
                          READ GEOM
                              |
                   GEOMETRY VIEWER
                              |-----------------------|
    __________________________|                DISPLAY IMAGE
    |              |          |
BACKGROUND    LUMINANCE       |
    |              |          |
    |           CONTRAST      |
    |              |------|   |
    |                   REPLACE ALPHA
    |                         |
    |----------------------|  |
                          COMPOSITE
                              |
                     IMAGE VIEWER
```

EXAMPLE 2

The following network reads in an image and then sends it to the **image viewer** module. This lets you apply all of the imaging techniques of the **image viewer** to the image.

```
     READ IMAGE
         |
         |
IMAGE VIEWER
```

EXAMPLE 3

The following network shows the **sketch roi** module connected to the **image viewer**. **sketch roi** is producing a region of interest to the **ip edge** module. The user is drawing the region of interest in the **image viewer** window. Notice how **ip edge**'s output is connected to both **image viewer** and **sketch roi**.

```
                READ IMAGE
                      |
                      |
       |-------|      |
       |    |--|---|  |
SKETCH ROI  |  |  IP EDGE
  |    |    |  |------|
  |    |----|         |
  |-----------|       |
              |       |
             IMAGE VIEWER
```

EXAMPLE 4

The following network shows the **image viewer** displaying a scalar byte image. The "image" started life as a 3D uniform byte, 3-vector field that is reduced to 2D with **orthogonal slicer**, and to scalar with **extract scalar**. Without **generate colormap**, the image would be displayed in grayscale.

```
                      READ FIELD
                           |
                           |
                  ORTHOGONAL SLICER
                           |
GENERATE COLORMAP          |
             |      EXTRACT SCALAR
             |             |
             |----------|  |
                        |  |
                  IMAGE VIEWER
```

RELATED MODULES

display image
read image
image to postscript

SEE ALSO

The "Image Viewer Subsystem" chapter in the *AVS User's Guide*, and the "Image Viewer" section of the *AVS 5 Update* document.

integer

NAME

integer – send a user-entered integer to one or more module(s) integer parameter port

SUMMARY

Name	integer				
Availability	Imaging, UCD, Volume, FiniteDiff module libraries				
Type	data				
Inputs	none				
Outputs	integer				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Integer Value	dial	0	*unbounded*	*unbounded*

DESCRIPTION

The **integer** module sends a single user-specified integer value to one or more integer-type parameter ports on one or more receiving modules. Its purpose is to make it possible for you to simultaneously control integer parameter input to more than one module using only a single input widget (whether the default dial, or a typein).

Before you can connect **integer** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter Editor window appears, click any mouse button on its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **integer** module icon in the usual way.

PARAMETERS

Integer Value (integer)

The single user-supplied integer value to be sent to the module(s) integer parameter port(s). The default value is 0. There is no minimum or maximum restriction on the value. You should be aware of the range of numbers that it is reasonable to send to the receiving modules. The default widget type is a dial.

OUTPUTS

Integer (integer)

The integer value is sent to all modules with integer-type parameter ports connected to the **integer** module

EXAMPLE 1

The following network reads a field, then creates two orthogonal slices through the field in different planes (one in I and one in J) using the **integer** module to specify the same offset slice plane to both slicers. The resulting planes are converted to meshes and composited together in the geometry viewer window.

```
                        INTEGER         READ FIELD
                          |                |
                          |     |----------|-----|
                          |     |                |
                          |-----|----------|     |
                          |     |          |     |
                         ORTHO SLICE      ORTHO SLICE
                                |                |
GENERATE COLORMAP               |                |
        |-----------------|-----|-----------|    |
                          |     |           |    |
                          FIELD TO MESH    FIELD TO MESH
                                |                |
                                |                |
                                ------------------
                                        |
                                        |
                                 GEOMETRY VIEWER
```

EXAMPLE 2

This example reads two different fields, uses the **integer** module to specify the same slice plane in both to the **orthogonal slicer** modules, then uses **field math** to produce a new field that is the difference between them.

```
           READ FIELD                      READ FIELD
                |                                |
INTEGER         |                                |
  |             |                                |
  |------|------|------------------------|       |
         |      |                        |       |
        ORTHOGONAL SLICE                ORTHOGONAL SLICE
                |                                |
                |                                |
                |------------|     |------------ |
                             |     |
                           FIELD MATH
                                |
                          FIELD TO MESH
                                |
                          GEOMETRY VIEWER
```

RELATED MODULES

Modules that can process **integer** output:

all modules with integer-type parameter ports

SEE ALSO

The example scripts INTEGER, FIELD TO BYTE, as well as others demonstrate the **integer** module.

interpolate

NAME

interpolate – compute intermediate values to change the size of a field

SUMMARY

Name	interpolate				
Availability	Imaging, Volume, FiniteDiff module libraries				
Type	filter				
Inputs	field 2D / 3D scalar *any-data any-coordinates*				
Outputs	field *same-dimension* scalar byte *same-coordinates*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	interp_sx	float	1.0	0.0	4.0
	interp_sy	float	1.0	0.0	4.0
	interp_sz	float	1.0	0.0	4.0
	sampling	choice	Point		

DESCRIPTION

The **interpolate** module arbitrarily changes the size of its input data, either by subsampling or interpolating it. This module is useful for smoothly scaling the data arbitrarily up and down.

The interpolation algorithm first selects, for each output point, its real (floating-point) position in the input data set:

```
New X = Old X * interp_sx
New Y = Old Y * interp_sy
New Z = Old Z * interp_sz
```

With the **Point** sampling method, it then selects the closest pixel (voxel) to the computed one. With **bilinear** (in 2D) or **trilinear** (in 3D) sampling, it finds the four pixels (2D) or eight voxels (3D) around the computed point and does a linear sampling for in-between pixels.

The point sampling mode is much quicker than the linear sampling and should be used when interactivity is more important than image quality.

INPUTS

Data Field (field 2D / 3D scalar *any-data any-coordinates*)
The input field may be 2D or 3D, but must be scalar. The data for each element can be of any type. The field can be uniform, rectilinear, or irregular.

PARAMETERS

interp_sx
interp_sy
interp_sz (does not appear for 2D input fields)
The interpolation factors for the coordinate dimensions.

sampling This choice determines the sampling method, **Point** or **Bi/Trilinear**, as described above.

OUTPUTS

Data Field The output field has the same form as the input field. Note that the extent is unmodified; this module changes the resolution of the data within the physcial space delimited by the extents. It does not alter the physical extents of the data.

RELATED MODULES

This module is similar to **downsize** (which does uniform, stride-based point sampling), **average down** (which averages data in specified chunks sizes, independently in the X, Y, and Z dimensions, and **crop** (which selects a subset of the data but doesn't change the resolution). Some advantages to using this module are: it can scale non-uniformly in each dimension; it can do high-quality linear sampling; and it can scale data up instead of only down.

LIMITATIONS

This module does the wrong thing when down-sampling (going from a large image to a small one) in the Bi/Trilinear mode. What it should do is "average" appropriately chosen regions down to each pixel. What is does is to choose the four pixels around the center of that region and interpolate between them. This is not a huge error, but it is not strictly correct.

SEE ALSO

The example script INTERPOLATE demonstrates the **interpolate** module.

ip absolute

NAME

ip absolute – absolute value of a field

SUMMARY

Name	ip absolute
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D uniform scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-type same-vector*
Parameters	none

DESCRIPTION

ip absolute calculates the absolute value of all the data elements in the input field, placing the result in the output field.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The input is a 2D or 3D uniform field of type byte, short, or float. The field can be any vector length. If the field is 3D, the absolute value operation is applied to Z successive XY slices.

Data Field (optional; field 2D uniform scalar byte)

This field is an optional region of interest. If connected, only the pixels designated by the ROI are affected in each XY slice. The region of interest must have the same XY dimensions as the input field.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field of the same dimensions, data type, and vector length as the input field. Its header min/max data value has been marked invalid.

EXAMPLE

```
  READ IMAGE
       |
       |
IP ARITHMETIC
       |
       |
IP ABSOLUTE
       |
       |
IMAGE VIEWER
```

RELATED MODULES

ip arithmetic
ip float math
ip logical
field math

SEE ALSO

The example script Imaging / IP ABSOLUTE demonstrates this module.

NAME

ip arithmetic – arithmetic operations on fields

SUMMARY

Name	ip arithmetic
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field uniform *same-dims same-data same-vector (optional)* field 2D scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Operation	choice	add constant		
constant	float dial	0	*unbounded*	*unbounded*

DESCRIPTION

ip arithmetic performs arithmetic between two uniform fields, or between one uniform field and a **constant** value.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)
This rightmost input port data field must be present. If it is the only field present, operations are performed with a constant value.

Data Field (optional; field uniform *same-dims same-data same-vector*)
If this second, optional input field is present, then operations can be performed between the two input fields. This field will be ignored if one of the constant operations is selected. This field must have the same dimensions, extents, data type, and vector length as the first input field. One field can be connected to both input ports.

Data Field (optional; field 2D scalar byte)
This field is an optional region of interest. If connected, only the pixels designated by the ROI are affected on each XY slice. The ROI must have the same extents as the input field(s).

PARAMETERS

Operation A series of radio buttons to select the operation.

These functions work with two input fields:

add
subtract
multiply
divide
min
max

These functions work with the rightmost input field and the **constant**:

add constant
mul constant
min constant
max constant
shift (only valid with byte or short input)

The functions are performed in the data type of the input fields.

- In the case of arithmetic overflow, the result's high order bits are clipped.
- If the **divide** function detects divide-by-zero, it sets the destination value to the maximum value for that data type. (Floats are set to a constant HUGE_VAL, which is defined on each platform as the largest value a float can hold.)
- When adding constants to byte and short input data, the fractional portion of the **constant** value is clipped.

constant A floating point dial that specifies the constant value to use against the rightmost input field. The default is 0; the range is unbounded.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)
The output field has the same dimensions, data type, and vector length as the input field(s). Its field header min / max data values are marked invalid.

EXAMPLE

```
                        READ VOLUME
                              |
                              |------------|
                              |            |
                              | IP ARITHMETIC
                              |            |
                              |     |------|
                        IP ARITHMETIC
                                    |
GENERATE COLORMAP    ORTHOGONAL SLICER
          |                         |
          |---------------------|   |
                            COLORIZER
                                    |
                         IMAGE VIEWER
```

RELATED MODULES

ip absolute
ip float math
ip logical
field math

SEE ALSO

The Imaging / IP ARITHMETIC examples script demonstrates this module.

NAME

ip blend – alpha or compositing blend of two fields

SYNOPSIS

Name	ip blend
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field uniform *same-dims same-data same-vector* field 2D [byte | short | float] scalar (*alpha mask*)
Outputs	field uniform *same-dims same-data same-vector*
Parameters	none

DESCRIPTION

ip blend performs a pixel-by-pixel composition of two fields, using an alpha mask field for the blending.

The equation used to composite the fields is:

```
output pixel =   (alpha for this pixel) * (field1 pixel)
               + (1.0 - (alpha for this pixel)) * (field2 pixel);
```

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)
Data Field (required; field uniform *same-dims same-data same-vector*)

The two fields to be blended. The fields must match in dimensions, extents, data type, and vector length. If the fields are 3D, the blending operation is applied to Z successive XY slices.

Data Field (required; field 2D [byte | short | float] scalar)

A 2D field used as the alpha mask. Its extents must match those of the input fields.

Byte, short, or float fields can be used as the alpha mask blending function. Byte or short fields will be scaled to vary from 0 to 1; float fields will be assumed to be in the range 0.0 to 1.0. If they are not, a warning is printed.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output field has the same dimensions, data type, and vector length as the input field. Its header min / max data values are set to invalid.

EXAMPLE

ip blend

```
                              READ IMAGE
                                  |
                      |-----------|
                      |           |---|
          EXTRACT SCALAR          |   |
                   |              |   |
       IP ARITHMETIC  IP ARITHMETIC   |
                   |              |   |
                   |------------| |   |
                                | |   |
                               IP BLEND
                                   |
                         IMAGE VIEWER
```

RELATED MODULES

alpha blend

SEE ALSO

The example script Imaging / IP BLEND demonstrates this module.

NAME

ip compare – compares two fields

SUMMARY

Name	ip compare	
Availability	Imaging module library	
Type	data output	
Inputs	field 2D uniform [byte | short | float] *n-vector* field 2D uniform [byte | short | float] *n-vector*	
Outputs	none	
Parameters	*Name*	*Type*
	Differences	string block

DESCRIPTION

ip compare compares two fields (usually images). It compares the sizes (extents) of the fields, their vector lengths, and data type, in that order. If the fields are otherwise identical, they are then compared on a pixel-by-pixel basis.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)
Data Field (required; field 2D uniform [byte | short | float] *n-vector*)

These are 2D fields of type byte, short, or float. Generally, these are images.

PARAMETERS

Differences (string block)

The output is reported in a string block parameter on the module's control panel. The comparison occurs in the order listed. **ip compare** stops reporting after the first difference is found.

The output reports:

Sizes differ	field extents are different
Number of channels differ	fields have different vector lengths
Data types differ	field data types are different
n pixels differ	fields are identical, but pixel values differ
No differences	fields are identical

EXAMPLE 1

```
IP READ VFF     IP READ VFF
        |              |
        |------|   |---|
               |   |
           IP COMPARE
```

EXAMPLE 2

The following network could be used to count the number of pixels whose value is greater than 127, where **ip logical** is set up to **AND** the constant 128:

ip compare

```
                    READ IMAGE
                          |
              |-------------|
              |             |
        IP LOGICAL          |
              |             |
              |----------|  |
                         |  |
                   IP COMPARE
```

RELATED MODULES

ip extrema
ip register
ip statistics
print field
compare field

SEE ALSO

The example script Imaging/IP COMPARE demonstrates this module.

NAME

ip contour – draw iso-level contours

SUMMARY

Name	ip contour
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector*
Outputs	field uniform *same-dims* byte *same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	selection	none | scalar		
level type	choice	N equal levels		
N Level Steps	int dial	3	1	*unbounded*
Level Value	float dial	0.0	*unbounded*	*unbounded*

DESCRIPTION

ip contour derives iso-level contours from the source field and draws the contours into the destination field.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The input field is uniform, 2D or 3D, of data type byte, short, or float. It can have any number of vector components. If the field is 3D, the contouring will be performed on Z successive XY slices, not on the field as a 3D whole (i.e., the input is treated as a series of 2D XY slices).

PARAMETERS

Channel A set of buttons that select which vector elements to contour. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be contoured in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

level type A pair of radio buttons that chooses how to make the contours:

1 level

If **1 level** is selected, then a single contour is produced for **Level Value**.

N equal levels

If **N equal levels** is selected, then **N Level Steps** contours are produced. The contour interval is: (*max* - *min*) / (**N Level Steps** + 1). **ip contour** uses whatever min / max values are contained in the input field's header without validation. If none are present, it calculates them. The contouring always starts from 0. The interval is calculated as a float.

If the source field has neighboring pixels that cross one of the given levels, the output pixel is set to the value of MAXBYTE. Otherwise, the output pixel is not affected.

N Level Steps

An integer dial that specifies the number of iso levels. This is used only when **N equal levels** is selected. The minimum is 0, the maximum is unbounded; and the default is 3.

Level Value

A float dial that specifies a single contour level to map. This is used only when **1 level** is selected. The default is 0.0; the range is unbounded.

OUTPUTS

Data Field (field uniform *same-dims* byte *same-vector*)

The output is a field with the same dimensions and vector length as the input field. It is always data type byte. Those vector elements not selected by the **Channel** choice are 0.

The value inserted into the output field to mark the contour levels is 255. This happens to produce red, green, and blue contour lines for ARGB input images.

EXAMPLE

```
     READ IMAGE
          |
          |
     IP CONTOUR
          |
          |
IMAGE VIEWER
```

SEE ALSO

The example script Imaging / IP CONTOUR demonstrates this module.

NAME

ip convolve – convolve with image float kernel

SUMMARY

Name	ip convolve
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D scalar float (*kernel*) field 2D uniform scalar byte (*optional, region of interest*)
Outputs	field *same-dims same-data same-vector*
Parameters	*Name* / *Type* / *Default* Channel / selection / none | scalar clear output / boolean / off

DESCRIPTION

ip convolve convolves a field with the specified kernel.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)
The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the convolution is performed on Z successive XY slices.

Data Field (required; field 2D scalar float)
This center port is the convolution kernel. The kernel is usually supplied either from a file via **ip read kernel**, or generated interactively with a module such as **generate filters**.

Be aware that larger convolution kernels can require geometrically longer processing times.

Data Field (optional; field 2D uniform scalar byte)
This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to convolve. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be convolved in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

clear output
A boolean switch. If on, the output field has the new data created by **ip convolve**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field 2D uniform *same-vector same-data*)
The output field has the same dimensions, vector, and data type as the input field. Its edge pixels are set to 0. The header min / max values are

set to invalid.

EXAMPLE 1

```
IP READ KERNEL  READ IMAGE
            |       |
            |----|  |
                 |  |
           IP CONVOLVE
                    |
                    |
         IMAGE VIEWER
```

EXAMPLE 2

```
IP READ KERNEL  READ VOLUME
            |       |
            |----|  |
                 |  |
           IP CONVOLVE
                    |
     ORTHOGONAL SLICER
                    |
         IMAGE VIEWER
```

RELATED MODULES

convolve
generate filters
ip read kernel

SEE ALSO

The example script Imaging / IP CONVOLVE demonstrates this module.

NAME

ip dilate – dilate a field

SUMMARY

Name	ip dilate
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D uniform scalar integer (*structuring element*) field 2D uniform scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-type same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	selection	none | scalar		
iterations	int dial	1	1	*unbounded*
clear output	boolean	on		

DESCRIPTION

ip dilate performs dilation or "region growing" morphological operations on fields based on an arbitrary structuring element.

The input field can be considered to be of two types:

logical

This is a field whose vector elements are bytes, each of which contains only one of two values: 0 or 255. Such logical or "binary" fields are produced by **ip threshold** and **ip morph**.

In the case of a logical field, the output of **ip dilate** is the logical "or" of all the neighborhood pixels selected by the structuring element.

grayscale

Any other input field is said to be a "grayscale", meaning only that each vector element ("band") contains data of any type that can be interpreted as a set of grayscale values. For a grayscale field, the output is the maximum of all the neighborhood pixels selected by the structuring element.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the dilation is performed on Z successive XY slices.

Data Field (required; field 2D uniform scalar integer)

The center input is for the 2D structuring element. This is usually obtained from a file via the **ip read sel** module. See that man page for a detailed description of its format.

The logical structuring element describes the neighborhood that will be used to determine which neighborhood pixels are used as input elements into the operation.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

ip dilate

PARAMETERS

Channel A set of buttons that select which vector elements to dilate. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be dilated in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

iterations An integer dial that specifies how many times the structuring element should be applied to the input. Allows for iterative morphological operations. The minimum is 1, the maximum is unbounded, and the default is 1. The Status bar reports the progress of the iterations.

clear output

A boolean switch. If on, the output field has the new data created by **ip dilate**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. Edge pixels in the destination field are set to 0. The header's min / max data values are set to invalid.

EXAMPLE 1

```
IP READ SEL      READ IMAGE
        |             |
        |-----------| |
                    | |
             IP DILATE
                      |
          IMAGE VIEWER
```

EXAMPLE 2

```
IP READ SEL      READ VOLUME
        |             |
        |-----------| |
                    | |
             IP DILATE
                      |
     ORTHOGONAL SLICER
                      |
          IMAGE VIEWER
```

RELATED MODULES

ip erode
ip median
ip morph
ip read sel

SEE ALSO

The example script Imaging / IP DILATE demonstrates this module.

NAME

ip edge – enhance edges in a field

SUMMARY

Name	ip edge
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D uniform scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	selection	none | scalar		
Method	choice	Prewitt		
hwidth	float dial	3.0	0.0	*unbounded*
vwidth	float dial	3.0	0.0	*unbounded*
clear output	boolean	on		

DESCRIPTION

ip edge performs an edge enhancement operation, using the specified algorithm.

The algorithms use convolution kernels to sharpen a field in the horizontal direction and then in the vertical direction. The algorithms then perform a quadratic add on the resulting images. All convolutions for the multiple kernels are performed in a single pass.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the edge enhancement is performed on Z successive XY slices. It is not performed on a 3D volume as a "whole," i.e., no edges are enhanced in a ZY plane, etc.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to edge enhance. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be edge enhanced in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

Method A series of radio buttons to select the edge-detection algorithm. The default is **Prewitt**. The choices are:

Prewitt
Roberts
Compass
Sobel
Frei Chen

Marr Hildreth
Nevatia Babu
Robinson 3
Robinson 5
Macleod
Argyle
Kirsh
Boxcar
Deriv(ative) of Gaussian
Weighted Line
Unweighted Line

hwidth
vwidth

hwidth and **vwidth** are floating point dial parameters to the algorithms that use variable width kernels: **Argyle**, **Macleod**, **Marr Hildreth** (just hwidth), **Boxcar**, and the **Deriv of Gaussian**. The variables specify the functional size of the kernel, not the actual size of a kernel.

A particular algorithm generates the actual kernel size from these values. A variable width kernel is useful because you can make the width smaller to detect smaller detail; or larger to ignore noisy edges in an image.

Be aware that you can supply widths that will produce large kernels, which will require large amounts of processing time. In these cases, you may find that you can perform an edge enhancement operation faster if you first perform a Fourier transform on the image.

clear output

A boolean switch. If on, the output field has the new data created by **ip edge**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. Edge pixels in the destination field are set to 0. The header's min / max data values are set to invalid.

EXAMPLE

```
          READ IMAGE
               |
               |
          IP EDGE
               |
               |
      IMAGE VIEWER
```

RELATED MODULES

ip convolve
ip kernel
sobel

SEE ALSO

The example script Imaging / IP EDGE demonstrates this module.

NAME

ip erode – erode a field

SUMMARY

Name	ip erode				
Availability	Imaging module library				
Type	filter				
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D uniform scalar integer (*structuring element*) field 2D uniform scalar byte (*optional, region of interest*)				
Outputs	field uniform *same-dims same-type same-vector*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Channel	selection	none | scalar		
	iterations	int dial	1	1	*unbounded*
	clear output	boolean	on		

DESCRIPTION

ip erode performs erosion or "region shrinking" morphological operations on fields based on an arbitrary structuring element.

The input field can be considered to be of two types:

logical

This is a field whose vector elements are bytes, each of which contains only one of two values: 0 or 255. Such logical or "binary" fields are produced by **ip threshold** and **ip morph**.

In the case of a logical field, the output of **ip erode** is the logical "and" of all the neighborhood pixels selected by the structuring element.

grayscale

Any other input field is said to be a "grayscale", meaning only that each vector element ("band") contains data of any type that can be interpreted as a set of grayscale values. For a grayscale field, the output is the minimum of all the neighborhood pixels selected by the structuring element.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the erosion is performed on Z successive XY slices.

Data Field (required; field 2D uniform scalar integer)

The center input is for the 2D structuring element. This is usually obtained from a file via the **ip read sel** module. See that man page for a detailed description of its format.

The logical structuring element describes the neighborhood that will be used to determine which neighborhood pixels are used as input elements into the operation.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

ip erode

PARAMETERS

Channel A set of buttons that select which vector elements to erode. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be eroded in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

iterations An integer dial that specifies how many times the structuring element should be applied to the input. Allows for iterative morphological operations. The minimum is 1, the maximum is unbounded, and the default is 1. The Status bar reports the progress of the iterations.

clear output

A boolean switch. If on, the output field has the new data created by **ip erode**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. Edge pixels in the destination field are set to 0. The header's min / max data values are set to invalid.

EXAMPLE

```
IP READ SEL    READ IMAGE
        |           |
        |---------| |
                  | |
             IP ERODE
                    |
                    |
         IMAGE VIEWER
```

RELATED MODULES

ip dilate
ip median
ip morph
ip read sel

SEE ALSO

The example script Imaging / IP ERODE demonstrates this module.

NAME

ip extrema – find data value extrema

SUMMARY

Name	ip extrema
Availability	Imaging module library
Type	data output
Inputs	field 2D uniform [byte | short | float] *n-vector* field 2D scalar byte (*optional, region of interest*)
Outputs	none
Parameters	*Name* / *Type* / *Default* Channel / choice / <channel 0> Extrema / string block

DESCRIPTION

ip extrema finds the minimum and maximum data values in one channel of a field.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)
The right input is a 2D uniform field of type byte, short, or float. It can be any vector length.

Data Field (optional; field 2D scalar byte)
This left input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of radio buttons that choose which vector element to calculate the extrema for. There are as many buttons as vector elements. One vector element can be selected at one time.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "channel 0", "channel 1," etc. The first selection is the default.

Extrema A string block widget that reports the data value extrema. It appears on the **ip extrema** module's control panel. Two floating point values, Minimum and Maximum, are reported.

EXAMPLE

```
          READ IMAGE
               |
               |
      IP THRESHOLD
               |
               |---------------|
               |               |
    EXTRACT SCALAR             |
               |----------|    |
               |          |    |
      IMAGE VIEWER      IP EXTREMA
```

ip extrema

RELATED MODULES

ip compare
ip register
ip statistics
print field
statistics

SEE ALSO

The example script Imaging/IP EXTREMA demonstrates this module.

NAME

ip fft – Fourier transform a field

SUMMARY

Name	ip fft		
Availability	Imaging module library		
Type	filter		
Inputs	field [1D | 2D | 3D] uniform [byte | short | float] *n-vector*		
Outputs	field uniform float *same-dims same-vector* (packed complex)		
Parameters	*Name*	*Type*	*Default*
	Channel	selection	none | scalar

DESCRIPTION

ip fft Fourier transforms a uniform field (not complex) and places the packed complex result in a uniform float output field. Generally, these fields are images.

The data will have the following format, typical of FFT algorithms, in the output field:

Re[0][0] Re[0][N / 2]	Re[1][0] Im[1][0]	...	Re[M / 2-1][0] Im[M / 2-1][0]
Re[0][1] Im[0][1]	Re[1][1] Im[1][1]	...	Re[M / 2-1][1] Im[M / 2-1][1]
.			
.			
Re[0][N / 2-1] Im[0][N / 2-1]	Re[1][N / 2-1] Im[1][N / 2-1]	...	Re[M / 2-1][N / 2-1]Im[M / 2-1][N / 2-1]
Re[M / 2][0] Re[M / 2][N / 2]	Re[1][N / 2] Im[1][N / 2]	...	Re[M / 2-1][N / 2]Im[M / 2][N / 2]
Re[M / 2][1] Im[M / 2][1]	Re[1][N / 2+1] Im[1][N / 2+1]	...	Re[M / 2-1][N / 2 + 1]Im[M / 2-1][N / 2+1]
.			
.			
Re[M / 2][N / 2-1] Im[M / 2][N / 2-1]	Re[1][N-1] Im[1][N-1]	...	Re[M / 2-1][N-1] Im[M / 2-1][N-1]

The complete MxN transform may be deduced from the fact that for float fields, the forward 2D FFT produces a field with conjugate symmetry, such that:

Re[M-i][N-j] = Re[i][j] and Im[M-i][N-j] = –Im[i][j]

These packed complex output fields are un-packed for further processing with **ip fft unpack**. They can also be viewed as magnitude / phase images by processing with **ip fft display**, or turned back into the original image with **ip ifft**.

INPUTS

Data Field (required; field [1D | 2D | 3D] uniform [byte | short | float] *n-vector*)

The input is a 1D, 2D, or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the input is not already in the proper format to perform an FFT, the module converts the data to float, forces its XY extents to be a power of 2, and centers the original field in this new area before calling the FFT function. 1D input can be generated by the **ip read line** module that interactively extracts a 1D subset from an image using a sampling line. If the field is 3D, then the FFT is performed on Z successive XY slices.

PARAMETERS

Channel A set of buttons that select which vector elements to FFT. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be FFT'd in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

OUTPUTS

Data Field (field uniform float *same-dims same-vector*)

The output field contains the packed complex representation of the Fourier transform result, stored as a float. Its dimensions, extents, and vector length equal those of the input field. The "excess" created if the input's XY extents were forced to be a power of 2 is clipped in this output field. Vector elements that were not selected by **Channel** are set to 0. The header's min / max data values are set to invalid.

This output should be processed with **ip fft unpack** for viewing. 1D output can be sent to the **graph viewer**.

EXAMPLE 1

This example displays an FFT in the Image Viewer along with the original image:

```
      READ IMAGE
           |----------|
           |          |
       IP FFT         |
           |          |
           |          |
  IP FFT DISPLAY      |
           |          |
           |----------|
  IMAGE VIEWER
```

RELATED MODULES

ip fft display
ip fft multiply
ip fft pack
ip fft unpack
ip ifft
ip read line

SEE ALSO

The example scripts Imaging / 1D FFT, Imaging / IP FFT, Imaging / filtering data with FFTs, and Imaging / doing convolutions with FFTs demonstrate this module.

ip fft display

NAME

ip fft display – calculate magnitude and phase of packed fft field

SUMMARY

Name	ip fft display
Availability	Imaging module library
Type	filter
Inputs	field [1D | 2D | 3D] uniform float (packed complex) *n-vector*
Outputs	field [1D | 2D | 3D] uniform float *same-vector* (magnitude) field [1D | 2D | 3D] uniform float *same-vector* (phase)

Parameters

Name	*Type*	*Default*
Channel	selection	none | scalar
calc magnitude	boolean	on
log magnitude	boolean	off
calc phase	boolean	off
normalize phase	boolean	on

DESCRIPTION

ip fft display converts the packed conjugate symmetric FFT representation written by **ip fft** to a displayable form by calculating the magnitude and/or phase of the packed input field.

INPUTS

Data Field (required; field [1D | 2D | 3D] uniform float (packed complex) *n-vector*)

The input field must be the packed conjugate symmetric array of the type produced by **ip fft**. (See that module's man page.) It can be 1D, 2D, or 3D, of any vector length. Generally, this is an image. If the field is 3D, then the unpacking is performed and on Z successive XY slices.

PARAMETERS

Channel A set of buttons that select which vector elements to unpack. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be unpacked in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

calc magnitude
calc phase Two boolean switches. If **calc magnitude** is on, then the magnitude of the input will be calculated and sent to the rightmost output field. If **calc phase** is on, then the phase of the input will be calculated and sent to the left output field. **calc magnitude** is on by default; **calc phase** is off.

log magnitude

A boolean switch that, if on, causes the module to compute the log (base 10) of the magnitude rather than just the magnitude. It is off by default.

normalize phase

A boolean switch that, if on, normalizes the phase to the range 0.0-255.0. This switch is on by default. If off, the phase may have a data range that makes it display as black in the **image viewer** window.

OUTPUTS

Data Field (field uniform float *same-dims same-vector*)

The right output port is a field containing the magnitude of the input field. It contains data only if **calc magnitude** is on. If **log magnitude** was selected, it contains the log (base 10) of the magnitude of the input field. It has the same dimensions, extents, data type, and vector length as the input field. Those vector elements not selected by **Channel** are set to 0. 1D output can be sent to the **graph viewer** for viewing.

The header's min / max data values are set to invalid.

Data Field (field uniform float *same-dims same-vector*)

The left output port is a field containing the phase of the input field. It contains data only if **calc phase** is on. It has the same dimensions, extents, data type, and vector length as the input field. Those vector elements not selected by **Channel** are set to 0. 1D output can be sent to the **graph viewer** for viewing.

The header's min / max data values are set to invalid.

EXAMPLE

This example displays an FFT in the Image Viewer along with the original image:

```
     READ IMAGE
           |----------|
           |          |
       IP FFT         |
           |          |
           |          |
  IP FFT DISPLAY      |
           |          |
           |----------|
  IMAGE VIEWER
```

RELATED MODULES

ip fft
ip fft multiply
ip fft pack
ip fft unpack
ip ifft
ip read line

SEE ALSO

The example scripts Imaging / 1D FFT, Imaging / IP FFT, and Imaging / doing convolutions with FFTs demonstrate this module.

NAME

ip fft multiply – multiply two packed complex fields

SUMMARY

Name	ip fft multiply
Availability	Imaging module library
Type	filter
Inputs	field [1D | 2D | 3D] uniform float *n-vector* (packed complex) field uniform float *same-dims same-vector* (packed complex)
Outputs	field uniform float *same-dims same-vector* (packed complex)
Parameters	*Name* / *Type* / *Default* Channel / selection / none | scalar

DESCRIPTION

ip fft multiply multiplies two packed complex fields. Multiplying in the frequency domain is the same as convolving in the spatial domain, but faster.

INPUTS

Data Field (required; field [1D | 2D | 3D] uniform float *n-vector*)
Data Field (required; field uniform float *same-dims same-vector*)

The input fields are floats, but in packed complex form as produced by **ip fft**. They must have the same dimensions, extents, and vector length. Generally, these are images. If the fields are 3D, then the multiplication is performed on Z corresponding, successive XY slices.

PARAMETERS

Channel A set of buttons that select which vector elements to multiply. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be multiplied in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

OUTPUTS

Data Field (field uniform float *same-dims same-vector*)

The output is a float field of the same dimensions, extents, and vector length as the input fields. It is in packed complex form. Those vector elements not selected by **Channel** are set to zero. The header's min / max data values are set to invalid.

EXAMPLE

```
GENERATE FILTERS         READ IMAGE
          |                   |
       IP FFT              IP FFT
          |                   |
          |---------|    |-----|
                    |    |
              IP FFT MULTIPLY
                       |
                       |
                    IP IFFT
                       |
                       |
              IMAGE VIEWER
```

ip fft multiply

RELATED MODULES

ip fft
ip fft display
ip fft pack
ip fft unpack
ip ifft
ip read line

SEE ALSO

The example script Imaging/doing convolutions with FFTs demonstrates this module.

NAME

ip fft pack – fold conjugate symmetric FFT representation

SUMMARY

Name	ip fft pack
Availability	Imaging module library
Type	filter
Inputs	field [1D | 2D | 3D] uniform float *n-vector* (real) field [1D | 2D | 3D] uniform float *n-vector* float (imaginary)
Outputs	field uniform float *same-dims same-vector*
Parameters	*Name* / *Type* / *Default* Channel / selection / none | scalar center DC / boolean / on

DESCRIPTION

ip fft pack folds real and imaginary input fields into a single output field appropriate for inverse transform using **ip ifft**.

INPUTS

Data Field (required, field [1D | 2D | 3D] uniform float *n-vector*)
Data Field (required, field [1D | 2D | 3D] uniform float *n-vector*)

The right input port supplies the real portion of a field. The left input port supplies the imaginary portion of a field. Both are generally images. The inputs are 1D, 2D, or 3D uniform float fields. They can be any vector length. The two fields must have the same dimensions, extents, and vector length. If the fields are 3D, the packing is performed on Z successive XY slices.

ip fft pack assumes the input fields exhibit conjugate symmetry. This means s[i,j] = s[N-i,M-j] for the real field and s[i,j] = - s[N-i,M-j] for the imaginary field, where N and M are the width and height of the source fields.

PARAMETERS

Channel A set of buttons that select which vector elements to pack. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be packed in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

center DC A boolean switch that specifies where the DC component of the source field should be taken from. If **center DC** is on, the DC value will be taken from the center of the source fields; if off, the DC value will be taken from the [0,0] pixel of the source fields. The default is on.

OUTPUTS

Data Field (field uniform float *same-dims same-vector*)

The output is a field with the same dimensions, extents, data type, and vector length as the input field. Those vector elements not selected by **Channel** are set to 0. The header's min / max data values are set to invalid.

EXAMPLE

ip fft pack

```
   READ IMAGE
         |
     IP FFT
         |
IP FFT UNPACK
     |   |
     |   |---------|
     |             |
     |    IP ARITHMETIC
     |             |
     |--|  |-------|
        |  |
    IP FFT PAC
           |
           |
      IP IFFT
           |
           |
  IMAGE VIEWER
```

RELATED MODULES

ip fft unpack
ip ifft
ip fft multiply
ip fft
ip fft display
ip read line

SEE ALSO

The example scripts Imaging/IP FFT, and Imaging/filtering data with FFTs demonstrate this module.

NAME

ip fft unpack – unfold conjugate symmetric FFT representation

SUMMARY

Name	ip fft unpack
Availability	Imaging module library
Type	filter
Inputs	field [1D | 2D | 3D] uniform float *n-vector* (packed complex)
Outputs	field [1D | 2D | 3D] uniform float *same-vector* (real) field [1D | 2D | 3D] uniform float *same-vector* (imaginary)

Parameters

Name	*Type*	*Default*
Channel	selection	none | scalar
center DC	boolean	on

DESCRIPTION

ip fft unpack unfolds the conjugate symmetric FFT representation written by **ip fft** into two fields that represent the real and imaginary components. The destination fields will be conjugate symmetric.

INPUTS

Data Field (required; field [1D | 2D | 3D uniform float *n-vector*)

The input is a field in packed complex form produced by the **ip fft** module. It is a 1D, 2D, or 3D uniform field of type float. It can be any vector length. Generally, this is an image. If the field is 3D, then the unpacking is performed on Z successive XY slices.

PARAMETERS

Channel A set of buttons that select which vector elements to unpack. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be unpacked in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

center DC A switch that specifies where the DC component of the source image should be placed. If **center DC** is on, the DC value will be placed at the center of the destination field; if off, the DC value will be placed in the [0,0] pixel of the destination field. The default is on.

OUTPUTS

Data Field (field uniform float *same-dims n-vector*)
Data Field (field uniform float *same-dims n-vector*)

The float output fields have the same dimensions, extents, and vector length as the input field. The right output port is the real component. The left output port is the imaginary component. Those vector elements not selected by **Channel** are set to 0. The header's min / max data values are set to invalid.

EXAMPLE

ip fft unpack

```
          READ IMAGE
                |
            IP FFT
                |
       IP FFT UNPACK
            |   |
            |   |---------|
            |             |
            |    IP ARITHMETIC
            |             |
            |--|  |-------|
               |  |
          IP FFT PACK
                  |
                  |
              IP IFFT
                  |
                  |
           IMAGE VIEWER
```

RELATED MODULES

ip fft pack
ip fft
ip ifft
ip fft multiply
ip fft display
ip read line

SEE ALSO

The example scripts Imaging/IP FFT, and Imaging/filtering data with FFTs demonstrate this module.

NAME

ip float math – floating point operations on a field

SUMMARY

Name	ip float math
Availability	Imaging module library
Type	filters
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D scalar byte (*optional, region of interest*)
Outputs	field uniform float *same-dims same-vector*

Parameters

Name	*Type*	*Default*	*Choices*
Channel	selection	none | scalar	
op	choice	log	log, log10, sqrt, exp, recip cos, sin, atan
clear output	boolean	on	

DESCRIPTION

ip float math performs floating-point operations on the input field (generally an image), placing the result in the output field. Whatever the data type of the input field, it is converted to float for the calculations, and the output field is float.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the operations are performed on Z successive XY slices.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to perform operations on. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be calculated in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

Operation A series of radio buttons to select the operation.

log generates a field containing the natural logarithms of the source field's pixels.

log10 generates a field containing the common (base 10) logarithms of the source field's pixels.

sqrt generates a field containing the square roots of the field's pixels.

exp generates a field containing the exponential values (e^{pixel_value}) of the source field's pixels.

recip generates a field containing the reciprocals of the source field's pixels.

cos generates a field containing the cosines of the source field's pixels.

sin generates a field containing the sines of the source field's pixels.

atan generates a field containing the arctangent of the source field's pixels.

clear output

A boolean switch. If on, the output field has the new data created by **ip fmath**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform float *same-dims same-vector*)

The output is a field with the same dimensions and vector length as the input field, but of type float. The header's min / max data values are set to invalid.

EXAMPLE

```
        READ IMAGE
             |
             |
   IP FLOAT MATH
             |
             |
       FIELD TO BYTE
             |
             |
    IMAGE VIEWER
```

RELATED MODULES

ip arithmetic
ip absolute
ip logical
field math

SEE ALSO

The example script Imaging / IP FLOAT MATH demonstrates this module.

NAME

ip histogram – create a histogram

SUMMARY

Name	ip histogram
Availability	Imaging module library
Type	mapper
Inputs	field 2D uniform [byte | short | float] *n-vector* field 2D scalar byte (*optional; region of interest*)
Outputs	field 1D scalar integer

Parameters

(*Name*	*Type*	*Default*	*Min*	*Max*
Channel	selection	<channel 0>		
N Bins	int dial	256	1	*unbounded*
Lower Limit	float dial	0.0	0.0	*unbounded*
Upper Limit	float dial	255.0	0.0	*unbounded*

DESCRIPTION

ip histogram takes the unnormalized histogram of the source field.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)

The right input is a 2D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image.

Data Field (optional; field 2D scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to count for the histogram. There are as many buttons as vector elements. Only one vector element can be selected at one time.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "channel 0", "channel 1," etc. The default is the first channel.

N Bins An integer dial that specifies how many bins to group the count in the output histogram field. The range is 1 to unbounded. The default is 256.

Lower Limit
Upper Limit

Floating point dials that specify the lower limit and upper limit of the data range to be examined when the histogram is compiled. An optimized special case exists for finding the entire histogram of an byte input. This optimized case will be invoked for byte fields when **N Bins** = 256, **Lower Limit** = 0, and **Upper Limit** = 255.

OUTPUTS

Data Field (field 1D scalar integer)

The output is a 1D scalar integer field, **N Bins** long. Its extents are set to **Lower Limit** and **Upper Limit**. Each element contains an integer count of the number of data values that fell into that bin. Each bin in the histogram covers a data range of (**Upper Limit - Lower Limit**) / **N Bins** in the source field. This should be used as input to **graph viewer**'s rightmost input port.

The header's min / max data values are set to invalid.

EXAMPLE

```
                     READ IMAGE
                          |
        |-------------------|
        |                   |
  IMAGE VIEWER    IP HISTOGRAM
                          |
                          |
                  GRAPH VIEWER
```

RELATED MODULES

generate histogram

SEE ALSO

The example script Imaging / IP HISTOGRAM demonstrates this module.

NAME

ip ifft – inverse Fourier transform for conjugate data sets

SUMMARY

Name	ip ifft		
Availability	Imaging module library		
Type	filter		
Inputs	field [1D | 2D | 3D] uniform float *n-vector* (packed complex)		
Outputs	field uniform float *same-dims same-vector*		
Parameters	*Name*	*Type*	*Default*
	Channel	selection	none | scalar

DESCRIPTION

ip ifft performs an inverse Fourier transformation on a conjugate symmetric field to produce a real (not complex) field.

INPUTS

Data Field (required; field [1D | 2D | 3D] uniform float *n-vector*)

The input is a 1D, 2D, or 3D uniform float field of any vector length. Generally, this is an image. 1D input can be generated by the **ip read line** module that interactively extracts a 1D subset from an image using a sampling line. If the field is 3D, then the inverse FFT is performed on Z successive XY slices.

Each XY slice of the input data must have the following format, typical of FFT algorithms:

Re[0][0] Re[0][N/2]	Re[1][0] Im[1][0]	...	Re[M/2-1][0]Im[M/2-1][0]
Re[0][1] Im[0][1]	Re[1][1] Im[1][1]	...	Re[M/2-1][1]Im[M/2-1][1]
.			
.			
Re[0][N/2-1] Im[0][N/2-1]	Re[1][N/2-1] Im[1][N/2-1]	...	Re[M/2-1][N/2-1] Im[M/2-1][N/2-1]
Re[M/2][0] Re[M/2][N/2]	Re[1][N/2] Im[1][N/2]	...	Re[M/2-1][N/2]Im[M/2][N/2]
Re[M/2][1] Im[M/2][1]	Re[1][N/2+1] Im[1][N/2+1]	...	Re[M/2-1][N/2 + 1]Im[M/2-1][N/2+1]
.			
.			
Re[M/2][N/2-1] Im[M/2][N/2-1]	Re[1][N-1] Im[1][N-1]	...	Re[M/2-1][N-1] Im[M/2-1][N-1]

The complete MxN transform may be deduced from the fact that for real fields, the forward 2D FFT produces a field with conjugate symmetry, such that:

Re[M-i][N-j] = Re[i][j] and Im[M-i][N-j] = –Im[i][j]

Channel A set of buttons that select which vector elements to inverse FFT. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be inverse FFT'd in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

OUTPUTS

Data Field (field uniform float *same-dims same-vector*)

The output field contains the real representation of the original data. It has the same dimensions, extents, and vector length as the input field. Vector elements that were not selected by **Channel** are set to 0. The header's min / max data values are set to invalid. 1D output can be sent to the **graph viewer** for viewing.

EXAMPLE

```
GENERATE FILTERS         READ IMAGE
          |                   |
       IP FFT             IP FFT
          |                   |
          |---------|   |-----|
                    |   |
             IP FFT MULTIPLY
                       |
                       |
                   IP IFFT
                       |
                       |
             IMAGE VIEWER
```

RELATED MODULES

ip fft
ip fft multiply
ip fft display
ip fft pack
ip fft unpack
ip read line

SEE ALSO

The example script Imaging / doing convolutions with FFTs demonstrates this module.

NAME

ip lincomb – inter-band linear combination

SUMMARY

Name	ip lincomb
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D scalar float (*transformation matrix*) field 1D scalar float (*optional, constant matrix*) field 2D scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-data same-vector*
Parameters	none

DESCRIPTION

ip lincomb operates on the vector elements ("bands") of a source field, combining the vector elements of each input pixel to produce pixels in the output field. The field is usually an image.

Each pixel in the input image is treated as a vector whose components are the bands of that pixel. This vector is multiplied by the transformation matrix contained in the second input field to produce a new vector whose components represent the bands of the output pixel. Then, if a constant matrix is provided in the third field input, this new vector is added to the constant matrix to create the output pixel. Expressed in matrix notation, **o** = **T•i** ‘‘+ C , where **i** and **o** are the input and output vectors, and **T** and **C** are the **tmatrix** and **cmatrix**, respectively.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The right input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the linear combination is performed on Z successive XY slices.

Data Field (required; field 2D scalar float)

This is the transformation matrix. It is treated as an array of floating point numbers. The field's width (first dimension) must be equal to the number of vectors in the input image. It can have any height. See the example transformation matrices below.

This input could be generated by a user-written module, or input as a field using **read field** or ADIA.

Data Field (optional; field 1D scalar float)

This is the constant matrix. It is treated as a 1D array of floating point numbers. The field's length must equal the height of the transformation matrix. The constant matrix is optional—it is applied only if present. See the example constant matrix below.

This input could be generated by a user-written module, or input as a field using **read field** or ADIA.

Data Field (optional; field 2D uniform scalar byte)

This left input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the dimensions and data type as the input field. Its vector length equals the height of the transformation matrix. The header's min / max data values are set to invalid.

EXAMPLE

To exchange second and third vector elements (indexes 1 and 2) of a field, use this transformation matrix and do not apply a constant matrix. This effectively swaps the red and green channels on an ARGB image.

1.0	0.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	0.0	1.0

To produce an output image that represents YUV (biased by 0.1 in Y) from an input floating point image whose vectors are normalized ARGB (values between 0.0 and 1.0), the appropriate transformation matrix would be:

0.0	0.1140	0.5870	0.2990
0.0	0.0813	0.4185	0.4998
0.0	0.4997	0.3311	0.1686

and the constant matrix would be:

0.1
0.0
0.0

EXAMPLE

This network reads the transformation and constant matrices from user-supplied data using the AVS Data Interchange Application's (ADIA) **file descriptor** module.

```
                    READ IMAGE
                         |
           FILE DESCRIPTOR |
                    |    |
     FILE DESCRIPTOR  |    |
               |    |    |
               |-|  |  |-|
                 |  |  |
              IP LINCOMB
                   |
                   |
            IMAGE VIEWER
```

RELATED MODULES

field math
ip float math
ip arithmetic
ip linremap

SEE ALSO

The example script Imaging / IP LINCOMB demonstrates this module.

NAME

ip linremap – linearly remap a field

SUMMARY

Name	ip linremap
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	choice	none | scalar		
constant	float dial	0.0	*unbounded*	*unbounded*
multiplier	float dial	1.0	*unbounded*	*unbounded*
clear output	boolean	on		

DESCRIPTION

ip linremap linearly remaps a field (generally an image) by first adding **constant** to the input pixels, then multiplying by **multiplier.** Byte and short fields are then clamped. Float fields are not.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The right input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the remapping is performed on Z successive XY slices.

Data Field (optional; field 2D uniform scalar byte)

This left input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to remap. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be remapped in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

constant A floating point dial that specifies the constant to add to the pixels. The range is unbounded; the default is 0.

multiplier A floating point dial that specifies the multiplier for the field. The range is unbounded; the default is 1.0.

clear output

A boolean switch. If on, the output field has the new data created by **ip linremap**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. The header's min / max data values are set to invalid.

EXAMPLE

```
   IP READ VFF
         |
         |
 FIELD TO FLOAT
         |
         |
IP LINREMAP
         |
         |
 FIELD TO BYTE
         |
         |
IMAGE VIEWER
```

RELATED MODULES

ip lincomb
ip threshold

SEE ALSO

The example script Imaging / IP LINREMAP demonstrates this module.

NAME

ip logical – bitwise logical operations

SUMMARY

Name	ip logical
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short] *n-vector* field uniform *same-dims same-data same-vector* (*optional*)
Outputs	field uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
op	choice	or constant		
constant	int dial	0	0	*maxval*

DESCRIPTION

ip logical does bitwise logical operations on each pixel of the two input fields and places the result at the output field. The two input fields must have the same dimensions, extents, type, and number of vectors. If there is only one input field, the logical function is performed against itself or a constant.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte or short. (Float is not accepted.) It can be any vector length. Generally, this is an image. If the field is 3D, then the operations are performed on Z successive XY slices.

Data Field (optional; field uniform *same-dims same-data same-vector*)

If this center, optional input field is present, then operations can be performed between the two input fields. This field will be ignored if one of the constant operations is selected. This field must have the same dimensions, extents, data type, and vector length as the first input field. One field can be connected to both input ports.

Data Field (optional; field 2D scalar byte)

This field is an optional region of interest. If connected, only the pixels designated by the ROI are affected on each XY slice. The ROI must have the same extents as the input field(s).

PARAMETERS

op A series of radio buttons to select the logical operation.

and
nand
or
nor
xor only work with two inputs; with one input they are ignored.

not
and constant
or constant
xor constant
only work with the right input field and the **constant** dial value.

The default is **or constant**.

constant An integer dial to set the constant value. The default is 0. The minimum is 0. The maximum is 255 for byte input; 65535 for short input.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)
The output field has the same dimensions, data type, and vector length as the input field. Its field header min/max data values are marked invalid.

EXAMPLE

```
      READ IMAGE
           |
           |
   IP LOGICAL
           |
           |
IMAGE VIEWER
```

RELATED MODULES

ip arithmetic
ip float math
ip absolute
field math

SEE ALSO

The example script Imaging/IP LOGIC demonstrates this module.

NAME

ip lookup – pass field through lookup table

SUMMARY

Name	ip lookup				
Availability	Imaging module library				
Type	filter				
Inputs	field [2D | 3D] uniform [byte | short] *n-vector* field 1D scalar integer (*lookup table*) field 2D scalar byte (*optional, region of interest*)				
Outputs	field uniform *same-dims same-data same-vector*				
Paramters	*Name*	*Type*	*Default*	*Min*	*Max*
	Channel	selection	none | scalar		
	clear output	boolean	on		

DESCRIPTION

ip lookup passes a byte or short field through an integer lookup table. The number in the field is used as an index into the lookup table. The original number is replaced by the number found at that index in the lookup table.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short] *n-vector*)

The right input is a 2D or 3D uniform field of type byte or short. (Float is not accepted.) It can be any vector length. Generally, this is an image. Because the numbers in this field are used as an array index, they should be unsigned. If the field is 3D, then the operations are performed on Z successive XY slices.

Data Field (optional; field 1D uniform scalar integer)

This center input is a 1D integer field containing the lookup table array. The field's X dimension should be long enough to satisfy any index value from the input field. Indexes outside the bounds of the field are undefined. Lookup tables for byte input fields should not exceed 256 in length; short lookup tables should not exceed 32767.

This input can be generated by another module, or read from a file using **read field** or ADIA.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to perform the lookup operation upon. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be used to create the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

clear output

A boolean switch. If on, the output field has the new data created by **ip lookup**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. The header's min / max data values are set to invalid.

EXAMPLE 1

```
               IP READ VFF
                    |
          |---------|
          |         |
IP HISTOGRAM        |
          |         |
          |------|  |
                 |  |
             IP LOOKUP
                    |
                    |
          IMAGE VIEWER
```

EXAMPLE 2

```
                            IP READ VFF
                                 |
          |----------------------|
          |                      |
IP THRESHOLD                     |
          |                      |
          |                      |
          |                      |
          |   FILE DESCRIPTOR    |
          |            |         |
          |            |------|  |
   EXTRACT SCALAR             |  |
          |                   |  |
          |-----------------| |  |
                            | |  |
                          IP LOOKUP
                                 |
                                 |
                       IMAGE VIEWER
```

RELATED MODULES

ip rescale

SEE ALSO

The example script Imaging / IP LOOKUP demonstrates this module.

NAME

ip median – median field filter

SUMMARY

Name	ip median
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D uniform scalar integer (*structuring element*) field 2D uniform scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-type same-vector*
Parameters	*Name* / *Type* / *Default* Channel / selection / none | scalar clear output / boolean / on

DESCRIPTION

ip_median finds the median value in a local collection of pixels using a structuring element.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the filtering is performed on Z successive XY slices.

Data Field (required; field 2D uniform scalar integer)

The center input is for the 2D structuring element. This is usually obtained from a file via the **ip read sel** module. See that man page for a detailed description of its format.

The logical structuring element describes the region "mask" to be used in performing the median filtering.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

This ROI does not limit the size of the median window in the source field.

PARAMETERS

Channel A set of buttons that select which vector elements to run through the median filter. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be filtered in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

clear output

A boolean switch. If on, the output field has the new data created by **ip median**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. Edge pixels in the destination field are set to 0. The header's min / max data values are set to invalid.

EXAMPLE

```
                READ IMAGE
                     |
IP READ SEL          |
        |            |
        |-------|    |
                |    |
            IP MEDIAN
                     |
          IMAGE VIEWER
```

RELATED MODULES

ip dilate
ip erode
ip read sel
local area ops

SEE ALSO

The example script Imaging / IP MEDIAN demonstrates this module.

NAME

ip merge – merge two fields

SUMMARY

Name	ip merge
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field uniform *same-dims same-data same-vector* field 2D scalar byte (*region of interest*)
Outputs	field uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*
Channel	selection	none | scalar
clear output	boolean	on

DESCRIPTION

ip merge merges two fields (generally, images) on a pixel-by-pixel basis, using a region of interest (ROI) field to specify which source field a given pixel in the output field comes from.

All inputs must have the same dimensions, extents, data type, and vector length.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the merge is performed on Z successive XY slices.

Data Field (required; field uniform *same-dims same-data same-vector*)

The center input must have the same dimensions, extents, data type, and vector length as the right input field.

Data Field (required; field 2D uniform scalar byte)

This leftmost input field is a region of interest. This input is required.

If the ROI value for a particular pixel is non-zero, then the pixel in the output field will come from the first (right input port) field. Otherwise the value will come from the second (center) input port.

If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input fields.

PARAMETERS

Channel A set of buttons that select which vector elements to merge. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be merged in the output field.

If the input field’s vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

clear output

A boolean switch. If on, the output field has the new data created by **ip merge**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

ip merge

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. The header's min / max data values are set to invalid.

EXAMPLE

```
                          READ IMAGE
                               |
         |---------------------|
IP THRESHOLD                   |-----------------|
         |                     |                 |
         |-------------|       |      IP ARITHMETIC
         |             |       |                 |
         |   EXTRACT SCALAR    |                 |
         |             |       |                 |
         |             |----|  | |---------------|
         |                  |  | |
DISPLAY IMAGE              IP MERGE
                               |
                               |
                     IMAGE VIEWER
```

RELATED MODULES

ip blend
composite

SEE ALSO

The example script Imaging / IP MERGE demonstrates this module.

NAME

ip morph – morphological operation

SUMMARY

Name	ip morph				
Availability	Imaging module library				
Type	filter				
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 1D scalar byte (*conditional morph table*) field 1D scalar byte (*optional, unconditional morph table*) field 2D scalar byte (*optional, region of interest*)				
Outputs	field uniform *same-dims same-vector* byte				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Channel	selection	none | scalar		
	iterations	int dial	1	1	*unbounded*
	clear output	boolean	on		

DESCRIPTION

ip morph performs various morphological operations on an input "logical" field and places the result in the output field.

INPUTS

Data Field (required, field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the filtering is performed on Z successive XY slices.

A "logical" field is one of any data type (but usually byte) whose data values are either 0 or 255.

Data Field (required; field 1D scalar byte)
Data Field (optional; field 1D scalar byte)

These fields are morphology table structures that tabulate an output for every possible bit pattern in a 3x3 neighborhood. The are usually read from an external file via **ip read mtable**. See that man page for a detailed description of their formats.

The second-from-the-right input field is a conditional morphology table. This input is required.

The third-from-the-right input field is an unconditional morphology table. This input is optional. The unconditional tables contain an extra bit which reflects whether the previous conditional operation produced a on-pixel; from this, it is possible to prevent connectivity breaking for certain thinning operators.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to morph. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be morphed in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

iterations An integer dial that specifies how many times the morph table should be applied to the input. Allows for iterative morphological operations. The minimum is 1, the maximum is unbounded, and the default is 1. The Status bar reports the progress of the iterations.

clear output

A boolean switch. If on, the output field has the new data created by **ip morph**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-vector* byte)

The output is a byte field with the same dimensions and vector length as the input field.

This output field is a "logical" field, meaning that each value is either 0 or 255. Edge pixels in the output field are set to 0. Those vector elements not selected by **Channel** are set to 0. The header's min/max data values are set to be invalid.

EXAMPLE

```
                    READ IMAGE
                         |
         IP READ MTABLE  |
                    |    |
IP READ MTABLE      |--| |
       |               | |
       |-------------| | |
                     | | |
                  IP MORPH
                         |
                         |
                         |
               IMAGE VIEWER
```

RELATED MODULES

ip dilate
ip erode
ip read mtable

SEE ALSO

The example script Imaging/IP MORPH demonstrates this module.

NAME

ip read kernel – read a convolution kernel from a file into a field

SUMMARY

Name	ip read kernel	
Availability	Imaging module library	
Type	data input	
Inputs	none	
Outputs	field 2D uniform float scalar	
Parameters	*Name*	*Type*
	Read Kernel Browser	file browser

DESCRIPTION

ip read kernel reads a convolution kernel from a file into a 2D uniform float scalar field. This kernel is used as an input to the **ip convolve** module.

PARAMETERS

Read Kernel Browser

A file browser widget to specify the kernel file. A kernel file has this format:

```
KERNEL                 <--1st line says KERNEL
size <x y>             <--2nd line defines X Y dimensions
datatype <datatype>    <--3rd line defines type; only float is supported
          .
        <data>         <--<x> lines of <y> columns,
          .               separated by blanks
          .
```

This, for example, is a 5x5 "boxcar" column kernel:

```
KERNEL
size 5 5
datatype float
0.1 0.1 0.1 0.1 0.1
0.1 0.1 0.1 0.1 0.1
0.0 0.0 0.0 0.0 0.0
-0.1 -0.1 -0.1 -0.1 -0.1
-0.1 -0.1 -0.1 -0.1 -0.1
```

There are sample kernel files in *$AVS_PATH/data/ip/kernel*.

OUTPUTS

Data Field (field 2D uniform float scalar)

The output is a field containing the convolution kernel. This kernel is used as an input to the **ip convolve** module.

RELATED MODULES

ip convolve
generate filters

SEE ALSO

The example script Imaging/IP CONVOLVE demonstrates the **ip read kernel** module.

ip read line

NAME

ip read line – read line of data between two image pixels

SYNOPSIS

Name	ip read line	
Availability	Imaging module library	
Type	mapper	
Inputs	field 2D uniform [byte | short | float] *n-vector* image viewer id structure (*invisible, autoconnect*) mouse info structure (*invisible, autoconnect*)	
Outputs	field 1D *n-vector* image draw structure	
Parameters	*Name*	*Type*
	set pick mode	oneshot

DESCRIPTION

ip read line reads a line of pixel data between two pixels of an image. The data is output as a 1D n-vector field.

Reading data involves an interaction between **ip read line** and the **image viewer** module. **ip read line**'s image draw structure output must be connected to the **image viewer** module's leftmost image draw structure input. See the "Example" below.

You specify the two pixels to measure interactively in the **image viewer** window as follows:

1. The **ip read line** module must have control of the left mouse button in the Image Viewer window. When **ip read line** is first connected and data first passes through it, it should have control of the left mouse button.
2. Press and hold down the left mouse button to select the starting pixel.
3. Move the cursor over the image. As you move the cursor, a line follows it anchored at the starting pixel.
4. To read data, release the left mouse button. The line disappears. There is now no starting pixel defined.

If there are multiple images in the Image Viewer window, and / or multiple sketching modules, then some other module or the Image Viewer itself may have control of the left mouse button. To get control back to **ip read line**:

1. Make the image the current image (use shift-left mouse button or left mouse button).
2. Press **set pick mode** on **ip read line**'s control panel.

   This tells the **image viewer** that the left mouse button will be drawing selection lines, not setting the current image.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)

The input is a 2D uniform field of type byte, short, or float. It can be any vector length.

Note: Though **ip read line** accepts *n-vector* and data type byte, short, or float, the input to image viewer can only be byte, 1-vector or 4-vector.

image viewer id structure (required; invisible, autoconnect)

This input port is invisible by default. It connects automatically to the image viewer module's image viewer id structure output. The two modules communicate the image viewer module's scene id on this connection. Normally, you can ignore its existance.

mouse info structure (required; invisible, autoconnect)

This input port is invisible by default. It connects automatically to the image viewer module's mouse info structure output. The two modules communicate image name, mouse pointer location and button up/down information on this connection. Normally, you can ignore its existance.

PARAMETERS

set pick mode A oneshot that sets the **image viewer**'s upstream mouse picking focus to this module. Use it to regain control of the mouse whenever the left mouse button doesn't seem to be working to draw selection lines.

OUTPUTS

Data Field (field 1D *n-vector*)

The data read.

image draw structure

The left output port contains the image draw structure that connects to the **image viewer** module's leftmost input port.

EXAMPLE

This example shows a simple network to read pixels. The invisible upstream connections coming from image viewer to **ip read line** are not shown.

```
                       READ IMAGE
                           |
                           |
                           |
           |---------------|
           |               |
   IP READ LINE            |
|-------|    |             |
|            |             |
|   EXTRACT SCALAR         |
|            |             |
|            |             |
|  GRAPH VIEWER            |
|                          |
|------------------|       |
                   |       |
                 IMAGE VIEWER
```

RELATED MODULES

image viewer
image probe
sketch roi

SEE ALSO

The example script Imaging/IP READ LINE demonstrates this module.

The upstream feedback mechanism that makes **ip read line** work is described in the *AVS 5 Update* document.

ip read mtable

NAME

ip read mtable – read a morphology table from a file into a field

SUMMARY

Name	ip read mtable	
Availability	Imaging module library	
Type	data input	
Inputs	none	
Outputs	field 1D uniform byte scalar	
Parameters	*Name*	*Type*
	Read Mtable Browser	file browser

DESCRIPTION

ip read mtable reads a morphology table from a file into a 1D uniform byte scalar field. These tables are used as inputs to the **ip morph** module.

PARAMETERS

Read Mtable Browser

A file browser widget to specify the morphology table file. The default directory location is defined by DataDirectory.

A morphology table is a binary file with the following format:

#EC2<*3 blanks*><*name of table*><1D byte stream, data values either 0 or 255>

The "<>" characters in the syntax above are not part of the file. Any XY "dimensioning" of the 1D stream occurs implicitly when it is applied to a field with **ip morph**.

There are sample morphology table files in *$AVS_PATH/data/ip/mtable.*

OUTPUTS

Data Field (field 1D uniform byte scalar)

The output is a 1D uniform byte scalar field containing the morphology table. These tables are used as inputs to the **ip morph** module.

EXAMPLE

```
                        READ IMAGE
                             |
               IP READ MTABLE  |
                        |    |
     IP READ MTABLE      |--| |
            |               | |
            |-------------| | |
                          | | |
                      IP MORPH
                             |
                             |
                             |
                   IMAGE VIEWER
```

RELATED MODULES

ip morph

SEE ALSO

The example script Imaging/IP MORPH demonstrates the **ip read mtable** module.

NAME

ip read sel – read a structuring element from a file into a field

SUMMARY

Name	ip read sel	
Availability	Imaging module library	
Type	data input	
Inputs	none	
Outputs	field 2D uniform integer scalar	
Parameters	*Name*	*Type*
	Read Sel Browser	file browser

DESCRIPTION

ip read sel reads a structuring element from a file into a 2D uniform integer scalar field. This structuring element is used as an input to the **ip dilate**, **ip erode**, and **ip median** modules.

PARAMETERS

Read Sel Browser

A file browser widget to specify the structuring file. This is an ASCII file, containing only 0's and 1's.

A structuring element file has this format:

```
SEL                       <--1st line says SEL
size <x y>                <--2nd line defines X Y dimensions
          .
        <data>            <--<x> lines of <y> columns,
          .                  separated by blanks
          .
```

This, for example, is a 3x3 "cross" structuring element:

```
SEL
size 5 5
0 0 1 0 0
0 0 1 0 0
1 1 1 1 1
0 0 1 0 0
0 0 1 0 0
```

There are sample structuring element files in *$AVS_PATH/data/ip/sel*.

OUTPUTS

Data Field (field 2D uniform integer scalar)

The output is a 2D integer field containing the structuring element. This structuring element is used as an input to the **ip dilate**, **ip erode**, and **ip median** modules.

RELATED MODULES

ip dilate
ip erode
ip median

SEE ALSO

The example scripts Imaging/IP DILATE, IP MEDIAN, and IP ERODE demonstrate this module.

ip read vff

NAME

ip read vff – import a SunVision *.vff*-format image file into an AVS field

SUMMARY

Name	ip read vff		
Availability	Imaging module library		
Type	data		
Inputs	none		
Outputs	field 2D uniform [byte | short | float] *n-vector*		
Parameters	*Name*	*Type*	*Default*
	Read VFF Image Browser	file browser	
	Gamma Correct	boolean	off

DESCRIPTION

ip read vff converts SunVision *.vff*-format image files into a 2D uniform AVS field with dimensions equal to size=*xsize ysize*, vector length equal to bands=*n*, and a data type that corresponds most closely to bits=*n*. These fields can be used in a network, and / or saved to disk with the **write field** module.

ip read vff reads the *.vff* file's header. It processes the information in the following way:

rank The input image must be of rank=2. This produces an AVS field with ndim=2.

type The input image must be type=raster.

size size is interpreted as dim1=*xsize* and dim2=*ysize*.

rawsize The rawsize is ignored. *.vff* files are assumed to contain just one 2D image. Fields do not store size information. The size of the resulting field is implicitly:

(dim1 x dim2 x veclen x sizeof(data)) + header + extent information

bands bands are taken as the output field's veclen.

bits AVS fields can contain byte, short, int, float, and double data types. (The actual size of these data types can vary from platform to platform; for example, 32 or 64 for int.) However, **ip read vff** assumes that the input is either a byte, short, or real, as these were the supported image data types in SunVision. Bit values are rounded up to the next-matching AVS data type. For example, a 12 bit image will be stored in a field with data type short. An 8 bit image is stored in a byte field. A 24 or 32 bit image is stored as a float.

AVS fields contain just one data type, not mixed data types. The largest bit value is taken as the target value for the remaining bit values. For example, bits= 8 16 would result in a 2-vector short output field.

format **ip read vff** assumes base image file order: all the bands of a pixel are stored together.

ip read vff will automatically swap an ABGR 4 band, 1 byte per band *.vff* input file to be a 4-vector byte ARGB AVS image.

It will also automatically swap a BGR 3 band, 1 byte per band into a 3-vector RGB field.

One band, 1 byte per band inputs are assumed to be monochrome. They produce 1-vector byte fields with a vector label "grey".

All other formats are simply copied to the output field, and their vector labels set to "band0", "band1", etc. Vector labels past the first four are not set. If this does not produce a useable field, you may still be able to import the *.vff* file with the **read field** module or ADIA's **file descriptor** module.

origin This field is ignored. **ip read vff** always produces a uniform field which assumes the origin is the upper left corner at 0,0.

extent These values are ignored. **ip read vff** uses the size=*xsize ysize* as the output field's header extents. Coordinate area extents are not set.

data_offset
data_scale
title These values are ignored.

PARAMETERS

Read VFF Image Browser
A file browser to select the *.vff* input file.

There are example *.vff* files in the directory *...avs/data/ip/vff*.

Gamma Correct
A boolean switch. If the input image is not gamma-corrected, then turning this on causes AVS to apply the gamma correction factor defined by the **-gamma** command line option or the **Gamma** *.avsrc* file keyword to the image. This is sometimes necessary because images that display well under SunVision on Sun workstations may appear too dark on other monitors. **Gamma Correct** lightens them. The default is off (no gamma correction).

OUTPUTS

Data Field (field 2D uniform [byte | short | float] *n-vector*)
The output is an AVS field.

EXAMPLE

```
          IP READ VFF
               |
               |
       IMAGE VIEWER
```

RELATED MODULES

read field
file descriptor
write field
write vffimage

SEE ALSO

The example script Imaging / IP READ VFF demonstrates this module.

See the discussions of the AVS field data type in: the **read field** module man page; the "Importing Data into AVS" chapter of the *AVS User's Guide*; and the "AVS Data Types" chapter of the *AVS Developer's Guide*.

NOTE

SunVision and AVS terminology differ somewhat. A *.vff* "band" is equivalent to a "vector element" in an AVS field. A "single-band image" is thus a "scalar field". A 4-band image is a 4-vector field, etc. Moreover, modules usually refer to the multiple vectors in a field as "channels". Thus, a 4-vector byte field containing alpha, red, green, blue vector elements has four channels. Channels/vector elements have optional labels that are specified in the field's header. Such specified labels will replace the default "Channel 0, Channel 1," etc. selections on module control panels.

A more subtle difference is the use of the term "image". In AVS, "image" refers specifically to 2D uniform 4-vector byte fields whose vector elements contain alpha, red, green, blue pixel information. There are also "image files" (*.x* suffix) that are a specific binary storage format for alpha, red, green, blue pixel values. (See the "AVS Module: read image" section in the "Importing Data into AVS" chapter of the *AVS User's Guide*.)

A SunVision "image" has a broader definition that corresponds to the broad use of the term "image" found in the image processing field. They are 2D, but can have one, two, or many bands. The data in the bands can represent alpha, red, green, blue, or *any* value, such as density or temperature. A "pixel" is just the data in all the bands at a particular x,y coordinate; not necessarily an ARGB. Data is not restricted to bytes, and can be of any type.

Thus, a SunVision "image" corresponds to a wide variety of 2D uniform AVS fields, of which an AVS "image" is just one particular type. When manipulating former SunVision images in AVS networks, you can do anything with them that you can do with a 2D uniform AVS field.

The main tool for breaking up a multi-banded image (n-vector field) into its component bands (vector elements) for individual manipulation is the **extract scalar** module. Bands (vector elements) are recombined with the **combine scalars** module.

LIMITATIONS

Complex image importation is not supported since AVS does not support a complex field data type.

NAME

ip reflect – rotate or transpose field

SUMMARY

Name	ip reflect
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector*
Outputs	field uniform *same-dims same-data same-vector*
Parameters	*Name* / *Type* / *Default* dir_code / choice / horizontal

DESCRIPTION

ip reflect reflects a field (usually an image) in one of seven different directions.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the reflection is performed on Z successive XY slices.

PARAMETERS

dir_code A set of radio buttons to select the direction of reflection. The default is **horizontal**. The choices are:

horizontal	across the Y axis
vertical	across the X axis
transpose main	across the main diagonal
transpose anti	across the anti-diagonal
90 degrees	counterclockwise 90 degrees
180 degrees	rotate counterclockwise 180 degrees
270 degrees	rotate counterclockwise 270 degrees

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output field has the same dimensions, vector length, and data type as the input field. When **horizonal**, **vertical**, or **180 degrees** are selected, the output field has the same extents as the input field. For other techniques, the output field will have different extents if the original dimensions were not square. The header's min / max data values are a copy of the input field's.

EXAMPLE

```
          READ IMAGE
               |
               |
          IP REFLECT
               |
               |
     IMAGE VIEWER
```

RELATED MODULES

ip rotate
ip translate
ip twarp
ip warp
ip zoom
transpose
mirror

SEE ALSO

The example script Imaging / IP REFLECT demonstrates this module.

NAME

ip register – determine maximum correlation position

SUMMARY

Name	ip register
Availability	Imaging module library
Type	data output
Inputs	field 2D uniform [byte | short | float] *n-vector* field 2D uniform [byte | short | float] *n-vector* (template)
Outputs	none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Input Channel	choice	<channel 0>		
Template Channel	choice	<channel 0>		
X Center	int dial	*max x/2*	0	*max x-1*
Y Center	int dial	*max y/2*	0	*max y-1*
X Range	int dial	*max x/2*	0	*max x-1*
Y Range	int dial	*max y/2*	0	*max y-1*
X Step	int dial	1	0	*max x-1*
Y Step	int dial	1	0	*max y-1*
Correlation	string block			

DESCRIPTION

ip register performs a sequential search correlation match of a field with a template.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)

The right input is a 2D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. This is the field that will be correlated against the template.

Data Field (required; field 2D uniform [byte | short | float] *n-vector*)

The left input is a template field that is to be correlated with the right input. This template field does not have to match the main input field's extents, data type, or vector length.

PARAMETERS

Input Channel

A set of radio buttons that selects which channel (vector element) of a multi-vector input field to perform the correlation on. There are as many buttons as vector elements. One vector element can be selected at one time. The default is the first channel listed.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "channel 0", "channel 1," etc.

Input Channel

A set of radio buttons that selects which channel (vector element) of a multi-vector template field to use as the correlation template. There are as many buttons as vector elements. One vector element can be selected at one time. The default is the first channel listed.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "channel 0", "channel 1," etc.

X Center
Y Center Two integer dials that define the location of the pixel in the input field about which the search is performed. The range is 0 to the X and Y extents (*max x-1, max y-1*), of the input field. The default is the midpoint (*max x/2, max y/2*).

X Range
Y Range Two integer dials that specify the bounds of the area in the input field over which the search takes place. The numbers on the dials are taken to be + and **-** from **X** and **Y Center**.

The range is 0 to the X and Y extents (*max x-1, max y-1*), of the input field. The default is the midpoint (*max x/2* and *max y/2*), respectively.

X Step
Y Step Two integer dials that specify the granularity of the search in pixels. The range is 0 to the X and Y extents (*max x-1, max y-1*), of the input field. The default is 1.

Image Correlation

Areas of the search region which require the template extend beyond the edge of the input field are not calculated.

A string block text widget that reports the results. The widget is located on the module's control panel.

Three floating values are reported:

X Offset
Y Offset
The XY location of the pixel in which the maximum correlation was found.

Maximum correlation
The maximum correlation data value.

EXAMPLE

```
 READ IMAGE
      |
      |------|
      |    CROP
      |  |---|
      |  |
IP REGISTER
```

RELATED MODULES

ip compare
ip extrema
ip statistics

SEE ALSO

The example script Imaging / IP REGISTER demonstrates this module.

NAME

ip rescale – rescale a field

SUMMARY

Name	ip rescale
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	selection	none | scalar		
src min	float dial	0.0	*unbounded*	*unbounded*
src max	float dial	255.0	*unbounded*	*unbounded*
dst min	float dial	0.0	*unbounded*	*unbounded*
dst max	float dial	255.0	*unbounded*	*unbounded*
clear output	boolean	on		

DESCRIPTION

ip rescale rescales fields (usually images) by linearly remapping the pixel values between **src min** and **src max** in the input field to the output field in the range **dst min** to **dst max**.

Source pixels whose values are outside the **src min** and **src max** range are mapped to the destination's corresponding limits ("clamped"). For example, if **src min** and **src max** are 20.0 and 100.0 and **dst min** and **dst max** are 40.0 and 80.0, all source values below 20.0 are mapped to 40.0 and all source values above 100.0 are mapped to 80.0.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the rescaling is performed on Z successive XY slices.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to rescale. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be rescaled in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

src min
src max
dst min
dst max Floating point dials that establish the range of input data values (**src min** and **src max**) to map to the range of output data values (**dst min** and **dst max**). The range is unbounded. The default is 0.0 for **src min** and **dst min**; and 255.0 for **src max** and **dst max**.

clear output

A boolean switch. If on, the output field has the new data created by **ip rescale**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, data type, and vector length as the input field. The header's min / max data values are set to invalid.

EXAMPLE 1

```
     READ IMAGE
          |
          |
  IP RESCALE
          |
          |
IMAGE VIEWER
```

EXAMPLE 2

```
                  READ FIELD
                       |
              ORTHOGONAL SLICER
                       |
             |------|  |
             | IP RESCALE
             |         |
        |--------------|
SKETCH ROI   |         |    GENERATE COLORMAP
     |  |----|         |            |
     |                 |            |
     |----------|   |---------------|
                |   |  |
                |   |  |
             IMAGE VIEWER
```

RELATED MODULES

ip linremap
contrast

SEE ALSO

The example script Imaging / IP RESCALE demonstrates this module.

NAME

ip rotate – rotate a field

SUMMARY

Name	ip rotate				
Availability	Imaging module library				
Type	filter				
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector*				
Outputs	field uniform *same-dims same-data same-vector*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	angle	float dial	0.0	-180.0	180.0
	interp	choice	point		

DESCRIPTION

ip rotate rotates a field about its center.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the rotation is performed on Z successive XY slices.

PARAMETERS

angle is the angle of rotation, in degrees, on a floating point dial. A positive angle indicates counter-clockwise rotation; a negative angle indicates clockwise rotation. Internally, **ip rotate** converts angles to radians. The relationship of an angle expressed in radians and degrees is:

angle(radians)=angle(degrees)*(pi / 180)

The default is 0.0; the range -180.0 to 180.0.

interp A set of radio buttons to select the interpolation method. The choices are **point**, **bilinear**, and **bicubic**. The default is **point**.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, extents, data type, and vector length as the input field.

A field rotated at an **angle** other than a multiple of 90 degrees produces a field that is clipped by the extents of the input field, with empty areas rendered as 0 (black) pixels.

The header's min / max data values are set to invalid.

EXAMPLE

```
        IP READ VFF
             |
             |
         IP ROTATE
             |
             |
    IMAGE VIEWER
```

ip rotate

RELATED MODULES

ip reflect
ip translate
ip twarp
ip warp
ip zoom
transpose
mirror

SEE ALSO

The example script Imaging / IP ROTATE demonstrates this module.

NAME

ip statistics – find field mean and variance

SUMMARY

Name	ip statistics
Availability	Imaging module library
Type	data output
Inputs	field 2D uniform [byte | short | float] *n-vector* field 2D scalar byte (*optional, region of interest*)
Outputs	none
Parameters	*Name* / *Type* / *Default* Channel / choice / <channel 0> Statistics / string block

DESCRIPTION

ip statistics finds the number of pixels, mean, and variance of one channel in a field. These outputs are displayed as floating point values in an output text widget on the module's control panel.

INPUTS

Data Field (required; field 2D uniform [byte | short | float] *n-vector*) The right input is a 2D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image.

Data Field (optional; field 2D uniform scalar byte)
This left input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector element to calculate the statistics for. There are as many buttons as vector elements. One vector element can be selected at one time. The default is the first channel listed.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "channel 0", "channel 1," etc.

Statistics A string block text widget that reports the results. The widget is located on the module's control panel.

Three floats are reported:

Number of Pixels
Mean
Variance

EXAMPLE

```
          READ IMAGE
               |
               |
      IP STATISTICS
```

RELATED MODULES

ip compare
ip extrema
ip register
statistics

SEE ALSO

The example script Imaging/IP STATISTICS demonstrates this module.

NAME

ip threshold – threshold field against a float value

SUMMARY

Name	ip threshold
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D scalar byte (*optional, region of interest*)
Outputs	field uniform *same-dims same-vector* byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	selection	none | scalar		
lo value	float dial	0.0	*unbounded*	*unbounded*
hi value	float dial	*maxval*	*unbounded*	*unbounded*
invert	boolean	off		
clear output	boolean	on		

DESCRIPTION

ip threshold thresholds a field against a floating point value, producing a bi-valued (logical) byte field as a result. A logical field is one in which all values are either 0 or 255.

If the values of **lo value** and **hi value** are equal, field values below the limit are set to 0 and field values that are greater than or equal to the limit are set to MAXBYTE. (MAXBYTE is defined as 255.)

If the values of **lo value** and **hi value** are different, field values that are less than or equal to the low limit are set to 0; field values that are greater than or equal to the high limit are also set to 0, and values within the high and low limits are set to MAXBYTE (255).

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the thresholding is performed on Z successive XY slices.

Data Field (optional; field 2D uniform scalar byte)

This leftmost input field is an optional region of interest. If connected, only the pixels designated by the ROI are affected. If the input is a 3D field, the ROI is applied to Z successive XY slices. The ROI must have the same XY extents as the input field.

PARAMETERS

Channel A set of buttons that select which vector elements to threshold. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be thresholded in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

lo value is the low-limit threshold value. This is a float dial. It is unbounded; the default is 0.0.

hi_value is the high-limit threshold value. This is an unbounded float dial. The default depends on the input data type. Byte input defaults to 255.0. Short input defaults to 65535.0. Float data causes the dial to remain truly unbounded (no maximum).

invert A boolean switch. If off, the destination bi-valued result is produced as described above. If **invert** is on, the bi-valued results are inverted (pixels are set to MAXBYTE instead of zero and vice versa). The default is off.

clear output
A boolean switch. If on, the output field has the new data created by **ip threshold**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-vector* byte)
The output is a field with the same dimensions and vector length as the input field. It is of type byte. It is a "logical" field, meaning that it contains only either 0 or MAXBYTE (set to 255) values. Those vector elements not selected by **Channel** are set to zero. The header's min / max data values are set to invalid.

EXAMPLE

```
          IP READ VFF
               |
               |
      IP THRESHOLD
               |
               |
      IMAGE VIEWER
```

EXAMPLE 2

```
                  READ VOLUME
                       |
             |----------|
             |          |
   IP THRESHOLD         |
             |          |
ORTHOGONAL SLICER       |
            |-------|   |
                    |   |
              IP THRESHOLD
                       |
          ORTHOGONAL SLICER
                       |
              IMAGE VIEWER
```

RELATED MODULES

ip dilate
ip erode
ip morph
define roi
threshold

SEE ALSO

The example script Imaging / IP THRESHOLD demonstrates this module.

ip translate

NAME

ip translate – field translation

SUMMARY

Name	ip translate				
Availability	Imaging module library				
Type	filter				
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector*				
Outputs	field uniform *same-dims same-data same-vector*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	dx	int dial	0	*-max x*	*max x*
	dy	int dial	0	*-max y*	*max y*

DESCRIPTION

ip translate copies the input field to the output field with the translation relative to the input specified by **dx** and **dy**.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the translation is performed on Z successive XY slices.

The translation is replicated across multiple vectors.

PARAMETERS

dx
dy Two bounded integer dials that specify how many pixels to shift the field in the **dx** or **dy** direction. Positive or negative translations may be specified.

The default for both is 0. The range is set dynamically to equal - and + the XY extents, respectively, of the input field.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)

The output is a field with the same dimensions, extents, data type, and vector length as the input field.

The translated input field is clipped against the output field's extents, and the exposed area is set to 0 value in the output field.

The header's min / max data values are set to invalid.

EXAMPLE

```
      READ IMAGE
           |
           |
     IP TRANSLATE
           |
           |
  IMAGE VIEWER
```

RELATED MODULES

ip reflect
ip rotate
ip zoom

SEE ALSO

The example script Imaging/IP TRANSLATE demonstrates this module.

ip twarp

NAME

ip twarp – arbitrary field warp using warp data from table

SUMMARY

Name	ip twarp
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 2D uniform 2-vector float (*warp table*)
Outputs	field uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel	selection	none | scalar		
interp	choice	point		
X Offset	int dial	0	0	*input max x - warp max x*
Y Offset	int dial	0	0	*input max y - warp max y*

DESCRIPTION

ip twarp performs an arbitrary warp using a warp table to designate which pixel in the input field corresponds to each pixel in the output field.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the warping is performed on Z successive XY slices.

Data Field (required; field 2D uniform 2-vector float)

The center input is the warp table. It is a 2D uniform 2-vector float field. The warp table can be any size; it does not have to equal the extents of the input field. The output field will have the same extents as the warp table.

Each "cell" of the table is a 2-vector float. The first vector element is the X coordinate of the input field. The second vector element is the Y coordinate of the input field. **ip twarp** takes the input pixel defined by this XY pair and transforms it (with a choice of interpolations) to the location in the output field implicitly defined by the location of the XY pair in the warp table.

For example, if warp table location (25,100) contained the XY vector element pair (30,90), then the pixel at (30,90) in the input field would be warped to position (25,100) in the output field.

To produce a warp table, one could:

- Create an ASCII file with the warp coordinates as defined below, and import it into a 2D uniform 2-vector AVS field using either **read field** or the ADIA application.

  In this table, x00 and y00 are the coordinates of the source pixel that corresponds to the first pixel in the first row of the destination field, and so on.

```
x00   y00   x01   y01   x02   y02   ...   x0n   y0n
x10   y10   x11   y11   x12   y12   ...   x1n   y1n
 .
 .
```

xm0 ym0 xm1 ym1 xm2 ym2 ... xmn ymn

- Write a module that generates a field of the correct type that contains the warp coordinates.

PARAMETERS

Channel A set of buttons that select which vector elements to warp. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be warped in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

interp Radio buttons that set the type of interpolation. The choices are **point**, **bilinear**, and **bicubic**. The default is **point**.

X Offset
Y Offset These integer dials are used when the input field is larger than the output field. If you supply nonzero offsets, for each pair of coordinates in the table, the function adds **X Offset** to the x coordinate and **Y Offset** to the y corrdinate before it reads a pixel value from the input field. For example, if you supply offsets of 10 (x) and 20 (y) and the first two values in your table are 125 and 40, the value of the pixel at 0,0 in the output field will be determined by the value of the pixel at 135,60 in the input. These offsets allow you to use one table to warp a number of subimages in the input field.

OUTPUTS

Data Field (field 2D uniform *same-dims same-data same-vector*)
The output is a field with the same vector, data type, and dimensions as the input field. The field's extents will equal those of the warp table field. The header's min / max data values are set to invalid.

EXAMPLE

This example uses the **read field** module to input a user-supplied warp table from an ASCII file. **read field** could be replaced with a user-supplied module that generated the warp table.

```
              READ IMAGE
                   |
                   |
                 CROP
                   |
      READ FIELD   |
            |      |
            |--|   |
               |   |
           IP TWARP
                   |
                   |
         IMAGE VIEWER
```

RELATED MODULES

ip reflect
ip rotate
ip warp
ip zoom

SEE ALSO

The example script Imaging/IP TWARP demonstrates this module.

NAME

ip warp – polynomial image warp

SUMMARY

Name	ip warp
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* field 1D uniform 2-vector float (*warp coefficients*)
Outputs	field uniform *same-dims same-data same-vector*
Parameters	*Name* / *Type* / *Default*

Name	*Type*	*Default*
Channel	selection	none | scalar
choice	choice	point
clear output	boolean	on

DESCRIPTION

ip warp applies a geometric transform to a field. (This field is generally an image.) The transform is defined as a polynomial mapping from an output pixel position to an input pixel position. The input and output fields need not have the same extents.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)

The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the warping is performed on Z successive XY slices.

Data Field (required; field 1D uniform 2-vector float)

The center input contains the polynomial warp coefficients. It is a 1D uniform 2-vector float field. The first vector element contains the x polynomial warp coefficients; the second vector element contains the y polynomial warp coefficients.

This field can be supplied by the **calc warp coeffs** module, or by a user-written module.

The field containing the warp coefficients has the following format:

1. It has a certain *degree*. *degree* is the maximum degree of x or y (not the maximum cross term degree). Degrees 1 and 2 are accepted. 1 is appropriate for linear polynomials (linear or bilinear). 2 is appropriate for quadratic polynomials (quadratic or biquadratic). For degree 1, 4 coefficients are used. For degree 2, 9 coefficients are used.

   Purely by convention, the number of coefficients should be stored as, and will be retrieved from the field's maximum X extent value. A module that is creating the warp field would set this value with the **AVSfield_set_extent** routine.

2. The body of the field is a 1D 2-vector float that contains the x and y coefficients. The first vector element contains the x coefficients; the second vector element contains the y coefficients. Each can be thought of as an array that contains $(\text{degree} + 1)^2$ polynomial coefficients.

   The ordering of the coefficients is in x major order.

   For degree 1 polynomials (i.e., linear and bilinear) there are four coefficients and their ordering is:

```
input_pixel_x =     cx[0] +
                    cx[1] * x +
                    cx[2] * y +
                    cx[3] * x * y

input_pixel_y =     cy[0] +
                    cy[1] * x +
                    cy[2] * y +
                    cy[3] * x * y
```

This shows the ordering for degree 2 polynomials (i.e., quadratic and biquadratic) with 9 coefficients:

```
input_pixel_x =     cx[0] +
                    cx[1]*x +
                    cx[2]*x*x +
                    cx[3]*y +
                    cx[4]*x*y +
                    cx[5]*x*x*y +
                    cx[6]*y*y +
                    cx[7]*x*y*y +
                    cx[8]*x*x*y*y

input_pixel_y =     cy[0] +
                    cy[1]*x +
                    cy[2]*x*x +
                    cy[3]*y +
                    cy[4]*x*y +
                    cy[5]*x*x*y +
                    cy[6]*y*y +
                    cy[7]*x*y*y +
                    cy[8]*x*x*y*y
```

For example, to warp an image according to the mapping:

```
x_src = 0.2*x_dst*x_dst – 512.0
y_src = 0.5*y_dst + 0.3*x_dst*y_dst – 128.0
```

with degree 2, the coefficients in the field would look like the following, where the first list is the first, x vector element, and the second list is the second, y vector element:

```
cx[0] = –512.0;
cx[1] = 0.0;
cx[2] = 0.2;
cx[3] = 0.0;
cx[4] = 0.0;
cx[5] = 0.0;
cx[6] = 0.0;
cx[7] = 0.0;
cx[8] = 0.0;

cy[0] = –128.0;
cy[1] = 0.0;
cy[2] = 0.0;
cy[3] = 0.5;
```

cy[4] = 0.3;
cy[5] = 0.0;
cy[6] = 0.0;
cy[7] = 0.0;
cy[8] = 0.0;

PARAMETERS

Channel A set of buttons that select which vector elements to warp. There are as many buttons as vector elements. More than one vector element can be selected at one time—each will be warped in the output field.

If the input field's vectors are labelled, then the labels will appear on the buttons. Otherwise, the buttons are labelled "Channel 0", "Channel 1," etc. There is no default selection unless the input is scalar.

choice Radio buttons that set the type of interpolation. The choices are **point**, **bilinear**, and **bicubic**. The default is **point**.

clear output
A boolean switch. If on, the output field has the new data created by **ip warp**, and the rest of the values are 0. If off, those vector elements not selected by **Channel** are copied intact to the output field. **clear output** is on by default.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)
The output is a field with the same vector lenth, data type, and dimensions as the input field. It may have different extents that the input field.

EXAMPLE

This networks shows the warp coefficients being generated interactively using **calc warp coeffs** and the **image viewer**. The automatically-created invisible upstream connections from **image viewer** are not shown.

```
                    READ IMAGE
                          |
                          |
              |-------------|
              |             |
  CALC WARP COEFFS          |
          |   |             |
          |   |---------|   |
          |             |   |
          |           IP WARP
          |                 |
          |---------|       |
                    |       |
                IMAGE VIEWER
```

EXAMPLE 2

This network shows the warp coefficients supplied through a field containing tiepoints, converted to warp coefficients with **calc warp coeffs**.

ip warp

```
    READ FIELD      READ IMAGE
        |                 |
        |                 |
        |   |-------------|
        |   |             |
CALC WARP COEFFS          |
            |             |
            |---------|   |
                      |   |
                     IP WARP
                          |
                          |
                IMAGE VIEWER
```

RELATED MODULES

calc warp coeffs
ip twarp
ip reflect
ip rotate
ip zoom

SEE ALSO

The example scripts Imaging/CALC WARP COEFFS and Imaging/IP WARP demonstrate this module.

NAME

ip write vff – save a AVS image-format field as a SunVision *vff*-format image file

SUMMARY

Name	ip write vff
Availability	Imaging module library
Type	data output
Inputs	field 2D uniform byte 4-vector
Outputs	none

Parameters	*Name*	*Type*	*Default*
	Write VFF Image Browser	file browser	
	Gamma Correct	boolean	on

DESCRIPTION

ip write vff converts an AVS field in image format into a SunVision binary *vff*-format image file and writes it to disk.

The output file's *vff* header will read:

```
ncaa
rank=2;
size= xdim ydim;
format=base;
bands=4;
bits=8 8 8 8;
type=raster;
```

AVS uses ARGB as its true color pixel ordering. This ARGB will be automatically converted to standard SunVision ABGR format in the output file.

INPUTS

Data Field (required; field 2D uniform byte 4-vector)

The input is a field in AVS "image" format.

PARAMETERS

Write VFF Image Browser

A file browser to specify the output file. The default is the **DataDirectory** startup value. No output is generated until an output file is specified. A *.iff* suffix is automatically appended to the output file's name.

Gamma Correct

A boolean switch. AVS images are normally gamma corrected by the factor defined by the **-gamma** command line option or the **Gamma** *.avsrc* file keyword. SunVision images are also normally gamma corrected. (Non-gamma corrected images will appear dark on many monitors.) If you wish to remove the gamma correction when the image is converted, turn off this switch. The default is on (keep gamma correction).

EXAMPLE

```
 READ IMAGE
      |
      |
IP WRITE VFF
```

ip write vff

RELATED MODULES

write field
read vff image

SEE ALSO

The example script Imaging/IP WRITE VFF demonstrates this module.

NAME

ip zoom – zoom field with interpolation

SUMMARY

Name	ip zoom
Availability	Imaging module library
Type	filter
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector*
Outputs	field 2D uniform *same-dims same-data same-vector*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
x factor	float dial	1.0	0.0	*unbounded*
y facator	float dial	1.0	0.0	*unbounded*
interp	choice	point		
x offset	float dial	0.0	0.0	*x-size*
y offset	float dial	0.0	0.0	*y-size*

DESCRIPTION

ip zoom zooms a field using one of four interpolation methods. The zooming can be done with floating-point offsets, which enables you to offset a zoomed image by fractional pixels.

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)
The rightmost input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. Generally, this is an image. If the field is 3D, then the zooming is performed on Z successive XY slices.

PARAMETERS

x factor
y factor are floating dials that set the x and y zoom factors. The range is 0.0 to unbounded; the default is 1.0.

interp Radio buttons to select the interpolation method. The choices are **point**, **bilinear**, **bicubic**, and **adaptive**. The default is **point**.

You can use **point**, **bilinear**, **bicubic** interpolation whether you are scaling a field up or down.

You can only use **adaptive** ("adaptive support") interpolation if you are scaling an image down by a factor of 2 or more; that is, your scaling factors must be equal to or less than 0.5. With **adaptive**, the value of each pixel in the output field is calculated by averaging the values of a block of pixels in the input field. The size of this block is determined by the scale factor such that all the pixels in the input field affect a pixel in the output field.

x offset
y offset are the coordinate offsets to the first pixel in the zoom area. The default is 0.0; the minimum is 0.0, and the maximum is the X/Y size of the image.

OUTPUTS

Data Field (field uniform *same-dims same-data same-vector*)
The output is a field with the same dimensions, data type, and vector length as the input field. The extents of the output field will vary depending upon the zoom factor. The header’s min / max data values are

set to invalid.

EXAMPLE

```
         READ IMAGE
              |
              |
           IP ZOOM
              |
              |
    IMAGE VIEWER
```

RELATED MODULES

ip reflect
ip rotate
ip warp
ip twarp
ip translate
interpolate
downsize

SEE ALSO

The example script Imaging/IP ZOOM demonstrates this module.

NAME

isosurface – generate an isosurface for a volume of data

SUMMARY

Name	isosurface
Availability	Volume, FiniteDiff module libraries
Type	mapper
Inputs	field 3D scalar *any-data any-coordinates* field 3D scalar *any-data* (optional) colormap (optional)
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Isosurface Level	float	128	*unbounded*	*unbounded*
Optimize Surface	toggle	off		
Optimize Wireframe	toggle	off		
Flip Normals	toggle	off		

DESCRIPTION

The **isosurface** module inputs a volume data set (3D field of values, either curvilinear, rectilinear, or uniform). It produces a geometric object that represents an isosurface of this object. An *isosurface* is a 3D generalization of a 2D contour line — it connects all field elements that have the same parameter-controlled data value.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)

The input data must represent a volume, with a single value of any primitive data type for each field element.

Auxiliary Data Field (optional; field 3D scalar *any-data*)

This port can be used to generate a colored isosurface; the color at each point on the surface indicates the value of another attribute of the volume. For instance, you could generate a pressure isosurface with colors indicating the temperature at each point on the surface.

In this case, the **Data Field** would be used to input the pressure data, and the **Auxiliary Data Field** would be used to input the temperature data. In all cases, both volume data sets must have the same dimensions.

Colormap (optional; colormap)

If you use an **Auxiliary Data Field**, you must also specify a colormap. Since the auxiliary volume data is floating-point, you must adjust the **lo value** and **hi value** parameters of the **generate colormap** module to correspond to the minimum and maximum data values of the auxiliary field.

For the pressure-temperature example described above, the temperature data set might have data values in the range 0.0–100.0 degrees. In this case, set the **lo value** to 0 and **hi value** to 100 in **generate colormap**.

PARAMETERS

Isosurface Level

A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the field element's data value equals the **Isosurface Level** value. The dial is unbounded. However, the resolution of the dial is rescaled to the minimum and maximum data range each time the input changes. The default is reset to minval if the previous setting is less than the new minimum value. The default is reset to maxval if the previous setting is greater than the new maximum value. Otherwise, it is left unchanged.

Optimize Surface
Optimize Wireframe

These two toggle parameters allow you to control a tradeoff between how efficiently the isosurface is computed and how efficiently it can be rendered. If you turn on **Optimize Surface**, extra time will be spent generating a more optimal surface description, containing fewer triangles.

Turn on **Optimize Wireframe** to generate a wireframe representation for the isosurface along with the shaded surface representation. If you want to view your surface as a wireframe (using the **Objects** selection in the **geometry viewer** control panel), you must toggle this on.

Flip Normals

Reverses the direction of each surface normal in the generated isosurface. If the normals point in the wrong direction, the outside of the isosurface will appear at the ambient light intensity. In this case, click this button or specify bi-directional lighting in the **geometry viewer** control panel (**Lights** selection).

OUTPUTS

Isosurface (geometry)

A shaded surface, optionally with an associated wireframe representation.

NOTES

The most important parameter is the **Isosurface Level** (threshold), which is defined in the unbounded floating-point data space of the volume. Whenever the input to the **isosurface** module changes, the range for the **Isosurface Level** parameter is set to be the range of the input data. If the current setting for the **Isosurface Level** parameter is outside this data range, the **Isosurface Level** parameter is changed to reflect the new range.

Because **isosurface** is compute-intensive, it is often advisable to include a **downsize** module in the network. This allows you to quickly select a proper isosurface level before generating one at full resolution.

Another technique is to use the **Action** capability of the Geometry Viewer (**geometry viewer** module) to save and play back a sequence of isosurfaces at different value levels.

EXAMPLE 1

```
   READ VOLUME
        |
     DOWNSIZE
        |
    ISOSURFACE
        |
  GEOMETRY VIEWER
```

EXAMPLE 2

This example uses an auxiliary data set.

```
                READ VOLUME (color volume)          READ VOLUME (surface volume)
                      |                                  |
                      |                                  |
                      ------------------|   |-------------
           GENERATE COLORMAP            |   |
                  |                     |   |
                  ------------------|   |   |
                                 ISOSURFACE
                                        |
                                  GEOMETRY VIEWER
```

RELATED MODULES

geometry viewer, render geometry, downsize, generate colormap, read field, read volume

LIMITATIONS

In some circumstances, the generated isosurface may have some of its normals pointing inward and some outward. There is no way to correct this situation, but usage of bi-directional lighting (**Lights** selection of the Geometry Viewer / **geometry viewer**) may be helpful.

SEE ALSO

The example script FIELD LEGEND demonstrates the **isosurface** module.

label

NAME

label – creates a title for flexible geometry viewer annotation

SUMMARY

Name	label
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data
Inputs	none
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Value	float dial	0.0	*unbounded*	*unbounded*
Title String	typein	*none*		
Font Number	int slider	0	0	20
Drop Shadow	boolean	off		
Text Alignment	choice	Center		
X Position	float slider	0.00	-1.00	1.00
Y Posiiton	float slider	0.00	-1.00	1.00
Text Height	float slider	0.10	0.00	1.00
Red	float slider	0.70	0.00	1.00
Green	float slider	0.70	0.00	1.00
Blue	float slider	0.70	0.00	1.00

DESCRIPTION

label creates a label style text string in GEOM format. This label is input to the **geometry viewer**. Once in the Geometry Viewer, the label behaves like a Geometry Viewer title. There are two advantages to using this module over the labelling facilities in the Geometry Viewer:

1. The labelling information is saved with a network, and
2. The optional floating point parameter (**Value**) can come from another module. It can represent some important variable such as time, animation step, some parameter, etc.

   The **Title String** can contain a '%f' (like C programs) to include this parameter. For example, the **Title String** can be "Time Step %f" and the value of **Value** will get transferred to the geometry title. Thus, titles become automatic and dynamic.

OUTPUTS

geom (geometry)

The text string as a geom title label.

PARAMETERS

Value (dial)

A floating point number that can appear in the label as long as the **Title String** contains a %f. If you make this parameter visible on the module icon (Module Editor **Parameter Editor**'s **Port Visible** toggle), then you can attach it to another module such as **animated float**.

Title String

The character string to appear as a title. If it contains a %f, the value of the **Value** parameter is included.

Font Number (islider)

A value from 0 to 20 for the available fonts. The actual font number to font mapping varies from system to system.

Drop Shadow (boolean)

When on, this produces a black drop shadow. Drop shadows may not be implemented on all renderers.

Text Alignment (choice)

Describes the start of the text relative to its position. The choices are **Left**, **Center** (default), and **Right**.

X Position

Y Posiiton Floating point sliders that position the title on the screen. (0.0, 0.0) is the center of the window.

Text Height

Floating point sliders to specify the font height. The range is from 0.0 to 1.0; the default is 0.10. The actual font sizes available varies from system to system.

Red

Green

Blue Floating point sliders that determine the color of the label.

EXAMPLE 1

```
LABEL          READ GEOM
  |                 |
  |-----------|-----|
              |
   GEOMETRY VIEWER
```

SEE ALSO

The example script LABEL illustrates the **label** module.

local area ops

NAME

local area ops - image processing based on pixel neighborhoods

SUMMARY

Name	local area ops					
Availability	Imaging module library					
Type	filter					
Inputs	field 2D 4-vector byte uniform (*image*) OR field 1-3D scalar *any-data any-coordinates*					
Outputs	field of same type as input					
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*	*Choices*
	kernel width	integer	3	3	31	
	choice	choice	Min			Min, Max, Median, Mean

DESCRIPTION

local area ops contains four operations used in image processing, each of which takes an input field and computes an output image using some function. In a "local area operation" the value of each pixel in the output image is based on the values of pixels in its immediate neighborhood. The kernel is the NxN neighborhood of pixels surrounding each pixel used to calculate each new pixel value. The "width" of the kernel thus determines the size of this neighborhood.

In the operation **Min**, for example, using a filter width of 3, the value of each pixel in the output image becomes the minimum value of the pixel and the 8 pixels surrounding it.

In the case of an image, which is a 2D field of 4-byte vectors, **local area ops** disregards the alpha bytes and separates the red, green, and blue bytes. Then it applies the operation separately to each color byte, before reassembling the bytes into 4-vector image format. The status bar shows the module processing three times, once for each color byte.

Apart from AVS images **local area ops** handles only scalar values of any data type. All data-types are converted to floats during computation and then converted back in the output of **local area ops**.

In order to handle edge effects, a border around the perimeter of the image is not operated on. The border is half the width of the kernel.

INPUTS

Data Field (required; field 2D 4-vector byte uniform (*image*)) OR

Data Field (required; field 1-3D scalar *any-data any-coordinates*) Typically, the input will be an AVS *image*, which is a 2D field of 4-vector bytes.

The input may be any 1-3D field of scalar values of *any-data any-type*.

PARAMETERS

choice sets which local area operation to apply. There are 4 options:

Min

In the **min** operation each pixel in the output image becomes the minimum of the pixels in its immediate neighborhood. This has the effect of shrinking light regions of an image, and is refered to as a "region shrinking" operation.

Max

In the **max** operation each pixel in the output image becomes the maximum of the pixels in its immediate neighborhood. This has the effect of enlarging light regions of an image, and is refered to as a "region growing" operation.

Median

In the **median** operation the pixels in the neighborhood are sorted. Then the pixel at the center of the neighborhood gets the value that is in the middle value of the sorted array. This has an effect similar to the mean operation, but it can be especially useful in removing noise from an image, since anomalies are not likely to effect the output image. Note: since the **median** calculation requires a sort, it is very compute intensive, especially when the filter width is large. AVS puts up a warning message when the **median** operation is selected.

Mean

In the **mean** operation each pixel in the output image becomes the average of the pixels in its immediate neighborhood. This has the effect of reducing the contrast of an image between the light and the dark regions.

kernel width

Determines the size of the neighborhood of pixels contributing to the value of each pixel in the output image.

OUTPUTS

Output Field

The output field is the same type as the input data field.

EXAMPLE 1

The following network reads in an image, applies the local area operations to it, and displays the resulting image:

```
   READ IMAGE
        |
        |
LOCAL AREA OPS
        |
        |
  IMAGE VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read image
pixmap to image
orthogonal slicer
any other module which outputs a field of scalars or an image

Modules that can process the output of **local area ops**:

display image
image viewer
any other module which takes a 2D field as input

Modules that have similar function:

ip convolve
ip read kernel

SEE ALSO

The example script LOCAL OPS demonstrates the **local area ops** module.

NAME

luminance – compute the luminance of an image

SUMMARY

Name	luminance
Availability	Imaging module library
Type	filter
Inputs	field 2D uniform 4-vector byte *(image)*
Outputs	field 2D uniform scalar byte
Parameters	none

DESCRIPTION

The **luminance** module computes the luminance (brightness) of an image, then outputs a 2-dimensional field of the same dimensions, but with a *scalar* byte value for each pixel in the original image instead of the full four-byte alpha, red, green, blue vector.

The **luminance** (I) is calculated as follows:

$$I = (0.299 * \text{red}) + (0.587 * \text{green}) + (0.114 * \text{blue})$$

This luminance byte value can be used to produce a black and white version of the original image (with **colorizer**), or substituted back into the alpha byte of the original image (with **replace alpha**) to produce transparency effects.

INPUTS

Image (required; field 2D uniform 4-vector byte)
The image whose luminance to calculate.

OUTPUTS

Data Field (field 2D uniform scalar byte)
The output field has the same dimension as the input image, but with a scalar byte value representing the image luminance at each original pixel instead of color value.

EXAMPLE 1

The following network reads an image, computes its luminance, colorizes the resulting field with the default black and white colormap, producing a black and white version of the original image. The result is displayed through the **image viewer**.

```
  READ IMAGE
      |
  LUMINANCE
      |
  COLORIZER
      |
 IMAGE VIEWER
```

EXAMPLE 2

This network takes a geometry, displays it on the screen, then converts the screen pixmap to an image, computes its luminance, uses that to create an alpha mask, renders a shaded background and composites the rendered image over the shaded background. The **contrast** modules controls should be set to : minimum and maximum input contrast, both 1; minimum output contrast 0, and maximum output contrast, 255. If the original geometry were *$AVS_PATH/data/geometry/jet.geom* and the **background** module were set to produce a sky-like pattern, this would produce a jet over a sky field.

```
                          READ GEOM
                              |
                       GEOMETRY VIEWER
                              |-----------------------|
                              |                       |
    __________________________|                DISPLAY IMAGE
    |              |          |
BACKGROUND    LUMINANCE       |
    |              |          |
    |           CONTRAST      |
    |              |------|   |
    |                   REPLACE ALPHA
    |                         |
    |----------------------|  |
                          COMPOSITE
                              |
                         IMAGE VIEWER
```

RELATED MODULES

Modules that could provide the **Image** input:

Any module that produces an image as output

Modules that can process **luminance** output:

colorizer
contrast
Any modules that can process a 2D scalar field

See also **background**, **composite**, **replace alpha**, and **extract scalar**

SEE ALSO

The example script LUMINANCE demonstrates the **luminance** module.

NAME

minmax – set min and max values of a selected vector in an AVS field

SUMMARY

Name	minmax
Availability	Supported, Volume, Imaging, Finite Diff module libraries
Type	filter
Inputs	field
Outputs	field (of the same type) min value (float) max value (float)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
channel	integer dial	0	0	*n-vectors - 1*
min value	float typein		*unbounded*	*unbounded*
max value	float typein		*unbounded*	*unbounded*

DESCRIPTION

The **minmax** module modifies the minimum and maximum values of a selected vector element (channel) of an n-vector AVS field. The output field is identical to the input field, except for the new vector minimum and maximum values. **minmax** also outputs the minimum and maximum values of a selected vector element in its output ports.

The **minmax** module has two main purposes:

- It can be used to provide min and max inputs to the **generate colormap** module's **hi value** and **lo value** parameters. These in turn will output a scaled colormap to the **color legend** module.
- It can be used to set the minimum / maximum range for animating a sequence of datasets with different minimum and maximum values (such as a time-series). In this application, setting a wide enough range will prevent such modules as **isosurface** and **field legend** from resetting their parameters every time a new dataset is read.

INPUTS

Input (field; required)

The input structure is any valid AVS field.

PARAMETERS

channel An integer dial that selects which channel of an n-vector field's min / max is being edited. For a scalar field, this dial is made invisible. For an n-vector dataset, the maximum value of the dial is set to be the vector length of the field -1. The default is 0.

min value A floating-point typein that specifies a new minimum value for the selected channel of the field. By default it is set to the minimum value of the first dataset read in. If a new field of the same type is read the parameter value is not updated. If a field of a different type (data type, vector length, dimensions, etc.) is read, then the module asks to be thrown away and reinstantiated.

max value A floating-point typein that specifies a new maximum value for the selected channel of the field. By default it is set to the maximum value of the first dataset read in. If a new field of the same type is read the parameter value is not updated. If a field of a different type (data type, vector length, dimensions, etc.) is read, then the module asks to be thrown away and reinstantiated.

OUTPUTS

Output (field)

The output field is exactly the same as the input field, except that the channel's minimum and maximum data values may be reset to the parameter **minimum** and **maximum** values.

EXAMPLE

The following network reads in a field and sets min / max values for a channel, which are used by **generate colormap** and **contour** modules. **generate colormap**'s **lo value** and **hi value** parameter ports must be made visible before they can be connected to **minmax**. To do this, bring up **generate colormap**'s Module Editor, click on the **lo value** parameter button, and then click on **Port Visible** on the resultant Parameter Editor panel. Repeat for **hi value**.

```
                      READ FIELD
                          |
                       MINMAX
                       | |  |
             |------------| |  |
             |      |-------|  |
             |      |          |
        GENERATE COLORMAP      |
             |                 |
             |     |-----------|-------|
             |     |                   |
           FIELD LEGEND           ISOSURFACE
                |                      |
                |                      |
                |----------------------|
                          |
                   GEOMETRY VIEWER
```

RELATED MODULES

ucd minmax

Modules that could provide the **Input field** input:

read field
read volume
Any module that outputs a field.

Modules that can process **minmax**'s output:

generate colormap, field legend, isosurface, etc.

SEE ALSO

The example script MINMAX demonstrates the **minmax** module.

NAME

mirror – reverse array indices in a 2D or 3D data set

SUMMARY

Name	mirror
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field 2D / 3D *n-vector any-data any-coordinates*
Outputs	field of same type as input

Parameters

Name	*Type*	*Default*	*Choices*
axis	choice	Original	Original, X, Y, Z

DESCRIPTION

The **mirror** module reverses the array indexes along one dimension of a 2D or 3D field. This has the effect of creating a mirror image of the data set.

For uniform fields, the data is mirrored "in place" in the data array. In a 50 x 100 uniform field, applying **mirror** to the X dimension does the following (in FORTRAN array notation):

```
INPUT(1,i) ---> OUTPUT(50,i)      (for all 100 values of i)
INPUT(2,i) ---> OUTPUT(49,i)
INPUT(3,i) ---> OUTPUT(48,i)
INPUT(4,i) ---> OUTPUT(47,i)
  ...
INPUT(50,i) ---> OUTPUT(1,i)
```

For rectilinear and irregular data, the coordinate data points array is mirrored about the selected axis. The data in the data array is unchanged.

mirror can be used to change the orientation of the data for display and / or processing purposes.

To perform a reversal in two or more dimensions, use two or more **mirror** modules in succession.

INPUTS

Data Field (field 2D / 3D *n-vector any-data any-coordinates*)
The input may be any 2D / 3D AVS *field*.

PARAMETERS

axis The choices for exchanging the data are:

Original Copies the input to the output; no transformation is performed.

X For uniform fields, reverses the array indices in the X dimension (first dimension). For rectilinear and irregular fields, the coordinate points array is mirrored about the X axis.

Y For uniform fields, reverses the array indices in the Y dimension (second dimension). For rectilinear and irregular fields, the coordinate points array is mirrored about the Y axis.

Z For uniform fields, reverses the array indices in the Z dimension (third dimension). (Equivalent to **Original** for a 2D field.) For rectilinear and irregular fields, the coordinate points array is mirrored about the Z axis.

OUTPUTS

Data Field The output field as the same form as the input field.

RELATED MODULES

This module combined with **transpose** can re-orient the data in any desired way.

ip reflect
ip rotate
ip translate

SEE ALSO

The example script GRAPH VIEWER demonstrates the **mirror** module.

NAME

Module Generator – interactively generate skeletal module source code

SUMMARY

Name	Module Generator
Type	data output
Inputs	*none*
Outputs	*none*
Parameters	*various, internal use*

DESCRIPTION

The **Module Generator** is an interface that a programmer can use to interactively generate skeletal AVS module source code in C or FORTRAN for both subroutine and coroutine modules. The **Module Generator** will also create makefiles and module man page documentation templates, compile modules, and assist the programmer with debugging. To use the **Module Generator**, simply drag its module icon into the Network Editor Workspace. It is not connected to other modules.

When creating output files or reading input files with the **Module Generator**, first specify a filename using the file browser widget controls, then press the appropriate **Write** or **Read** button.

AVS modules have a basic structure:

```
global defines
module description routine
compute routine
AVSinit_modules initialization routine
utility routines
```

Coroutine modules have a main() routine before the specifcation routine, in lieu of a compute routine.

The **Module Generator**'s control panel allows the programmer to specify the module's name, input/output ports, and parameters, parameter widgets, and parameter ranges and defaults. From this information it automatically generates:

- The correct include files for the module.
- A reserved area for user-supplied global defines.
- A module description routine with all of the AVS *libflow.a* library routines to create the input and output ports and parameters.
- A reserved area for user-supplied additions to the module description/specification routine.
- A module compute function definition with input, output, and parameters correctly declared.
- Optionally, an area of code that provides "hints" as to how memory should be allocated and deallocated for the output data.
- A reserved area for user-supplied code that will make up the body of the compute routine.
- A correct module initialization routine. This routine is called by the AVS flow executive when a module is moved from the Network Editor Palette into the Workspace. It "activates" the module's description information and informs the flow executive of the module's compute routine's name so that the flow executive can call it when its turn in to process data flowing through the network

comes.

- A reserved area for user-supplied subroutines, functions, and utility routines.

The programmer can generate a makefile for this code, edit it skeletal source code using their choice of local text editors, compile it, debug it, and create true *troff* or ASCII pseudo-man pages, all from within the AVS environment.

SEE ALSO

The **Module Generator** is described in detail in the "Module Generator" section of the *AVS Applications* document.

LIMITATIONS

More detailed "hints" are provided for C routines than FORTRAN routines.

NAME

offset – deform, or "blow up" a geometry object based on vector values at each node

SUMMARY

Name	offset				
Type	filter				
Inputs	geometry				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	offset	float	0.0	none	none

DESCRIPTION

The **offset** module transforms an AVS *geometry*, so that each vertex of each polygon is translated along its vertex normal. It is useful for emphasizing surface discontinuities (e.g. cusps) and for producing "blow ups" of objects.

INPUTS

Geometry (required; geometry)
An AVS geometry, created with the *libgeom* library or by another AVS module.

PARAMETERS

offset The amount by which each vertex is translated along its normal. Positive values create a "blow-up" of the geometry. Negative values collapse it.

OUTPUTS

Geometry A geometry that represents that same object(s) as the input data.

EXAMPLE

```
   READ GEOM
       |
     OFFSET
       |
GEOMETRY VIEWER
```

RELATED MODULES

read geom, flip normal, tube, geometry viewer, render geometry

LIMITATIONS

This module works only for polytriangle strips and meshes, not for polyhedra. It has no effect on objects that do not have surface normals.

SEE ALSO

The example script OFFSET demonstrates the **offset** module.

oneshot

NAME

oneshot - send a oneshot value to one or more module(s) "oneshot" parameter port(s)

SUMMARY

Name	oneshot				
Availability	Imaging, UCD, Volume, FiniteDiff module libraries				
Type	data				
Inputs	none				
Outputs	oneshot				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	oneshot	oneshot	0	0	unbounded

DESCRIPTION

The **oneshot** module sends a single user-specified "oneshot" value to one or more "oneshot" parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control "oneshot" parameter input to more than one module using only a single "oneshot" input widget.

oneshot outputs an integer which represents the number of times that **oneshot**'s parameter button was clicked in a certain time period. The length of the time period is not user controllable, but depends on the speed with wich AVS executes the network to which **oneshot** is connected. Thus, if AVS were executing a compute intensive network, you could click **oneshot**'s button 10 times. Then, **oneshot** will output the number 10 the next time it executes. Typically, **oneshot** is used as a signal to perform some operation.

Since oneshot data-type is not identical to an integer, **oneshot** can not be used to pass integer parameters.

Before you can connect **oneshot** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **oneshot** module icon in the usual way one connects modules.

PARAMETERS

oneshot (integer)

The single "oneshot" value, specified through a "oneshot" button, to be sent to the receiving module(s) oneshot parameter port(s). The default value is zero.

OUTPUTS

oneshot (integer)

The "oneshot" value is sent to all modules with oneshot-type parameter ports that are connected to the **oneshot** module.

RELATED MODULES

Modules that can process **oneshot**'s output:

all modules with oneshot-type parameter ports

SEE ALSO

The example scripts WRITE VOLUME and WRITE IMAGE demonstrate the **oneshot** module.

orthogonal slicer

NAME

orthogonal slicer – slice through 3D or 2D field with plane perpendicular to coordinate axis

SUMMARY

Name	orthogonal slicer
Availability	Volume, FiniteDiff module libraries
Type	mapper
Inputs	field 3D or 2D *n-vector any-data any-coordinates*
Outputs	field 2D or 1D *n-vector same-data same-coordinates*

Parameters	*Name*	*Type*	*Default*	*Min*	*Max*	*Choices*
	slice plane	int	0	0	255	
	axis	choice	K			I, J, K

DESCRIPTION

The **orthogonal slicer** module takes a 2D slice from a 3D array, or a 1D slice from a 2D array. It does so by holding the array index in one dimension constant, and letting the other index(es) vary. For instance, a data set might include a volume of 5000 points, arranged as follows (using FORTRAN notation):

```
DATA(I,J,K)          I = 1,10
                     J = 1,20
                     K = 1,25
```

You can take a 2D "I-slice" from this data set by setting *I*=4 and letting the other indices vary:

```
DATA(4,J,K)          J = 1,20
                     K = 1,25
```

The notation used in the example above assumes that the field's data values are scalars (in FORTRAN, DATA(4,5,6) must be a scalar). If fact, however, the **orthogonal slicer** module can takes slices of vector-valued fields, also. It passes through whatever data type is presented to it; e.g. if the input is a "field 3D 3-vector float", the output is a "field 2D 3-vector float".

INPUTS

Data Field (field 2D / 3D *n-vector any-data any-coordinates*)
The input may be any 3D or 2D *field.*

PARAMETERS

slice plane Determines the value of the array index to be held constant. This value is reset to zero each time a new data field is input.

axis Selects the dimension (I, J, or K) in which the array index is to be held constant.

OUTPUTS

Data Field (field 1D / 2D *n-vector any-data any-coordinates*)
The output field is 2D instead of 3D (or 1D instead of 2D), and has the same type of data as the input field.

Appropriate new values for **min_ext** and **max_ext** are written to the output field.

EXAMPLE 1

The following network takes a slice from a scalar volume and displays it:

```
            READ VOLUME
                |
          ORTHOGONAL SLICER
                |            GENERATE COLORMAP (optional)
                |   |-----------------|
                |   |
             COLORIZER
                |
           DISPLAY IMAGE
```

The **colorizer** module is necessary because the output of **orthogonal slicer** is a "field 2D scalar byte", which must be cast into an AVS *image* field for display.

EXAMPLE 2

For reasonably small volumes, a better way to construct this network is:

```
       READ VOLUME
           |          GENERATE COLORMAP (optional)
           |   |----------------|
           |   |
        COLORIZER
           |
     ORTHOGONAL SLICER
           |
      DISPLAY IMAGE
```

This network has the effect of colorizing the entire volume once, which make the slicing operation more efficient. It does this at the expense of allocating more memory up front.

EXAMPLE 3

Irregular Data: orthogonal slicer supports the passing of "points" data for *rectilinear* and *irregular* data. This is an important module for visualizing curved data sets. For example:

```
                              READ FIELD (irregular data)
                        ____________|____________
                        |                       |
                ORTHOGONAL SLICER               |
GENERATE COLORMAP       |                       |
        |_________      |               VOLUME BOUNDS
                 |      |                       |
               FIELD TO MESH                    |
                    |___________        ________|
                               |       |
                             GEOMETRY VIEWER
```

(This is the reason for labeling the axis control with "I, J, and K": frequently, the data is *not* aligned to the X, Y, and Z axes. **orthogonal slicer** takes slices through the logical data set, not the physical one.)

EXAMPLE 4

The following network shows how to use **orthogonal slicer** to plot the values of one scan-line of an image:

orthogonal slicer

```
   READ IMAGE
        |
        |
  EXTRACT SCALAR
        |
        |
ORTHOGONAL SLICER (set to middle of image)
        |
        |
  GRAPH VIEWER
```

RELATED MODULES

field to mesh
colorizer

SEE ALSO

The example scripts ANIMATED INTEGER, COLOR RANGE, and VECTOR CURL demonstrate the **orthogonal slicer** module.

NAME

output postscript – convert pixmap to PostScript™ and store in file

SUMMARY

Name	output postscript			
Availability	this module is in the unsupported library			
Type	data output			
Inputs	pixmap (required; pixmap)			
Outputs	*none*			
Parameters	*Name*	*Type*	*Default*	*Choices*
	filename	typein		
	mode	choice	laserwriter	laserwriter, color, mathematica
	Mathematica Options:			
	monochrome	toggle	off	
	8 bit	toggle	off	
	compress	toggle	off	
	dither	toggle	off	

DESCRIPTION

Note: output postscript is similar to **image to postscript**. The main difference is that **output postscript** takes an input pixmap from the **render geometry** module, which may have been dithered down to 8-bits on pseudocolor systems, while **image to postscript** takes an input image from various modules including the **geometry viewer** and **graph viewer** modules. **image to postscript**'s image will be in 24-bit true color even on pseudocolor systems if the **geometry viewer**'s software renderer option is in effect. Thus, **output postscript** (along with **render geometry**) is obsolete. It is retained in the unsupported module library for backward compatibility only.

The **output postscript** module converts its input pixmap to the PostScript™ page description language and stores it in a file.

On most platforms, the window that you are dumping should be wholly on the screen and unobscured by other windows. On some platforms, the window containing the picture to be output is mapped before the picture is saved.

After the file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

Three types of PostScript output are supported:

- An 8-bit gray scale image suitable for sending to a gray-scale PostScript-compatible laser printer such as a **laserwriter**.
- A 24-bit true color RGB **color** image suitable for sending to a PostScript-compatible laser printer that supports the Level 1 PostScript **colorimage** operator color extensions, or any PostScript Level 2 color printer. The actual format is 3-component (RGB) with 8 bits per component, in *multi* format, with a line of red values, then green values, then blue values for each scan line.
- Mathematica™ compatible. Mathematica PostScript-format files are usually readable only by Mathematica and its utilities.

All files are formatted as left-to-right, top-to-bottom scan lines.

The PostScript files are not "encapsulated;" that is, they are formatted as PostScript "main" routines that can be sent directly to the printer. To include the files in other PostScript files (e.g., documents) they should be run through a PostScript encapsulation program that will convert them into a PostScript subroutine.

The files are scaled and translated to produce a centered, page-filling image. This can be altered by manually editing the file, or by using parameters usually provided by the encapsulation program.

INPUTS

pixmap (pixmap)
Any AVS pixmap.

PARAMETERS

filename A typein that allows you to specify the name of the PostScript file to be created. After the file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

Mode Selects the type of PostScript output: **laserwriter**, **color**, or **mathematica**.

The following toggle parameters control the creation of Mathematica PostScript files *only*:

monochrome
If ON, produces monochrome output. **If OFF**, produces color output.

8 bit **If ON**, produces 8-bit output. **If OFF**, produces 4-bit output.

compressed
If ON, produces compressed output. **If OFF**, produces uncompressed output.

dither
If ON, produces dithered output. **If OFF**, produces undithered output.

EXAMPLE

This example converts a display in the **render geometry** module into a PostScript file:

```
 any network
      |
RENDER GEOMETRY
      |-------------|
      |             |
DISPLAY PIXMAP   OUTPUT POSTSCRIPT
```

RELATED MODULES

image to postscript
render geometry

LIMITATIONS

The Mathematica compress option is not supported in any released version of Mathematica.

The dither option produces visual artifacts on some images.

COPYRIGHT

Mathematica is a copyright of Wolfram Research.

SEE ALSO

The example script OUTPUT POSTSCRIPT demonstrates the **output postscript** module.

NAME

particle advector – release grid of particles into velocity field

SUMMARY

Name	particle advector
Availability	FiniteDiff module library
Type	mapper
Inputs	field 3D 3-vector float *any-coordinates* field irregular 3-space *(optional, from samplers module)* upstream transform *(optional, invisible, autoconnect)* integer *(optional, invisible)*
Outputs	particles geometry tracers geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Mesh Res	integer dial	5	2	100
Tracer Length	integer dial	0	0	100
Time Step	float dial	0.2	*unbounded*	*unbounded*
Size	float dial	0.0	*unbounded*	*unbounded*
Advect Batch	oneshot			
Stop Advection	toggle	off		
Replay Advect	toggle	off		
Reset Particles	oneshot			
Show Bounds	toggle	on		
Color	toggle	off		
Surface	toggle	off		
Method	radio	Euler		
Tracer Style	choice	cap		

DESCRIPTION

The **particle advector** module takes as input a 3D 3-vector field of *float*s (e.g. fluid flow simulation data), and treats it as a velocity field. A batch of zero mass (the "sample") particles is *advected* (placed into the field at various initial positions with no initial direction or speed). The particles move through the velocity field according to the magnitude and direction of the vectors at the nodes in the volume. A forward differencing method is used to estimate the next position of each particle as a function of the current position and velocity.

This module is an AVS *coroutine* — it generates new data continuously, rather than waiting for a module upstream to pass it new data.

The starting position of the sample of particles is user controlled. If **particle advector**'s **Show Bounds** parameter is turned on, and **particle advector** is not connected to the **samplers** module (see description of **Upstream Transform** input, below), the sample object, from which particles are advected, is visible. This object can be manipulated like any other geometry object. To select it, click on it with the left mouse button, or enter the Geometry Viewer and make it the current object.

particle advector can receive input from the **samplers** module. **samplers** outputs a list of points in space, and these points become the starting location for advecting particles. When **particle advector** receives input from the **samplers** module, the **Mesh Res** dial, and the **Show Bounds** and **Surface** buttons disappear from the control panel. If **particle advector** does not receive input from the **samplers** module, particles can only be advected from a plane sample; the point, circle, and space options are not available.

Note that, using the **Stop Advection** button, it is possible to advect a batch of particles, stop their progress, reposition the sample plane, and then advect another batch with new parameter settings from a different location. Turn **Stop Advection** off to set both groups of particles in motion.

On systems without hardware sphere rendering, you can represent the polyhedrons that render more quickly using the spheres **Subdivision** slider on the Geometry Viewer's **Objects** submenu.

INPUTS

Data Field (required; field 3D 3-vector float *any-coordinates*)

The input data must be a 3D field, representing a volume of points. The data value for each point must be a 3D vector of *float*s. The input field can be uniform, rectilinear, or irregular.

Sample Input (optional; field irregular, from **samplers** module)

This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **particle advector** uses these locations as the starting positions for advecting particles. If **particle advector** does not receive input from the **samplers** module, particles can only be advected from a plane sample; the point, circle, and space options are not available.

Upstream Transform (optional, invisible, autoconnect)

When the **particle advector** and **geometry viewer** modules coexist in a network, they communicate through a normally-invisible data port. "particle.advect" shows up as an object in the Geometry Viewer. When you select the particle.advect object and move it, **geometry viewer** informs the **particle advector** module what the sample's new location is, and the **particle advector** module recalculates the location and data it is displaying accordingly. This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **particle advector** module's sample of locations. Note that, when **particle advector** receives sample input from the **samplers** module, the bounds of the "particle.advect" object are not visible, and **particle advector**'s **Show Bounds** parameter is disabled.

Synchronize (optional, invisible)

The **particle advector** is an asynchronous coroutine module. There may be some instances when you will want to synchronize the module to the rest of your network. When this input port is connected to another module's output port, the **particle advector** module will only fire when the input port changes value. By disconnecting the input port, the module will go back to asynchronous computation.

PARAMETERS

Various aspects of the particle advection process can be adjusted interactively.

Mesh Res The number of particles is controlled by the **mesh res** parameter. The total number in each batch is **mesh_res** * **mesh_res**.

Tracer Length

Integer dial which controls the length of the tracer output which shows the trajectory of each advected particle. The default is 0; higher numbers produce longer tracers.

Time Step Adjusts a scalar that multiplies the magnitude of the vector along which each particle is travelling. This causes successive positions of particles to be more widely spaced. (See also the **Color** parameter.)

size Controls the radius of the particles, which are rendered as spheres.

The default *size* is zero; this causes the particles to be rendered as points (individual pixels).

Advect batch
Triggers the release of a batch of particles.

Stop Advection
Temporarily halts this module.

Replay Advection
Restarts the advection using the current settings of all parameters.

Reset Particles
Sets the total number of particles to zero.

Show Bounds
(toggle) Controls the visibility of the mesh of particles.

Color (toggle) If **ON**, colors the line segments to indicate how fast the particles are travelling through the velocity field:

red	fastest
yellow	
green	
cyan	
blue	stopped

Surface Creates a solid shaded mesh. The coloring scheme is the same as that used with the **Color** parameter.

method (radio buttons) The buttons **Euler** and **Runge-Kutta** select the method used to calculate the next position of a sample particle. The **Euler** method is faster, involving a single vector in the input field. The **Runge-Kutta** method involves an interpolation, and produces considerably more accurate results.

Tracer Style
(radio buttons) Specifies the form of the **tracers** output:

cap Short lines that show the beginning trajectory of each advected particle. The particles eventually "break free" of these lines, after which the particles continue to move, but the lines do not.

cycle Short lines that show the last few interations of the flow. These lines appear to be "tails" attached to the advected particles.

end Continuous lines that show the entire trajectories of the particles.

OUTPUTS

Particles (geometry)
This output is an AVS *geometry* that represents the batch of particles advected into the input vector field.

Tracers (geometry)
This output is a set of tracer lines (analogous to stream lines) produced by the sample particles. The **tracer style** parameter controls the form that these lines take.

particle advector

EXAMPLE

In the following network, **read field** reads in a 3D scalar field, and **compute gradient** calculates a 3-vector for every field location.

```
READ FIELD
        |
        +------------------+
        |                  |
COMPUTE GRADIENT    VOLUME BOUNDS
        |                  |
        |                  |
PARTICLE ADVECTOR          |
        |                  |
        |   +--------------+
        |---|
            |
GEOMETRY VIEWER
```

RELATED MODULES

Vector operations:

vector curl, vector div, vector grad, vector mag, vector norm

Additional geometries:

volume bounds
arbitrary slice
isosurface

Geometric rendering:

geometry viewer
render manager
render geometry
display pixmap

SEE ALSO

The example script PARTICLE ADVECTOR demonstrates the **particle advector** module.

NAME

pdb to geom – create molecule geometry from Protein Data Bank(PDB) file

SUMMARY

Name	pdb to geom		
Availability	this module is in the unsupported library		
Type	data		
Inputs	none		
Outputs	geometry		
Parameters	*Name*	*Type*	*Choices*
	Data file	browser	
	Render Mode	choice	ball and stick, ball, stick, colored stick, colored residue

DESCRIPTION

The **pdb to geom** module reads the description of a molecule from a file in the Brookhaven Protein Data Bank (PDB) data format. Typically, such files have a *.pdb* filename suffix. The output is an AVS *geometry* description of the molecule.

PARAMETERS

Data File A file browser allows you to specify the name of the *.pdb* file containing the molecule description.

Mode The type of geometry produced:

ball and stick
Small spheres represent the atoms, and white lines represent the bonds.

ball
Large spheres represent the atoms.

stick
White lines represent the bonds.

colored stick
Colored lines represent the atoms and their bonds.

colored residue
Colored lines represent the atoms and their bonds. The color of the lines represents the type of amino acid that the molecule is in.

OUTPUTS

Molecule (geometry)
An AVS *geometry* description of the molecule.

EXAMPLE

This example shows a simple application of **pdb to geom**:

```
   PDB TO GEOM
        |
GEOMETRY VIEWER
```

RELATED MODULES

geometry viewer, render geometry

LIMITATIONS

If you read in the same *.pdb* file name twice, you will get only one instance of the geometry, not two.

Since the *.pdb* file does not contain any bond information, bonding is determined by the distances between atoms.

The render **Mode** is only applied to the last structure if more than one structure is present.

Readings stops on end-of-file, or "END" line, or any line with just a period "." character.

Atom coordinates are from ATOM and HETATM records only.

No futher processing is applied to the atom coordinates. I.e., it is assumed: 1) that the structure contains only one segment; and 2) that all non-protein atoms (solvent, inhibitors) and non-realistic atoms (disorder atoms) are protein atoms.

SEE ALSO

The example script PDB TO GEOM demonstrates the **pdb to geom** module.

NAME

pixmap to image – transform AVS pixmap to AVS image

SUMMARY

Name	pixmap to image
Availability	this module is in the unsupported library
Type	mapper
Inputs	pixmap
Outputs	image (field 2D 4-vector byte)
Parameters	none

DESCRIPTION

Note: The **geometry viewer** module superceded **render geometry** in AVS 4. **geometry viewer** outputs an AVS image directly. There is thus little need for this older **pixmap to image** module. It is retained in the unsupported module library for backward compatibility only.

The **pixmap to image** module takes an AVS pixmap as input and outputs an AVS image ("field 2D 4-vector byte"). The pixmap is an X Window System resource used to store image data in the X server. This reduces the amount of data AVS must pass between modules: a pixmap id and window id.

The 4-vector byte representation for the image consists of pixels that look like this:

auxiliary	red	green	blue

this field interpreted as pixel's opacity value

these three fields make up pixel's color value

The high-order byte field (auxiliary) is generally unused, but sometimes contains alpha (opacity) information on a per-pixel basis.

The pixmap must be entirely on screen and unobscured by other windows or the results of the conversion will be unpredictable.

INPUTS

pixmap (required; pixmap)

The input is any AVS *pixmap*.

OUTPUTS

image (field 2D 4-vector byte)

The output data is a 2D block of pixels. The data set at each point of the 2D field will be a 4-vector of bytes in the AVS *image* format.

EXAMPLE

This module is useful for converting the output of data output modules (e.g. **render geometry**) into images for writing to a file.

pixmap to image

```
  READ GEOM
       |
       |
RENDER GEOMETRY
       |
       |
PIXMAP TO IMAGE
       |
       +_________________+
       |                 |
DISPLAY IMAGE        WRITE IMAGE
```

RELATED MODULES

Image processing:

contrast, threshold, histogram stretch, clamp, interpolate,
colorizer, generate colormap

Renderers which generate pixmaps:

render geometry

Display an image:

display image
image viewer

Pixmap manipulation and display:

transform pixmap, display pixmap

LIMITATIONS

The "Refine" function in a **transform pixmap** module that is upstream of a **pixmap to image** module does not work.

NAME

print field – create an ASCII printable / readable version of an AVS field

SUMMARY

Name	print field
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Display Header	switch	on		
Display Data	switch	on		
Max Elements	integer dial	1	1	5000
Output File	typein	*/tmp/pfield...*		
Min X	typein	0	0	1000000
Max X	typein	-1	-1	1000000
Min Y	typein	0	0	4096
Max Y	typein	-1	-1	4096
Min Z	typein	0	0	4096
Max Z	typein	-1	-1	4096
Min W	typein	0	0	1000
Max W	typein	-1	-1	1000

DESCRIPTION

The **print field** module creates a human-readable version of a portion of the contents of an AVS field. The information takes two forms: it is displayed in an **Output Browser** widget on the AVS control panel, and it is written to a online file. **print field** is useful whenever you need to inspect the actual contents of an AVS field. For example, if you are using the **import to field** module, **print field** can show whether you importing the data correctly.

If the **Display Header** toggle is on, **print field** displays just the header information, showing the number of dimensions (Ndim), the size of each dimension (Dims), the number of coordinate dimensions (Nspace), the vector length (Veclen), the data type (real, integer, byte, etc.), the size of each data element in bytes (Size), the coordinate type (uniform, rectilinear, or curvilinear), and the minimum and maximum data extent. If the information is present, it will also display any labels, any units and minimum or maximum data values associated with the field.

If the **Display Data** switch is toggled, **print field** also displays the data contents of the field and its coordinate values. An integer dial regulates how many values (to a maximum of 5000) are shown. A scrollbar lets you scroll vertically through the data elements outside the normal scope of the display widget.

By default, **print field** starts at X, Y, Z values 0, 0, 0 and starts counting up with the X value turning over most quickly. However, you can display any rectangular section of the data by setting the minimum and maximum coordinate values for X, Y, Z, and (if present) W.

Whenever you change any of the parameter settings, **print field** rewrites the **Output File**, as well as changing the display in the **Output Browser** widget.

The window in which **print field** displays its output can be resized, like any other widget, using the AVS Layout Editor. For a detailed description of how to do this, see the section titled "Layout Editor," in the chapter The Network Editor Subsystem of the *AVS User's Guide*.

print field

INPUT

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)
The input AVS field can be 1, 2, 3, or 4 dimensional.

PARAMETERS

Display Header
A toggle switch that controls whether **print field** displays and writes the field's header information (dimensionality, type, etc.) It is on by default.

Display Data
A toggle switch that controls whether **print field** displays and writes the field's data and coordinate information. It is off by default.

Max Elements
An integer dial that controls how many elements of the field are displayed and written to the output file. The default is 1, which displays and writes one value. The maximum for any one display and file write is 5000 elements. You can use the scrollbar at the side of the **Output Browser** widget to see values vertically outside the window. You can look at the file output version of the field if too much data is clipped horizontally by the **Output Browser** widget. or resize the widget using the Layout Editor.

Output File
An ASCII typein for specifying the output file. By default, **print field** writes to a file in the */tmp* directory called *pfield_nnnn*, where nnnn is the process id of the **print field** module. The **Output File** is rewritten whenever any of the other parameters change.

Min X
Max X
Min Y
Max Y
Min Z
Max Z
Min W
Max W Integer typeins that define a rectangular section of the field to display and write to the **Output File**. Whatever values are entered here, **Max Elements** regulates the total number of elements that will be output. **print field** does not check to see that the values entered are within the actual dimensions of the field, or that the number of dimensions match, but it will not exceed the actual dimensions of the field. 1, 2, 3 and 4 dimensional fields are supported. By default, minimum values are set to 0, while the maximum values are -1, causing as much of the field in that dimension to be displayed as **Max Elements** allows.

EXAMPLE 1

The following network converts some data into an AVS field, displays the contents of the new field, and gives the person the option of writing the new AVS field permanently to disk. For details on converting data into AVS field format, see the man page for **read field**.

```
 READ FIELD
     |
     |--------------|
     |              |
PRINT FIELD    WRITE FIELD
```

RELATED MODULES

compare field

LIMITATIONS

print field writes to */tmp* by default. This can cause problems if: (1) there is no */tmp* mounted on your system, (2) the */tmp* directory does not have very much room in it or has inaccessible protections.

SEE ALSO

The example scripts PRINT FIELD, and FIELD MATH demonstrate the **print field** module.

probe

NAME

probe – interactively show numeric data values in a geometry rendered field

SUMMARY

Name	probe
Availability	Volume, FiniteDiff module libraries
Type	mapper
Inputs	field 3D *n-vector any-data any-coordinates* colormap *(optional)* field irregular *(optional, from samplers module)* upstream transform *(optional, invisible, autoconnect)* upstream geometry *(optional, invisible, autoconnect)*
Outputs	geometry upstream transform *(optional, invisible, autoconnect)*

Parameters

Name	*Type*	*Default*
Sampling Style	choice	Point
Probe Type	choice	Cursor
Pick Geometry	boolean	off

DESCRIPTION

Scientific visualization converts numbers into colored pictures. However, after you have a picture, you often want to be able to get back and examine the numbers that are producing it.

The **probe** module displays the numeric data values in a field at a location in space. It works for fields that have been rendered as an AVS geometry. It works for uniform, rectilinear, and irregular coordinates, and for any data type. It works for both scalar and vector fields.

probe works by creating a cursor-like object titled "probe" that coexists in the Geometry Viewer window with the rendered version of the field data. Its initial position is 0,0,0; the origin. You deal with this probe object just like any other object in the Geometry Viewer. As you move the "probe" object through space, it reports its location and the data value at that location.

There are two major ways to use the **probe**:

- With the **Pick Geometry** option *off*, the "probe" object in the Geometry Viewer acts like any other object. To find a data value at a particular location in space, you make "probe" the current object and move it to that location. The movement can be direct manipulation using the usual Geometry Viewer mouse-button commands (e.g., right button moves object left and right); or, if that is too awkward and imprecise, you can use the Geometry Viewer's "Transformation Selection" panel and have the "probe" object jump to any absolute or relative point in space. As the **probe** travels, it continuously reports its location and the data value beneath it.

- With the **Pick Geometry** option *on*, data sampling is more a "point the mouse cursor and click" technique. Select "probe" as the current object in the Geometry Viewer, point at the object surface you want to sample with the *mouse* cursor, then press the left mouse button. The probe object snaps *to the surface* beneath the cursor and reports the data value.

  The Geometry Viewer tells the **probe** module what vertex the mouse cursor was over when the button was pressed, and **probe** reports the original data value at that vertex.

When reporting data values for vector fields, **probe** lists the values of all the vector elements. If the **probe** is being colored with the data values., the color shown is SQRT(vec0**2 + vec1**2 + vec2**2 ...), in other words, the magnitude of the data vector, mapped to the range of the current colormap.

INPUTS

Data Field (required; field 3D *n-vector any-data any-coordinates*)

The input field is 3D, scalar or vector, uniform or rectilinear or irregular, of any data type.

Colormap (optional)

If an AVS colormap is supplied to the center input port, the color of the probe object in the Geometry Viewer will change according to the data value it is pointing at. I.e., if it is pointing at a "low" value with the default colormap from **generate colormap**, the probe object will be blue; it it is pointing at a "high" value, it will be red.

Data Field (optional; field irregular)

This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **probe** will take these locations and display the data values associated with them.

Upstream Transform (optional, invisible, autoconnect)

When the **probe** and **geometry viewer** modules coexist in a network, they communicate through a normally-invisible data port. "Probe" shows up as an object in the Geometry Viewer. When you select the probe object and move it, **geometry viewer** informs the **probe** module what the probe's new location is, and the **probe** module recalculates the location and data it is displaying accordingly. This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **probe** module's "probe" object.

Upstream Geometry (optional, invisible, autoconnect)

Used by the **Pick Geometry**'s "point cursor and click" technique, this normally invisible port is what the **geometry viewer** module uses to inform **probe** of the geometry vertex selected so it can display the data value for it. The module connection occurs automatically.

PARAMETERS

Sampling Style

A pair of radio buttons that specify what sampling technique to use to report the data values.

point means that, if the probe/cursor is pointing *between* actual nodes on the data lattice, it will display the *real* data value for the *nearest* node. This is the faster sampling technique.

Trilinear means that, if the probe/cursor is pointing between actual nodes on the data lattice, it will *calculate* a data value that is a trilinear interpolation of the *eight* nearest real node data values.

Probe Type

A set of radio buttons that control what the "probe" object looks like in the Geometry Viewer.

Cursor creates a probe that looks like a miniature XYZ axis.

Crosshair creates a probe that looks like half of a miniature XYZ axis. The crosshair stays aligned with the axis, and its endpoints lie in the XY, YZ, and XZ planes.

Probe creates a probe that looks like an electronic probe or a dissecting needle.

Pick Geometry

A boolean switch that controls whether one moves the "probe" object like any Geometry Viewer object by selecting it as the current object and translating it with mouse button commands or the Transformation Selections panel (the default, off); or whether one selects data by pointing to an object's verticies with the mouse cursor and pressing the left mouse button.

OUTPUTS

Geometry (geometry)

The output geometry has two parts:

The rendering of the "probe" object, and;

The rendering of the "Text for Probe" that lists the data value and coordinate position.

Upstream Transform (optional, invisible, autoconnect)

If **probe** is connected to the **samplers** module, it uses this port to relay movement information from **render geometry** back up the network to **samplers**.

EXAMPLE 1

The following network inputs a curvilinear scalar field, scales the color values to the actual data range, displays it through **arbitrary slicer**, with a colorized "probe" object, surrounded by volume bounds:

```
                      read field
                          |
generate colormap         |
         |                |
         |     |----------|
         |     |          |
      color range         |----------------|---------------|
             |            |                |               |
             |-----|----- | -------------| |               |
                   |      |              | |               |
                   |      |         |--| | |               |
             arbitrary slicer       |  probe         volume bounds
                       |            |    |                 |
                       |-----------------|-----------------|
                       |            |
                 geometry viewer    |
                              |-----|
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
read field
read plot3d

Modules that could provide the **colormap** input:

generate colormap
color range

Modules that could provide the Sample field input:

samplers

Modules that can process **probe** output:

geometry viewer
render geometry

SEE ALSO

The example script PROBE demonstrates the **probe** module.

read field

NAME

read field – read AVS field from a disk file, or import data files into AVS field format

SUMMARY

Name	read field		
Availability	Imaging, UCD, Volume, FiniteDiff module libraries		
Type	data		
Inputs	none		
Outputs	field *same-dimension same-vector same-data same-coordinates*		
Parameters	*Name*	*Type*	*Default*
	Read File	browser	
	Auto/ Portable(XDR)	choice	Auto

DESCRIPTION

The **read field** module has two input modes:

- In its first input mode, it reads an AVS *field* data structure from a disk file into a network. The format of an AVS *field file* is discussed below in the "Native Field Input" section.
- In its second input mode, it converts data stored in ASCII, Fortran unformatted, or pure binary data files into AVS field format. **read field** can thus be used to import *some* datasets into the AVS system. (The **file descriptor** module also performs this function, but with more flexibility.)

The two input modes—"native field input" and "data-parsing input"—are described separately in the sections below.

PARAMETERS

Read File A file browser window to specify the name of the file to be read.

Auto/Portable(XDR)

A pair of radio buttons that control how **read field** will interpret binary AVS field input files.

Auto

If **Auto** is selected, then **read field** will examine the ASCII header's "data=" line. If the file is described as just "data=integer", or "data=float", then **read field** assumes that the field file's binary data format is compatible with the system on which the **read field** module is executing. If the file is described as "data=xdr_float", "data=xdr_integer", or "data=xdr_double", then **read field** assumes that the binary area of the field file is written in machine-independent XDR (external data representation) format and will translate the binary portion of the field file into the binary format of the system on which the **read field** module is executing.

Portable(XDR)

If this is selected, then **read field** assumes that the binary portion of the field file is written in machine-independent XDR format (no matter what the ASCII header says) and will translate the binary portion of the field file into the binary format of the system on which the **read field** module is executing.

See the "Binary Compatibility on Different Hardware Platforms" section below for more information on this feature.

NATIVE FIELD INPUT

read field can read files in the native AVS field file format into an AVS network. An AVS field file (suffix *.fld*) has the following components:

- An ASCII header that describes the field
- Two separator characters that divide the ASCII header from the data and coordinate information
- A binary area containing the data and coordinate information

The **write field** module creates files in this format.

ASCII Header

The ASCII header contains a series of text lines, each of which is either a comment or a *TOKEN=VALUE* pair. For example, the following header created by the **write field** module defines a field of type "field 2D 4-vector byte", which is the AVS image format:

```
# AVS field file
# creation date: Fri Aug 23 11:23:27 1991
#
ndim=2                          # number of dimensions in the field
dim1=500                        # dimension of axis 1
dim2=480                        # dimension of axis 2
nspace=2                        # number of physical coordinates per point
veclen=4                        # number of components at each point
data=byte                       # portable data format
field=uniform                   # field type (uniform, rectilinear, irregular)
min_ext=0.000000 0.000000                       # coordinate space extent
max_ext=499.000000 479.000000                   # coordinate space extent
label= alpha red green blue
min_val=0 0 0 0                 # minimum data values for each data component
max_val=0 255 255 255           # maximum data values for each data component
```

The first three lines are comments, indicated by the **#** character. Note that the first line of the header *must* begin as follows:

```
# AVS
```

In this example, comments also occur at the end of each line. Any characters following (and including) **#** in a header line are ignored. Comments are not required.

Separator Characters

The ASCII header must be followed by two formfeed characters (i.e. **Ctrl-L**, octal 14, decimal 12, hex 0C), in order to separate it from the binary area. This scheme allows you use the **more**(1) shell command to examine the header. When **more** stops at the formfeeds, press **q** to quit. This avoids the problem of the binary data garbling the screen.

Binary Area

The size (in bytes) of the binary area depends on the field type:

- For **uniform** fields, the binary area contains data values followed by the coordinate values.

  Coordinate information is limited to minimum and maximum extent fullword values for each physical dimension (n-space) of the data. The minimum and maximum extent values in the coordinate binary area are copies of the **min_ext** and **max_ext** values in the field data structure, *except* when the field has been cropped, downsized, or interpolated. Then the field data structure contains the original field's **min_ext** and **max_ext** values, while the coordinate section of the binary area contains the minimum and maximum extent of the subsetted data. Mapper modules can use this additional extent information to properly locate their geometric representation of the subsetted data in world coordinate space. The extents in the coordinate binary area are stored in this order: minimum x, maximum x, minimum y, maximum y, minimum z...etc.

Thus, the size of the binary area is the product of the following numbers:

value of **dim1**	(product of sizes of computational dimensions
value of **dim2**	yields total number of field elements)
...	
value of **dim***x*	
value of **veclen**	*(number of data values per field element)*
size of **data**	*(byte size of primitive data type)*

Plus:

8 ∗ value of **nspace**	*(2 coordinates per dimension, 4 bytes per coordinate)*

In the stream of data values:

- All the data values for a field element are stored together.
- The first array index varies most quickly (*FORTRAN-style*).

- For **rectilinear** fields, the binary area contains both data values and coordinates for each scalar data value or vector of data values. The data values occupy the same amount of space as for a **uniform** field. Each coordinate is a single-precision floating-point number (4 bytes), and there is one coordinate for each array index in each dimension of computational space. Thus, the size of the coordinates area is:

  $(dim1 + dim2 \ldots + dimx) * 4$

  All of the X-coordinates are stored together, at the beginning of the coordinates area. Following these are all the Y-coordinates, and so on.

- For **irregular** fields, the data area contains both data values and coordinates. The data values occupy the same amount of space as for a **uniform** field. Each coordinate is a single-precision floating-point number (4 bytes), and each field element is mapped to a point in *nspace*-dimensional physical space. Thus, the size of the coordinates area is:

  $(dim1 * dim2 \ldots * dimx) * nspace * 4$

  As with **rectilinear field**, all of the X-coordinates are stored together, at the beginning of the coordinates are. Following these are all the Y-coordinates, and so on.

Binary Compatibility on Different Hardware Platforms

Memory addressing on 32-bit systems is usually divided into two major hardware classes:

"**Big-endian**"

32-bit words are divided into 4 8-bit bytes, where the high-order byte is byte 0. Systems with this organization include Sun, Hewlett-Packard, and IBM workstations.

"**Little-endian**"

32-bit words are divided into 4 8-bit bytes, where the low-order byte is byte 0. Systems with this organization include Digital Equipment Corporation workstations.

Binary byte data are compatible between the two kinds of systems. Binary integer, floating point, and double-precision floating point data are *not* compatible between the two kinds of systems. For example, an integer AVS field file written on a Sun workstation would not normally be readable on a DEC workstation.

To make AVS field data interchangeable among platforms, the **write field** module has a **Native/Portable(XDR)** switch. Selecting **Portable(XDR)** will write the binary area of the field in Sun's external data representation (XDR). The field header will show "data=xdr_integer|xdr_float|xdr_double". If **Native** is selected, the field header will contain a comment at the end of the "data=" line stating what platform the field file was created on. **read field** uses its **Auto/Portable(XDR)** switches to either examine the ASCII header for the "data=xdr_" flag, or to force reading the data file as XDR format no matter what the ASCII header says. (Note: XDR format is simply 32-bit "big-endian" integers and IEEE standard format floating point.)

EXAMPLE 1

The following ASCII header describes a volume (3D uniform field) with a single byte of data for each field element. This format might be used to represent CAT scan data.

```
# AVS field file
ndim=3          # number of dimensions in the field
dim1=64         # dimension of axis 1
dim2=64         # dimension of axis 2
dim3=64         # dimension of axis 3
nspace=3        # number of physical coordinates per point
veclen=1        # number of components at each point
data=byte       # data type (byte, integer, float, double)
field=uniform   # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

$$(64 * 64 * 64) * 1 * 1 = 262{,}144 \text{ bytes}$$

The coordinates area occupies $(2 * 4) * 3$ bytes. The total binary area occupies 262,168 bytes.

EXAMPLE 2

The following ASCII header describes a volume (3D uniform field) whose data for each field element is a 3D vector of single-precision values. This format might be used to represent the wind velocity at each point in space. This field file is written in XDR format.

```
# AVS field file
ndim=3          # number of dimensions in the field
dim1=27         # dimension of axis 1
dim2=25         # dimension of axis 2
dim3=32         # dimension of axis 3
nspace=3        # number of physical coordinates per point
veclen=3        # number of components at each point
data=xdr_float      # portable data format
field=uniform   # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

$$(27 * 25 * 32) * 4 * 3 = 259{,}200 \text{ bytes}$$

The coordinates area occupies $(2 * 4) * 3$ bytes. The total binary area occupies 259,224 bytes.

EXAMPLE 3

The following ASCII header describes an irregular volume (3D irregular field) with one single-precision value for each field element. The binary area includes an (X,Y,Z) coordinate triple for each field element, indicating the corresponding point in physical space. This format might be used to represent fluid flow data.

```
# AVS field file
ndim=3            # number of dimensions in the field
dim1=40           # dimension of axis 1
dim2=32           # dimension of axis 2
dim3=32           # dimension of axis 3
nspace=3          # number of physical coordinates per point
veclen=1          # number of components at each point
data=float        # data type (byte, integer, float, double)
field=irregular   # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

$(40 * 32 * 32) * 4 * 1 = 163{,}840$ bytes

The coordinates area occupies this amount of space:

$(40 * 32 * 32) * 4 * 3 = 491{,}520$ bytes

DATA-PARSING INPUT MODE

In its second input mode, **read field** can convert a certain class of data stored in ASCII, Fortran unformatted, or pure binary data files into AVS field format. To import data into AVS, you must create an ASCII description file that defines the structure of the AVS field to make. The first part of this description file is identical in format and meaning to the ASCII header file described above.

The second part of this file contains commands that specify which files contain the data or coordinate information, its data type (ASCII, binary, or Fortran unformatted) and simple parsing instructions. **read field** can read a file that is parseable by this general scheme:

> **skip** *n* lines or bytes
> move over an **offset** of *m* columns on this line (ASCII only)
> read the value
> do until *# of values needed*
> > {
> > take *p* **stride**(s) to the next value
> > read the value
> > }

The ASCII description file, data, and coordinate information for rectilinear and irregular data can all be read from different files. If the resulting AVS field contains a vector of data values at each point, each vector element can also be read from a separate file.

The ASCII description file must have a *.fld* file suffix or the **read field** file browser will not display the file.

read field data parsing capablility is meant to be used only once, in order to convert data to AVS field format. The parsing activity makes **read field** run more slowly than when it reads a file that is already in AVS field format. Once you have read your data using **read field**'s data-parsing mode, you should use the **write field** module to store it permanently on disk in AVS field file format.

Suggestion: While experimenting with **read field**'s ASCII description file, connect its output port to the **print field** module's input port and use **print field**. This allows you to examine the results online, to see whether the data is being interpreted correctly.

read field chronicles its progress in a status display below the file browser widget as it works through the input files to assemble the AVS field.

ASCII Description File

As the example below shows, the ASCII description file contains a series of text lines that define the AVS field to construct. Each line is either:

- A comment
- A required line in the form *token*=*value*
- An optional line in the form *token*=*value*
- A **variable** or **coord** parsing specification

The following ASCII description file imports three dimensional curvilinear data with a vector of values at each point into an AVS field of type "field 3D 3-vector irregular float". This type of data often occurs in computational fluid dynamics applications. The data and coordinate information are in separate files, both of which were written as straight binary data. Both files happen to have a serial organization. In the data file, all of vector element 1's values appear, then all of vector element 2's, then all of vector element 3's values. In the X, Y, Z coordinate file, all the X coordinate values appear, then all the Y's, then all the Z's.

Each line's meaning is explained in detail below.

```
# AVS field file     the string "# AVS" must be the first
#                    five characters in the file
#                    when a '#' character appears in a line,
#                    the rest of the line is a comment
#
ndim=3                     # REQUIRED--the number of dimensions in the field
dim1=40                    # REQUIRED--dimension of axis 1
dim2=32                    # REQUIRED--dimension of axis 2
dim3=32                    # REQUIRED--dimension of axis 3
nspace=3                   # REQUIRED--number of coordinates per point
veclen=3                   # REQUIRED--number of components at each point
data=float                 # REQUIRED--data type (byte,integer,float,double)
field=irregular            # REQUIRED--field type (uniform, rectilinear,irregular)
min_ext=-1.0 -1.0 -1.0     # OPTIONAL--coordinate space extent
max_ext=1.0 1.0 1.0        # OPTIONAL--coordinate space extent
label=x-velocity           # OPTIONAL--component label for variable 1
label=y-velocity           # OPTIONAL--component label for variable 2
label=z-velocity           # OPTIONAL--component label for variable 3
unit=miles-per-second      # OPTIONAL--describes unit of measure for variable 1
unit=miles-per-second      # OPTIONAL--describes unit of measure for variable 2
unit=miles-per-second      # OPTIONAL--describes unit of measure for variable 3
min_val=-2.18 -0.32 -3.73  # OPTIONAL--minimum data values per component
max_val=5.79 3.54 1.50     # OPTIONAL--maximum data values per component
#
# For each coordinate X, Y, and Z, where to find it and how to read it
#
coord 1 file=/usr/userid/data/wing.bin filetype=binary skip=12
coord 2 file=/usr/userid/data/wing.bin filetype=binary skip=163852
coord 3 file=/usr/userid/data/wing.bin filetype=binary skip=327692
#
# For each value in the vector, where to find it and how to read it
#
```

```
variable 1 file=/usr/userid/data/wdata.bin filetype=binary skip=28
variable 2 file=/usr/userid/data/wdata.bin filetype=binary skip=163868
variable 3 file=/usr/userid/data/wdata.bin filetype=binary skip=327708
```

Any characters following (and including) **#** in a header line are ignored.

NOTE: The first five characters in the ASCII description file *must* be "# AVS" or **read field** will not recognize the file as valid.

The example above shows all of the required *TOKEN=VALUE* token names: an ASCII description file that is missing one or more of these lines causes **read field** to generate an error. Required *TOKEN=VALUE* pairs are stored in the AVS field that **read field** produces as output.

Optional *TOKEN=VALUE* pairs are stored in the output AVS field as well, if they are provided. **min_ext** and **max_ext** are stored in the output AVS field even if they are not specified, as **read field** calculates them if they are not provided.

The **variable** and **coord** lines are not stored in the output AVS field. They are only instructions to **read field**.

With the exception of filenames, ASCII description file specifications are *not* case-sensitive.

- You can surround the **=** character with any amount of white space (including none at all). For example, "dim2 = 32", "DIM 2 =32", and "Dim2=32" are all equivalent.
- Value strings *do not* have to be padded out to 11 characters.

ndim = *value* (required)

The number of computational dimensions in the field. For an image, **ndim** = 2. For a volume, **ndim** = 3.

dim1 = *value* (required)
dim2 = *value* (required, depending on total number of dimensions)
dim3 = *value* (required, depending on total number of dimensions)
...

The dimension size of each axis (the array bound for each dimension of the computational array). The number of **dim***x* entries must match the value of **ndim**. For instance, if you specify a 3D field (**ndim**=3), you must specify the length of the X dimension (**dim1**), the length of the Y dimension (**dim2**), and the length of the Z dimension (**dim3**).

Note that counting is 1-based, not 0-based.

nspace = *value* (required)

The dimensionality of the physical space that corresponds to the computational space (number of physical coordinates per field element).

In many cases, the values of **nspace** and **ndim** are the same — the physical and computational spaces have the same dimensionality. But you might embed a 2D computational field in 3D physical space to define a manifold; or you might embed a 1D computational field in 3D physical space to define an arbitrary set of points (a "scatter").

veclen = *value* (required)

The number of data values for each field element. All the data values must be of the same primitive type (e.g. **integer**), so that the collection of values is conceptually a **veclen**-dimensional vector. If **veclen**=1, the single data value is, effectively, a scalar. Thus, the term *scalar field* is often used to describe such a field.

data = byte (one of the four options is required)
data = integer
data = float
data = double

The primitive data type of all the data values. It is possible to specify "data=xdr_integer|xdr_float|xdr_double" in data parsing input mode as well as native field input mode. However, it will only work correctly in the case where the original binary file is in 32-bit big-endian format. The reverse case will not work.

field = uniform (one of the three options is required)
field = rectilinear
field = irregular

The field type. A **uniform** field has no computational-to-physical space mapping. The field implicitly takes its mapping from the organization of the computational array of field elements.

For a **rectilinear** field, each array index in each dimension of the computational space is mapped to a physical coordinate. This produces a physical space whose axes are orthogonal, but the spacing among elements is not necessarily equal.

For an **irregular** field, there is no restriction on the correspondence between computational space and physical space. Each element in the computational space is assigned its own physical coordinates.

min_ext = *x-value [y-value] [z-value]...* (optional)
max_ext = *x-value [y-value] [z-value]...* (optional)

The minimum and maximum coordinate value that any member data point occupies in space, for each axis in the data. If you do not supply this value, **read field** calculates it and stores it in the output AVS field data structure. This value can be used by modules downstream to, for example, size the **volume bounds** drawn around the data in the Geometry Viewer or put minimum and maximum values on coordinate parameter manipulator dials (**probe**). Values can be separated by blanks and/or commas.

If you do not know the extents, don't guess — let **read field** calculate them. Most downstream modules use whatever values are supplied, without checking their validity. If the wrong numbers are specified, incorrect results will be computed.

label = *string1 [string2] [string3]...* (optional)

Allows you to title the individual elements in a vector of values. These labels are stored in the output AVS field data structure. Subsequent modules that work on the individual vector elements (for example, **extract scalar**) will label their parameter widgets with the strings provided here instead of the default "Channel 0, Channel 1...", etc. You can either use one **label** line as shown here, or separate label lines as shown in the example above. In either case, the labels are applied to the elements of the vector in the order encountered. You can also label single scalar values, though downstream modules may ignore such a label. Any alphanumeric string is acceptable. Strings can be separated by blanks and/or commas.

unit = *string1 [string2] [string3]...* (optional)

Allows you to specify a string that describes the unit of measurement for each vector element. You can either use one *unit* line as shown here, or separate unit lines as shown in the example above. In either case, the unit specifications are applied to the elements of the vector in the order encountered. You can also specify the unit for a single scalar value, though downstream modules may ignore it. Any alphanumeric string is acceptable. Strings can be separated by blanks and / or commas.

min_val = *value [value] [value]...* (optional)
max_val = *value [value] [value]...* (optional)

For each data element in a scalar or vector field, allows you to specify the minimum and maximum data values. These values are stored in the output AVS field data structure. This is used by subsequent modules that need to normalize the data. Values can be separated by blanks and / or commas.

read field does not calculate these values if you do not supply them (unlike **min_ext** and **max_ext**). If you do not know these values, don't guess — just leave these optional lines out. In this case, you can use the **write field** module to compute these values when it creates an AVS field file. Most downstream modules use whatever values are supplied, without checking their validity. If the wrong numbers are specified, incorrect results will be computed.

variable *n* **file=***filespec* **filetype=***type* **skip=***n* **offset=***m* **stride=***p*
coord *n* **file=***filespec* **filetype=***type* **skip=***n* **offset=***m* **stride=***p*

variable specifies where to find *data* information, its type, and how to read it.

coord specifies where to find *coordinate* information, its type, and how to read it. It is used when the data is **rectilinear** or **irregular**.

The individual parameters are interpreted as follows:

n — An integer value that specifies which element of a data vector or which coordinate (1 for x, 2 for y, 3 for z, etc.) the subsequent read instructions apply to. **n** does not default to 1 and must be specified.

file = *filespec*

The name of the file containing the data or coordinates. The *filespec* can be an absolute full pathname to a file, or it can be a *filespec* relative to the directory that contains the field ASCII header. For example, an absolute pathname might be */home/myuserid/experiment/data1*. **Note:** the *$AVS_PATH* environment variable is not recognized nor interpreted correctly. You must use a full absolute pathname.

In a relative pathname specification, if the ASCII file of field parsing instructions exists in the file */home/myuserid/experiment/readit.fld* and the data and coordinate files are in the subdirectory */home/myuserid/experiment/data*, you can name these files as *data/xyzs* and *data/values*. The advantage of this second approach is that you can move the directories containing your data around without having to change the contents of the ASCII parsing instruction file.

filetype = ascii
filetype = unformatted
filetype = binary

ascii means that the data or coordinate information is in an ASCII file. In ASCII files, float data can be specified in either real (0.1) or scientific notation (1.00000e-01) format interchangeably.

unformatted means that the data or coordinate information is in a file that was written as Fortran unformatted data. (Fortran unformatted data is binary data with additional words written at the beginning and end of each data block stating the number of bytes or words in the data block.). When you are figuring out the **skip** and **stride** values below, you must count the additional words surrounding any header information that must be **skip**ped over; but ignore the size words when reading the actual data. See the example below.

binary means that the file is written in straight binary format. such as that produced by Unix output routines, write and fwrite.

Note the warning on binary compatibility among different hardware platforms earlier on this man page.

In each case, **read field** will use the data type specified in the earlier **data={byte,float,integer,double}** statement when it interprets the file.

skip = *n*

For **ascii** files, **skip** specifies the number of *lines* to skip over before starting to read the data. Lines are demarked by newline characters.

For **binary** or **unformatted** files, **skip** specifies the number of *bytes* to skip over before starting to read the data.

There are two motivations for **skip**. First, data files often include header information irrelevant to the AVS field data type. Second, if the file contains, for example, all X data values, then all Y data values, **skip** provides a way to space across the irrelevant data to the correct starting point.

skip can only be used once at the start of the file. There is no way to **skip**, read, **stride**, then **skip** again.

You must simply know what value to use for **skip** based on your knowledge of the software that produced the original data file, the number of data elements, and the type (byte, float, double, integer, etc.)

skip defaults to 0.

offset = *m*

offset is only relevant to ASCII files; it is ignored for binary or unformatted files. **offset** specifies the number of columns to space over before starting to read the first datum. (The **stride** specification determines how subsequent data are read.) Hence, to read the fourth column of numbers in an ASCII file, use **offset=3**.

In ASCII files, columns must be separated by one or more blank characters. Commas, semicolons, TAB characters, etc., are *not* recognized as delimiters. If necessary, edit ASCII files to meet this restriction.

offset defaults to 0 (the first column, no columns spaced over).

stride = *p*

stride assumes you are "standing on" the data value just read. **stride** specifies how many "strides" must be taken to get to the next data value. In ASCII files, **stride** means stride forward *p* delimited items. In binary and unformatted files, **stride** means stride forward $p \times$ *the size of the data type* (byte, float, double, integer). In a file where the data or coordinate values are sequential, one after the other, the **stride** would be 1. Note that this presumes homogeneous data in binary and unformatted files — double-precision values could not be intermixed with single precision values.

stride defaults to 1.

The stride value will be repeatedly used until the number of data items indicated by the product of the dimensions (e.g. dim1 × dim2 × dim3) have been read.

Here are some **skip**, **offset**, and **stride** examples for ASCII data. "A's" are vector component 1; "B's" are vector component 2. There are more examples at the end of this manual page.

ASCII file organization 1:

```
X    Y    Z    A    B
1    1    1    A1   B1
2    2    2    A2   B2
3    3    3    A3   B3
4    4    4    A4   B4
5    5    5    A5   B5
```

to read A: skip=1, offset=3, stride=5
to read B: skip=1, offset=4, stride=5

ASCII file organization 2:

```
A1   A2   A3   A4   A5
A6   A7   A8   A9   A10
A11  A12  A13  A14  A15
B1   B2   B3   B4   B5
B6   B7   B8   B9   B10
B11  B12  B13  B14  B15
```

to read A: skip=0, offset=0, stride=1
to read B: skip=3, offset=0, stride=1

ASCII file organization 3:

```
A1     B1     A2     B2     A3     B3
A4     B4     A5     B5     A6     B6
A7     B7     A8     B8     A9     B9
A10    B10    A11    B11    A12    B12
```

to read A: skip=0, offset=0, stride=2
to read B: skip=0, offset=1, stride=2

ASCII file organization 4:

```
TEMP1=A1 TEMP2=A2 TEMP3=A3 TEMP4=A4
TEMP5=A5 TEMP6=A6 TEMP7=A7 TEMP8=A8
PRESS=B1 PRESS=B2 PRESS=B3 PRESS=B4
PRESS=B5 PRESS=B6 PRESS=B7 PRESS=B8
```

read field cannot read this file until
the data labels and equal signs are edited out.

EXAMPLE 4

You have some 3-dimensional, curvilinear data that projects the amount and location of wood that will be eaten after five years by a colony of termites that has entered a 14th century Scandanavian grain silo structure at a particular spot in its base. The data is in one ASCII file, *decay.dat*, as a long sequential, numbered list of 1250 consumed-wood values that looks like this:

```
1,1002.707;
2,1443.971;
3,1307.069;
4,1240.354;
5,1778.715;
    ...
```

The coordinates that correspond to the data values are in a separate ASCII file, *where.coord*, that looks like this:

```
LOC,1,0,0.2500000,0.0000000e+00,1.105255,0.0000000e+00;
LOC,2,0,0.2500000,0.0000000e+00,1.000000,0.0000000e+00;
LOC,3,0,0.5000000,0.0000000e+00,1.552552,0.0000000e+00;
LOC,4,0,0.5000000,0.0000000e+00,1.442042,0.0000000e+00;
LOC,5,0,0.5000000,0.0000000e+00,1.331531,0.0000000e+00;
    ...
```

In the data file, the second column represents the data. In the coordinate file, the fourth through sixth columns are the x, y, and z coordinates, respectively.

First, to read this data, you must use a text editor to globally edit out the commas and semi-colons, changing them to spaces. The files now look like:

```
1 1002.707
2 1443.971
  ...

LOC 1 0 0.2500000 0.0000000e+00 1.105255 0.0000000e+00
LOC 2 0 0.2500000 0.0000000e+00 1.000000 0.0000000e+00
  ...
```

The following ASCII description file, *decay.fld*, would import the data into AVS field format.

```
# AVS Field File
#
# Termite Decay after Five Years
#
  ndim=3          # number of dimensions in the field
  dim1=25         # dimension of axis 1
  dim2 =10        # dimension of axis 2
  dim3 =5         # dimension of axis 3
  nspace=3        # number of physical coordinates
  veclen=1        # number of elements at each point
```

```
  data=float       # data type (byte, integer, float, double)
  field=irregular  # field type (uniform, rectilinear, irregular)
coord 1 file = where.coord filetype=ascii offset = 3 stride = 7
coord 2 file = where.coord filetype=ascii offset = 4 stride = 7
coord 3 file = where.coord filetype=ascii offset = 5 stride = 7
variable 1 file = decay.dat filetype=ascii offset =1 stride = 2
```

In this example, the ASCII description file *decay.fld* is in the same directory as the *where.coord* and *decay.dat* files. If it were in a different directory, you could either give a pathname relative to *decay.fld*'s position, (e.g., *../data/where.coord* or *data/decay.dat*, etc.), or an absolute pathname to the files.

EXAMPLE 5

The following ASCII description file specifies how to convert the volume data in the file *$AVS_PATH/data/volume/hydrogen.dat* into an AVS field. *hydrogen.dat* is a series of binary byte values that represent the probability of finding an electron at various locations around a hydrogen nucleus. The first three bytes in the file give the X, Y, and Z dimensions of the data—however, this information is not part of the actual data and must be skipped over. You could examine these three bytes and determine what to use for the dimensions in the ASCII description file. Thereafter, it is just a matter of reading successive bytes. **offset** is not used because this is not an ASCII file. **stride** is allowed to default to 1. Note that, because the *$AVS_PATH* construct is not recognized, the example uses a full absolute pathname of */usr/avs/...* to find the file.

```
# AVS field file
ndim=3             # number of dimensions in the field
dim1=64            # dimension of axis 1
dim2=64            # dimension of axis 2
dim3=64            # dimension of axis 3
nspace=3           # number of physical coordinates per point
veclen=1           # number of components at each point
data=byte          # data type (byte, integer, float, double)
field=uniform      # field type (uniform, rectilinear, irregular)
variable 1 file=/usr/avs/data/volume/hydrogen.dat filetype=binary skip=3
```

EXAMPLE 6

This ASCII description file specifies how to use **read field** to convert the image data in *$AVS_PATH/data/image/mandrill.x* into an AVS field. The first two words in *mandrill.x* are 32-bit integers that specify the horizontal and vertical dimensions of the image. This information must be skipped over — you must supply it in the ASCII description file. Thereafter, *mandrill.x* is a succssion of 32-bit straight binary words, one word per pixel. However, in AVS, each of these words is considered to be a vector of 4 bytes. The first byte is the "alpha" (or "transparency") value for the pixel, and the second through fourth bytes are the red, green, and blue values for each pixel. Thus, this whole file is treated as a series of binary bytes. Note that, because the *$AVS_PATH* construct is not recognized, the example uses a full absolute pathname of */usr/avs/...* to find the file.

```
# AVS field file
#
ndim = 2                        # number of dimensions in the field
nspace=2                        # number of physical coordinates
dim1=500                        # dimension of axis 1
dim2=480                        # dimension of axis 2
veclen=4                        # number of components at each point
data=byte                       # data type (byte, integer, float, double)
```

```
field=uniform                   # field type (uniform, rectilinear, irregular)
label = alpha, red, green, blue     # labels the vector elements
variable 1 file=/usr/avs/data/image/mandrill.x filetype=binary skip=8 stride=4
variable 2 file=/usr/avs/data/image/mandrill.x filetype=binary skip=9 stride=4
variable 3 file=/usr/avs/data/image/mandrill.x filetype=binary skip=10 stride=4
variable 4 file=/usr/avs/data/image/mandrill.x filetype=binary skip=11 stride=4
```

EXAMPLE 7

This ASCII description file reads a FORTRAN unformatted ARC 3D dataset. The file is 34x34x34, made up of floating point numers. It is irregular, therefore there is both computational and coordinate data, in this case in two separate files. The vector length is six. The data file is written as a 24 byte header that must be skipped over followed by all vector 1 values, all vector 2 values, etc. The coordinate file is written as a 12 byte header (a fullword for each of the X, Y, and Z dimensions) followed by all X coordinates, all Y coordinates, then all Z coordinates. The person is using a relative file specification—the filenames will be interpreted relative to the directory of the ASCII description file.

```
# AVS field file
# to read an Arc 3D FORTRAN unformatted file that's 34x34x34
ndim = 3
dim1 = 34
dim2 = 34
dim3 = 34
nspace = 3
veclen = 6
data = float
field = irregular
#
coord 1 file=for003.dat filetype=unformatted skip=20 stride=1
coord 2 file=for003.dat filetype=unformatted skip=157236 stride=1
coord 3 file=for003.dat filetype=unformatted skip=314452 stride=1
#
variable 1 file=for004.dat filetype=unformatted skip=32 stride=1
variable 2 file=for004.dat filetype=unformatted skip=157248 stride=1
variable 3 file=for004.dat filetype=unformatted skip=314464 stride=1
variable 4 file=for004.dat filetype=unformatted skip=471680 stride=1
variable 5 file=for004.dat filetype=unformatted skip=628896 stride=1
variable 6 file=for004.dat filetype=unformatted skip=786112 stride=1
```

Given that the coordinate file header is 12 bytes, why is the **skip** value 20? It is 20 because **read field** must be directed to skip over the one word FORTRAN unformatted header, and the one word FORTRAN unformatted record trailer (12+4+4=20). The same 20 bytes must be added to the **skip** value for **coords** 2 and 3. Similarly, the data file's 24 byte header must have 8 bytes added to it for a total of 32. **read field** correctly deals with the remaining "invisible" FORTRAN unformatted record header and trailer words in the rest of the file, provided that all values pertaining to a dimension (X, Y, or Z) and/or all values pertaining to a vector (e.g., all x-momentums) were written as one record. It will also work if the records were written as repeating groups (e.g., X, Y, Z; X, Y, Z; etc.). It will not work if the output was generated as "first half of X's; second half of X's", since the intermediate FORTRAN length words will throw of its **stride**s.

RELATED MODULES

The **file descriptor** module can also be used to import data into AVS. It has some additional capabilities such as the ability to read 16-bit halfword data, to read some parsing information (such as the dimensions of the data) directly from the data file itself, and to use variables and expressions for skips, offsets, and strides. The **data dictionary** modules can use the data forms that **file descriptor** constructs to repeatedly read external format data.

The **write field** module will take the AVS field produced by **read field** and write it to disk as a permanent AVS field file. The **read field** module can then read the data much more quickly whenever you need to use it.

The **print field** module displays the ASCII header and contents of an AVS field interactively on the screen. Connect it to **read field**'s output port while experimenting with ASCII description files to verify that the data is being read correctly.

ERROR CHECKING

read field performs a significant amount of error checking. If an error is detected while reading the field, an error dialog box appears on the screen, indicating the line in which the error occurred (if it was in the ASCII header), along with the type of error.

SEE ALSO

The example scripts PRINT FIELD, CONTRAST, FIELD MATH, as well as others demonstrate the **read field** module.

NAME

read geom – reads a data file containing an AVS ´geometry´

SUMMARY

Name	read geom
Availability	FiniteDiff module library
Type	data
Inputs	*none*
Outputs	geometry
Parameters	*Name* *Type* Read Geometry browser

DESCRIPTION

The **read geom** module reads a file containing an AVS *geometry* and outputs the geometry to one or more modules connected to its output port. The resulting object will be named after the file from which it was read. Since AVS replaces geometries based on the object name, if you read in the same filename twice, you will only get one representation of the object.

Since the Geometry Viewer subsystem (also accessible as the **geometry viewer** module) has a built-in **Read Object** function, you rarely need to use this module. It is most useful when used in conjunction with a filter module that processes geometric data (e.g. **shrink**).

PARAMETERS

filename A file browser allows you to specify the name of the file that contains an AVS *geometry*.

OUTPUTS

geometry The output is the *geometry* that was read from the specified file.

EXAMPLE

```
    READ GEOM
        |
     SHRINK
        |
 GEOMETRY VIEWER
```

RELATED MODULES

shrink, offset, geometry viewer, render geometry, wireframe, tube

LIMITATIONS

This module reads GEOM-file files only. It cannot read *.obj* script files or *.scene* scene files that can be created with the Geometry Viewer Script Language (see Appendix B).

The object is always named after the file from which it is read. This makes it awkward to create animation loops, for which you might want to direct multiple files to the same name or to read in multiple instances of the same object.

SEE ALSO

The example scripts CONTRAST, OFFSET, PROBE, as well as others demonstrate the **read geometry** module.

read image

NAME

read image – read image file from disk into a field

SUMMARY

Name	read image				
Availability	Imaging module library				
Type	data				
Inputs	none				
Outputs	field 2D 4-vector byte				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Read Image	Browser	not applicable		

DESCRIPTION

The **read image** module reads an image file from disk and outputs the image as a "field 2D 4-vector byte". Each field element represents a pixel. The data value for each element is a 4D vector of bytes, laid out as follows:

auxiliary	red	green	blue

this field interpreted as pixel's opacity value — these three fields make up pixel's color value

The auxiliary field ("alpha") is sometimes used to store opacity information on a per-pixel basis.

PARAMETERS

Read image

A file browser window that allows you to specify the name of the image file to be read.

OUTPUTS

Data Field The output data is a 2D block of pixels. The data set at each point of the 2D field will be a 4-vector of bytes in the AVS *image* format.

IMAGE FILE FORMAT

read image expects its input file to be in the following format:

```
4-byte integer       nx: number of pixels in X dimension
4-byte integer       ny: number of pixels in Y dimension
nx * ny * 4 bytes    pixel data (4 bytes per pixel)
```

RELATED MODULES

Image processing:

contrast, threshold, histogram stretch, clamp, interpolate
luminance, generate filters, sobel, convolve, local area ops

Decompose / compose images from separate bands:

extract scalar
combine scalars

Display picture:

display image

Turn image data into a pixmap for more powerful viewing techniques:

image to pixmap
transform pixmap
display pixmap

SEE ALSO

The example scripts CONTRAST, FIELD IMAGE, PRINT FIELD, as well as others demonstrate the **read image** module.

read plot3d

NAME

read plot3d – read a PLOT3D format file into an AVS field

SUMMARY

Name	read plot3d
Unsupported	this module is in the unsupported library
Type	data
Inputs	*none*
Outputs	field 1D, 2D, or 3D irregular 3-, 4-, or 5-vector float

Parameters

Name	*Type*	*Default*
X[YZ] Grid File	browser	
Q Solution File	browser	
Multigrid	boolean	false
w/IBLANK	boolean	false
Data Format	choice	binary
Organization	choice	3D/whole
Grid number	integer	1

DESCRIPTION

The **read plot3d** module reads computational fluid dynamics data files in the National Aeronautics and Space Administration's PLOT3D format (see reference) and converts them into AVS field format. There are two types of PLOT3D files, the XYZ grid files that specify the irregular coordinate information, and the Q solution files that contain a vector of values for each point in the grid.

XYZ and Q file pairs can contain a single set of grid/data mappings, or multiple grid/data mappings. The XYZ file can also contain an IBLANK value for each point. The data within the files can be in either binary, or FORTRAN formatted or unformatted format. XYZ grid file and Q solution file formats must match in all respects.

read plot3d requires that you know the format (dimensionality, whole/plane, number of grids, binary/formatted/unformatted, and whether IBLANK values are present) of the PLOT3D files that you are trying to read. It does not check to verify that the values it is given map reasonably to the data.

Q solution files contain three to five floating point values for each point in the grid: X momentum (1D), Y momentum (1D and 2D), Z momentum (1D, 2D, and 3D), density, and stagnation. The four header values (FSMACH, ALPHA, RE and TIME) are ignored.

read plot3d does impose some practical limits to the size of the data: No one dimension can be larger than 1,000,000; the output data can have no more than 1,000,000,000 points in any one grid; and the maximum number of data grids is 50.

read plot3d displays a control panel with a set of radio button switches for specifying the multigrid attribute, the IBLANK attribute, dimensionality and organization, a set for the input file type, and an integer dial for the grid number (this dial is not displayed for single-grid files). You specify the Q solution file and XYZ grid file through two separate file browsers. The file selections are cancelled whenever the selection of data format or organization is changed. In addition, if the module has successfully produced an output field, and subsequently one of the file browsers is used to select a file, the file selection for the other browser is cancelled. These actions prevent the module from attempting to mesh unrelated XYZ and Q files when you change from one data set to another.

PARAMETERS

multigrid A toggle that specifies whether the file has a single grid or multiple grids.

grid number
Which grid, in multi-grid files, to use to produce the AVS field.

w/IBLANK
A toggle that specifies whether or not the XYZ file contains an array of IBLANK values for each point in the grid.

data format
A set of radio buttons to specify how *both* the X[YZ] grid file and Q solution file are organized:

binary
The file is written in binary format, that is, the machine's native representation for integers (for the indices) and single precision floating point (for the points and values).

formatted
The file is written as FORTRAN formatted ASCII output.

unformatted
The file is written as FORTRAN unformatted output, including any framing values used by the machine's native FORTRAN compiler.

Organization
A set of radio buttons to specify the dimensionality and organization of the data for both the X[YZ] grid file and the Q solution file.

1D Input files are each a sequence of 1-dimensional arrays of values.

2D Input files are each a sequence of 2-dimensional arrays of values, stored in natural FORTRAN order.

3D/whole
Input files are each a sequence of 3-dimensional arrays of values, stored in natural FORTRAN order.

3D/planes
Input files are each a sequence of sets of 2-dimensional arrays of values, where each set of arrays corresponds to a single plane from the entire array.

X[YZ] File A file browser widget for specifying the grid file.

Q (solution) File
A file browser widget for specifing the solution file.

OUTPUTS

Data Field (field irregular float 1D, 2D, or 3D of 3-, 4-, or 5-vector)
The AVS field output will match the dimensionality of the original PLOT3D dataset. At each point in the grid will be three to five floating point values: density, X momentum (and Y momentum, and Z momentum, if appropriate), and stagnation, in that order. The output AVS field represents only the one specified grid of multi-grid parameter files. There is no way to pack multiple grids into an AVS field.

EXAMPLE

The following example shows how **cfd values** and **read plot3d** can be used. The **extract scalar** on the right extracts one value from the 12-vector that **cfd values** outputs. **isosurface** computes the isosurface for this scalar output, and **volume bounds** is used to draw a bounding box for the data. The left hand **extract scalar** module extracts another value from **cfd values** output. This second scalar field is used to color the isosurface. The **color range** module is used to scale the colormap to the range of the extracted cfd value. This network will allow you, for example, to generate an isosurface of the density in a field, and then color this isosurface based on the temperature values at each point on the isosurface.

```
                              READ PLOT3D
                                   |
                                   |
                              CFD VALUES
                                   |
GENERATE COLORMAP         |---------|--------|
        |                 |                  |
        |          EXTRACT SCALAR     EXTRACT SCALAR
        |                 |                  |
        |-----|   |-------|----|             |
              |   |            |   |---------|---------|
           COLOR RANGE         |   |                   |
                  |            |   |            VOLUME BOUNDS
                  |---------|  |   |                   |
                            |  |   |                   |
                          ISOSURFACE                   |
                               |                       |
                               |----------------------|
                               |
                        GEOMETRY VIEWER
```

RELATED MODULES

The **cfd values** modules is particularly designed to compute 7 common CFD values such as temperature, pressure, enthalpy, mach number, and energy from the five values provided by this and any other CFD input modules.

Modules that can process **read plot3d** output:

cfd values
extract scalar
extract vector
volume bounds
isosurface
arbitrary slicer

REFERENCES

Pieter Buening, **PLOT3D Reference Manual**.

SEE ALSO

The example scripts READ PLOT3D and CFD VALUES demonstrate the **read plot3D** module.

NAME

read ucd – read UCD structure from a disk file

SUMMARY

Name	read ucd	
Availability	UCD module library	
Type	data	
Inputs	none	
Outputs	ucd structure	
Parameters	*Name*	*Type*
	Read UCD	browser

DESCRIPTION

read ucd reads a UCD structure from a file, which must have a *.inp* suffix. The file may be ASCII or binary. The cell connectivity list is calculated automatically.

Binary UCD files have a different format than ASCII UCD files. Specifically, if a file is binary then it is assumed that it is in the format output by the module **write ucd**.

ASCII UCD files have a simple format described below under "ASCII File Format". For a more detailed description of both ASCII and binary file formats, see the "Unstructured Cell Data" appendix of the *AVS Developer's Guide*.

PARAMETERS

Read UCD A file browser window to specify the name of the UCD file to be read. Files must have a *.inp* suffix or they will not appear in the browser.

OUTPUTS

UCD structure

The output structure is in AVS unstructured cell data format.

ASCII FILE FORMAT

If a UCD file is in ASCII, it has the following format. For a more complete description of UCD file formats, as well as a discussion of UCD data in general, see the "Unstructured Cell Data" appendix of the *AVS Developer's Guide*.

Comments, if present, must precede all data in the file—comments within the data will cause read errors. The general order of the data is:

1. Numbers defining the overall structure, including the number of nodes, the number of cells, and the length of the vector of data associated with the nodes, cells, and the model.
2. For each node, its node-id and the coordinates of that node in space. Node-ids must be integers, but any number including non-sequential numbers can be used. Mid-edge nodes are treated like any other node.
3. For each cell: its cell-id, material, type (hex, prism, pyr, tet, quad, tri, line, pt), and the list of node-ids that correspond to each of the cell's verticies. (The UCD appendix shows the order in which cell verticies are numbered.)
4. For the data vector associated with nodes, how many components that vector is divided into (e.g., a vector of 5 floating point numbers may be treated as 3 components: a scalar, a vector of 3, and another scalar, which would be specified as 3 1 3 1).
5. For each node data component, a component label / unit label pair, separated by a comma.

6. For each node, the vector of data values associated with it.
7. That is the end of the node definitions. Cell-based data descriptions, if present, then follow in the same order and format as items 4, 5, and 6.
8. The single model-based data descriptions, if present, comes last.

The input file cannot contain blank lines or lines with leading blanks. The numbers down the left correspond to the above descriptions and are not part of the ASCII file.

<comment 1>
.
.
.
<comment n>
1. <num_nodes> <num_cells> <num_ndata> <num_cdata> <num_mdata>
2. <node_id 1> <x> <y> <z>
<node_id 2> <x> <y> <z>
.
.
.
<node_id num_nodes> <x> <y> <z>
3. <cell_id 1> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
<cell_id 2> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
.
.
.
<cell_id num_cells> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>

Note: valid strings for <cell-type> are: pt, line, tri, quad, tet, pyr, prism, and hex.

4. <num_comp for node data> <size comp 1> <size comp 2>...<size comp n>
5. <node_comp_label 1> , <units_label 1>
<node_comp_label 2> , <units_label 2>
.
.
.
<node_comp_label num_comp> , <units_label num_comp>
6. <node_id 1> <node_data 1> ... <node_data num_ndata>
<node_id 2> <node_data 1> ... <node_data num_ndata>
.
.
.
<node_id num_nodes> <node_data 1> ... <node_data num_ndata>
7. <num_comp for cell's data> <size comp 1> <size comp 2>...<size comp n>
<cell-component-label 1> , <units-label 1>
<cell-component-label 2> , <units-label 2>
.
.
.
<cell-component-label n> , <units-label n>
<cell-id 1> <cell-data 1> ... <cell-data num_cdata>
<cell-id 2> <cell-data 1> ... <cell-data num_cdata>
.

.
.
<cell-id num_cells> <cell-data 1> <cell-data num_cdata>
8. <num_comp for model's data> <size comp 1> <size comp 2>...<size comp n>
<model-component-label 1> , <units-label 1>
<model-component-label 2> , <units-label 2>
.
.
.
<model-component-label n> , <units-label n>
<model-id> <model-data 1> <model-data num_mdata>

The UCD structure and library will support either integer or character node-, cell-, and model-ids, (referred to in the library documentation as **names**). However, the **read ucd** module only accepts integer node-ids, cell-ids, and model-ids. This is shown in the example below. The ids do not have to be consecutively numbered.

Also note that, at present, most of the UCD modules do not make use of cell and model-based data, thus the input data examples all show "0" for <num-cdata> and <num-mdata>. User-written modules can use the UCD library to manipulate cell- and model-based data.

SAMPLE UCD FILE

The following is an example of a simple UCD file. This UCD structure has 8 nodes in 1 hexahedral cell. Associated with each node is a single scalar data value, making up one component that this person labels "stress," and specifies a "lb/in**2" unit label. There is no cell- or model-based data. See the "Unstructured Cell Data" appendix in the *Developer's Guide* for more examples.

```
8 1 1 0 0                    1.  8 nodes, 1 cell, 1 component of node data
1  0.000   0.000   1.000     2.  for each node, its id and node coordinates
2  1.000   0.000   1.000
3  1.000   1.000   1.000
4  0.000   1.000   1.000
5  0.000   0.000   0.000
6  1.000   0.000   0.000
7  1.000   1.000   0.000
8  0.000   1.000   0.000
1  1 hex 1 2 3 4 5 6 7 8     3.  cell id, material id, cell type, cell vertices
1 1                          4.  num data components, size of each component
stress, lb/in**2             5.  component label, units label
1  4999.9999                 6.  data vector for each node
2  18749.9999
3  37500.0000
4  56250.0000
5  74999.9999
6  93750.0001
7  107500.0003
8  5000.0001
```

EXAMPLE

The following network reads in a UCD ASCII file (.*inp* suffix), and displays it:

```
GENERATE COLORMAP         READ UCD
         |                   |
         |      |------------|
       UCD CONTOUR           |
                |            |
                |--------|   |
                         |   |
                       UCD TO GEOM
                             |
                             |
                             |
                     GEOMETRY VIEWER
```

RELATED MODULES

Modules that can process **read ucd**'s output:

ucd to geom, ucd crop, ucd threshold, ucd extract, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d, ucd legend, ucd probe, ucd streamline, write ucd, ucd tracer.

SEE ALSO

The example script READ UCD demonstrates the **read ucd** module.

NAME

read volume – read volume file from disk into a field

SUMMARY

Name	read volume	
Availability	Volume, FiniteDiff module libraries	
Type	data	
Inputs	none	
Outputs	field 3D scalar byte	
Parameters	*Name*	*Type*
	Read Volume	Browser

DESCRIPTION

The **read volume** module reads a disk file in *volume data* format and outputs the data as a "field 3D scalar byte". It is used to read data files containing scalar-valued volume data (e.g. CAT scan data, NMR data).

PARAMETERS

read volume

A file browser allows you to specify the name of the file that contains the volume data set.

OUTPUTS

Data Field (field 3D scalar byte)

The output is the byte data cast as the scalar data in a 3D *field*.

VOLUME DATA FILE FORMAT

read volume expects its input file to be in the following format:

```
(1 byte)   nx:        number of voxels in X
(1 byte)   ny:        number of voxels in Y
(1 byte)   nz:        number of voxels in Z

(nx * ny * nz bytes): volume data elements
```

EXAMPLE

This simple example displays a volume data set.

```
    READ VOLUME
         |
     COLORIZER
         |
 ORTHOGONAL SLICER
         |
   DISPLAY IMAGE
```

RELATED MODULES

Colormaps:

generate colormap, read colormap

Filters:

clamp, contrast, crop, downsize, field to byte, field to double,
field to float, field to int, histogram stretch, interpolate,
mirror, offset, transpose, colorizer, compute gradient, gradient shade

Mappers:

dot surface, arbitrary slicer, bubbleviz, orthogonal slicer,
field to mesh, isosurface, volume bounds

Renderers:

alpha blend, display image, render geometry

SEE ALSO

The example scripts ANIMATED FLOAT, BRICK, and THRESHOLDED SLICER demonstrate the **read volume** module.

NAME

render geometry – convert geometric description to pixmap (Geometry Viewer)

SUMMARY

Name	render geometry
Availability	this module is in the unsupported library
Type	data output
Inputs	geometry (optional, multiple) field 2D / 3D 4-vector byte (optional, requires 3D texture mapping support)
Outputs	pixmap
Parameters	*Name* add to object transform

DESCRIPTION

Note: the **render geometry** module has been superceded by the **geometry viewer** module. Please read the documentation for the **geometry viewer** module. **render geometry** is retained in the unsupported module library for backward compatibility only.

The **render geometry** module provides access within an AVS network to the complete Geometry Viewer subsystem. Many different modules can supply input geometries. That is, many *geometry*-format outputs can be connected to **render geometry**'s geometry input port. All the objects will be combined into a single scene. Each module providing input to **render geometry** can define attributes and geometries for any number of objects. Each of these modules can also define a hierarchical relationship among its objects.

You can also invoke **render geometry** with no inputs, so that the "scene" is initially empty. Objects can be added to a scene either by upstream modules or by the **Read Object** selection on the **render geometry** control panel. Geometries and descriptions sent by upstream modules can be saved to files using the **Save Object** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Object**.

SPECIAL CONSIDERATIONS

This module is special: instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multi-level application menu of the Geometry Viewer subsystem. Thus, when you add the **geometry viewer** icon to a network, no page is added to the Network Control Panel. There are two ways to access the Geometry Viewer menu:

- Click the small square in **render geometry** icon with the left mouse button.
- Click the **Geometry Viewer** button located at the top of the Network Control Panel. This button is always visible, even when there is no active network.

In some circumstances, it is useful to be able to access both the Geometry Viewer control panel and the Network Control Panel simultaneously. They both occupy the same screen position, along the left edge of the screen. In these cases, use the X Window System window manager to move the one of these menu windows out of the way.

The **geometry viewer**'s control panel also differs from that of other modules in these ways:

- The Network Editor's **Layout Editor** cannot be used to rearrange Geometry Viewer controls.
- If a network includes more than one instance of **render geometry**, AVS does *not* create a separate control panel for each instance. Each **render geometry** sends its output to a different window, but the same Geometry Viewer application menu controls all the windows. The module whose output window is currently highlighted in red is the one being controlled. To switch the focus to another **render geometry** output window, just click in it with any mouse button.

INPUTS

Geometry (optional, multiple; geometry)

The input data can be any AVS *geometry*. More than one geometry can be input to this port. All the geometries will be combined into the same "scene".

Texture (optional; field 2D/3D 4-vector byte uniform)

This input port requires 2D/3D texture mapping support. 2D/3D texture mapping is supported on only a few hardware renderers (see the release note information that accompanies AVS on your platform). The software renderer does support 2D/3D texture mapping.

The optional input provides a way to perform "dynamic texture mapping". The AVS 2D or 3D field of color values input to this port it is available as a dynamic texture. From within the "Edit Texture" submenu under **Objects**, you can bind this texture map to a particular object.

PARAMETERS

add to object transform

This parameter can be attached to the dialbox or the Spaceball, allowing these devices to control object transformations. In such cases, you can still control transformations using the mouse:

Mouse	**Transform**
middle	rotate
right	translate in plane of screen
middle with SHIFT key	scale
right with SHIFT key	translate perpendicular to plane of screen

OUTPUTS

pixmap The output is a pixmap containing a *scene* that includes all the input objects.

EXAMPLE 1

This network creates a tube version of an object:

```
   READ GEOM
       |
   WIREFRAME
       |
     TUBE
       |
RENDER GEOMETRY
       |
 DISPLAY PIXMAP
```

RELATED MODULES

geometry viewer, display pixmap, read geom

SEE ALSO

The *Geometry Viewer* chapter of the *AVS User's Guide*.

render manager

NAME

render manager – share geometries among subnetworks

SUMMARY

Name render manager

Unsupported this module is in the unsupported library

Type data output

Inputs geometry

Outputs none

Parameters

Name	*Type*
Create New Window	one shot
Active Windows	choice

DESCRIPTION

The **render manager** module takes geometries as input, uses the AVS Geometry Viewer to render them, and displays the results in one or more windows. This module is very similar to the **render geometry** module, with these differences:

- **render manager** creates its own pixmap and window on the screen, rather than relying on **display pixmap**. An initial window is created by default.
- **render manager** has a built-in mechanism for creating and selecting output windows. A set of windows is shared among **render manager** modules in separate subnetworks. At any moment, one of them — the *current output window* — is shared by all the **render manager** modules in all subnetworks. This window displays the combined results of all these modules.

It is possible to create a new output window, which automatically becomes the shared current output window. This provides a powerful capability for exploring differences between datasets, or different mappings of the same dataset. See the **Create New Window** parameter below.

This module is used by the AVS Image Viewer and Volume Viewer subsystems.

INPUTS

Geometry (geometry)
Any AVS *geometry*.

PARAMETERS

Create New Window
Click this button to create a new output window, which becomes the current output window. Subsequent geometric input is rendered into this window, until such time as you change the current output window again (perhaps by creating yet another window).

Active Windows
A choice menu that lists all the output windows, showing which one is current. You can also make an output window current by pressing any mouse button in the window itself.

EXAMPLE

Suppose you have built the following two networks:

```
                READ VOLUME           READ VOLUME
                      |                    |
    GENERATE COLORMAP |              VOLUME BOUNDS
            |         |                    |
          ARBITRARY SLICER                 |
                  |                        |
            RENDER MANAGER          RENDER MANAGER
```

When you select a volume dataset (e.g. *hydrogen.dat*) for the **arbitrary slicer** subnetwork, the slice is rendered by the Geometry Viewer, and a window is created to display the picture. If you select the same dataset in the **volume bounds** subnetwork, the bounds are rendered and displayed in the same window.

If you click **Create New Window**, and then select a new dataset was selected in the **arbitrary slicer** subnetwork, it (and it alone) is displayed in the new window. The geometries in the original window do not change.

RELATED MODULES

Same as for **render geometry**.

NOTES

The output window(s) are not destroyed until *all* **render manager** modules are destroyed.

replace alpha

NAME

replace alpha – replace the alpha channel (transparency) in an image

SUMMARY

Name	replace alpha
Availability	Imaging mobule library
Type	filter
Inputs	field 2D uniform 4-vector byte *(image)* field 2D uniform scalar byte *(new alpha)*
Outputs	field 2D uniform 4-vector byte *(image)*
Parameters	none

DESCRIPTION

The **replace alpha** module replaces the alpha (opacity) byte of all the pixels in an image with the byte value from a 2D uniform *scalar* field of the same dimensions. This 2D uniform scalar field is usually produced by passing the image through the **luminance** or **extract scalar** module, then perhaps performing further imaging techniques on the scalar value (e.g. **contrast**), The modified alpha is then rejoined with the original image using **replace alpha**.

INPUTS

Image (required; field 2D uniform 4-vector byte)

The image whose alpha byte will be replaced. This is the right input port on the **replace alpha module.**

Data Field (required; field 2D uniform scalar byte)

The field of byte values, with the same dimensions as the input image, to use as the replacement alpha values. This is the left input port on the **replace alpha** module.

OUTPUTS

Image (field 2D uniform 4-vector byte)

The output image has the same dimension as the input image.

EXAMPLE 1

The following network reads an image, computes its luminance, uses that to create an alpha mask, generates a shaded background, and composites the rendered image over the shaded background image.

```
                          READ IMAGE
                              |
    __________________________|
    |              |          |
BACKGROUND    LUMINANCE       |
    |              |          |
    |          CONTRAST       |
    |              |------|   |
    |                   REPLACE ALPHA
    |                         |
    |----------------------|  |
                          COMPOSITE
                              |
                        DISPLAY IMAGE
```

RELATED MODULES

Modules that could provide the **Image** input:

- contrast
- pixmap to image
- read image
- threshold
- Any module that produces an image as output

Modules that could provide the 2D scalar field:

- luminance
- extract scalar
- Any modules that can output a 2D scalar field

Modules that can process **replace alpha** output:

- composite
- write image
- image to pixmap

See also **background**, **luminance**

SEE ALSO

The two example BACKGROUND scripts demonstrate the **replace alpha** module.

ribbons

NAME

ribbons – generate ribbon representation for streamlines

SUMMARY

Name	ribbons
Availability	FiniteDiff module library
Type	filter
Inputs	geometry (from stream lines module only) field 3D 3-vector 3-space float (optional; from vector curl or similar) field 3D scalar 3-space float (optional; scalar to control colors) colormap (optional; to apply colors to scalar field)
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
width	float dial	0.5	*unbounded*	*unbounded*
length	int dial	128	4	128
texture	float dial	0.0	0.0	1.0
Mode	choice	none		
flip orientation	boolean	off		

DESCRIPTION

The **ribbons** module generates a set of geometric ribbons by taking the polyline output of the **stream lines** module and replacing them with finite width, colored, and textured polytriangle ribbons. The orientation is optionally controlled by a secondary vector field, usually derived from the streamline field by the **vector curl** module. This allows the ribbon orientation to show field vorticity. If an optional scalar field and associated colormap are connected, and the choice button is set to **scalar field**, the ribbon color will reflect the values in the field. The ribbon output can also contain u-v texture coordinate information, so that the ribbons can be overlayed with a meaningful texture image.

The ribbon representation can be animated by moving the stream lines base position, altering the length parameter, or by changing the texture offset dial to make the texture "crawl" along the ribbon.

The access to the vorticity and scalar fields uses tri-linear interpolation. If the fields are irregular, a block table is built within the ribbons module, which may take some time when these fields change.

The texture mode requires several things to be set up. First, select **texture** on the **Mode** control list. Second, connect an image source, such as **read image**, to the second optional field port on the **geometry viewer** module. Next, select an image that will "tile" vertically. The u-v coordinate specifications generated by the ribbons module only shows half of the image at a time. The image is scrolled vertically, across each facet of the ribbon, by using the **texture** offset dial. If the input image has the same picture on both the top and bottom halves, and is tall and narrow in aspect, then animation cycles can be constructed by animating the **texture** dial.

INPUTS

Geometry (required; geometry)

This should receive the geometry output of the **stream lines** module.

Data Field (optional; field 3D 3-vector 3-space float)

This optional port is used to control ribbon surface orientation. The 3D float field is typically generated by the **vector curl** module.

Data Field (optional; field 3D scalar 3-space float)

This scalar field can optionally be connected to map a second field value onto the ribbons using the colormap input to determine local ribbon color. If a field is present, a colormap must also be present. The **vector mag** module, for example, can be used here to map vector magnitude onto the ribbons.

Colormap (optional; colormap)

This optional colormap is used with the scalar field input. If the colormap is connected, a scalar field must also be connected. The lower and upper values in the colormap control the scalar field mapping. Either set these manually with **generate colormap**, or use the **color range** module to set them automatically.

PARAMETERS

Width The width of each ribbon, centered on the stream line. This float dial is unbounded; the default is 0.5.

Length How much of the stream line to show. This matches the **Length** control on **stream lines**, but allows a shorter ribbon to be selected. This can be animated from 4 to the current stream line length to show ribbon growth, without having to re-calculate the stream lines. The default is 128.

Texture Determines the u-v texture offset factor for which part of the image should appear on each ribbon panel. This can be animated to make a "crawl" effect.

mode (radio buttons)

With the default **none**, the ribbon has no color (white). If **color** is selected, a separate color is assiged to each edge, so the number of rotations of a ribbon can be seen. In **checker** mode, every other panel along the ribbon gets a different color. In **texture** mode, color is deferred to the renderer, so that a texture image can be used. In **scalar field** mode, the ribbon color is by data sampled in the input scalar field.

flip orientation

This choice button determines if the ribbon orientation is controlled by the input field vorticity vector, or a cross product of this and the velocity vector. It has the visual effect of flipping the ribbon 90 degrees.

OUTPUTS

Ribbons (geometry)

A set of polytriangles with colors, normals, and u-v coordinates.

EXAMPLE

The following network reads in a 3D vector field and calculates streamlines for the field. **ribbons** generates ribbon representations and **volume bounds** shows the field extent. Set the stream line object to **Hide** in the **geometry viewer**, leaving it selected, so that the base positions can be moved.

ribbons

```
         READ FIELD
             |
             |-------------------|
             |                   |
        STREAM LINES             |
             |              VOLUME BOUNDS
|------------|                   |
|            |                   |
|         RIBBONS                |
|            |                   |
|------------|                   |
             |-------------------|
             |
             |
  GEOMETRY VIEWER
```

RELATED MODULES

animated float
hedgehog
particle advector
stream lines
tube
ucd streamlines
vector curl

SEE ALSO

The example script RIBBONS demonstrates the ribbons module.

NAME

samplers – extract a subset of locations from a 3-vector 3D field

SUMMARY

Name	samplers				
Availability	UCD, FiniteDiff module libraries				
Type	data				
Inputs	field 3D float *any-coordinates* upstream transform (*optional, invisible, autoconnect*)				
Outputs	field 3D irregular *(locations)*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Choice	choice	point		
	N Segment	integer dial	16	2	64

DESCRIPTION

The **samplers** modules extracts a subset of coordinates from a 3D AVS field of floating point data, producing an output field that is "3-space irregular," i.e., it contains a series of coordinates (also called "scattered data") in 3-space (which can correspond to a uniform, rectilinear, or irregular grid) but without any data values associated with them.

samplers's main purpose is to simultaneously control two or three of the **hedgehog/particle advector/stream lines** modules. For example, you can show the streamlines and hedgehog vectors for the same sample set of points together.

samplers can extract a single location coordinate point, a series of points along a line through the 3D field, a series of points along a circle in a 3D field, a series of points on a plane in a 3D field, or a series of points in a volume of a 3D field.

How many points **samplers** extracts (the sample resolution) depends upon the **N Segment** dial setting.

When the output "field of locations" is connected to the *left* input port of the three volume-of-vectors mapping modules (**hedgehog**, **particle advector**, and **stream lines**), these modules will calculate and display only the subset of points in the input field.

If you don't connect **samplers** to the left input port on **hedgehog/particle advector/stream lines**, these modules create their own internal parameters that function identically to the **samplers** module, like the other parameters-as-data modules (**integer**, etc.),

When **samplers** and **geometry viewer** coexist in a network, the two are connected automatically through an invisible "upstream transform" port. "samplers" becomes a selectable object in the Geometry Viewer. If you select and move the "samplers" geometry object, **geometry viewer** informs the **samplers** module of the new location of the sample subset, **samplers** recalculates the "field of locations" and the **hedgehog/particle advector/stream lines** module redraws the data at the new location. The effect is direct mouse-manipulation control over a line, circle, plane, or volume sampling subset.

If you want less than a whole plane or whole volume sample, use the **crop** module on the input to **samplers**, while letting the full field through to **hedgehog/particle advector/stream lines**'s *right* input port. You can then move the subset volume around the whole volume of the field.

INPUTS

Data Field (required; field 3D 3-vector *any-data any-coordinates*)

The input field is a 3D 3-vector of any coordinate type and any data type.

Upstream Transform (optional, invisible, autoconnect)

When the **samplers** module coexists with the **geometry viewer** module in a network, **geometry viewer** feeds information on how the "samplers" object has been moved in the Geometry Viewer back to this input port on the **samplers** module. The information is relayed through the **hedgehog**, **particle advector**, or **stream lines** module. The modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over the **samplers** sample set.

PARAMETERS

point
line
circle
plane
space A set of radio button choices that determines what type of geometric construct the sample locations will be taken from. You can move each of the structures listed below around the volume of data using the Geometry Viewer's transformations.

point causes a single data location to be output, no matter what the **N Segment** parameter value is. This is the default.

line causes **N Segment** sample locations to be taken along a line through the volume.

circle causes **N Segment** sample locations to be taken around a "ring" within the volume space.

plane causes **N*N Segment** sample locations to be taken along a plane slice through the volume space.

space causes **N*N*N segment** sample locations to be taken throughout the whole volume space. The only way to subset the volume is to pass it through the **crop** module before it reaches **samplers**.

N Segment An integer dial that determines how many sample locations to extract from the volume. It is ignored for **point**. The default is 16, the minimum is 2, and the maximum is 64.

OUTPUTS

Data Field (field 3D irregular)

The output field is a 3D lattice of locations from the original input field, with no data values at each node. It is passed down to the **hedgehog**, **particle advector**, or **stream lines** *left* input port, telling them what subset of their complete data to map.

EXAMPLE 1

The following network reads in a 3-vector field, extracts a sample subset, then maps it as both a **hedgehog** and **stream lines** representation, finally displaying it surrounded by volume bounds.

```
                         READ FIELD
                             |
                             |
   |-------------------------|
   |                         |
SAMPLERS                     |
   |                         |
   |                         |
   |-------------------|     |
   |     --------------|-----|-------------------|
   |     |             |     |                   |
   |     |             |     |                   |
  HEDGEHOG           STREAM LINES          VOLUME BOUNDS
     |                     |                     |
     |---------------------|---------------------|
                           |
                     GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read field

Modules that can process **sampler** output:

hedgehog
particle advector
stream lines

Modules that can be used instead of *samplers*:

create geom / generate grid

SEE ALSO

The example script PARTICLE ADVECTOR demonstrates the **sampler** module.

scatter dots

NAME

scatter dots – generate spheres at points in 3D space

SUMMARY

Name	scatter dots					
Availability	Volume, FiniteDiff module libraries					
Type	mapper					
Inputs	field 1D real 3-space irregular (a "scatter" field)					
Outputs	geometry					
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*	*Choices*
	Connect the dots	toggle	off			on, off
	Radius	Real	0.0	0.0	1.0	

DESCRIPTION

The **scatter dots** module generates spheres of various radii at the coordinate locations in a specified field. For a scalar field, each sphere's radius is proportional to the scalar value, and the sphere is always colored white. If the field is a 4-vector float (such as that produced by the **bubbleviz** module), only the first element of the vector determines the sphere's radius. The other three elements are interpreted as red-green-blue color values (normalized to the range 0..1).

Sphere rendering is both compute and memory intensive. Use the **downsize** module to reduce the amount of data to render. Also, use the Geometry Viewer's **Subdivision** slider to render spheres as less demanding polygonal shapes.

INPUTS

Point List (required; field 1D 3D float *irregular*)

The input field must be a list of points in 3D space, with a *float* value specified at each point.

PARAMETERS

Connect the dots (toggle)

- **If OFF**, a sphere is drawn at each point in the field. The radius of the sphere is specified by the field element's scalar data value. (If the field has vector data, the value of the first vector element is used and the other values determine the sphere's color.
- **If ON**, the points are represented as dots, connected with a single polyline (in the order specified by the 1D array). If the input field has 4-vector float data, the last three vector elements are ignored. No spheres are drawn in this case.

Radius (real)

Radius is a floating-point multiplier factor for the sphere radii.

OUTPUTS

Geometry (geometry)

The output is an AVS *geometry*.

EXAMPLE 1

The **scatter dots** module can be used in combination with the **dot surface** module as follows:

```
             READ VOLUME
                |
             DOT SURFACE
                |
            SCATTER DOTS
                |
          GEOMETRY VIEWER
```

EXAMPLE 2

The **scatter dots** module is required to make **bubbleviz** work properly:

```
GENERATE COLORMAP          READ VOLUME
      |                         |
      |---------|   |-----------|
                |   |
                |   |
             BUBBLEVIZ
                 |
                 |
             SCATTER DOTS
                 |
                 |
          GEOMETRY VIEWER
```

RELATED MODULES

scatter to ucd, read geom, tube, wireframe, geometry viewer, render geometry

SEE ALSO

The example scripts BUBBLEVIZ, and DOT SURFACE demonstrate the **scatter dots** module.

scatter to ucd

NAME

scatter to ucd – convert a scatter field to a tetrahedral UCD structure

SUMMARY

Name	scatter to ucd
Availability	UCD module library
Type	filter
Inputs	field 1D irregular 3-space *n-vector any-data*
Outputs	ucd structure
Parameters	none

DESCRIPTION

The **scatter to ucd** module converts a scatter field to a single UCD structure of tetrahedral cells using a Delauney tesselation algorithm. The scatter data points become the nodes of the tetrahedral UCD cells. Each vector element becomes a node data component in the output structure.

AVS, as shipped, contains only a few modules useful for visualizing scatter fields (**bubbleviz/scatter dots**, for example). If you convert scatter data to a UCD structure, you can then use all of the UCD modules to visualize the data.

INPUTS

Data Field (required; field 1D irregular 3-space *n-vector any-data*)

The input is a scattered field of any data type. A scattered field is a 1D array of scalar or vector data values, where each array element has an X, Y, Z location specified for it in space.

OUTPUTS

UCD Structure

The output is a UCD structure composed of tetrahedral cells..

EXAMPLE 1

This is the most basic UCD visualization network. The scatter field is converted to a UCD structure, and then to a colorized geometry.

```
                         READ FIELD
                             |
                             |
                      SCATTER TO UCD
GENERATE COLORMAP            |
          |                  |
          |     |------------|
          |     |            |
         UCD CONTOUR         |
                |            |
                |------|     |
                       |     |
                    UCD TO GEOM
                          |
                          |
                   GEOMETRY VIEWER
```

EXAMPLE 2

The following network reads in a field and converts it to a UCD structure of tetrahedral cells. This structure is then passed to **ucd tracer** to produce a ray traced volume rendering. The module **euler transformation** allows you to rotate the volume to produce views from any angle.

```
                           READ FIELD
                               |
                               |
                         SCATTER TO UCD
EULER TRANSFORMATION           |
                  |--------|   |
GENERATE COLORMAP          |   |
             |---------|   |   |
                       |   |   |
                       UCD TRACER
                             |
                             |
                      DISPLAY IMAGE
```

RELATED MODULES

Modules that could provide the **field** input:

read field
any other module which outputs a field.

Modules that can process **scatter to ucd**'s output:

any module that inputs a UCD field.

SEE ALSO

The example script SCATTER TO UCD demonstrates the **scatter to ucd** module.

set view

NAME

set view – view objects in geometry viewer from fixed orthogonal orientations

SUMMARY

Name	set view
Availability	Imaging, UCD, Volume, FiniteDiff module libraries
Type	data input
Inputs	none
Outputs	none

Parameters

Name	*Type*	*Default*
User	oneshot	
Top	oneshot	
Bottom	oneshot	
Front	oneshot	
Back	oneshot	
Right	oneshot	
Left	oneshot	
Bounds	boolean	off
Persp	boolean	off

DESCRIPTION

The **set view** module provides simplified, push-button control of the user's view of the top-level object in the **geometry viewer** module's output window. It is intended primarily to be used by the AVS Data Viewer. When used in a network by the **Data Viewer** module, it surrounds the **geometry viewer**'s display window with its push button controls. When used without the Data Viewer, it places its controls on the control panel like all other modules.

The **set view** module does not connect to other modules in a network through standard data flow connections. Rather, it performs its functions by sending CLI commands to the **geometry viewer** module through the AVS kernel.

PARAMETERS

User A oneshot control. The first time this is selected, it remembers the current orientation of the top-level object in the view window. Subsequently, it will return the top-level object to this orientation from wherever the user has moved it with the buttons below. The **User** value is cleared when the top-level object is next directly transformed with the mouse.

Top/Bottom
Front/Back
Right/Left A series of oneshot controls that instantly transform the top-level object to a fixed orientation orthogonal to the scene's X, Y, and Z axis. The top-level object is also normalized, if necessary, to fit entirely within the field of view.

Top/Bottom produce views looking directly along the Z axis.

Front/Back produce views looking directly along the Y axis.

Right/Left produce views looking directly along the X axis.

Bounds A switch that turns on Bounding Box mode for efficiently rendering object transformations.

Persp A switch that turns on a perspective view of the scene.

EXAMPLE

The following network reads an AVS field, then maps it as an orthogonal slice in the Geometry Viewer. The **set view** module, though not connected to any other module in the network, can be used to control the view of the object in the Geometry Viewer's display window.

```
                        READ FIELD
                             |
GENERATE COLORMAP            |---------------|
       |                     |               |
       |           ORTHOGONAL SLICER         |
       |                     |               |
       |--------------|      |               |
                   FIELD TO MESH   VOLUME BOUNDS
                             |               |
                             |---------------|
                   GEOMETRY VIEWER
                                                 SET VIEW
```

RELATED MODULES

geometry viewer
data viewer

shrink

NAME

shrink – make polygons of a geometry object smaller

SUMMARY

Name	shrink
Availability	FiniteDiff module library
Type	filter
Inputs	geometry
Outputs	geometry

Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	offset	float	1.0	0.0	1.0

DESCRIPTION

The **shrink** module transforms an AVS *geometry*, so that each vertex of each polygon is translated towards (or away from) the polygon's centroid (center of gravity). This has the effect of creating spaces between polygons, and is useful for visualizing the internal geometry of an object.

INPUTS

Geometry (required; geometry) An AVS geometry, created with the *libgeom* library or by another AVS module.

PARAMETERS

offset The amount by which each vertex is translated. Positive values collapse the geometry inward. Negative values create a "blow-up" of the geometry.

OUTPUTS

Geometry A geometry that represents the same object(s) as the input data.

EXAMPLE

```
     READ GEOM
         |
      SHRINK
         |
  GEOMETRY VIEWER
```

RELATED MODULES

read geom, flip normal, tube, geometry viewer, render geometry

LIMITATIONS

This module works only for polytriangle strips and meshes; it does not work for polyhedra.

This module doesn't copy UV data, used in texture mapping.

This module can increase the size of the data: it can generate up to five times the number of triangles for polytriangle objects, and up to three times the number of vertices for meshes.

SEE ALSO

The example script SHRINK demonstrates the **shrink** module.

NAME

sketch roi – create a region of interest field

SYNOPSIS

Name	sketch roi
Availability	Imaging module library
Type	data input
Inputs	field [2D | 3D] uniform [byte | short | float] *n-vector* image viewer id structure (*invisible, autoconnect*) mouse info structure (*invisible, autoconnect*)
Outputs	field 2D uniform scalar byte (*region of interest*) image draw structure

Parameters

Name	*Type*	*Default*
inside	boolean	on
accumulate	boolean	off
invert	oneshot	
clear region	oneshot	
set pick mode	oneshot	

DESCRIPTION

sketch roi creates the region of interest (ROI) field that modules such as **ip edge**, **ip twarp**, and **ip convolve** use to restrict their operation to a subset of their input image.

Creating a ROI involves an interaction between **sketch roi** and the **image viewer** module. **sketch roi** must be receiving the same image input as the **image viewer** module. **sketch roi**'s left **image draw structure** output must be connected to the **image viewer** module's leftmost **image draw structure** input. **sketch roi**'s right ROI output is connected to the ROI input of the image processing module that wants the ROI. (See "Example" below).

To draw the region of interest in the Image Viewer window:

1. The **sketch roi** module must have control of the left mouse button in the Image Viewer window. When **sketch roi** is first connected and data first passes through it, it should have control of the left mouse button.
2. Press and hold down the left mouse button, moving the cursor over the image to sketch the region of interest. Release the left mouse button when you are done.

If there are multiple images in the Image Viewer window, and / or multiple sketching modules, then some other module or the Image Viewer itself may have control of the left mouse button. To get control back to **sketch roi**,

1. Make the image the current image (use shift-left mouse button or left mouse button).
2. Press **set pick mode** on **sketch roi**'s control panel.

Some points to note:

- **sketch roi** will close an open area by creating a line between the end of the sketched area and its beginning by the shortest distance.
- ROI boundaries are de-composed into line segments, not smooth curves.
- Part of a sketch can be outside of an image's boundaries to create ROIs that include edge areas.

- With **accumulate** on, ROIs can overlap. If thought of as a Venn diagram, the areas are treated as "or's".

INPUTS

Data Field (required; field [2D | 3D] uniform [byte | short | float] *n-vector*)
This input is a 2D or 3D uniform field of type byte, short, or float. It can be any vector length. **sketch roi** uses this input field for only one purpose: to extract the X and Y extent information it needs to create a ROI that is the same size as the image that the image processing module wants masked.

image viewer id structure (required; invisible, autoconnect)
This input port is invisible by default. It connects automatically to the **image viewer** module's **image viewer id structure** output. The two modules communicate the **image viewer** module's scene id on this connection. Normally, you can ignore its existance.

mouse info structure (required; invisible, autoconnect)
This input port is invisible by default. It connects automatically to the **image viewer** module's **mouse info structure** output. The two modules communicate image name, mouse pointer location and button up / down information on this connection. Normally, you can ignore its existance.

PARAMETERS

inside This is a boolean switch. When on, the space "inside" the area is the ROI. When off, the space "outside" the area is the ROI. The default is on.

accumulate
This is a boolean switch. When on, subsequent areas that one draws are added to the ROI. When off, each area that one draws is a new ROI, and the previous area is deleted.

clear region
This is a oneshot. It erases the existing ROI.

invert This is a oneshot. When pressed, the ROI is inverted--the area formerly inside the ROI is now outside, and the area outside the ROI is now the ROI.

set pick mode
A oneshot that sets the **image viewer**'s upstream mouse picking focus to this module.

OUTPUTS

Data Field (field 2D uniform scalar byte)
The left output field is a 2D uniform scalar byte field that is the region of interest. The ROI has the same XY extents as the input field. All byte values are either 0 (not part of ROI) or 1 (part of ROI). This field should be connected to the ROI input port of the imaging module that needs the ROI.

image draw structure (required)
The left output port contains the **image draw structure** that connects to the **image viewer** module's leftmost input port. It is required.

EXAMPLE

This example shows a simple network to define a region of interest that is used with the **ip arithmetic** module. The invisible upstream connections coming from **image viewer** to **sketch roi** are not shown. Note that **sketch roi** must take the same image as input that **image viewer** is receiving.

```
                       READ IMAGE
                             |
                             |
          |---------|        |
          |      |--|---|    |
   SKETCH ROI    |  |  IP ARITHMETIC
        |  |-----|  |----|
        |                |
        |--------|       |
                 |       |
              IMAGE VIEWER
```

RELATED MODULES

ip threshold
image viewer
any module with a region of interest input
image measure
image probe

SEE ALSO

The example script Imaging / SKETCH ROI demonstrates this module.

The upstream feedback mechanism that makes **sketch roi** work is described in the *AVS 5 Update* document.

sobel

NAME

sobel - apply an edge detecting filter to 2D field

SUMMARY

Name	sobel
Availability	Imaging mobule library
Type	filter
Inputs	field 2D *n-vector any-data any-coordinates* ("image")
Outputs	field of same type as input
Parameters	none

DESCRIPTION

sobel uses the "sobel operator" for finding edges in a 2D byte field. The typical use is to find edges in images prior to some segmentation operation, such as dividing the image into regions that correspond to the individual objects in the picture. The Sobel operator consists of two 3x3 filters. One detects changes in an image in the x direction; thus detecting vertical edges. The other detects changes in the y direction, and thus is used to detect horizontal edges.

sobel takes the two sobel filters and applies them to a source field to produce a destination field. Both the source and destination fields must be 2D. Typically, the source and destination fields will be AVS *images*, but they might also be 2D slices of 3D fields.

sobel accepts vectors of any size containing data of any type. In the case of an image, which is a 2D field of 4-byte vectors, **sobel** disregards the alpha bytes and separates the red, green and blue bytes. Then it applies the filter separately to each color byte, before reassembling the bytes into 4-vector image format.

In the case of non-image data, for example a 2D field of 5-vector floats, **sobel** handles one component of the vector at a time. All data-types are converted to floats during computation and then converted back in **sobel**'s output.

In order to handle edge effects, a border around the perimeter of the source field is not operated on. The border is one pixel wide.

INPUTS

Data Field (required; field 2D *n-vector any-data any-coordinates*)
A 2D AVS field, typically an image, to be operated on.

OUTPUTS

Output Field
The output field is the same type as the input data field.

EXAMPLE 1

The following network reads in an image, applies the sobel operation to it, and displays the resulting image:

```
READ IMAGE
     |
     |
   SOBEL
     |
     |
IMAGE VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read image
pixmap to image
orthogonal slicer
any other module which outputs a 2D field

Modules that can process **sobel**'s output:

display image
image viewer
any other module which takes a 2D field as input

Also related:

ip edge
generate filters
convolve
local area ops

SEE ALSO

The example script SOBEL demonstrates the **sobel** module.

NAME

statistics – display statistics on AVS field contents including min and max values

SUMMARY

Name	statistics
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	none
Parameters	*Name* / *Type* / *Default*
	Compute Median / switch / off

DESCRIPTION

The **statistics** module displays global statistical information about field data. **statistics** scans the input field and produces a small output table like the following:

```
Field Statistics
================
Dimensions:         628 184 (x4)
Min/Max:            0.000000 255.000000
Mean:               58.934429
Median:
Standard Deviation: 76.030327
Skewness:           1.328104
Kurtosis:           0.686514
```

The output is displayed in an output text widget. Calculating the Median value is compute-intensive; it is only calculated if the **Compute Median** switch is turned on.

Use the **statistics** module when you need to know what a field's min / max are. This information is often useful if you wish to scale the dials in downstream modules which are operating on the same input field. The output values mean:

Dimensions
The dimensions of the field, with vector length, if applicable.

Min/Max
The lowest and highest values in the data set.

Mean
The average of the data.

Median
The center value of a sorted list of the data.

Standard Deviation
The square root of the sum of the squares of the deviations.

The next two values are derived from comparing the distribution of the values to an ideal Gaussian "standard" distribution.

Skewness
When positive, the right side of the distribution curve is "steeper" than the left. When negative, the left side is "steeper."

Kurtosis
When positive, the data is more "spikey" than a standard distribution. When negative, the data is more broadly-distributed than a standard distribution.

INPUT

Data Field (required; field *any-dimension n-vector any-data any-coordinates*)

The input AVS field can be any dimension, with any vector length, and of any data type.

PARAMETERS

Compute Median

A toggle switch that makes **statistics** also go through the compute-intensive calculation of the field's median. It is off by default.

EXAMPLE 1

The following network computes statistics on an image.

```
          READ IMAGE
               |
               |
          STATISTICS
```

EXAMPLE 2

The following network shows how you might use the **statistics** module to determine the min and max values in a 3Dfield, so that you could scale the dials on the **thresholded slicer** module accordingly.

```
                       READ VOLUME
GENERATE COLORMAP           |
           |                |--------------------|
           |--------|       |                    |
                THRESHOLDED SLICER       STATISTICS
                            |
                            |
                 GEOMETRY VIEWER
```

RELATED MODULES

ip statistics
print field
compare field

SEE ALSO

The example script STATISTICS demonstrates the **statistics** module.

stream lines

NAME

stream lines – generate stream lines for a vector field

SUMMARY

Name	stream lines
Availability	FiniteDiff module library
Type	mapper
Inputs	field 3D 3-vector float *any-coordinates* field irregular *(optional, from samplers module)* upstream transform *(optional, invisible, autoconnect)* field 3d scalar *(optional, for coloring streamlines)* colormap *(optional, for coloring arrows)*
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
width	integer dial	12	4	32
length	integer dial	12	4	128
step	float dial	0.02	0.0	1.0
N segment	integer dial	16	2	64
Sample	radio	point		
Mode	radio	lines		
Method	radio	Euler		
Show Bounds	toggle	on	off	on

DESCRIPTION

The **stream lines** module generates streamlines based on a *field* that is a volume of 3D vectors. It places a "sample" of points at a parameter-controlled starting location in the volume. The number of points is also parameter-controlled; their orientation is mouse-controlled, using the same "virtual trackball" paradigm as the Geometry Viewer.

Then, for every time step, **stream lines** advances each sample point through space, based on the interpolated value of the vector field at its present position. The result is a set of stream lines showing the progress of massless particles through a vector field.

This module is similar to the **particle advector** module, except that the result is a static set of lines (or a surface) instead of a dynamically updated set of spheres.

A bounding diagram is generated to show you the region in which the samples are generated. For the **point** sample, this bounds is represented as a 3-dimensional cross-hair. For other representations, it is represented as a line, a circle, a rectangle, and a retangular prism, depending on which sampling option is chosen. This bounding hull is generated by default, but may be turned off using the **Show Bounds** button.

INPUTS

Data Field (required; field 3D 3-vector float *any-coordinates*)

The input field must be a 3D 3-vector field. The data for each field element must be a 3D vector of type float, representing the components of a velocity vector.

Sample Field (optional; field irregular)

This leftmost input port is meant to connect to the output of the **samplers** module. **samplers** creates a field that is nothing but a series of locations. **stream lines** will take these locations and use them as the sample of starting for points for the stream lines.

Note that, when the **stream lines** module receives input locations from **samplers**, **stream lines**'s **N segments** dial, and its **Sample** buttons disappear from the control panel.

Upstream Transform (optional, invisible, autoconnect)

When the **stream lines** and **geometry viewer** modules coexist in a network, they communicate through a normally-invisible data port. "streamline" shows up as an object in the Geometry Viewer. When you select the streamline object and move it, **geometry viewer** informs the **stream lines** module what the sample's new location is, and **stream lines** recalculates the location and streamlines it is displaying, accordingly. This module connection occurs automatically. The effect is to give you direct mouse manipulation control over the **stream lines** module's sample of locations.

Scalar Field (optional)

This is the port you fill when you want to color the streamlines by a second, scalar field. This field must be topologically identical to the required vector field (i.e. it must have the same dimensions, n-space, etc.). If this port is used, then a colormap must be supplied as well.

Colormap (optional)

If a scalar field is provided to color the streamlines with, then a colormap must also be provided to act as a mapping from data space to color space. In order for this to happen, it is important that the range of the colormap be related to the range of the scalar data. This is most easily accomplished by using the **color range** module which adjusts the effective range of the colormap to the field.

PARAMETERS

Width The density of points in the sample set.

Length A scale factor, which multiplies the length of the streamline segments generated during each time step.

Step Determines the time step for the interactive computation. The larger the value, the greater the interval.

N segments

An integer value which determines the number of points for which stream lines are computed. This controls the density of the stream lines output by **stream lines**.

Sample (radio buttons) Specifies the configuration of points from which stream lines will be drawn: point, line, circle, plane, or space.

mode (radio buttons) The default mode, **lines**, causes a stream line to be produced from each point in the sample set. The **mesh** mode applies only to line and circle samples. In this mode, a sample line or circle sweeps out a surface (manifold or cylinder) instead of a set of stream lines. If plane or space is selected as the sample, the **lines**, and **mesh** buttons disappear from the control panel. This is true even when the sample is received from the **samplers** module.

method (radio buttons) The buttons **Euler** and **Runge-Kutta** select the method used to calculate the next position of a sample particle. The **Euler** method is faster, involving a single vector in the input field. The **Runge-Kutta** method involves an interpolation, and produces considerably more accurate results.

Show Bounds

A bounding hull for the sample points is typically produced so that you can easily see the extent of the sample positions. This can be disabled with the **Show Bounds** toggle. When on (the default mode), this option causes the bounding hull to be generated as a wireframe geometry. When off, no hull is generated.

OUTPUTS

Streamlines (geometry)

A set of disjoint lines.

EXAMPLE 1

The following network reads in a 3D vector field, and calculates streamlines for the field. **animate lines** is used to dynamically represent the output of **stream lines**.

```
  READ FIELD
      |
      |-------------------|
      |                   |
 STREAM LINES             |
      |             VOLUME BOUNDS
      |                   |
 ANIMATE LINES            |
      |                   |
      |-------------------|
      |
GEOMETRY VIEWER
```

EXAMPLE 2

The following network uses the READ PLOT3D module to read in a 5-vector CFD (Computational Fluid Dynamics) field. Three of these componants are extracted to generate the stream lines and another is extracted to color the streamlines.

```
GENERATE COLORMAP                READ PLOT3D
        |                |-------------|
        |         EXTRACT SCALAR       |
        |______  ________|             |
             |   |       |        EXTRACT VECTOR
          COLOR RANGE    |             |
               |_______  |  ___________|
                      |  |  |          |
                    STREAMLINES    VOLUME BOUNDS
                        |______________|
                               |
                        GEOMETRY VIEWER
```

RELATED MODULES

animate lines
hedgehog
particle advector
samplers

SEE ALSO

The example script STREAMLINES demonstrates the **stream lines** module.

NAME

3D bar chart – 3D bar chart with average statistics and annotation

SUMMARY

Name	3D bar chart				
Availability	Imaging, Volume, FiniteDiff module libraries				
Type	mapper				
Inputs	field 2D uniform scalar float colormap				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Z scale	float dial	1.0	*unbounded*	*unbounded*
	width	float dial	0.8	0.0	1.0
	offset	float dial	0.1	0.0	1.0
	threshold	float dial	0.0	*unbounded*	*unbounded*
	tic scale	float dial	1.0	*unbounded*	*unbounded*

DESCRIPTION

The **3D bar chart** module converts a two-dimensional floating point field into a group of 3D blocks, represented as a geometry. Each element in the field is mapped to a 3D bar. The height of each bar above each point is proportional to the scalar value of the field.

Side panels show the Min, Max and Average along each row and column in the 2D data.

A threshold transparent sheet can be moved through the graph to highlight specific values. Only values above the threshold will protrude above the sheet.

Line tic marks and text labels show the row and column numbers, and the vertical scale.

This module does not normalize the Z-height. The XY plane is approximately 0.0 to 1.0 on each side. Use the **Z scale** dial to set the vertical scale. The dials are unbounded to allow any data range. If the dials prove too sensitive for small numbers, press the blue dot in the center of the dial to bring up the Dial Editor, and either type in a specific value, or reset the dial resolution with the **Min** and **Max** typeins.

INPUTS

Data Field (required; field 2D scalar uniform float)
The input data must be a 2D field with a scalar float data value at each element.

Colormap Colors each bar in the chart a specific color according to the data value at that point.

PARAMETERS

Z scale Floating point dial that controls the height scale for the entire chart. The default is 1. This often needs to be reset. For example, byte data ranging from 0 to 255 displays well with a **Z scale** value of .001.

width Floating point dial that controls the relative width of each bar, or the "space" between the bars. The default is .8.

offset Floating point dial that controls how far away the shadow planes that display each row/column's minimum, maximum and average are from the main bar chart. The default is .1.

threshold Floating dial that controls the height of the threshold sheet in the chart. The default is 0.0.

tic scale Floating dial that controls the vertical scale tic marks. There are ten tic marks. **tic scale** specifies the interval between tic marks, scaled by **Z scale**. The default is 1.

OUTPUTS

Geometry (geometry)
The output is an AVS geometry.

EXAMPLE

The following network inputs a 3D uniform vector field (such as *$AVS_PATH/data/field/radm/hour011.fld*, downsizes it, extracts one vector element, then removes one 2D plane (**orthogonal slicer**) from the volume and sends it to **3D bar chart** to be converted into a 3D geometric bar graph. The bars are colored by a colormap that is scaled to the data range. If the input field were byte or integer data, there would be a **field to float** module inserted between **extract scalar** and **color range** and **orthogonal slicer**.

```
                        READ FIELD
                            |
                         DOWNSIZE
                            |
GENERATE COLORMAP           |
        |               EXTRACT SCALAR
        |                   |
        |      |-----------|-----------|
        |      |                       |
      COLOR RANGE              ORTHOGONAL SLICER
            |                          |
            |-----------|    |---------|
                        |    |
                      3D BAR CHART
                             |
                             |
                      GEOMETRY VIEWER
```

RELATED MODULES

Modules that can provide the **Data Field** input:

any module that outputs a field

Modules that can provide the **Colormap** input:

generate colormap
color range

Modules that can process the output:

geometry viewer

SEE ALSO

The example script 3D BAR CHART demonstrates the **3D bar chart** module.

NAME

threshold – restrict values in data field

SUMMARY

Name	threshold
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field *any-dimension n-vector any-data any_coordinates*
Outputs	field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
thresh_min	float	0.0	none	none
thresh_max	float	255.0	none	none

DESCRIPTION

The **threshold** module transforms the values of a field as follows:

- Any value less than the value of the **threshold_min** parameter is set to 0.
- Any value greater than the value of the **threshold_max** parameter is set to 0.
- All values within the **threshold_min**-to-**threshold_max** range are not changed.

After being **threshold**'ed, a data set's values are all either zero, or in this range:

thresh_min ≤ *value* ≤ **thresh_max**

Note the difference between the **clamp** and **threshold** modules:

- **threshold** sets values outside the specified range to be zero.
- **clamp** sets values outside the specified range to be the range's minimum and maximum values.

INPUTS

Data Field (required; field *any-dimension n-vector any-data any_coordinates*)
The input data may be any AVS field.

PARAMETERS

thresh_min
The minimum threshold value.

thresh_max
The maximum threshold value.

OUTPUTS

Field Data The output field has the same dimensionality as the input field.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
any other filter module

Modules that could be used in place of **threshold**:

ip threshold
clamp

Modules that can process **threshold** output:

colorizer
any other filter module

SEE ALSO

The example scripts CONTOUR GEOMETRY, and THRESHOLDED SLICER demonstrate the **threshold** module.

NAME

thresholded slicer – slice through volume data with high / low values invisible

SUMMARY

Name thresholded slicer

Availability Volume, FiniteDiff module libraries

Type mapper

Inputs field 3D scalar *any-data any-coordinates* (volume)
upstream transform (optional, invisible, autoconnect)
colormap (required)

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Resolution	int dial	12	12	64
Distance	float dial	0.0	unbounded	unbounded
Low Threshold	float dial	0.0	unbounded	unbounded
High Threshold	float dial	255.0	unbounded	unbounded
Sampling Style	choice	point	point,trilinear	
Refine	choice	coarse	coarse, fine	

DESCRIPTION

The **thresholded slicer** module extracts a 2D slice from a 3D volume of data. It differs from the **arbitrary slicer**, **orthogonal slicer**, and **brick** modules in that one can establish that numerical values below a **Low Threshold** value and above a **High Threshold** value will not be mapped—they will be given zero values in the output 2D slice. One can thus "edit out" or "crop" high and low values from a volume rendering.

thresholded slicer's slice plane is moveable through the Z axis with its **Distance** parameter dial.

It is also possible to move the slice plane arbitrarily within the volume using the mouse or the Geometry Viewer's transformation panel This is because **thresholded slicer** has an invisible "Upstream Transform" input port that allows it to automatically receive information from the **geometry viewer** module about how the "thresholded slice" object has moved,

The mapping technique for **thresholded slicer** is the same as **arbitrary slicer**. That is, the volume of data is represented as a 3D scalar field, defining a lattice within the volume. The slice plane is represented as a 2D grid, with parameter-controlled resolution. The intersection of the volume and the grid is a *mesh* of vertices in 3D space.

Each vertex in the mesh is assigned a color (with the input from **generate colormap** or the **colormap manager**) that corresponds to one or more values of the scalar field. Values below and above the **Low Threshold** and **High Threshold** settings are set to zero. Since, in general, the mesh vertices *do not* coincide with the original lattice points, an interpolation method can be used — see the *trilinear* input parameter below.

By default, the volume is placed at the origin and the slice plane is an X-Y plane placed midway through the Z dimension of the data.

You can control the resolution of the mesh using the **Resolution** parameter. At lower resolutions, fewer original data points are used in the computations; at higher resolutions, more points are used.

The optimal way to use this module is to start off with a low resolution mesh, position it as desired, then increase the resolution and turn on trilinear mapping and the Fine level of refinement.

INPUTS

Data Field (required; field 3D scalar *any-data any-coordinates*)
The input data must be a 3D field, with a byte value at each location in the field. The field must be uniform.

Upstream Transform (optional, invisible, autoconnect)
When the **thresholded slicer** module coexists with the **geometry viewer** module in a network, **geometry viewer** feeds information on how the "thresholded slice" object has been moved in the Geometry Viewer back to this input port on the **thresholded slicer** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **thresholde slicer**'s slice plane.

Colormap (required; colormap)
By default, the value computed for each vertex of the mesh is used as the hue in HSV space. The values are transformed to the range 0..255, and are then used as indexes into the colormap.

PARAMETERS

Resolution An integer dial that controls how many sampling points are taken through each dimension of the volume data. The default is a fairly low resolution 12. The maximum value is 64.

Distance A floating point dial widget that controls the movement of the slice surface in the Z direction. The 0.0 initial value is defined to be *midway* through the volume. Hence, a volume with a Z dimension of 64 has 0.0 in the middle, with +32.0 and -32.0 in either direction. The dial itself is unbounded. If you enter a value outside the actual volume, the slice surface disappears.

Low Threshold
A floating point dial, set by default to 0.0. Values in the volume below this dial setting do not generate any polygons.

High Threshold
A floating point dial, set by default to 255.0. Values in the volume above this dial setting do not generate any polygons.

Refine The intersection of the contour with the voxel is computed in a refinement loop. This selection chooses how many levels of refinement are performed. **coarse** is 2; **fine** is 8. Fine gives more accurate contours.

Sampling Style
A choice of two styles that control how each vertex in the output mesh is assigned a color:

- **If Point**, a nearest-neighbor algorithm is used. Each mesh vertex is assigned the byte value of the nearest point in the lattice.
- **If Trilinear**, a trilinear interpolation is performed. The value at each vertex depends on the byte values at the eight lattice points that are the corners of the "enclosing cube".

The trilinear interpolation method is more accurate but takes longer to compute, particularly with larger meshes.

OUTPUTS

Geometry (geometry)

The output is an AVS *geometry*.

EXAMPLE

This example shows the typical usage of the **thresholded slicer** module for byte data in the range 0-255:

```
                  READ VOLUME
                       |
                       |
GENERATE COLOR         |------------------|
       |               |                  |
       |----------|    |                  |
                  |    |                  |
                  |    |                  |
            THRESHOLDED SLICER     VOLUME BOUNDS
                       |                  |
                       |  |---------------|
                       |  |
                  GEOMETRY VIEWER
```

The **volume bounds** modules gives a reference frame for orienting the slice plane. Often, an **isosurface** is also input to the **geometry viewer** module.

SEE ALSO

The example script THRESHOLDED SLICER demonstrates the **thresholded slicer** module.

time sampler

NAME

time sampler – extract 3D time slices from 4D time series field with interpolation

SUMMARY

Name	time sampler				
Availability	Volume, FiniteDiff module libraries				
Type	filter				
Inputs	field 4D *n-vector any-data any-coordinates*				
Outputs	field 3D *same-vector same-data same-coordinates*				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	time step	float dial	0.0	0.0	*number of slices*
	choice	choice	slice		

DESCRIPTION

time sampler extracts two sequential 3D fields from a 4D field and interpolates between their computational data values by one of three techniques, producing a single 3D field as output. (**time sampler** does not interpolate coordinate data.) The input field is intended to be a time series of 3D fields packed into a single 4D field. Using **time sampler**'s **time step** parameter, it is possible to generate the data interpolations that are required to animate time series data with the **AVS Animator** module.

INPUTS

Data Field (required; field 4D *n-vector any-data any-coordiates*)

The input is a 4D field of any type. Such fields can be created, for example, by concatenating a time series of data and coordinate files together and then using the **read field** module's data input parsing option to produce a single 4D field.

PARAMETERS

time step

time step is a floating point dial that specifies which time slices to interpolate between. When **slice** (below) is selected, **time step** only accepts whole integer values; intermediate floating point values are floored. The range is from 0.0 to the number of 3D time-steps in the 4D input field (counting from 0). The default is 0.0.

choice

A set of radio buttons that determines the interpolation method. **slice** is the default.

slice No interpolation is performed. Instead, just the 3D slice specified by **time step** is extracted and output.

linear Interpolate linearly between adjacent 3D slices. The formula used to generate the output value is:

((slice2 - slice1) ∗ (time step - floor(time step)) + slice1

Where *slice2* and *slice1* are the data values in the adjacent slices, *time step* is the value selected by the **time step** parameter, and *floor(time step)* is the integer portion of *time step.*

For example, if the **slice1** value were 10, and the **slice2** value were 1, selecting a **time step** of 1.33 would yield an interpolated value of:

```
   ( ( 1 - 10 ) * ( 1.33 - 1) ) + 10
=  (     -9     *     .33     ) + 10
=  -2.97 + 10
=  7.03
```

or the difference between 10 and 1 occurring 1/3 (1.33) of the way between the adjacent data values.

cubic Interpolate between adjacent 3D slices using cubic splines. This produces a non-linear, smoothed curve between the actual data values. The input field must have at least four time steps to compute cubic interpolation because **cubic** smooths using the two previous and two following data values. At the first and last time slices interpolation is linear.

OUTPUTS

Data Field (field 3D *same-vector same-data same-coordinates*)
The output is a 3D field of the same type as the input field.

EXAMPLE 1

The following network animates time series data using the **AVS Animator** module. Select one **time step** in the **time sampler** module, then set a keyframe in the **AVS Animator**. Select a second **time step** in **time sampler**, then set the next keyframe in the **Animator**. You can also use **animated float** to send data values to the **time step** parameter to create an animation without the **AVS Animator**.

Note that the **color range** module is connected to the field before it is time sampled.

```
                  READ FIELD
                      |
                      |
GENERATE COLORMAP     |                   AVS ANIMATOR
         |      -------|-------
         |      |             |
      COLOR RANGE       TIME SAMPLER
             |                |-------------|
             |                |             |
             |-----------|    |             |
                         |    |             |
                      BUBBLEVIZ     VOLUME BOUNDS
                           |                |
                     SCATTER DOTS           |
                           |                |
                           |----------------|
                           |
                GEOMETRY VIEWER
```

RELATED MODULES

orthogonal slicer

tracer

NAME

tracer – perform ray-traced volumetric rendering on volume data

SUMMARY

Name	tracer
Availability	Volume, FiniteDiff module libraries
Type	mapper
Inputs	field uniform 3D byte, scalar *or* 4-vector field 2D scalar float (transformation matrix, optional, autoconnect) colormap (optional; used with scalar input)
Outputs	field 2D 4-vector byte (image)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
alpha scale	float dial	1.0	0.0	1.0
perspective	float dial	0.0	0.0	1.0
width	int typein	64		
height	int typein	64		
interpolate	toggle	off		

DESCRIPTION

tracer belongs to a family of modules (along with **x-ray** and **cube**) that render volume data. **tracer** takes a volume, which can be visualized as a block of cubic "voxels" (volume elements), and generates a 2D image using ray tracing. Each voxel in the volume has color and opacity values associated with it.

The ray tracing method is as follows. For each pixel in the output image a ray is "shot" into the volume. Each voxel the ray passes through makes some contribution to the color of the pixel. How much a voxel contributes depends on its opacity. The ray travels through the volume until the opacity of all the cubes it has passed through adds up to 1.0. This is an "additive light model", because the rays accumulate voxel color contributions as they travel through a volume.

For example, if a ray were to hit a completely opaque red voxel then it would travel no further, and the pixel associated with that ray would be colored red. On the other hand, if the voxel were nearly transparent, then it would confer only a fraction of its color to the pixel, and the ray would pass deeper into the volume, summing the color values of the other voxels it intersects.

Volumetric rendering such as this allows you to penetrate beneath the surface of 3D data, and see depths surrounded by "translucent" outer layers. The degree of opacity of the volume can be controlled by changing the alpha scale parameter, or by using **generate colormap**'s widget's **opacity** control to edit the opacities associated with the data.

tracer has two input field options. Both are required to be uniform 3D byte fields. However, the byte fields can be either a scalar (a single byte of data at each node), or a 4-vector of bytes.

If the input field is scalar, then each 8-bit data value represents itself. The 0 to 255 data range will be interpreted as transparency and gray scale values (0 = transparent/white, 255 = opaque/black). To add color, connect a **generate colormap** module to **tracer**'s optional leftmost input port.

If the input field is a 4-vector of bytes, then the original data value (byte, integer, float, or double) has been translated into an alpha (transparency), red, green, blue "field of colors" by a module such as **colorizer** or **compute shade**.

The scalar byte field uses less memory than the 4-vector of bytes. Thus, for a given system memory size, it is possible to render a larger dataset.

On the other hand, 4-vector byte fields can be gradient-shaded with **compute shade** while scalar byte fields cannot.

The method used by **tracer** avoids the image anomalies that **alpha blend** displays when volumes are rotated.

tracer includes a "Performance Stats" output widget that reports the number of voxels and pixels rendered, and the wall-clock seconds required to produce them.

INPUTS

Data field (required; field 3D byte, scalar *or* 4-vector)

The input data must be a 3D uniform byte field. It may be either a scalar byte field, or a 4-vector of bytes in the alpha-red-green-blue format used in AVS images. Scalar byte fields use less memory. 4-vector alpha-red-green-blue input data is produced by passing 3D field data through the module **colorizer** or **compute shade** before it enters **tracer**. While using more memory, 4-vector fields can be gradient-shaded.

The **tracer** network structures differ slightly between the two input types. See the examples below.

Transformation matrix (optional; field 2D scalar float, autoconnect)

The center port on **tracer** can receive a 4x4 transformation matrix describing rotations and translations to apply to the volume data. This matrix (field 2D scalar float) can come from an appropriate downstream module such as **display tracker**, or from the **euler transformation** or **track ball** modules. These mechanisms allow you to rotate the volume in 3-space.

For example, when the **tracer** module is connected to the **display tracker** module in a network, **display tracker** sends a transformation matrix back to this port on **tracer**. This allows you to directly manipulate the volume by moving the mouse in **display tracker**'s window, using the "virtual spaceball" paradigm. For a more detailed description of direct manipulation see the section titled "Transforming Objects" in the "Geometry Viewer" chapter of the *AVS User's Guide*.

Colormap (optional; colormap)

Use this optional input port to colorize scalar data. If unused, the scalar byte data is rendered in gray scale. This port is ignored with 4-vector data.

PARAMETERS

alpha scale (float dial)

A floating point value between 0.0 and 1.0 which is multiplied by the alpha byte of every voxel in the volume. This determines how transparent the the volume will seem. The default of 1.0 results in all the voxels' alpha bytes remaining unchanged. As the value of alpha scale approaches 0.0 the volume becomes more transparent, allowing rays to penetrate deeper into the volume, and making inner regions visible.

The **generate colormap** module's **opacity** channel also controls transparency. It produces the "alpha byte" that **alpha scale** scales.

perspective (float dial)

With perspective set to the default 0.0, the rays sent into the volume emanate from an "eye point" at infinity. This means that when a ray passes through the image plane it is orthogonal to that plane, resulting in a parallel projection (i.e. non-perspective) view of the volume. As the perspective value increases the point from which rays emmanate moves closer to the image plane, resulting in an increase in perspective. Selecting a high value for perspective may result in part of the volume moving outside the bounds of the image window.

width (integer typein)

Value which determines the width in pixels of the output image. Another way of thinking of this is the width determines the number of rays that will be projected into the volume along the x direction. This changes the shape of the window through which you view the volume. With **perspective** on, changing the width can bring clipped regions of the window back into view.

Note: Downstream modules such as **display tracker** have controls that will enlarge the image in the output window without computing at higher resolution.

height (integer typein)

Value which determines the height in pixels of the output image. Another way of thinking of this is the height determines the number of rays that will be projected into the volume along the y direction. This changes the shape of the window through which you view the volume. With **perspective** selected, changing the height can bring clipped regions of the window back into view.

interpolate (toggle)

Allows you to choose between two ray-tracing algorithms:

Voxel approximation (default)

This is the default. The 3D field is broken into cells, or voxels, as described above, i.e. the volume is decomposed into blocks, each with eight corners. Each voxel has a single opacity—and, with 4-vector, a color—and set of shading parameters. These values are taken from the vertex at the voxel's upper lefthand corner, and are assumed to be uniform throughout the voxel.

The length of a ray's path through a voxel is computed. Thus if a ray just nicks the corner of a green voxel, only a little green is added to the ray's accumulated color. This method is faster than the trilinear interpolation method. Use it to get a quick look at the data.

Trilinear Interpolation

In this algorithm it is not assumed that each voxel has a uniform color and opacity. Rather, the field values of the voxel's eight corners are interpolated. These interpolated values are then used to determine the actual opacity and color values of the points at which a ray enters and exits a voxel. As in the voxel approximation method, the length of the ray's path through the voxel affects that voxel's contribution to the ray's color. This method produces a more accurate rendering of the volume.

OUTPUTS

Data Field (field 2D 4-vector byte)

The output field is an AVS image.

EXAMPLE 1

The following network reads a scalar 3D uniform byte field (a volume) and ray traces it. **generate colormap** colors the otherwise gray scale bytes. The module **euler transformation** allows you to rotate the volume to produce views from any angle. If the input was not originally byte values, it could be converted with the **field to byte** module.

```
                                           READ VOLUME
EULER TRANSFORMATION                            |
                |                               |
                |----------------------------|  |
                                             |  |
    GENERATE COLORMAP                        |  |
                |--------------------------| |  |
                                           | |  |
                                           TRACER
                                              |
                                              |
                                        DISPLAY IMAGE
```

EXAMPLE 2

The following network is identical to the previous, except the uniform input field has been translated into a 4-vector field of colors prior to entering **tracer**.

```
GENERATE COLORMAP                          READ FIELD
        |                                       |
        |-------------------------------|       |
                                        |       |
                                        COLORIZER
EULER TRANSFORMATION                            |
                |                               |
                |----------------------------|  |
                                             |  |
                                           TRACER
                                              |
                                              |
                                        DISPLAY IMAGE
```

EXAMPLE 3

Another interesting technique is to apply a light source to the data. In order to do this the gradient of the data (which approximates the "surface normal") must be computed. Note that the **compute shade** module has been modified to accept a transformation matrix. This prevents the light source from rotating relative to the data, when the object is rotated using **display tracker**, **euler transformation**, or **track ball**. Without connecting **display tracker** (or **euler transformation**, etc.) to **compute shade**, the light source would appear "attached" to any object transformations. A network for doing this gradient shading is:

```
                           READ VOLUME
                                |
GENERATE COLORMAP               |
               |                |
               |                |
|-----------|  |                |
|         COMPUTE SHADE
|                             |
|---------------| |
|             TRACER
|               |
|               |
|         DISPLAY TRACKER
|----------|
```

Note that this network uses the module **display tracker**, which allows you to directly manipulate the volume being viewed by moving the mouse. **display tracker** feeds information on the mouse's movements back to **tracer** through its lefthand data port.

RELATED MODULES

Modules that could be used in place of **tracer**:

x-ray
cube

Modules that could provide the **Data Field** input:

read volume
read field
colorizer
compute shade
any other module which outputs a 3D byte field, scalar or 4-vector.

Modules that could provide the **Transformation Matrix** input:

euler transformation
track ball
display tracker (using upstream data)

Modules that can process **tracer**'s output:

display tracker
display image
image viewer
image to postscript
any other module which takes an AVS image as input.

SEE ALSO

Garrity, M., "Raytracing Irregular Volume Data," (Proceedings of the 1990 San Diego Workshop on Volume Visualization), *Computer Graphics*, Volume 24, Number 5, November 1990, pp. 35-40. ACM SIGGRAPH.

The example scripts TRACER and COMPUTE SHADE demonstrate the **tracer** module.

NAME

track ball - send object transformation matrix to other modules

SUMMARY

Name	track ball
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data (coroutine)
Inputs	none
Outputs	field uniform 2D scalar float (transformation matrix) geometry
Parameters	*Name* *Type* track field

DESCRIPTION

track ball generates a 4x4 transformation matrix. It produces the same transformation output as **euler transformation**. The difference is that you specify the transformation with a direct manipulation trackball widget, rather than with dial values. Thus, **track ball** is a kind of hybrid between **euler transformation** and the direct manipulation facilities of **display tracker**.

track ball produces its output transformation matrix in two forms: as a transformation matrix that can be sent to modules like **tracer**; and as a geometry edit list transformation that can be sent to the **geometry viewer** module to control its objects.

track ball is particularly useful when you want to apply the same transformation to objects in two or more downstream modules.

PARAMETERS

track A trackball widget. To generate the transformation, move the trackball with the mouse and cursor. **track ball** uses the same buttons as the Geometry Viewer:

rotate	center button
translate	right button
scale	shift-middle button

OUTPUTS

Transformation Matrix (field 2D uniform scalar float)
The output is a 4x4 array of floating point values which specifies rotations and scaling operations that can be applied to transform an object around the origin of its own coordinate system.

geometry (geometry)
This output is a geometry edit list containing the transformation.

EXAMPLE 1

The following network performs volumetric ray-tracing using **tracer**. **track ball** is used to move the object.

track ball

```
                TRACK BALL   READ VOLUME
                          |  |
                          |  |
GENERATE COLORMAP         |  |
                |------|  |  |
                        TRACER
                          |
                          |
                    DISPLAY IMAGE
```

RELATED MODULES

Modules that accept **track ball**'s output:

tracer
compute shade
gradient shade
geometry viewer

NAME

transform pixmap – perform 3D transformation on pixmap (hardware texture mapping systems only)

SUMMARY

Name	transform pixmap	
Availability	this module is in the unsupported library requires texture mapping and other hardware support	
Type	data output	
Inputs	pixmap	
Outputs	pixmap	
Parameters	*Name*	*Type*
	image transform	4x4 matrix
	transform image	toggle
	reset	toggle
	refine	toggle

DESCRIPTION

The **transform pixmap** module maps its pixmap input onto a rectangle that has been arbitrarily transformed in three dimensions. The resulting pixmap is then output. The transformation allows for rotation, scaling, translation, or shearing of the image (or any combination thereof).

A benefit of using **transform pixmap** in a network is that it automatically scales its output pixmap size to fit the output window of a **display pixmap** module downstream. For example, if you read in a 512x512 pixmap, you can display the entire pixmap in any size window.

AVAILABILITY

For **transform pixmap** to work, and for it to appear in the module palette, the system it is running on must support texture mapping in both graphics software and hardware. (See the release note information that accompanies AVS on your platform). The software renderer does not support **transform pixmap**.

INPUTS

pixmap The input can be any AVS *pixmap*.

PARAMETERS

image transform

Controls the 3D transform to be applied to the pixmap. The control widget is a window containing a colored cube, annotated with coordinate axis information. Transforming this cube with the following mouse buttons causes the pixmap to be transformed accordingly:

Mouse	**Transform**
left	cycle among three views: along X-axis, along Y-axis, along Z-axis
middle	rotate
right	translate in plane of screen
middle with SHIFT key	scale
right with SHIFT key	translate perpendicular to plane of screen

The mouse button mapping is the same as in the Geometry Viewer.

transform image (toggle)

This toggle parameter controls whether you can transform the image directly (i.e. in its window), or must use the **transformation** widget described above.

- **If ON:** The **transform pixmap** module "grabs" button press events in the associated output window, allowing you to transform the image directly.

  NOTE: For pixmaps generated by a **render geometry** module, button clicks in the window will no longer transform the geometry, but will transform the pixmap instead.

- **If OFF:** The mouse buttons have the same meanings, but you cannot "grab" the image in the output window directly. Instead, you must transform the cube in the transform control widget, which appears in the module's control panel.

refine (toggle)

Controls the use of point sampling to improve the quality of the output pixmap.

- **If ON**, A "successive refinement" algorithm is used to improve picture quality. When there is no other work left to do, **transform pixmap** applies nine refinement passes, each of which incrementally improves the picture. This is especially useful when small images are to be displayed in very large windows, or vice-versa.
- **If OFF**, the transformation applied to the image uses a "point sampling" algorithm.

reset (one-shot)

Resets the transformation of the image to be the identity transformation.

OUTPUTS

pixmap The output is a pixmap containing a *scene* that includes all the input objects.

EXAMPLE

```
   READ IMAGE
       |
 IMAGE TO PIXMAP
       |
TRANSFORM PIXMAP
       |
 DISPLAY PIXMAP
```

RELATED MODULES

image to pixmap, transform pixmap, display pixmap

LIMITATIONS

When you transform an image directly (**transform image** toggle) or use the **Reset** function, the transform control widget is not updated.

SEE ALSO

The example script CONTOUR GEOMETRY demonstrates the **pixmap to image** module.

NAME

transpose – exchange dimensions in a 2D or 3D data set

SUMMARY

Name	transpose
Availability	Imaging, Volume, FiniteDiff module libraries
Type	filter
Inputs	field 2D / 3D *n-vector any-data any-coordinates*
Outputs	field of same type as input

Parameters	*Name*	*Type*	*Default*	*Choices*
	axis	choice	Original	Original, YZ, XZ, XY

DESCRIPTION

The **transpose** module exchanges the data in two dimensions of a 2D or 3D field. It can be used to change the orientation of the data for display and / or processing purposes.

INPUTS

Data Field (required; field 2D / 3D *n-vector any-data any-coordinates*)
The input data may be any 2D or 3D AVS field.

PARAMETERS

axis The choices for exchanging the data are:

- **Original** Copies the input to the output; no transformation is performed.
- **YZ** Swaps the Y and Z dimensions. (Equivalent to "Original" for a 2D field.)
- **XZ** Swaps the X and Z dimensions. (Equivalent to "Original" for a 2D field.)
- **XY** Swaps the X and Y dimensions.

OUTPUTS

Data Field (field 2D / 3D *n-vector any-data any-coordinates*)
The output field has the same dimensionality and type as the input field.

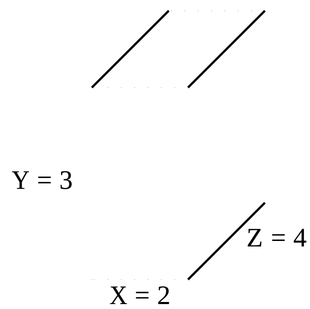
EXAMPLE 1

The following network reads in an image and then swaps the XY dimensions:

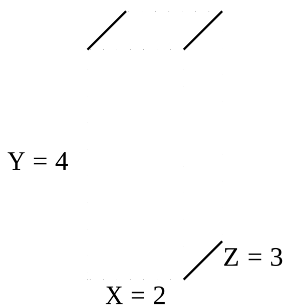
```
  READ IMAGE
      |
  TRANSPOSE
      |
DISPLAY IMAGE
```

EXAMPLE 2

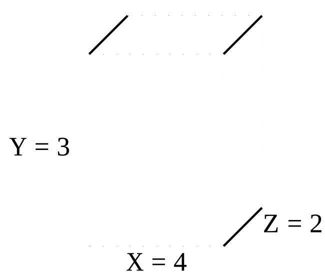
These drawings illustrate the transposition choices:



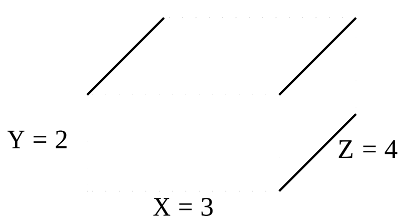
Original Data



After YZ Transpose



After XZ Transpose



After XY Transpose

RELATED MODULES

This module combined with **mirror** can re-orient the data in any desired way. See also **ip reflect**.

NAME

tristate - send a tristate value to one or more module(s) tristate parameter port(s)

SUMMARY

Name	tristate				
Type	data				
Inputs	none				
Outputs	tristate				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	tristate	tristate	0	0	2

DESCRIPTION

The **tristate** module sends a single user-specified tristate value to one or more tristate parameter ports on one or more receiving modules. Its purpose is to make it possible for a user to simultaneously control tristate parameter input to more than one module using only a single tristate input widget.

The tristate data-type is a variant of the boolean data-type. A tristate variable has three possible values: 0, 1 or 2. It is used to make selections when there are only three possible choices.

Before you can connect **tristate** to the receiving module, you must make that receiving module's parameter port visible. To make a parameter port visible, call up the module's Editor Window panel by pressing the middle or right mouse button on the module icon dimple. Next, look under the "Parameters" list to find the parameter you want to plug into. Position the mouse cursor over that parameter's button and press any mouse button. When the Parameter's Editor Window appears, click any mouse button over its "Port Visible" switch. A white parameter port should appear on the module icon. Connect this parameter port to the **tristate** module icon in the usual way one connects modules.

PARAMETERS

tristate (integer)

The single tristate value (0, 1, or 2), specified through a tristate widget, to be sent to the receiving module(s) tristate parameter port(s). The default value is zero.

OUTPUTS

tristate (integer)

The tristate value (0, 1, or 2) is sent to all modules with tristate-type parameter ports that are connected to the **tristate** module.

RELATED MODULES

Modules that can process **tristate**'s output:

modules with tristate-type parameter ports

tube

NAME

tube – convert lines to cylindrical tubes

SUMMARY

Name	tube				
Availability	UCD, FiniteDiff module libraries				
Type	filter				
Inputs	geometry				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	radius	float	0.1	0.0	4.0

DESCRIPTION

The **tube** module transforms an AVS *geometry*, replacing a set of disjoint lines with "tubes" constructed out of eight polygons.

INPUTS

Geometry (required; geometry) An AVS geometry, created with the *libgeom* library or by another AVS module.

PARAMETERS

radius The radius to be used for the tube. Only values in the range 0.0 – 0.4 produce an acceptable result.

OUTPUTS

Geometry (geometry)
The output is an AVS *geometry*, representing each input line as a set of polygons.

EXAMPLE

In this example, the original geometry includes no disjoint lines. The **wireframe** module is used to add disjoint lines, which are then converted to tubes.

```
   READ GEOM
       |
   WIREFRAME
       |
     TUBE
       |
GEOMETRY VIEWER
```

RELATED MODULES

read geom, offset, shrink, flip normal, wireframe, render geometry

LIMITATIONS

Only **radius** values in the range 0.0 – 0.4 produce acceptable results.

The cylinders are not capped and adjacent line segments are not joined. For thick cylinders, there may be quite a bit of surface intersections at the joins.

SEE ALSO

The example script TUBE demonstrates the **tube** module.

NAME

ucd anno – show data values of cells or nodes of a UCD structure

SUMMARY

Name	ucd anno
Availability	UCD module library
Type	mapper
Inputs	ucd structure upstream geometry (optional, invisible, autoconnect)
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Node Data	boolean	on		
components	choice	coords		
Cell Data	boolean	off		
components	choice	coords		
label id	boolean	on		
label value	boolean	off		
cell nodes	boolean	off		
title	boolean	off		
Text Size	integer dial	2	1	5
Text Offset	float dial	0.0	-10.0	10.0

DESCRIPTION

ucd anno makes it possible to see the values of specific cells and nodes of a UCD structure simply by clicking on the structure. The cell or node values of the cell that is clicked on are output as geometry labels, and can be viewed along with the UCD structure using the **geometry viewer** module. The **ucd anno** module thus provides a way to directly view data values contained in a UCD structure.

In a UCD structure, nodes and cells may have an arbitrary number of data components associated with them. **ucd anno** displays the values of one data component at a time, whether it is a scalar or a vector.

1. Use the **node data** and **cell data** choice buttons to select which type of data, node or cell, you wish to view.
2. Use the radio buttons beneath **node data** and **cell data** to select the data components you want **ucd anno** to display. Both may be selected. Note that the first choice is **coord**, which selects the coordinates of the node or cell rather than its component data values.
3. Choose the values you wish to see from the Label Options menu: any combination of the label's id, value, and cell nodes.
4. If necessary, use the **Text Size** and **Text Offset** parameters to size and position the text annotations so that you can read them.

ucd anno takes two inputs: a UCD structure, and an upstream geometry which it receives when it is in a network with **geometry viewer**. When you click the left mouse button on the image of the UCD structure the **geometry viewer** module sends information upstream telling **ucd anno** where on the structure the mouse was clicked. From this information **ucd anno** calculates which cell or node is being selected, and displays the data for that cell or node.

The labels that **ucd anno** outputs appear as geometry objects in 3-space attached to the nodes they are associated with. If the UCD structure is rotated the node and cell labels will rotate along with it. As they rotate they remain oriented parallel to

geometry viewer's window. This may cause a label to intersect the volume of the UCD structure and be partly or wholly hidden by the structure. Rotating the structure further will usually bring the label above the structure's surface. Alternatively:

- Use the **Text Offset** parameter to move the label;
- Use the **ucd to geom** module's **External Edges** parameter to display the ucd structure as a wireframe box;
- Or use the **Transparency** slider on the Geometry Viewer's **Edit Property** panel to make the structure semi-transparent and let the annotations show through. If your platform does not support hardware transparency, switch to **Software Renderer** on the **Cameras** menu.

INPUTS

UCD Structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

upstream geometry (optional, invisible, autoconnect)

When the **ucd anno** module coexists with the **geometry viewer** module in a network, **geometry viewer** feeds information on where the mouse has been clicked back to this input port on the **ucd anno** module. The two modules connect automatically, through a data pathway that is normally invisible. This makes it possible to see the values of specific cells and nodes simply by clicking on them.

PARAMETERS

Node Data Selects node data display. This is the default. Once this is selected, you may use the radio buttons to choose one data value to display, either the coordinates of the node, or one of its data components.

coords

Displays the coordinates of the node.

<component...>

Selects which of the node's data components to display. The buttons show the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

Cell Data Selects cell data display. Once this is selected, you may use the radio buttons to choose one data value to display, either the coordinates of the cell, or one of its data components.

coords

Displays the coordinates of the midpoint of the cell. This choice is present only if there is cell based data associated with the UCD structure.

<component...>

Selects which of the cell's data components to display. The buttons show the label attached to each cell data component. Before the module has received data, the default "<data 1>, <data 2>, ..." is displayed. If there is no cell based data in the structure "<no data>" is displayed on the button.

Label Options

label id

When **label id** is selected the integer or string that identifies a cell or node is displayed.

label value

When **label value** is selected the floating point value associated with one data component of a cell or node is displayed.

cell nodes

When **cell nodes** is selected, **ucd anno** displays the data for all the nodes of the cell that has been clicked on. Thus, for a hexadehron, **ucd anno** would display the node data at each of the cell's 8 nodes.

title

When **title** is selected, if the UCD structure has a title, it is displayed in the top-left corner of **display pixmap**'s window.

Text Size An integer dial that controls the font size of the output strings.

Text Offset

A floating point dial that offsets the text from the UCD node or cell, making it easier to read. The default is 0.0 (no offset); the min is -10.0 and the max is 10.0.

OUTPUTS

Geometry **ucd anno**'s outputs consist of the selected UCD structure values output as a geometry.

EXAMPLE

The following network reads in a UCD structure and annotates it. The selected values are displayed by **geometry viewer** along with the UCD structure itself:

```
        READ UCD
           |
           |-------------|
           |             |
     UCD TO GEOM    UCD ANNO
           |             |
           |-------------|
           |
    GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

field to ucd
ucd crop
ucd threshold
ucd extract
ucd hex to tet
Any module that outputs a UCD structure.

Modules that can process **ucd anno**'s output:

geometry viewer

SEE ALSO

The example script UCD ANNO demonstrates the **ucd anno** module.

ucd cell color

NAME

ucd cell color – color ucd structure based on cell or material id values

SUMMARY

Name	ucd cell color		
Availability	UCD module library		
Type	mapper		
Inputs	ucd structure colormap		
Outputs	field 1D 3-vector real		
Parameters	*Name*	*Type*	*Default*
	cell data	choice	<data 1>

DESCRIPTION

ucd cell color is used to color a UCD structure based upon either the *cell* data values, or the data values of the structure's material ids. It is thus almost identical in function to **ucd contour**, except that the latter colors a UCD structure based upon *node* data values.

Its output is passed to **ucd to geom**'s leftmost input port to produce a colored representation of a UCD structure. Essentially, **ucd cell color** associates colors with the values at each cell of a UCD structure—either the cell data values or the cell's material id.

A UCD structure has a number of cells. Each of these cells may have an arbitrary number of data components associated with it. Furthermore, each of these components itself can be a vector or a scalar. **ucd cell color** can only color the values of scalar cell components.

Use the **cell data** radio buttons to select one of the scalar data components, or the material id. The labels associated with the data components will be displayed on the radio buttons.

If the UCD structure has no cell data, then only **Materials** is displayed.

ucd cell color takes each cell or material id value and colors it in proportion to the range of values in the structure using the formula:

```
                cell_value - min_cell_value
color_index =  ------------------------------  *  max_colormap_value
              max_cell_value - min_cell_value
```

The "color index" is an index into the input colormap, and is used to compute the 3-vector real value for a given color.

Thus, **ucd cell color** scales the colormap to the range of values of the cell component or material id that has been selected. In other words, the lowest cell or material id value present in the structure will get colored with the lowest colormap value, and the highest cell or material id value will get colored with the highest colormap value. Of course you may change the input colormap using **generate colormap**'s colormap widget. The **Color Field** output by **ucd cell color** does not include the "alpha" or opacity information contained in an AVS colormap.

It should be noted that the **Color Field** output by **ucd cell color** is not an AVS colormap.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Colormap (required; colormap)

An AVS colormap. **ucd cell color** maps node values in the input structure to colors in the colormap.

PARAMETERS

cell data Selects which of the cell's data components to display. A set of radio buttons shows the label attached to each cell data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no cell data in the structure, then only **Materials** is displayed as a choice, indicating colorization by material ids.

OUTPUTS

Color Field (field 1D 3-vector real)

The output field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green, and blue. This should be connected to **ucd to geom**'s leftmost input port. (**ucd contour** uses the center input port.)

Note: if **ucd contour** or **ucd legend** is connected to **ucd to geom**'s center input port, by default it will be used instead of **ucd cell color** to color the output geometry. To override this, press the **Color Cells** button on **ucd to geom**'s control panel.

EXAMPLE 1

The following network reads in a UCD structure, colors each cell based on the cell's value, and displays the result. The sample ucd file *cell_data.inp* illustrates the functionality with cell data, and *shock.inp* illustrates the functionality with material ids.

```
GENERATE COLORMAP         READ UCD
         |                   |
         |     |-------------|
         UCD CELL COLOR      |
               |             |
               |-----|       |
                     |       |
                    UCD TO GEOM
                        |
                        |
                  GEOMETRY VIEWER
```

RELATED MODULES

ucd contour

Modules that could provide the **UCD Structure** input:

read ucd
ucd crop
ucd threshold
Any module that outputs a UCD Structure.

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd cell color**'s output:

ucd to geom

SEE ALSO

The example script UCD CELL COLOR demonstrates the **ucd cell color** module.

NAME

ucd cell to node – convert ucd cell-based data into node data

SUMMARY

Name	ucd cell to node			
Availability	UCD module library			
Type	filter			
Inputs	ucd structure			
Outputs	ucd structure			
Parameters	*Name*	*Type*	*Default*	*Choices*
	Method	choice	Average	Average, Interpolate

DESCRIPTION

The **ucd cell to node** module accepts a ucd structure with cell-based data components as input and computes node data components based on the cell data values. This module is necessary to visualize ucd cell-based data, since almost no other currently-implemented AVS ucd modules access cell data.

ucd cell to node uses one of two Methods to compute node values: **Average** or **Interpolate**. If the **Average** parameter is selected, a node value is computed by averaging values at all adjoining cells. If the **Interpolate** parameter is selected, then a node value is computed by interpolating values using distances from the node to the adjoining cell centroids. The output is a ucd structure containing only node data.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

Method (choice)

Selects method of converting cell-based data into node-based data: **Average** or **Interpolate**.

OUTPUTS

UCD structure

The output UCD structure contains node-based values.

EXAMPLE

The following network reads in a UCD structure that has cell-based components, converts cell-based components into the node data, and colors each node based on the value of that component:

ucd cell to node

```
                            READ UCD
                               |
                               |
 GENERATE COLORMAP    UCD CELL TO NODE
           |                   |
           |     |---------|
          UCD CONTOUR          |
                 |             |
                 |-----|       |
                       |       |
                     UCD TO GEOM
                               |
                               |
                   GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd

Modules that can process **ucd cell to node**'s output:

ucd to geom, ucd crop, ucd threshold, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d, ucd legend, ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD CELL TO NODE demonstrates the **ucd cell to node** module.

NAME

ucd contour – generate list of color values associated with unstructured cell data

SUMMARY

Name	ucd contour		
Availability	UCD module library		
Type	mapper		
Inputs	ucd structure colormap		
Outputs	field 1D 3-vector real		
Parameters	*Name*	*Type*	*Default*
	node data	choice	<data 1>

DESCRIPTION

ucd contour is used to create a color contour of a UCD structure. Its output is passed to **ucd to geom** to produce a colored representation of a UCD structure. Essentially, **ucd contour** associates colors with the values at each node of a UCD structure.

Typically a UCD structure has a number of nodes. Each of these nodes may have an arbitrary number of data components associated with it. Furthermore each of these components itself can be a vector or a scalar.

ucd contour can only color the values of scalar node components. By using the **node data** radio buttons you can select a scalar data component for **ucd contour** to color. If a UCD structure has both scalar and vector components, only the scalar components will be displayed. The labels associated with the data components will be displayed on the radio buttons.

ucd contour takes each node value and colors it in proportion to the range of values in the structure using the formula:

```
                   node_value - min_node_value
 color_index =  ------------------------------  *  max_colormap_value
                 max_node_value - min_node_value
```

The "color index" is an index into the input colormap, and is used to compute the 3-vector real value for a given color.

Thus **ucd contour** scales the colormap to the range of values of the node component that has been selected. In other words, the lowest node value present in the structure will get colored with the lowest colormap value, and the highest node value will get colored with the highest colormap value. Of course you may change the input colormap using **generate colormap**'s colormap widget. The **Color Field** output by **ucd contour** does not include the "alpha" or opacity information contained in an AVS colormap.

It should be noted that the **Color Field** output by **ucd contour** is not an AVS colormap.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Colormap (required; colormap)

An AVS colormap. **ucd contour** maps node values in the input structure to colors in the colormap.

ucd contour

PARAMETERS

node data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

OUTPUTS

Color Field (field 1D 3-vector real)
The output field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green, and blue.

EXAMPLE

The following network reads in a UCD structure, colors each node based on the node's value, and displays the result:

```
GENERATE COLORMAP      READ UCD
          |               |
          |     |---------|
          UCD CONTOUR     |
                |         |
                |-----|   |
                      |   |
                      UCD TO GEOM
                          |
                          |
                    GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD Structure** input:

read ucd
ucd crop
ucd threshold
Any module that outputs a UCD Structure.

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd contour**'s output:

ucd iso
ucd probe
ucd rslice
ucd to geom
ucd slice 2D

SEE ALSO

The example scripts UCD ISO, UCD PROBE, UCD EXTRACT, as well as others demonstrate the **ucd contour** module.

NAME

ucd crop – subset UCD structure data using slice plane or box

SUMMARY

Name	ucd crop
Availability	UCD module library
Type	filter
Inputs	ucd structure upstream transform (invisible, optional, autoconnect)
Outputs	ucd structure geometry

Parameters

Name	*Type*	*Default*
Crop Tool	choice	plane
Crop Direction	choice	inside
Do Crop	boolean	off

DESCRIPTION

ucd crop allows you to cut away portions of a ucd structure leaving behind the cells you are interested in. You can use either a slice plane or a wireframe box as your tool for subsetting UCD structures. Two notes: First, before cropping a UCD structure, the subsetting tool must be moved from its default location. Second, to initiate the actual cropping operation, you must press the "Do Crop" button.

The slice plane is initially oriented in the xy plane. If you rotate the slice plane, you will see that one side has a highlighted area. The highlighted surface is on the side that will be cropped if the Crop Direction is set to **inside**. If the Crop Direction is set to **outside**, the unhighlighted side of the plane will be cropped. In other words, any cells in the input structure which lie on the highlighted (or unhighlighted) side of the slice plane will not appear in the structure output by **ucd crop**. If a cell has even one node lying on the outside of the slice plane, that cell will be cropped from the output. Similarly, when using the cubic **space** tool, any cells that are inside or outside the bounds of the wireframe box are cropped from the output structure.

The **ucd crop** module is similar to the module **ucd threshold**. **ucd crop**, however, eliminates nodes from a UCD structure based on their x, y, z coordinates—**ucd threshold** eliminates nodes based upon their values.

ucd crop outputs both the cropped ucd structure and a geometry that represents the subsetting tool currently selected. Typically, the **ucd to geom** module is used to convert the structure output by **ucd crop** to a geometry so it can be visualized using the **geometry viewer** module.

Since **ucd crop** outputs the slice plane and box subsetting tools as geometry objects, they can be sent directly to **geometry viewer**, and they can be manipulated directly using the mouse just like any other geometry objects; simply enter the Geometry Viewer and select the crop tool object as the current object. When **ucd crop** is linked in a network to **geometry viewer**, manipulating the subsetting tools with the mouse causes **geometry viewer** to send an upstream transform to **ucd crop**. This tells **ucd crop** how the slice plane or box tool has been reoriented relative to the input structure. Then **ucd crop** can recalculate what portions of the structure to cut away.

INPUTS

Structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Upstream Transform (invisible, optional, autoconnect)
When the **ucd crop** module coexists with the **geometry viewer** module in a network, **geometry viewer** feeds information on how the "plane" or "space" subsetting object has been moved in the Geometry Viewer back to this input port on the **ucd crop** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **ucd crop**'s subsetting tools.

PARAMETERS

plane A radio button that selects the slice plane as the subsetting tool.

space A radio button that selects the wireframe box as the subsetting tool.

inside A radio button that selects the inside side of croping tool.

outside A radio button that selects the outside side of croping tool.

Do Crop A boolean switch that initiates the cropping function. This button allows you to manipulate the subsetting tool until you are satisfied with its position, and only then perform the cropping.

OUTPUTS

Structure The output structure is the cropped AVS unstructured cell data (UCD).

Geometry The geometry object that **ucd crop** outputs represents the subsetting tool currently selected, i.e., either the slice plane or the box tool.

EXAMPLE 1

The following network reads in a UCD structure and crops it. The **ucd crop** module outputs a geometry (the cropping tool) which gets passed directly to **geometry viewer**; it also outputs the cropped UCD structure from which a geometry is formed. This cropped UCD structure is both colored with **generate colormap** and **ucd contour**, and converted into a geometry with **ucd to geom**. Both the cropping tool and the cropped UCD structure are displayed together in the **geometry viewer**. Again, you must first move the cropping tool, then press **Do Crop** before the cropping will occur.

```
READ UCD
    |
    |              GENERATE COLORMAP
    |                      |
UCD CROP                   |
  |   |                    |
  |   --------|-------------------|
  |           |            |      |
  |           |         UCD CONTOUR
  |           |                   |
  |        |--|-------------------|
  |        |  |
  |     UCD TO GEOM (1)
  |        |
GEOMETRY VIEWER
```

EXAMPLE 2

This network is the same as the first, with two changes. First, the colorizing clause has been eliminated for clarity. Second, **read ucd** also sends the original UCD structure to a second **ucd to geom** module labelled (2). You can use this second geometry of the original, uncropped UCD structure to act as a wireframe volume bounds around the structure. To do this, either switch to the **geometry viewer** and specify a wireframe representation mode for the geometry output by this module, or press this second **ucd to geom** module's **External Edges** button. (Until you "wireframe" this second, overlapping UCD structure's representation, it will obscure the cropped version of the UCD.)

```
                 READ UCD
         |---------------|
         |               |
         |           UCD CROP
         |              |   |------|
         |              |          |
    UCD TO GEOM (2)     |      UCD TO GEOM (1)
         |              |---|      |
         |------------------|------|
                            |
                            |
                            |
                   GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
ucd_extract
Any module that outputs a UCD structure.

Other modules that subset UCD structures:

ucd threshold
ucd rslice

Modules that can process the cropped UCD structure output:

ucd to geom
Any module that inputs a UCD structure.

Modules that can process the subsetting tool output:

geometry viewer

SEE ALSO

The example script UCD CROP demonstrates the **ucd crop** module.

ucd curl

NAME

ucd curl – compute the curl of a vector UCD structure

SUMMARY

Name	ucd curl		
Availability	UCD module library		
Type	filter		
Inputs	ucd structure		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	Node Data	choice	<data 1>

DESCRIPTION

The **ucd curl** module accepts a UCD structure with vector node data components as input and computes the curl of that structure as output.

To reach the final result, **ucd curl** traverses the structure by cells, calculating the curl for each node in the cell, as affected by the other nodes in the cell. Because nodes are members of multiple cells, during this pass they accummulate an array of *n* curl values, one value coming from each cell of which the node is a member. Finally, **ucd curl** traverses the structure by nodes, averaging the array of results at each node to produce the final curl value for the node.

A UCD structure with only scalar data components should first be converted to contain vector components with **ucd extract vector**. The **Node Data** choice selects among multiple vector node data components.

Computation is a finite difference approximation based on a central difference scheme. Where the input is the vector function:

$\{F_x, F_y, F_z\}(i, j, k)$

The equation used to compute the curl is:

$$curl = \left\{ \left[\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right], \left[\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right], \left[\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right] \right\}$$

INPUTS

UCD Structure (required)

The input is a UCD structure containing 3-vector node data components.

PARAMETERS

Node Data Radio buttons to select which of the node's vector data components to display. The buttons show the label attached to each vector node data component. Before the module receives data, the default "<data 1>, <data 2>,..." is displayed. If there are no vector components in the node data, **ucd curl** complains. If there are several vector data components, these buttons let you select which componenet to use in calculating the curl. If there is no node data in the structure, "<no data>" is displayed on the button.

OUTPUTS

UCD Structure

The output structure has a single 3-element vector node data component representing the curl at each node.

EXAMPLE

```
                     READ UCD
                         |
                 UCD EXTRACT VECTOR
                         |
                      UCD CURL
                         |
GENERATE COLORMAP        |-------------|
             |   |-------|             |
             |   |       |             |
             |   |   UCD VECMAG        |
             |   |       |             |
             |---|----|  |             |
             |   |    |  |             |
             |   |    UCD ISO          |
             |   |       |        UCD TO GEOM
            UCD HOG      |             |
                 |-------|-------------|
                         |
             GEOMETRY VIEWER
```

RELATED MODULES

ucd div
ucd grad
ucd hog
ucd streamlines

SEE ALSO

The example script UCD CURL demonstrates the **ucd curl** module.

ucd div

NAME

ucd div – compute the divergence of a vector UCD structure

SUMMARY

Name	ucd div		
Availability	UCD module library		
Type	filter		
Inputs	ucd structure		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	Node Data	choice	<data 1>

DESCRIPTION

The **ucd div** module accepts a UCD structure containing 3-vector node data components as input and computes the divergence of the vector component as output.

Divergence is computed at each given node by averaging the divergences computed for surrounding cells at the given node.

To reach the final result, **ucd div** traverses the structure by cells, calculating the divergence for each node in the cell, as affected by the other nodes in the cell. Because nodes are members of multiple cells, during this pass they accummulate an array of *n* divergence values, one value coming from each cell of which the node is a member. Finally, **ucd div** traverses the structure by nodes, averaging the array of results at each node to produce the final divergence value for the node.

A UCD structure with only scalar data components should first be converted to contain vector components with **ucd extract vector**. The **Node Data** choice selects among multiple vector node data components.

The equation used to compute the divergence is:

$$divergence = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}$$

where (F_x, F_y, F_z) is a vector node data component.

INPUTS

UCD Structure (required)

The input is a UCD structure containing 3-vector node data components.

PARAMETERS

Node Data Radio buttons to select which of the node's vector data components to display. The buttons show the label attached to each vector node data component. Before the module receives data, the default "<data 1>, <data 2>,..." is displayed. If there are no vector components in the node data, **ucd div** complains. If there are several vector data components, these buttons let you select which component to use in calculating the divergence. If there is no node data in the structure, "<no data>" is displayed on the button.

OUTPUTS

UCD Structure

The output structure has a single floating-point value for each input structure node.

EXAMPLE

The following network reads in a UCD structure with vector node data components and computes its divergence. The divergence is then displayed as an isosurface.

```
                         READ UCD
                             |
                             |-------------|
                     UCD EXTRACT VECTOR    |
                             |             |
                          UCD DIV          |
 GENERATE COLORMAP           |        UCD TO GEOM
              |              |             |
              |---------|    |             |
                        |    |             |
                        UCD ISO            |
                             |-------------|
                             |
                 GEOMETRY VIEWER
```

RELATED MODULES

ucd curl
ucd grad
ucd streamlines
ucd hog

SEE ALSO

The example script UCD DIV demonstrates the **ucd div** module.

ucd extract

NAME

ucd extract – extract single node component from a UCD structure

SUMMARY

Name	ucd extract		
Availability	UCD module library		
Type	filter		
Inputs	ucd structure		
Outputs	structure		
Parameters	*Name*	*Type*	*Default*
	node data	choice	<data 1>

DESCRIPTION

The **ucd extract** module takes a ucd structure that has several data components at each node and outputs a structure that has only one data component at each node. The output UCD structure is identical to the input structure, except for the extraction.

Each node in a UCD structure may have an arbitrary number of data components associated with it. Furthermore each of these components itself can be a vector or a scalar. For example, a UCD structure may have 100 nodes. Each node consists of 3 components, labeled "temperature", "pressure", and "velocity". The first two components are scalar float values, but velocity is represented as a vector of three values.

ucd extract will extract any single component of the node data, whether that component is a vector or a scalar. If **ucd extract** takes a vector component, it extracts the entire vector of values. This means that **ucd extract** does not let you take a single element from a vector component. (Use **ucd extract scalars** instead.)

Note that if the input ucd structure has only one component, the **ucd extract** module will pass it to its output automatically.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

node data Selects which of the node's data components to extract. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure, "<no data>" is displayed on the button.

OUTPUTS

UCD structure

The output structure is the same as the input structure, except that the node data is reduced to one component.

EXAMPLE

The following network reads in a UCD structure, extracts one component of the node data, and colors each node based on the value of that component:

```
                       READ UCD
                          |
                          |
GENERATE COLORMAP    UCD EXTRACT
          |               |
          |     |---------|
          UCD CONTOUR     |
                |         |
                |-----|   |
                      |   |
                     UCD TO GEOM
                          |
                          |
                   GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that can be used in place of **ucd extract**:

ucd extract scalars, ucd extract vector

Modules that can process **ucd extract**'s output:

ucd to geom, ucd crop, ucd threshold, ucd hex to tet, ucd anno,
ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d,
ucd legend, ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD EXTRACT demonstrates the **ucd extract** module.

ucd extract scalars

NAME

ucd extract scalars – extract scalar node components from scalar and vector components of a UCD structure

SUMMARY

Name	ucd extract scalars		
Availability	UCD module library		
Type	filter		
Inputs	ucd structure		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	Channel 0	boolean	off
	Channel 1	boolean	off
	Channel 2	boolean	off
		.	
		.	
		.	
	Channel 24	boolean	off

DESCRIPTION

The **ucd extract scalars** module takes a ucd structure that has scalar and/or vector data components at each node, extracts a specified subset of the components, producing an output structure that has only scalar data components at each node. Each element of a selected vector component becomes an individual scalar component.

Each node in a UCD structure may have an arbitrary number of data components associated with it. Furthermore, each of these components itself can be a vector or a scalar. The **ucd extract scalars** module allows you to select both scalar and vector components and it converts all the selected components into scalar components. It is useful when you want to operate on a scalar component of a dataset that has vector components.

ucd extract scalars can handle up to 25 components. You can extract any number of them.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

Channel 0
Channel 1
Channel 2 ...

A series of on/off switches that specify which of the input scalar or vector node components to extract into the output ucd structure. If the input components have been labelled, then their labels will appear instead of the default "Channel *n*". Only as many switches will appear as there are input data components. By default, all of the switches are "off". There is no way to change the order of scalar components in the output structure; if X preceded Y in the input ucd structure, it will do so in the output ucd structure.

OUTPUTS

UCD structure

The output structure is the same as the input structure, except that the node data consists of the selected scalar components and selected vector components converted into scalars.

Labelled input components that were vectors (e.g. vect), will have each output component automatically labelled (e.g., vect1, vect2, vect3).

EXAMPLE

The following network extracts the x, y, and z momentum scalar components from a ucd dataset that has a momentum vector component. It then colors each node based on the value of one of the components:

```
                       READ UCD
                          |
                          |
GENERATE COLORMAP    UCD EXTRACT SCALARS
          |               |
          |     |---------|
         UCD CONTOUR      |
                |         |
                |-----|   |
                      |   |
                     UCD TO GEOM
                          |
                          |
                   GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
ucd extract vector
field to ucd
Any module that outputs a UCD structure.

Modules that can process **ucd extract**'s output:

ucd to geom, ucd crop, ucd threshold, ucd hex to tet, ucd anno,
ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d,
ucd legend, ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD EXTRACT SCALARS demonstrates the **ucd extract scalars** module.

ucd extract vector

NAME

ucd extract vector – extract single vector node component from scalar components of a UCD structure

SUMMARY

Name	ucd extract vector
Availability	UCD module library
Type	filter
Inputs	ucd structure
Outputs	structure

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Vector Length	integer dial	3	1	25
Channel 0	boolean	off		
Channel 1	boolean	off		
Channel 2	boolean	off		
	.			
	.			
	.			
Channel 24	boolean	off		

DESCRIPTION

The **ucd extract vector** module takes a ucd structure that has several scalar data components at each node and extracts a structure that has only one vector data component at each node.

Each node in a UCD structure may have an arbitrary number of data components associated with it. Furthermore each of these components itself can be a vector or a scalar. For example, a UCD structure may have 100 nodes. Each node consists of 5 scalar components, labelled "temperature", "pressure", "velocity_x", "velocity_y", and "velocity_z". The components "velocity_x", "velocity_y", and "velocity_z" are scalar values, but they can be represented as a vector of three values to be used as an input for ucd modules accepting only vector components, such as **ucd hog**, **ucd streamline**, **ucd vecmag**.

ucd extract vector can handle up to 25 scalar components. You can extract any subset of the components.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

Vector Length

An integer dial that specifies the vector length of the *output* ucd structure. The default is 3, the minimum is 1, and the maximum is 25.

Channel 0
Channel 1
Channel 2 ...

A series of on/off switches that specify which of the input scalar node components to extract into the output ucd structure. If the input scalar components have been labelled, then their labels will appear instead of the default "Channel *n*". Only as many switches will appear as there are input scalar components. By default, all of the switches are "off". There is no way to change the order of vector elements; if X preceded Y in the input ucd structure, it will do so in the output ucd structure.

OUTPUTS

UCD structure

The output structure is the same as the input structure, except that the node data is reduced to one vector component.

EXAMPLE

The following network extracts the x, y and z momentum vector elements from a ucd dataset, then plots their sum vector using **ucd hog**

```
     READ UCD
        |
        |
UCD EXTRACT VECTOR
        |
        |
     UCD HOG
        |
        |
  GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that can process **ucd extract vector**'s output:

ucd hog, ucd streamline, ucd vecmag, ucd extract scalars, ucd anno,
ucd offset, ucd probe, ucd streamline

SEE ALSO

The example script UCD EXTRACT VECTOR demonstrates the **ucd extract vector** module.

ucd grad

NAME

ucd grad – compute the vector gradient of a UCD structure

SUMMARY

Name	ucd grad		
Availability	UCD module library		
Type	filter		
Inputs	ucd structure		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	Node Data	choice	<data 1>

DESCRIPTION

The **ucd grad** module computes the gradient of a UCD structure. The input structure should contain scalar node data components. Vector node data components should be convert to scalar with **ucd extract scalars** prior to entering this module.

The output structure has a 3-vector float data component at each node that represents the gradient.

$$gradient(F) = \left\{ \frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right\}$$

This module does *not* normalize the output.

ucd grad is designed for input into the other vector UCD modules.

INPUTS

UCD Structure (required)

The input is a UCD structure containing scalar node data components.

PARAMETERS

Node Data Radio buttons to select which of the node's scalar data components to use in the computation. The buttons show the label attached to each scalar node data component. Before the module receives data, the default "<data 1>, <data 2>,..." is displayed. If there are no scalar components in the node data, **ucd grad** complains. If there are several scalar data components, these buttons let you select which componenet to use in calculating the gradient. If there is no node data in the structure, "<no data>" is displayed on the button.

OUTPUTS

UCD Structure

The output structure has a 3-vector node data component for each input structure node.

EXAMPLE

The following network reads a UCD structure with scalar node data components, computes its gradient and then uses the **ucd hog** module to display the resulting vector structure, together with an isosurface:

```
                       READ UCD
                           |
                           |----------|-------------|
                           |          |             |
                       UCD GRAD       |             |
GENERATE COLORMAP          |          |             |
           |------|-----|  |          |             |
                  |     UCD HOG       |        UCD TO GEOM
                  |        |          |             |
                  |--------|-------|  |             |
                           |      UCD ISO           |
                           |          |             |
                           |----------|-------------|
                                      |
                                      |
                          GEOMETRY VIEWER
```

RELATED MODULES

ucd curl
ucd div
ucd hog
ucd streamlines

SEE ALSO

The example script UCD GRAD demonstrates the **ucd grad** module.

ucd hex to tet

NAME

ucd hex to tet– convert a UCD structure from hexahedral cells to tetrahedral cells

SUMMARY

Name	ucd hex to tet
Availability	UCD module library
Type	filter
Inputs	ucd structure
Outputs	ucd structure

Parameters	*Name*	*Type*	*Default*
	24 Tet	boolean	off
	Node Data	choice	<data 1>

DESCRIPTION

The module **ucd hex to tet** takes a UCD structure with hexahedral cells and converts it to a structure with tetrahedral cells.

To perform the conversion, **ucd hex to tet** must recompute the structure's node connectivity list. Hexahedral cells can be subdivided into 5 tetrahedra or into 24 tetrahedra. When data cannot be properly decomposed into 5 tetrahedra, it needs to be divided into 24 by adding a new node at the center of each face in the cell. These new nodes are added to the UCD structure, and data for them is computed by averaging the values at the corners of the face they are in.

ucd hex to tet is designed to work with the module **ucd tracer**, which performs ray-traced rendering on UCD structures. **ucd tracer** requires that its input structure contain tetrahedral cells.

INPUTS

Structure (required)

The input is a UCD structure which has cells that are hexahedral.

PARAMETERS

24 Tet (boolean)

When **24 Tet** is selected, hexahedral cells are decomposed into 24 tetrahedra, instead of the default, which is 5.

node data Selects which of the node's data components to use . A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button. When the **24 Tet** option is being used to subdivide hexahedral cells into 24 tetrahedra, the **node data** parameter determines which component is used to compute the data associated with the new nodes.

OUTPUTS

Structure The output is a UCD structure which has cells that are tetrahedral.

EXAMPLE 1

The following network reads in a UCD structure, which is converted from hexahedral cells to tetrahedal cells. This structure is then passed to **ucd tracer**. The module **euler transformation** allows you to rotate the volume to produce views from any angle:

```
                           READ UCD
                               |
                               |
                         UCD HEX TO TET
EULER TRANSFORMATION           |
                  |--------|   |
GENERATE COLORMAP          |   |
             |---------|   |   |
                       |   |   |
                       UCD TRACER
                             |
                             |
                      DISPLAY IMAGE
```

EXAMPLE 2

The following network shows how **ucd hex to tet** can be used with modules other than **ucd tracer**.

```
                         READ UCD
                            |
                            |
                       UCD HEX TO TET
GENERATE COLORMAP           |
           |                |
           |     |----------|
           |     |          |
          UCD CONTOUR       |
                 |          |
                 |------|   |
                        |   |
                     UCD TO GEOM
                          |
                          |
                    GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **ucd structure** input:

read ucd
any other module which outputs a hexahedral UCD structure.

Modules that can process **ucd hex to tet**’s output:

ucd tracer

ucd hog

NAME

ucd hog – show UCD node vector values as line segments in 3D space

SUMMARY

Name	ucd hog
Availability	UCD module library
Type	mapper
Inputs	ucd structure colormap upstream transform (optional, invisible, autoconnect)
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
scale	float	0.2	0.0	10.0
arrows	boolean	off		
N segment	integer dial	16	2	64
Sample	choice	point		
Normalize Vectors	boolean	off		

DESCRIPTION

ucd hog takes in a ucd structure whose node values include a 3-vector float component. **ucd hog** interprets the 3 values of the vector as the x, y, z components of a vector in space and then displays these 3D vectors as small line segments with a particular length, direction and color. "hog" is short for "hedgehog", a reference to the bristly appearance of the output geometry vectors.

ucd hog gives you a sample probe, which you can manipulate in the object space of the UCD structure. Vector lines are displayed at a number of sample points (not node points) along the sample line, circle, plane, or space.

Since arbitrarily oriented sample locations do not, in general, coincide with the position of the UCD structure's nodes in space, an interpolation method is used to determine which nodes are nearest to the sample locations.

To move the sample probe, select it by clicking on it with the left mouse button. You can get the same effect by entering the Geometry Viewer, and making the probe object the current object. Then the probe can be moved like any other geometry object. As it moves, **ucd hog** will recompute the line segments it outputs.

Alternatively, you can display hedgehog vectors at each real node location by selecting **node**.

ucd hog only operates on vector components, thus it complains if the input structure has only scalar values at the nodes. If the nodes of a structure have more than one 3-vector component, use the **node data** radio buttons to select which component to use in calculating the hedgehog.

By default, **ucd hog** does not display the vector for every node in the structure. Instead **ucd hog** takes an arbitrarily-oriented (user-controlled) sample of locations within the bounds of the UCD structure. You can choose this sample to be:

- A single point
- A set of points on a line segment

- A set of points on a circle
- A set of points on a plane
- A volume of points

The module outputs the line segment(s) representing the node value at the sample location(s).

To see vectors at every actual node point, select **node**.

ucd hog uses the input colormap to associate a color with each line segment vector based on the magnitude of the vector. The colormap is scaled to the range of values in the structure.

INPUTS

UCD structure (required)
The input data must be a UCD structure. The structure must include a node data component which is a 3-vector of floats to be interpreted as vectors in 3-space.

colormap (required; colormap)
An AVS colormap which is used by **ucd hog** to associate colors with vector values. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd field legend**.

Upstream Transform (optional, invisible, autoconnect)
When the **ucd hog** module coexists with the **geometry viewer** module in a network, **geometry viewer** feeds information on how the point, circle or other sampling probe has been moved back to this input port on the **ucd hog** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **ucd hog**'s sampling probe.

PARAMETERS

node data Selects which of the node's data components to represent as vectors. A set of radio buttons shows shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

Vector Scale
The lengths of the line segments output by this module are proportional to this value.

arrows When **arrows** is selected the line segments are drawn with arrows at their heads, indicating their direction.

N segment An integer value which determines the number of points sampled by the line, circle, plane, or space sampling probe. This controls the density of line segments output by **ucd hog**.

Sample (radio buttons) Specifies the type of sample taken from the vector field: **point**, **line**, **circle**, **plane**, or **space**. If the last choice, **node**, is selected, the structure is not sampled. Rather, vectors are drawn at each real node location.

Normalize Vectors
When **Normalize Vectors** is selected the magnitude of the vectors is not indicated by the length of their line-segment representation. Instead, the vectors are all the same length, and only their color indicates their magnitude.

OUTPUTS

hog (geometry)

The output *geometry* is a collection of line segments representing the 3-vector component of nodes near the sample locations.

EXAMPLE

The following network reads in a UCD structure with a 3-vector float value as one of the components of the node data. **ucd hog** displays the values as line segment vectors. Note that the module **ucd to geom** is used to provide a frame within which to view the hedgehog of vectors. To do this, use **ucd to geom**'s **External Edges** parameter to convert the ucd structure's representation to a wireframe. You can also edit the color properties for this object to make it dimmer and more transparent. This will improve your view of the line segments output by **ucd hog**. You may want to similarly edit the properties of the sample probe, especially if it is a plane.

```
GENERATE COLORMAP       READ UCD
         |                 |
         |--------------|  |-------------|
                        |  |             |
                        UCD HOG       UCD TO GEOM
                           |             |
                           |-------------|
                           |
                    GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
ucd curl
ucd grad
Any module that outputs a UCD structure.

Modules that can process **ucd hog**'s output:

geometry viewer
Any module that inputs an AVS geometry.

Other related modules:

ucd curl
ucd div
ucd grad

SEE ALSO

The example script UCD HOG demonstrates the **ucd hog** module.

NAME

ucd iso – generate an isosurface for a UCD structure with scalar node data

SUMMARY

Name	ucd iso				
Availability	UCD module library				
Type	mapper				
Inputs	ucd structure colormap (optional) info (from **ucd legend**; optional)				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Node Data	choice	<data 1>		
	Iso Level	float	*max+min/2*	*min val*	*max val*
	map scalar	boolean	off		

DESCRIPTION

The **ucd iso** module takes a UCD structure as input. The structure must have at least one component of its node data that is a scalar value. It produces a geometry object that represents an isosurface of this structure. An isosurface is a 3D generalization of a 2D contour line — it connects all structure elements that have the same value. You can use the **Node Data** buttons to select which component of the node data to use when computing the isosurface.

The **Iso Level** value can be set in two ways. The value can be set using **ucd iso**'s floating-point parameter dial. **ucd iso** also can accept an **Info** input from the module **ucd legend**.

By default, the isosurface generated by **ucd iso** is not colored. To color the isosurface, **ucd iso** must receive its optional colormap input, and the **map scalar** parameter must be selected. If the input field has more than one scalar component of its node data, you can use the buttons beneath **map scalar** to select which component's values to use in determining the isosurface's color.

For example, if a structure's node data consisted of three scalars, temperature, pressure, and density, you might compute an isosurface for a given temperature throughout the structure. It would be intuitive to color this isosurface based on the temperature variable. However, it is also possible to color the temperature isosurface using the values of the pressure or density node data, thus indicating the pressure or density that hold for a fixed temperature.

Note: **ucd legend** outputs either a single float value or two float values representing a range. **ucd iso** can only use **ucd legend**'s single float output. Also, when **ucd iso** is connected to **ucd legend**, the selections of **ucd legend**'s node data buttons override **ucd iso** settings.

INPUTS

UCD structure (required)

The input data must be a UCD structure. The structure must include a scalar node data component.

colormap (optional)

An AVS colormap which is used by **ucd iso** to associate colors with the output isosurface. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd field legend**.

Info **ucd iso** can receive input from the module **ucd legend** through its left-most input port. This tells **ucd iso** what the value of the isosurface level should be.

PARAMETERS

node data Selects which of the node's scalar data components to use in constructing the isosurface. A set of radio buttons shows the label attached to each scalar node data component. Before the module receives any data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure, "<no data>" is displayed on the buttons.

Iso Level A floating-point value that specifies the common data value on the isosurface: for each point on the isosurface, the UCD structure's data value equals the **Iso Level** value. Before the module receives data, the dial shows a minimum of 1.0 and a maximum of 9.0. Once data flows into the module, these are reset to the minimum and maximum data values of the selected scalar node data component.

map scalar When the "map scalar" parameter is selected, and the optional colormap input is received, the isosurface that **ucd iso** outputs is colored using the values of the selected node component. By default it is off, and the isosurface is uncolored.

OUTPUTS

Isosurface (geometry)
A shaded surface which represents the isosurface.

EXAMPLE

The following network reads in a UCD structure and generates an isosurface for some node value. The **generate colormap** module provides a colormap to color the isosurface.

```
GENERATE COLORMAP              READ UCD
           |                      |
    |------|    |-----------------|----------|
    |      |    |                 |          |
    |    UCD LEGEND               |          |
    |       |                     |          |
    |-------|-------------------| |          |
            |                   | |          |
            |----------------|  | |        UCD TO GEOM
                             |  | |          |
                             UCD ISO         |
                                |            |
                                |------------|
                                |
                                |
                         GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Module that provides **Color Field** and **Info** inputs:

ucd legend

Modules that can process **ucd iso**'s output:

geometry viewer

SEE ALSO

The sample script UCD ISO demonstrates the **ucd iso** module.

ucd isolines

NAME

ucd isolines – generate isolines on the exterior boundary of a UCD structure

SUMMARY

Name	ucd isolines
Availability	UCD module library
Type	mapper
Inputs	ucd structure colormap (optional) Info (struct_ucd_legend. from **ucd legend**; optional)
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Node Data	choice	<data 1>		
Low Level	float dial			
High Level	float dial			
Isoline Number	integer dial	10	0	100

DESCRIPTION

The **ucd isolines** module takes a UCD structure as input. The structure must have at least one component of its node data that is a scalar value. It produces a geometry object that represents a set of isolines (lines of the constant data value) in the specified range of values on the external boundary of the UCD structure. You can use the **Node Data** buttons to select which component of the node data to use when computing the isosurface.

The **Low Level** and **High Level** values can be set in two ways. The values can be set using **ucd isolines**'s floating-point parameter dials, or **ucd isolines** can accept an Info input from the module **ucd legend**.

By default, the isolines generated by **ucd isolines** are not colored. To color the isolines, **ucd isolines** must receive its optional colormap input.

Note: ucd legend outputs either a single float value or two float values representing a range. **ucd isolines** can only use **ucd legend**'s range output. Also, when **ucd isolines** is connected to **ucd legend**, the selections of **ucd legend**'s **node data** buttons override **ucd isoline**'s settings.

INPUTS

UCD structure (required)

The input data must be a UCD structure. The structure must include a scalar node data component.

colormap (optional)

An AVS colormap which is used by **ucd isolines** to associate colors with the output isolines. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd field legend**.

Info — **ucd isolines** can optionally receive input from the module **ucd legend** through its leftmost input port. This tells **ucd isolines** what the low and high value of the isolines levels should be.

PARAMETERS

Node Data — Selects which of the node's scalar data components to use in constructing the isolines. A set of radio buttons shows the label attached to each scalar node data component. Before the module receives any data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure, "<no data>" is displayed on the buttons.

Low Level A floating-point dial that specifies the low level value of the isolines range.

High Level A floating-point dial that specifies the high level value of the isolines range.

Isoline Number
An integer dial that specifies the number of isolines between low and high levels. Note that if **Isoline Number** is set to 1, the isoline level is controlled by the **Low Level** parameter.

OUTPUTS

Geometry (geometry)
A set of lines that represent isolines.

EXAMPLE

The following network reads in a UCD structure and generates an isolines for some node value. The **generate colormap** module provides a colormap to color the isolines.

```
GENERATE COLORMAP            READ UCD
          |                     |
    |------|    |---------------|----------|
    |      |    |               |          |
    |    UCD LEGEND             |          |
    |       |                   |          |
    |-------|------------------| |          |
            |                  | |          |
            |----------------|  | |       UCD TO GEOM
                             |  | |          |
                           UCD ISOLINES    |
                                |          |
                                |------------|
                                |
                                |
                         GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that provide **colormap** and **Info** inputs, respectively:

generate colormap
ucd legend

Modules that can process **ucd isolines**'s output:

geometry viewer

SEE ALSO

The sample script UCD ISOLINE demonstrates the **ucd isolines** module.

ucd legend

NAME

ucd legend - creates a color legend relating UCD data to a color scale

SUMMARY

Name	ucd legend
Availability	UCD module library
Type	mapper
Inputs	ucd structure colormap
Outputs	range (struct_ucd_legend) color field (field 1D 3-vector real)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
range	boolean	off		
value	float	0.0	0.0	100.0
hi value	float	(not applicable)		
lo value	float	(not applicable)		

DESCRIPTION

The **ucd legend** module performs two functions. First it is used to color unstructured cell data (UCD). To do this it takes in an AVS colormap, and outputs an array of colors — one for each node in the UCD structure.

Second, **ucd legend** creates a "color legend" widget relating UCD data to a color scale. This widget displays the input colormap as a horizontal spectrum. Beneath this color table **ucd legend** prints the range of the node values of the UCD structure. Like a "legend" on a map, the color legend shows you which color represents each value. This widget is used, like a floating-point dial, to pick specific values.

ucd legend works with modules that take UCD structures and allow you to visualize subsets of the data, or specific values in the data. Such modules include: **ucd iso**, and **ucd thresh**. Typically, using a dial, you specify a single value, or a range of values, say from 1.6 to 4.3. With **ucd legend** you can specify the subset by numerical value or, by color range, for example, as ranging from green to blue. Manipulating colored data using **ucd legend**'s color legend is often more intuitive than using a floating-point parameter widget.

By dragging a "radio tuner" dial along the color legend you select a specific value for **ucd legend** to output. If the **range** parameter is selected you can move two radio tuner dials along the color legend to select both minimum and maximum values for the range that **ucd legend** outputs. The middle mouse button controls the maximum dial; the left controls the minimum dial.

Typically a UCD structure has a number of nodes. Each of these nodes may have an arbitrary number of data components associated with it. Furthermore each of these components itself can be a vector or a scalar.

ucd legend only works with scalar node components. By using the **node data** radio buttons you can select a scalar data component for **ucd legend** to use in its color legend. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. When data has been input the labels associated with the node components are displayed on the buttons. If there is no node data in the structure "<no data>" is displayed on the button.

ucd legend prints the scale representing the range of values associated with the selected node component, e.g. temperature, in scientific notation. The input colormap is normalized to the range of values of the selected node component. The label associated with the selected scalar is printed as title of the color legend.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Colormap (required; colormap)

An AVS colormap. **ucd legend** uses the colormap to associate colors with each node in the input UCD structure. The colormap is also used to generate the "color legend" widget.

PARAMETERS

node data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

range A boolean switch. If it is selected **ucd legend** outputs two values representing the minimum and maximum of a range. If it is off, **ucd legend** outputs a single floating point value. By default it is off.

value If the range parameter is not selected, a single floating-point dial appears. This functions in an identical manner to the **ucd legend**'s color widget; you can use it to select specific output values. In particular you can use the dial to type in specific values, by opening the dial's Dial Editor.

lo value
hi value If the range parameter is selected, two floating point dials appear. Using them you can specify the minimum and maximum values of the range that **ucd legend** outputs. The values shown on these dials are scaled to the range of values present in the input structure.

OUTPUTS

Selection (struct_ucd_legend)

ucd legend outputs either a single floating-point value, or two values representing the minimum and maximum of a range.

Color Field (field 1D 3-vector real; optional)

The color field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green and blue. Note that the **Color Field** is not the same as an AVS colormap. This output is usually connected to the **ucd to geom** module's matching input port. **ucd contour** can also be used to generate Color Fields.

EXAMPLE 1

The following network reads in a UCD structure. **ucd legend**'s leftmost output port generates a structure specifying either a single isosurface level, or a range of numbers. **ucd iso** can only use the single level value, not the range. The level can be set using **ucd legend**'s dials, or with the mouse and **ucd legend**'s colored value selection widget. This structure is input to the **ucd iso** module. The resulting isosurface is uncolored.

```
GENERATE COLORMAP     READ UCD
          |               |
          |  |------------|
          |  |            |
      UCD LEGEND          |
        |                 |
        |--------------|  |
                       |  |
                      UCD ISO
                         |
                         |
                  GEOMETRY VIEWER
```

EXAMPLE 2

This example has the same structure as the previous example. Three elements have been added. There is now a **ucd to geom** module. This will produce a picture of the original ucd structure. Toggle **External Edges** on the **ucd to geom** control panel so that the structure does not obscure the isosurface. Second, **ucd legend** now sends a Color Field on its rightmost output port to **ucd to geom**'s leftmost input port. This colors the ucd structure. Third, **ucd iso** takes a colormap input from **generate colormap**. This will color the isosurface itself by the value of one of the node data components, as selected with **ucd iso**'s **Map Scalar** controls.

```
GENERATE COLORMAP        READ UCD
          |                  |
          |  |---------------|-----------|
          |  |               |           |
          |---------------|  |           |
          |  |            |  |           |
      UCD LEGEND          |  |           |
        |    |            |  |           |
        |    -------------|--|-------|   |
        |--------------|  |  |    UCD TO GEOM
                       |  |  |          |
                      UCD ISO           |
                          |             |
                          |-------------|
                          |
                  GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD Structure** input:

read ucd
field to ucd
any other module which outputs a UCD structure

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd legend**'s output:

ucd iso
ucd thresh
ucd to geom

Modules that can produce Color Fields:

ucd contour

SEE ALSO

The example script UCD THRESHOLD, as well as others, demonstrates the **ucd legend** module.

ucd math

NAME

ucd math – perform math operations between UCD structures

SUMMARY

Name	ucd math				
Availability	UCD module library				
Type	filter				
Inputs	ucd structure ucd structure (optional)				
Outputs	ucd structure				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	choice	choice	none		
	Constant	float typein	0.0	*unbounded*	*unbounded*

DESCRIPTION

The **ucd math** module performs unary and binary operations upon UCD structures. It works with both node and cell data. It operates across all data components, scalar or vector, that are present in the UCD structure.

The unary operations are **+, -,** *, **/, Square, Sqrt, Pow(er),** and **Log.** The binary operations are **+, -,** *, and **/**.

When the right input port is connected, the unary operations appear as choices, plus a typein for a **Constant**. When the left input port is also connected, only the binary operation choices appear.

The input structures must be identical: they must have the same number of cells and nodes, and the cell and node data components must be the same length.

INPUTS

UCD structure (required)

The rightmost input field is used as the input to unary operations, or the first operand for binary operations.

UCD structure (optional)

The left structure is the second operand in binary operations. It must have the same number of cells, number of nodes, cell data component length, and node data component length as the first UCD structure.

PARAMETERS

+
-
*
/
Square
Sqrt
Pow(er)
Log
NONE

A choice of operations. If the left port structure (struct2) is not provided, the **Constant** parameter is used as the second operand. I.e., struct2 is replaced by **Constant**.

```
+               struct1  +  struct2
-               struct1  -  struct2
*               struct1  *  struct2
/               struct1  /  struct2 (result is 0 if struct2 is 0)
Square          struct1 * struct1
```

```
Sqrt            sqrt(struct1)
Pow(er)         struct1 ** Constant
Log             log e (struct1)
```

Constant A floating point typein to specify the constant value to be used as the second operand when there is just one input. If two structs are connected to the module, **Constant** is ignored, and it disappears from the control panel. The default is 0.0. There is no upper or lower limit.

OUTPUTS

UCD Structure

The output structure has the same form as the input structure.

EXAMPLE 1

This example performs a mathematical operation on a UCD structure. The result is mapped as an isosurface superimposed upon the picture of the UCD structure produced by **ucd to geom**. The two **ucd extract** modules extract single components from the node data. Without these modules, **ucd math** would operate across all of the data in the input structures, not just the components of interest, thus using more memory and computation time.

```
                     READ UCD
                        |
               |--------|-----------|---------------|
               |                    |               |
          UCD EXTRACT          UCD EXTRACT          |
               |                    |               |
               |--------|    |------|               |
                        |    |                      |
                      UCD MATH               UCD TO GEOM
  GENERATE COLORMAP        |                        |
            |----------|   |                        |
                       |   |                        |
                     UCD ISO                        |
                           |------------------------|
                           |
                 GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** inputs:

Any module that outputs a UCD structure.

Modules that can process **ucd math** output:

Any module that inputs a UCD structure.

SEE ALSO

The UCD MATH example script demonstrates the **ucd math** module.

ucd minmax

NAME

ucd minmax – set min and max values of a component in a UCD structure

SUMMARY

Name	ucd minmax				
Availability	UCD module library				
Type	filter				
Inputs	ucd structure				
Outputs	ucd structure min value (float) max value (float)				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	node data	choice	<data 1>		
	min value	float typein	0.0	*unbounded*	*unbounded*
	max value	float typein	0.0	*unbounded*	*unbounded*

DESCRIPTION

The **ucd minmax** module modifies the minimum and maximum values of a selected scalar node data component in a UCD structure. The output UCD structure is identical to the input structure, except for the new component minimum and maximum values. **ucd minmax** also outputs the minimum and maximum values of the selected component to its output ports.

The **ucd minmax** module has two main purposes:

- It can be used to provide min and max inputs to the **generate colormap** module's **lo value** and **hi value** parameters. These in turn will output a scaled colormap to the **color legend** module, making **color legend** useable with UCD data.
- It can be used to set the mininum / maximum range for animating of a sequence of datasets with different minimum and maximum values (such as a time-series). In this application, setting a wide enough range will prevent modules such as **ucd iso**, **ucd legend**, etc., from resetting their parameters every time a new dataset is read.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

node data Selects which of the node's data component's min / max is being edited. A set of radio buttons shows the label attached to each node data component. Only scalar components are shown; vector components will need to be converted to scalar with **ucd extract scalars**. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If components are labeled, the labels will appear on the buttons instead. If there is no node data in the structure, "<no data>" is displayed on the button. The first component is the default.

min value A floating-point typein value that specifies a new minimum value for a selected node data component of an input ucd structure. By default it is set to a "real" minimum value of the data component. If a new dataset having the same component name is read the parameter value is not updated.

max value A floating-point typein value that specifies a new maximum value for a selected node data component of an input ucd structure. By default it is set to a "real" maximum value of the data component. If a new dataset having the same component name is read the parameter value is not updated.

OUTPUTS

UCD structure

The output structure is the same as the input structure, except that the component's node data minimum and maximum values are reset to the parameter **minimum** and **maximum** values.

EXAMPLE

The following network reads in a UCD structure, sets the min / max values for a data component, which are used by **generate colormap** and **ucd contour** modules. **generate colormap**'s **lo value** and **hi value** parameter ports must be made visible before they can be connected to **ucd minmax**. To do this, bring up **generate colormap**'s Module Editor, click on the **lo value** parameter button, and then click on **Port Visible** on the resultant Parameter Editor panel. Repeat for **hi value**.

```
                 READ UCD
                    |
               UCD MINMAX
               |    |   |
     |---------|    |   |
     |      |-------|   |
     |      |           |
GENERATE COLORMAP       |
        |               |-----|------|
        |                     |      |
        |----------------|    |      |
        |                |    |      |
  COLOR LEGEND       UCD CONTOUR     |
        |                     |      |
        |                     |      |
        |                  UCD TO GEOM
        |                          |
        |-----------|-------------|
                    |
                    |
             GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that can process **ucd minmax**'s output:

generate colormap, ucd contour, ucd legend, ucd iso, ucd isolines,
ucd rslice, ucd slice2d, write ucd

SEE ALSO

The example script UCD MINMAX demonstrates the **ucd minmax** module.

ucd offset

NAME

ucd offset – deform a UCD structure based on vector values at each node

SUMMARY

Name	ucd offset				
Availability	UCD module library				
Type	filter				
Inputs	ucd structure				
Outputs	ucd structure				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	offset factor	float	1.0	*unbounded*	*unbounded*
	Node Data	choice	<data 1>		

DESCRIPTION

ucd offset "physically" deforms a ucd structure based upon the values at each of the structure's nodes.

The nodes of a UCD structure may contain several data components. Each of these components may itself be either a vector or a scalar value. **ucd offset** only operates on vector components, thus it complains if the input structure has only scalar values at the nodes. If the nodes of a structure have more than one 3-vector component, then use the **node data** radio buttons to select which component to use in calculating the deformation.

ucd offset takes the selected 3-vector component of each node and uses the three elements of that vector to translate the node in space. The first element of the vector translates the node's x coordinate, the second translates the y coordinate, and the third translates the z coordinate. The magnitude of each translation is proportional to the values at the nodes scaled by an **offset factor** between 0.0 and 1.0.

For example, if an unstructured cell dataset has a node component which is a 3-vector of values representing velocity in the x, y, z directions, **ucd offset** translates the x, y, and z location of each node proportional to the velocity values at that node.

INPUTS

UCD Structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

offset factor
A floating point value that is used to scale the magnitude of the deformation.

Node Data A set of radio buttons shows the label attached to any vector components of the node data. Before the module receives data, the default "<data 1>, <data 2>,..." is displayed. If there are no vector components of the node data **ucd offset** complains. If there are several vector data components, these buttons let you select which component to use in calculating the offset of the UCD structure.

OUTPUTS

UCD Structure
The output structure is the deformed UCD structure.

EXAMPLE

The following network reads in a UCD structure and deforms it, then displays the result:

```
     READ UCD
        |
        |
    UCD OFFSET
        |
        |
    UCD TO GEOM
        |
        |
  GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that can process **ucd offset**'s output:

ucd extract, ucd extract vector,
ucd to geom, ucd crop, ucd threshold, ucd anno,
ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d,
ucd legend, ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD OFFSET demonstrates the **ucd offset** module.

ucd plot

NAME

ucd plot – create a field to graph a linear sample through a UCD structure

SUMMARY

Name	ucd plot
Availability	UCD module library
Type	mapper
Inputs	ucd structure colormap (*optional*) upstream transform (*optional, invisible, autoconnect*)
Outputs	field 2D scalar real uniform geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Node Data	choice	<data 1>		
Abscissa Mapping	choice	Dist		
N Segment	int dial	20	2	1000

DESCRIPTION

The **ucd plot** module samples the node data along a line through a UCD structure, producing a 2D field that is used as input to the **graph viewer** module's rightmost (XY plot) port. The line is represented by a linear sampling object. The Y axis plots the data values in the structure against an X axis that can be either the distance along the linear sampling object, or the linear sampling object's points projected upon the UCD structure's X, Y, or Z object coordinates.

ucd plot represents the linear sampling object as a line geometry that is output to the **geometry viewer** module. The linear sampling object can be moved through the volume of the UCD structure using **geometry viewer** direct manipulation.

INPUTS

UCD structure (required)

The input data is a UCD structure with scalar node data components.

colormap (optional)

An AVS colormap. This colormap colors the linear sampling object in the **geometry viewer** by the data values encountered. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd legend**.

upstream transform *(optional, invisible, autoconnect*)

The **upstream transform** port receives the sampling object transformation (movement) information from the **geometry viewer** module, allowing **ucd plot** to track the user's placement of the sampling object. This port is normally invisible.

PARAMETERS

Node Data Selects which of the node's scalar data components to sample. This is a set of radio buttons that shows the label attached to each scalar node data component. Before the module receives any data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure, "<no data>" is displayed on the buttons. Vector components should first be converted to scalar components with **ucd extract scalars**.

Abscissa Mapping

This controls which values are used to construct the X axis of the output plot. **Dist** represents the simple distance of the linear sampling object through the UCD structure. For example, a structure extending from 0,0 to 0,2 would produce an X axis extending from 0 to 2. If **X**, **Y**, or **Z** is selected, then the **N Segment**s along the linear sampling object are projected down to the UCD structure's X, Y, or Z axis.

N Segment An integer dial that controls the resolution of the linear sampling object. The range extends from 2 to 1000, with a default of 20.

OUTPUTS

Data Field (field 2D scalar real uniform)

This 2D field is the "Plot as XY" data input to the **graph viewer** module's rightmost input port. It can be viewed as a two-column table of X-Y pairs that is **N Segments** long. The first column is location of a data value as determined by **Abscissa Mapping**. It is used as the X values in the output plot. The second column is the data values. It is used as the Y values in the output plot.

geometry A geometry representing the linear sampling object, initially centered within the extents of the UCD structure along the X axis.

EXAMPLE

The following network reads in a UCD structure and generates a plot of its data values. **ucd to geom** creates a colored representation of the UCD structure. **ucd plot** then produces the sampling object, which is superimposed over the UCD structure in the **geometry viewer** window. The **generate colormap** module provides a colormap that colors both the UCD structure and the sampling object. The **graph viewer** window displays the resulting plot.

```
                      READ UCD
                          |
    GENERATE COLORMAP     |
            |             |
            |--------|    |
            |   |----|----|
        UCD TO GEOM  |    |
             |       |    |
             |     UCD PLOT
             |        |   |
             |--------|   |------|
             |                   |
GEOMETRY VIEWER         GRAPH VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Module that provides **Colormap** inputs:

generate colormap

Modules that can process **ucd plot**'s output:

graph viewer (field)
print field (field)

statistics (field)
geometry viewer (geometry)

SEE ALSO

The example script UCD PLOT demonstrates the **ucd plot** module.

NAME

ucd print – create a readable format of a UCD structure.

SUMMARY

Name	ucd print
Availability	UCD module library
Type	data output
Inputs	ucd structure
Outputs	none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Output File	typein	*/tmp*		
Component	integer dial	-1	-1	*unbounded*
Start node	integer dial	-1	-1	*unbounded*
Start cell	integer dial	-1	-1	*unbounded*
Display Mode	choice	*none*		

DESCRIPTION

The **ucd print** module creates a human-readable version of the contents of an AVS ucd structure. The information is displayed in a Node Browser widget on the AVS control panel. **ucd print** is useful whenever you need to inspect the actual contents of a ucd structure.

By default, **ucd print** displays UCD structure header information. The control panel also contains a radio-button selector that allows you to display different additional pieces of the ucd data in the Node Browser window. The selection possibilities are: Node Data, Node positions, cell lists, node lists, and cell info. Each of these selections is explained under the **Display Mode** parameter below.

The header consists of the following information, as returned by the **UCDstructure_get_data** routine (*AVS Developer's Guide*, "Unstructured Cell Data Library" appendix): ucd name, data vector length, name flag, number of cells, cell vector length, cell mix, number of nodes, node vector length, node mix, util flag, XYZ extents, and the ranges for each node component and cell component, if present.

The cell mix and node mix are the vector lengths of the individual components of the cell or node data. The sum of these lengths will be the node data vector length (or the cell data vector length). The util flag is the util_flag field in the ucd struct as defined in *ucd_defs.h*. The X,Y, and Z extents are the extents of the mesh portion of the ucd (node positions). The node component and cell component ranges are ranges on the values stored either in the node or cell data sections of the ucd. **ucd print** does not calculate the ranges; they appear only if some upstream module has calculated them. In the language of analysis, the XYZ range can be thought of as the dimensions of the smallest box that will contain the domain of the function represented by the ucd, while the node/cell ranges are the dimensions of the smallest n-dimensional box that contains the image of the function represented by the ucd.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format

PARAMETERS

Output File

A typein that determines the temporary file used by **ucd print** to cache the browser info. This file can be changed by the user if storage on */tmp* is a problem for any reason.

Component
This parameter selects the data component to display. It is an integer dial.

Start node Selects the starting node for which to display the data. The node selected will be the first one placed in the browser window. Ten nodes are displayed at a time.

Start cell Selects the starting cell for which to display the data. The cell selected will be the first one placed in the browser window. Ten cells are displayed at a time.

Display Mode
This parameter is a radio button that selects the display mode. The choices are:

Node data
The **Node data** selection displays the data associated with the ucd nodes. The component selected by the component dial will be displayed in the browser along with the vector length of the component, its units, and the data itself. If the ucd only contains cell data, this information may not be available.

Node positions
The **Node positions** selection displays the node positions in XYZ coordinates. This data is always present in a UCD.

cell list
The cell list is the connectivity information.

node list
The node list information is the list of nodes comprising each cell in the ucd.

cell info
The cell info selection allows the user to display the material type, individual cell names, cell types and mid_edge flag for each cell in the ucd being examined.

EXAMPLE

The following network displays the contents of the ucd structure:

```
READ UCD
    |
    |--------------|
    |              |
UCD PRINT      WRITE UCD
```

The Node Browser widget is usually too narrow. To resize it: enter the Network Editor. Press **Layout Editor** on the Network Editor menu. The browser widget will be bordered in red. Move the mouse into it. Use your window manager to move the widget as though it were a window to *outside* the control panel. Release the mouse buttons, then resize the Node Browser widget like any other window. Leave the Layout Editor.

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
ucd extract
Any module that outputs a UCD structure.

The **print field** module performs a similar function for fields.

SEE ALSO

The example script UCD PRINT demonstrates the **ucd print** module.

ucd probe

NAME

ucd probe – interactively show numeric data values in a geometry rendered UCD structure

SUMMARY

Name ucd probe

Availability UCD module library

Type mapper

Inputs ucd structure
color field *(field 1D 3-vector float; optional)*
upstream transform *(optional, invisible, autoconnect)*
upstream geometry *(optional, invisible, autoconnect)*

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
x	float typein	0.0	*min-extent*	*max-extent*
y	float typein	0.0	*min-extent*	*max-extent*
z	float typein	0.0	*min-extent*	*max-extent*
Node Data	choice	<data 1>		
Probe Type	choice	Cursor		
Pick Geometry	boolean	off		
label nodes	boolean	off		
label id	boolean	off		
label value	boolean	off		
label cell	boolean	off		
Text Size	integer dial	2	1	7
Text Offset	float dial	0.0	-10.0	10.0

DESCRIPTION

The **ucd probe** module displays the numeric data values associated with the nodes of a specific cell in a UCD structure. It works for structures that have been rendered as AVS geometries.

ucd probe works by creating a cursor-like object titled "probe" that coexists in the Geometry Viewer window with the rendered version of the UCD structure. Its initial position is aligned with the first cell in the structure.

The **ucd probe** module lets you see the values in a UCD structure in three ways:

Typein values

You can specify an explicit **x**, **y**, and **z** cell location by typing into these parameters. The probe object will move to this location within the UCD structure and display the node data values for that cell.

Pick Geometry

If **Pick Geometry** is selected, then you simply point the mouse at the cell you are interested in and click the left mouse button. The probe object will "snap" to the UCD cell which is below the mouse cursor and display the node data values for that cell. This is a "point the mouse and click" technique of data sampling.

Follow Mouse

In the third method, the probe must be the Geometry Viewer's current object. Then, with **Pick Geometry** off (not selected), you use the right and shift right mouse buttons to move the probe object around the volume of the UCD structure. The probe "follows" the cursor around the display window, continuously reporting its position and the specified values of cells it passes through.

The Geometry Viewer tells the **ucd probe** module which UCD cell the mouse cursor is over as the buttons are pressed. **ucd probe** then reports the data values of the nodes which make up the vertices of the selected cell.

It is usually helpful to view the selected cell together with a wireframe rendering of the structure it belongs to. This can be achieved by adding the module **ucd to geom** to your network. **ucd to geom** outputs the entire UCD structure as a geometry object. By setting **ucd to geom**'s Geometry Display Mode to **External Edges**, you can produce a wireframe representation of the structure. The example network below demonstrates this. You can also use the **ucd crop** module to expose interior cells to make them easier to click upon. You can use **ucd probe**'s **Text Offset** parameter to move the labels away from the cell/node. If your platform supports transparency, you can use the Geometry Viewer's **Edit Properties** window's **Transparency** slider to make the UCD object semi-transparent.

INPUTS

UCD structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

Color Field (field 1D 3-vector real; optional)
The color field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green and blue. The color field input is used by **ucd probe** to render the geometry object which represents the selected UCD cell. Two modules output the color field data type, **ucd contour** and **ucd legend**. Note that the color field is not the same as an AVS colormap.

Upstream Transform (optional, invisible, autoconnect)
Used by **ucd probe**'s continuous tracking technique, this normally invisible port is what the **geometry viewer** module uses to inform **ucd probe** of the location of the probe in space so it can display the data value for it. The module connection occurs automatically.

Upstream Geometry (optional, invisible, autoconnect)
Used by the **ucd probe**'s "point cursor and click" technique, this normally invisible port is what the **geometry viewer** module uses to inform **ucd probe** of the geometry vertex selected so it can display the data value for it. The module connection occurs automatically.

PARAMETERS

x
y
z Floating point typeins that specify where, in the coordinate system of the UCD structure, the sampling should be taken. Setting these will move the probe object to this location, or, alternatively, they will display the location of the probe object if it is moved manually. The initial value is 0.0 in **x**, **y**, and **z**. The minimum and maximum values are restricted to the extents of the UCD structure.

Node Data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button.

Probe Type

A set of radio buttons that control what the probe object looks like in the Geometry Viewer.

Cursor

creates a probe that looks like a miniature XYZ axis.

Crosshair

creates a probe that looks like half of a miniature XYZ axis. The crosshair stays aligned with the axis, and its endpoints lie in the XY, YZ, and XZ planes.

Probe

creates a probe that looks like an electronic probe or dissecting needle.

label nodes

Marks the nodes of a picked cell as small x's.

label id When **label id** is selected, the integer or string node id which identifies the nodes is displayed.

label value When **label value** is selected the floating point value associated with one data component of a node, as determined by **Node Data**, is displayed.

label cell Displays the picked cell as a separate geomtry object colored by nodal values using the color field input.

Text Size An integer dial that controls the font size of the output strings.

Text Offset

A floating point dial that offsets the text from the UCD node, making it easier to read. The default is 0.0 (no offset); the min is -10.0 and the max is 10.0.

OUTPUTS

Geometry (geometry)

The output geometry has three parts:

The rendering of the UCD cell that was selected,

The rendering of the "probe" object,

The rendering of the "Text for Probe" that lists the data values and coordinate position.

EXAMPLE

The following network reads in a UCD structure with scalar component values, (e.g., *$AVS_PATH/data/ucd/scalar.1000.inp*) which is passed both to **ucd to geom** and to **ucd probe**. The **ucd probe** outputs a geometry object representing the cell that has been selected. The **ucd to geom** outputs the entire UCD structure. By setting the representation mode for the entire structure to **External Edges**, you can produce a rendering of the selected cell within a wireframe model of the structure:

```
GENERATE COLORMAP          READ UCD
          |                    |
          |  |-----------------|------------|
     UCD CONTOUR               |            |
            |                  |            |
            |-------------|--- | --------|  |
                          |    |         |  |
                       UCD TO GEOM     UCD PROBE
                               |            |
                               |-------|------|
                                       |
                                       |
                                 GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read ucd
field to ucd

Modules that could provide the **Color Field** input:

ucd contour
ucd legend

Modules that can process **ucd probe** output:

geometry viewer

SEE ALSO

ucd anno module

The example script UCD PROBE demonstrates the **ucd probe** module.

ucd reverse cell

NAME

ucd reverse cell – repair topology of imported UCD structures; reverse cell normals

SUMMARY

Name	ucd reverse cell		
Availability	UCD module library		
Type	filter		
Inputs	ucd structure		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	choice	choice	Correct Topology
	Reverse TRIANGLE	boolean	off
	Reverse QUADS	boolean	off
	Reverse TETRAS	boolean	off
	Reverse HEXAS	boolean	off
	Reverse PRISMS	boolean	off

DESCRIPTION

AVS's UCD structure defines a particular ordering of the nodes that make up cells (see "Unstructured Cell Data" appendix in the *AVS Developer's Guide*). Other parties' UCD structures, though they may support the same cell types, have a different node ordering. When such a dataset is imported into an AVS UCD structure without correcting the node ordering, the structure of the individual UCD cells and the dataset's overall structure appear correct. However, because the cell is effectively inside-out, the cells' normals will be wrong. That is, though the cell has the right shape, it appears as a featureless gray outline of a cell in the structure that is unaffected by coloring with **ucd contour**, or by Geometry Viewer lighting. The entire structure may be incorrect, or just individual cells. The geometry output looks "wrong"; it is full of gray "nothing" holes.

ucd reverse cell corrects these mistakes in cell topology. It will either traverse the entire structure, reordering all incorrect cells to match AVS's ordering (**Correct Topology**), or it will reverse the normals on all cells of a particular type (**Reverse Cell**). The repaired UCD structure can be saved permanently with **write ucd**.

ucd reverse cell has another use. Because it reverses cell normals, it can also be used to make isolines that are obscured by cell surfaces visible.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

choice A set of radio buttons that chooses the basic operation of the module. There are two choices.

Correct Topology

Causes **ucd reverse cell** to correct the ordering of nodes. "Correcting" means swapping the node ordering by what is *most likely* wrong with it, since only a few basic node orderings are in common use—they are never totally random. If **Correct Topology** is selected, then **ucd reverse cell** uses the Jacobian matrix determinant of each cell to determine if the cell has the right topology. If it is wrong, it swaps the nodes:

```
hexahedral:    4567 0123  -->  0123 4567
tetrahedral:        0123  -->       0213
prism:          345  012  -->   012  345
```

Triangle and quadrilateral cells are not adjusted.

Reverse Cell

A switch that causes **ucd reverse cell** to change the node ordering. You must select which types of cells to reverse. More than one can be selected. No effort is made to determine correctness; all of the nodes are swapped.

Reverse TRIANGLE

210 --> 012

Reverse QUADS

3210 --> 0123

Reverse TETRAS

0123 --> 0231

Reverse HEXAS

4567 0123 --> 0123 4567

Reverse PRISMS

345 012 --> 012 345

OUTPUTS

UCD Structure

The output is a UCD structure identical to the input structure except for its node ordering.

EXAMPLE 1

The following network corrects a UCD structure.

```
                        READ UCD
                           |
   GENERATE COLORMAP       |
            |              |
       UCD CONTOUR     UCD REVERSE CELLS
            |              |
            |-----------|  |------------|
                        |  |            |
                  UCD TO GEOM       WRITE UCD
                           |
                GEOMETRY VIEWER
```

EXAMPLE 2

This network makes the lines produced by **ucd isolines** visible if they are obscured by the UCD structure because the "top" of the cell is away from the viewer.

ucd reverse cell

```
GENERATE COLORMAP              READ UCD
           |                      |
    |------|    |-----------------|----------|
    |      |    |                 |          |
    |    UCD LEGEND      UCD REVERSE CELL    |
    |       |                     |          |
    |-------|-------------------| |          |
            |                   | |          |
            |----------------|  | |       UCD TO GEOM
                             |  | |          |
                            UCD ISOLINES     |
                                |            |
                                |------------|
                                |
                                |
                         GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD Structure** input:

read ucd
Any module that outputs a UCD Structure.

Modules that can process **ucd reverse cell**'s output:

any module that inputs a UCD structure

SEE ALSO

The example script UCD REVERSE CELL demonstrates the **ucd reverse cell** module.

NAME

ucd rslice – slice away portions of a UCD structure

SUMMARY

Name	ucd rslice
Availability	UCD module library
Type	mapper
Inputs	ucd structure color field (field 1D 3-vector real)
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Do Slice	boolean	off		
x-rot	float	0.0	0.0	360.0
y-rot	float	0.0	0.0	360.0
Distance	float	0.0	-2.0	2.0

DESCRIPTION

The **ucd rslice** module cuts through a UCD structure along an arbitrarily positioned slice plane. **ucd rslice** outputs the structure minus the portions that have been sliced away. The slice plane can be rotated around the x and y axes, and moved back and forth along the normal to the plane. Note that to initiate the slicing operation you must press the "Do Slice" button.

ucd rslice is similar to the modules **ucd crop** and **ucd threshold**, which also subset UCD structures. However, these two modules cut away the cells that make up the UCD structure; they do not cut *through* cells. **ucd rslice**, on the other hand, slices through any cells which the slice plane intersects. When you slice through hexahedral cells, for example, you may produce cells that do not look like hexahedrons. This is especially true if the UCD structure is being rendered as a wireframe.

By default, the UCD structure is placed at the origin and the slice plane is in the X-Z plane. The orientation of the slice plane is controlled by two floating point parameter dials, **x-rot** and **y-rot**. If you rotate the slice plane, you will see that one side has a highlighted area. The highlighted surface is on the side that will be removed.

Each time the slice plane is reoriented the boolean parameter **Do Slice** is turned off. This lets you adjust the slice plane until it is where you want, and only then perform the slicing operation. The slice plane can be moved back and forth through the UCD structure along the normal to the plane, using the **Distance** floating-point dial. This lets you take a series of parallel slices through a UCD structure in any direction.

INPUTS

Structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Color Field (field 1D 3-vector real)

This input field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green, and blue. Note that the color field is not a regular AVS colormap. Two modules output color fields: **ucd contour** and **ucd legend**.

PARAMETERS

Do Slice A boolean switch that initiates the slicing operation. This button allows you to manipulate the slice plane until you are satisfied with its position, and only then slice the UCD structure.

x-rot A floating point value which rotates the slice plane around the UCD structure's x axis.

y-rot A floating point value which rotates the slice plane around the UCD structure's y axis.

Distance A floating point value between -2.0 and 2.0 which moves the slice plane back and forth in the direction of the normal to the plane. This value is scaled by the largest dimension of the UCD structure. Consequently, you can move the slice plane along the normal from -(2 * max dimension) to (2 * max dimension).

OUTPUTS

Geometry **ucd rslice** outputs a geometry which includes the slice plane, and the portion of the UCD structure remaining after the slice operation is performed.

EXAMPLE

The following network reads in a UCD structure and slices it. The **ucd rslice** module outputs the sliced structure and the slice plane as geometries, which are rendered using **geometry viewer**:

```
GENERATE COLORMAP      READ UCD
          |               |
          |     |---------|
         UCD CONTOUR      |
                |         |
                |-----|   |
                      |   |
                UCD RSLICE
                      |
                      |
              GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
ucd_extract
Any module that outputs a UCD structure.

Other modules that subset UCD structures:

ucd threshold
ucd crop

Modules that can process **ucd rslice**'s output:

geometry viewer, render geometry

SEE ALSO

The example script UCD RSLICE demonstrates the **ucd rslice** module.

NAME

ucd rubber sheet – map values as a 3D surface with height proportionate to value

SUMMARY

Name	ucd rubber sheet
Availability	UCD module library
Type	mapper
Inputs	ucd structure colormap (*optional*)
Outputs	geometry (sampling plane) geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Node Data	choice	<data 1>		
Do Slice	toggle	off		
x-rot	float dial	0.0	0.0	360.0
y-rot	float dial	0.0	0.0	360.0
Distance	float dial	0.0	-1.0	1.0
Offset	float dial	0.0	-1.0	1.0
Scale	float dial	1.0	-10.0	10.0

DESCRIPTION

ucd rubber sheet maps node data component values as a 3D surface. **ucd rubber sheet** produces a plane sampling object that can be positioned anywhere in the volume of a UCD structure using **x-rot**, **y-rot**, and **Distance** dials. Once positioned, pressing **Do Slice** creates a 3D output "rubber sheet" geometry surface. The output surface is created by offsetting the points on the sampling plane by a distance that is proportional to the data values (interpolated) through which the sampling plane passes. The output surface reflects these values in two ways:

1. The surface's color is mapped according to the optional colormap.
2. The surface's offset from the sampling plane is scaled linearly to the data value (hence the name "rubber sheet"). For example, a sampling plane passing through these values:

   ```
   10 5 10
   5  0  5
   10 5 10
   ```

   would produce a squared-off concave "bowl" shape, with the bottom of the bowl at 0, the bowl's corners stretched 10x's away from the bowl bottom, and the centers of the edges of the bowl stretched half as far away from the bowl bottom as the bowl corners.

ucd rubber sheet thus uses the third dimension to illustrate the magnitude of the differences between node values. The height used is always scaled so that the output surface will fit within the volume of the UCD structure. You may multiply the resulting height by a dial-controlled **Scale** factor.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Colormap (optional; colormap)

An AVS colormap. **ucd rubber sheet** maps node values in the input structure to colors in the colormap.

ucd rubber sheet

PARAMETERS

Node Data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there is no node data in the structure "<no data>" is displayed on the button. Vector node data components should be converted to scalar with **ucd extract scalars**.

Do Slice Once the sampling plane is positioned, press **Do Slice** to generate the 3D surface.

x-rot A floating point dial that rotates the sampling plane around the structure's X axis. The range is 0.0 to 360.0; the default is 0.0.

y-rot A floating point dial that rotates the sampling plane around the structure's Y axis. The range is 0.0 to 360.0; the default is 0.0.

Distance A floating point dial that controls the Z axis position of the sampling plane. The range is -1.0 to 1.0; the default is 0.0 (centered).

Offset A floating point dial that controls how far away the new 3D surface will appear from the original sampling plane. The range is -1.0 to 1.0; the default is 0.0 (on the original sampling plane).

Scale A floating point dial that controls the height distortion. Internally, **ucd rubber sheet** creates a scaling factor (internal_scale) that will keep the rubber sheet within the extents of the UCD structure. This **Scale** parameter is used to multiply the final result:

(internal_scale * value) * Scale

The range is -10.0 to 10.0; the default is 1.0.

OUTPUTS

geometry (geometry)

This is the sampling plane.

geometry (geometry)

This is the 3D "rubber sheet" surface. It is generated or re-generated whenever **Do Slice** is pressed.

EXAMPLE

The following network reads in a UCD structure. This is fed to **ucd contour**, **ucd to geom** and **ucd rubber sheet**. **ucd to geom** uses it to create a colorized picture of the original UCD structure. **ucd rubber sheet** uses it to create its sampling plane and the 3D colorized surface. Both feed into the **geometry viewer**. In order to see the sampling object and the 3D surface, you should toggle **External Edges** on **ucd to geom**'s control panel.

```
GENERATE COLORMAP             READ UCD
          |                       |
          |     |------|----------|
          |-----|------|------|   |
          |     |      |      |   |
          UCD CONTOUR  |      |   |
                  |    | UCD RUBBER SHEET
                  |    |         |
                  |    |         |
               UCD TO GEOM       |
                        |--------|
                                 |
                      GEOMETRY VIEWER
```

RELATED MODULES

ucd rubber sheet is roughly the UCD equivalent to the field data module **field to mesh**.

Modules that could provide the **UCD Structure** input:

read ucd
ucd crop
ucd threshold
Any module that outputs a UCD Structure.

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd rubber sheet**'s output:

geometry viewer

SEE ALSO

The example script UCD RUBBER SHEET demonstrates the **ucd rubber sheet** module.

ucd slice 2D

NAME

ucd slice 2D – extract 2D slice from a UCD structure

SUMMARY

Name	ucd slice 2D
Availability	UCD module library
Type	mapper
Inputs	ucd structure colormap
Outputs	geometry geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
Interaction Mode	choice	Wait		
Do Slice	boolean	off		
x-rot	float	0.0	0.0	360.0
y-rot	float	0.0	0.0	360.0
Distance	float	0.0	-1.0	1.0
Transform Slice	boolean	off		

DESCRIPTION

The **ucd slice 2D** module extracts a 2D colored slice from a UCD structure. The slice plane can be rotated around the X and Y axes, and moved back and forth along the normal to the plane.

By default, the UCD structure is placed at the origin and the slice plane is in the X-Z plane. The orientation of the slice plane is controlled by two floating point parameter dials, **x-rot** and **y-rot**.

Interaction Mode offers a choice of **Immediate** and **Wait**.

Immediate generates output whenever a parameter or input port changes, such as during an animation.

Wait causes the **Do Slice** button to appear. In this mode, **ucd slice 2D** only generates output when **Do Slice** is pressed. This lets you adjust the slice plane until it is where you want, and only then perform the slicing operation. **Do Slice** does allow for successive automatic slices along one axis using the **Distance** parameter.

ucd slice 2D outputs two geometry objects. One is the slice plane, the other is the 2D slice of the UCD structure.

There are two different ways to use **ucd slice 2D**, one with only the left output port connected, and one with both output ports connected to different **geometry viewer** modules. These two configurations are illustrated in the "Examples" sections below.

INPUTS

UCD Structure (required)

The input structure is in AVS unstructured cell data (UCD) format. The structure can contain only scalar node data components.

Colormap (colormap)

This input colors the 2D slice according to an AVS colormap.

PARAMETERS

node data A set of radio buttons that selects which of the scalar node data components to output. If the components are unlabelled, this displays "<data 1>", "<data 2>", etc. If they are labelled, it displays the actual labels. The default is the first component.

Interaction Mode
A pair of radio buttons. **Immediate** generates output whenever a parameter or input port changes. **Wait** generates output whenever **Do Slice** is pressed. **Wait** is the default.

Do Slice A boolean switch that initiates the slicing operation. This button appears only when **Wait** is selected. It allows you to manipulate the slice plane until you are satisfied with its position, and only then extract the slice. **Do Slice** is off by default.

Each time the slice plane is reoriented the boolean parameter **Do Slice** is turned off. Once the slice plane is oriented as desired, and **Do Slice** is selected, the slice plane can be moved back and forth through the UCD structure along the normal to the plane with **Distance**. **Do Slice** remains "on" as you take successive slices along the normal. This lets you rapidly take a series of parallel slices through a UCD structure in any direction.

x-rot A floating point value which rotates the slice plane around the UCD structure's X axis. The range is 0.0 to 360.0. The default is 0.0.

y-rot A floating point value which rotates the slice plane around the UCD structure's Y axis. The range is 0.0 to 360.0. The default is 0.0.

Distance A floating point value between -1.0 and 1.0 which moves the slice plane back and forth in the direction of the normal to the plane. This value is scaled by the largest dimension of the UCD structure. Consequently, you can move the slice plane along the normal from **-** *max dimension* to + *max dimension*.

Transform Slice (boolean)
When selected, the 2D slice of the UCD structure is transformed so that it is parallel to the viewing plane. This must be turned **off** when **ucd slice 2D** is sending both its output geometries to a single **geometry viewer** module. It must be turned **on** when **ucd slice 2D** is sending its slice plane output to one **geometry viewer** module and its 2D slice output to another **geometry viewer** module.

This boolean is off by default.

OUTPUTS

Geometry The geometry object that **ucd slice 2D** outputs from the left output port represents the 2D slice of the UCD structure.

Geometry The geometry object that **ucd slice 2D** outputs from the right output port represents the slice plane.

EXAMPLE 1

In the following network **ucd slice 2D** sends both the slice plane output and the 2D slice output to a single **geometry viewer** module. This module also receives a model of the entire UCD structure from the **ucd to geom** module. Use **ucd to geom**'s **External Edges** parameter to create a wireframe representation of the object. These geometries are all rendered together. In this configuration, when you move the slice plane, the 2D slice will move with it.

Note that for the 2D slice to be correctly oriented, the **Transform Slice** parameter must be *off*. Note also that in this setup, **ucd slice 2D**'s lefthand output port is not connected to anything. If this port is connected to **geometry viewer** the results will be unpredictable.

```
GENERATE COLORMAP            READ UCD
         |                       |
         |                       |-----------------|
         |                       |                 |
         |                       |                 |
         |                       |                 |
         |------------------|    |                 |
                            |    |                 |
                       UCD SLICE 2D          UCD TO GEOM
                                |   |              |
                                |---|--------------|
                                    |
                          GEOMETRY VIEWER
```

EXAMPLE 2

In the second configuration, two **geometry viewer** modules are used. **ucd slice 2D** outputs both the 2D slice of the UCD structure and the slice plane. The 2D slice is viewed alone using the lefthand **geometry viewer** module. The 2D slice is transformed so that it is parallel to the view plane. This is done by turning **ucd slice 2D**'s **Transform Slice** parameter *on*.

The slice plane itself is sent to the righthand **geometry viewer** module, where it is rendered along with the UCD structure as a whole. This lets you see the position of the slice plane relative to the entire UCD structure. To display the structure as a wireframe model, switch **ucd to geom**'s **External Edges** parameter on.

```
GENERATE COLORMAP            READ UCD
      |                          |
      |                          |-----------------|
      |                          |                 |
      |                          |                 |
      |--------------------|     |                 |
                           |     |                 |
                       UCD SLICE 2D          UCD TO GEOM
                             |     |               |
                             |     |-----------|   |
                             |                 |   |
                     GEOMETRY VIEWER       GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that could provide the **Colormap** input:

generate colormap

Modules that can process **ucd slice2D**'s output:

geometry viewer

Any module that inputs a geometry

SEE ALSO

The example script UCD SLICE 2D demonstrates the **ucd slice2D** module.

ucd streamline

NAME

ucd streamline – generate stream lines or stream ribbons for a UCD structure

SUMMARY

Name	ucd streamline				
Availability	UCD module library				
Type	mapper				
Inputs	ucd structure colormap field irregular 3-space float *(optional, from create geom)* upstream transform *(optional, invisible, autoconnect)*				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	Node Data	choice	<data 1>		
	N Segment	integer dial	16	2	64
	Sample Style	choice	point		
	N Steps	integer dial	2	2	10
	Integration	choice	1st order		
	Backward	boolean	off		
	Color Streams	boolean	off		
	Ribbons	boolean	off		
	Ribbon Angle	float dial	0.0	0.0	360.0
	Ribbon Width	float dial	0.1∗*max dim*	0.0	20∗*default*
	Interaction Mode	choice	Immediate		
	Start Streams	boolean	off		

DESCRIPTION

The **ucd streamline** module generates colored stream lines or stream ribbons based on the vector node data in a UCD structure.

The stream lines are generated at selected sample points. For every time step, **ucd streamline** advances each sample point through space, based on the interpolated value of the node vectors surrounding the point. The result is a set of stream lines showing the progress of massless particles moving under the influence of the vector field at the nodes of the UCD structure. Stream ribbons behave similarly, except that their width and rotation also reflect the divergence and rotation of the flow at each point.

The sample points can be any scatter field. Usually, they come from two sources: from a sample probe generated by the **ucd streamline** module; or from arbitrarily-placed points defined by a field generated interactively by the **create geom** module.

ucd streamline's sample probe places a sample of points at a starting location in the UCD structure. The number of points is parameter-controlled. The sample probe's points can be moved around in space like any other geometry object, using the "virtual trackball" paradigm. To move the probe, select it by clicking on it, or by entering the Geometry Viewer and making it the current object.

There are three different modes to initiate the calculation of stream lines: **Immediate**, **Wait**, and **Button Up**. In **Immediate** mode, any change to a parameter or probe displacement will cause stream lines to be calculated immediately. In **Wait** mode you must press the **Start Streams** button to initiate streamlines calculation. This mode is useful when the streamline calculation requires a long time and you want to change a parameter or move the probe without immediate calculation. In **Button Up** mode, you can move probe with the mouse, keeping the button down. When you set the probe in a proper position and release the button, the module will then calculate streamlines. This mode can be useful when the streamline calculation requires a long time and you want to move the probe without immediate calculation.

A UCD structure consists of cells with nodes at their vertices. Each node may have data associated with it. **ucd streamline** only works with structures that have a vector component in their node data, thus it complains if the input structure has only scalar values at the nodes. (Scalar components can be converted to vector components with **ucd extract vector**.) If the nodes of a structure have more than one 3-vector component, use the **Node Data** radio buttons to select which component to use in calculating the stream lines.

INPUTS

UCD structure (required)
The input structure is in AVS unstructured cell data (UCD) format. It must have at least one node component which is a 3D vector, representing the components of a velocity vector.

colormap (required; colormap)
An AVS colormap that is used by **ucd streamline** to associate colors with vector values. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd legend**.

Data Field (optional, field irregular 3-space float)
ucd streamline generates its own data sampling probe that is manipulated from the **geometry viewer** module. Optionally, one can also input a field that defines an arbitrarily-placed set of sample points. This field is usually created interactively with the **create geom** module, but can be saved and reused as an AVS field.

Upstream Transform (invisible, optional, autoconnect)
When the **ucd streamline** module coexists with the **geometry viewer** module in a network, **geometry viewer** feeds information on how the point, circle or other sample probe has been moved back to this input port on the **ucd streamline** module. The two modules connect automatically, through a data pathway that is normally invisible. This gives direct mouse manipulation control over **ucd streamline**'s sample probe.

PARAMETERS

Node Data Selects which of the node's data components to display. A set of radio buttons shows the label attached to each node data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. If there are no vector components of the node data **ucd streamline** complains. If there are several vector data components, these buttons let you select which component to use in calculating the stream lines. If there is no node data in the structure "<no data>" is displayed on the button.

N Segment Integer dial that controls the density of points in the sample set.

Sample Style (radio buttons)
Specifies the configuration of points from which stream lines are drawn: **point**, **line**, **circle**, or **plane**.

N Steps Integer dial that specifies the number of time steps for which stream lines are computed within each cell of the UCD structure. As the number of time steps increases, so does the accuracy of the stream lines.

Integration Method

Selects the integration method used to advance sample points through space: **1st order** uses an euler integration method, **2nd order** uses a 2nd order Runge-Kutta method, and **3rd order** uses a 3rd order Runge-Kutta method.

Backward (boolean)

If **Backward** is selected, stream lines are extrapolated in the opposite direction that the UCD structure's vectors are pointing. By default this switch is off.

Color Streams (boolean)

If **Color Streams** is selected, the stream lines are colored based on the magnitude of the interpolated vectors used to generate the stream lines. By default this switch is off.

Ribbons (boolean)

A toggle switch that turns on stream ribbons rather than stream lines. The default is **off**.

Ribbon Angle (float dial)

This control only appears if **Ribbons** is selected. It controls the initial angle at which the ribbon is drawn. By default, ribbons are drawn with their width parallel to the X axis. The range is 0 to 360; the default is 0.

Ribbon Width (float dial)

This control only appears if **Ribbons** is selected. It scales the width of the ribbon. The range and default shown on the dial is calculated based on the size of the UCD structure. The default is 0.1*the maximum dimension of the structure. The minimum is 0.0. The maximum is scaled to be 20 times the default. (Ribbon width will vary along its length according to the divergence of the flow at each node.)

Interaction Mode (radio buttons)

Selects a mode to initiate the calculation of stream lines: **Immediate**, **SWait** or **Button Up**. In **Immediate** mode, any change to a parameter or probe displacement will cause stream lines to be calculated immediately. In **Wait** mode you must press the **Start Streams** button to calculate streamlines. In **Button Up** mode you can move the probe with the mouse, keeping the mouse button down. When you set the probe in a proper position and release the button, the module will calculate streamlines.

Start Streams (boolean)

A boolean switch that initiates the calculation of stream lines in **Wait** mode. This button allows you to manipulate the sample probe until you are satisfied with its position, or change other parameters and only then begin computing stream lines.

OUTPUTS

Stream Lines, Sampling Object (geometry)

ucd streamlines outputs two geometries: a set of colored disjoint lines or ribbons, and the sampling object.

EXAMPLE 1

The following network reads in a UCD structure with a 3-vector float value as one of the components of the node data. **ucd streamline** displays colored stream lines. Note that the module **ucd to geom** is used to provide a frame within which to view the streamlines. To do this, select the "External Edges" parameter in the **ucd to geom** module

```
GENERATE COLORMAP       READ UCD
        |                   |
        |---------------|   |-----------|
                        |   |           |
                   UCD STREAMLINE    UCD TO GEOM
                            |           |
                            |-----------|
                            |
                            |
                   GEOMETRY VIEWER
```

EXAMPLE 2

This network is identical to the first, except that the sample points are taken from a field that was originally generated with the **create geom** module. This field could have been saved with **write field**, in which case **read field** would replace **create geom** in the network.

```
GENERATE COLORMAP     READ UCD
            |           |
            |-------|   |-----------|
                    |   |           |
CREATE GEOM         |   |           |
     |              |   |           |
     |----------|   |   |           |
            UCD STREAMLINE    UCD TO GEOM
                        |           |
                        |-----------|
                        |
                        |
               GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
scatter to ucd
ucd curl
ucd grad
Any module that outputs a UCD structure.

Modules that could provide the **Colormap** input:

generate colormap

Modules that could provide the **Data field** input:

create geom
read field

Modules that can process **ucd streamline**'s output:

geometry viewer

SEE ALSO

The example script UCD STREAMLINE demonstrates the **ucd streamline** module.

ucd threshold

NAME

ucd threshold – get subset of UCD structure based on node values

SUMMARY

Name	ucd threshold
Availability	UCD module library
Type	filter
Inputs	ucd structure info (required; from **ucd legend**)
Outputs	ucd structure

Parameters

Name	*Type*	*Default*
below	boolean	off
inclusive	boolean	off

DESCRIPTION

ucd threshold takes a subset of the cells in a ucd structure based on the values at cell nodes. Input structure cells with nodes values that fall outside a user specified range do not appear in the structure which **ucd threshold** outputs.

The input received from **ucd legend** tells **ucd threshold** what range to restrict values to. This information can either be a single floating point number representing the cut-off value, or it can be two floating point numbers representing both a high and a low threshold.

The **ucd threshold** module is similar to the module, **ucd crop**. **ucd crop**, however, eliminates nodes from a UCD structure based on their x, y, z coordinates — **ucd threshold** eliminates nodes based upon their values.

INPUTS

Structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Info (from **ucd legend**)

ucd threshold must receive input from the module **ucd field legend** through its left input port. This tells **ucd threshold** what range to restrict values to.

PARAMETERS

below A boolean switch, which has meaning only when the **info** input is a single floating-point value. If it is selected, **ucd threshold** outputs the subset of the UCD structure that is below the threshold value. If it is not selected **ucd threshold** outputs the subset of the UCD structure that is above the threshold value.

inclusive A boolean switch; if it is selected, then **all** the nodes of a given cell must satisfy the threshold condition for that cell to be passed to the output. In other words, if a cell has even one node whose value falls outside the threshold range, that cell is eliminated from the output. If the **inclusive** switch is turned off, only one node of a given cell needs to satisfy the threshold condition for the cell to be included in the output structure.

OUTPUTS

Structure The output structure is the threshold filtered AVS unstructured cell data (UCD).

EXAMPLE

The following network reads in a UCD structure. The structure is passed to the **ucd field legend** module, which outputs a threshold value or range. It is also passed to the **ucd threshold** module, which restricts the structure's values to the threshold range. Note that **ucd legend** also outputs a color field that gets passed to **ucd to geom** so that the data is colored.

```
GENERATE COLORMAP      READ UCD
    |                     |
    |    |----------------|
    |    |                |
  UCD LEGEND              |
    |    |                |
    |----|------------|   |
         |            |   |
         |         UCD THRESHOLD
         |                |
         |-------------|  |
                     UCD TO GEOM
                          |
                          |
                          |
                   GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

Modules that provides the **info** input:

ucd legend

Modules that can process **ucd threshold**'s output:

ucd to geom, ucd crop, ucd anno,
ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d,
ucd probe, ucd streamline, write ucd.

SEE ALSO

The example script UCD THRESHOLD demonstrates the **ucd threshold** module.

ucd to geom

NAME

ucd to geom – convert a UCD structure into an AVS geometry

SUMMARY

Name	ucd to geom
Availability	UCD module library
Type	mapper
Inputs	ucd structure field 1D 3-vector real *(color field from ucd contour/ucd legend)* field 1D 3-vector real *(color field from ucd cell color)*
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Shrink	boolean	off		
Shrink Factor	integer dial	10	0	100
Geometry Display Mode	choice	External Faces		
Explode Materials	boolean	off		
Explode Factor	integer dial	5	0	100
Save Geometry	boolean	on		
Color Cells	boolean	off		

DESCRIPTION

ucd to geom converts a ucd structure into an AVS geometry that can be rendered using the **geometry viewer** module.

At the lowest level, unstructured cell data consists of nodes located in 3-space. These nodes may have a vector of values associated with them. Nodes form the vertices of polyhedral cells, which themselves may have cell based data associated with them.

ucd to geom takes the input structure's node location data, as well as a node connectivity list telling which nodes connect to form which cells. Each cell thus defined is converted into geometry format and is added to the geometry object that the module outputs.

A UCD structure may have hundreds of nodes and cells, many of which are likely to be "interior" and thus hidden. You can select the **External Faces** Geometry Display Mode to restrict **ucd to geom**'s output to the "exterior", visible faces of the UCD structure's cells. This makes converting the structure to a geometry and rendering it much faster.

In some cases, you may want to see objects that are inside the ucd structure, such as isosurfaces, streamlines, probes, and so on. In this case you can select the **External Edges** Geometry Display Mode to restrict **ucd to geom**'s output to the exterior edges, representing the wireframe boundary of the ucd structure. If this mode is selected, the **shrink factor** parameter changes its meaning and becomes the **Edge Angle** parameter which controls the accuracy of the boundary representation on the base of the angle between two adjoining faces.

When **All Faces** is selected, all faces of all cells will be displayed.

The cells can be shrunk using the **shrink factor** parameter. If the cells in a structure are packed close together, this creates gaps between cells and lets you see how cells are really shaped.

The **Explode Materials** parameter is useful for displaying ucd structures containing cells with different materials. For example, different materials can be assigned to different parts of an assembly. If the **Explode Materials** parameter is **on**, the module will create different geometry objects for each of the materials. Each of the geometry objects can be manipulated separately using the Geometry Viewer. The **Explode Factor** parameter controls how far apart these geometry objects are displayed initially.

The **Save Geometry** parameter is useful when you are changing the colors but not the geometric coordinates of the structure. For example, selecting different components of the data or animating time dependent data.

ucd to geom can receive optional color fields. A color field is an array of color values—one color for each node or cell in the input UCD structure. This results in the structure being rendered as a colored geometry object. The center input port is used to color *node* data. Its input field is generated by the modules **ucd contour** or **ucd legend**. The leftmost input port is used to color *cells*, either based upon the cell data or the material id of the cell. Its input field is generated by the **ucd cell color** module.

INPUTS

Structure (required)

The input structure is in AVS unstructured cell data (UCD) format.

Color Field (field 1D 3-vector real; optional)

This is the center input port. The color field is a 1 dimensional array of color values. There is one color for each node in the input UCD structure. Each color value is a triple of floating point numbers representing red, green and blue. The **Color Field** input is produced by the modules **ucd contour** and **ucd legend**. Note that it is not the same as an AVS colormap.

If both the center node color field input port and the leftmost cell color field input port are connected, this center input port will be used to color the data; the left cell color field will be ignored. Press the **Color Cells** switch to switch to coloring by cells or material ids.

Color Field (field 1D 3-vector real; optional)

This is the leftmost input port. The color field is a 1 dimensional array of color values. There is one color for each cell or material id in the input UCD structure. Each color value is a triple of floating point numbers representing red, green and blue. The **Color Field** input is produced by the **ucd cell color** module. Note that it is not the same as an AVS colormap.

If both the center node color field input port and this leftmost cell color field input port are connected, the center input port will be used to color the data and this left cell color field will be ignored. Press the **Color Cells** switch to switch to coloring by cells or material ids.

PARAMETERS

Shrink (boolean)

When this is selected each cell in the UCD structure is shrunk by the factor specified by the **shrink factor** parameter. By default **Shrink** is off.

shrink factor

An integer is used to scale the cells of the UCD structure. Values of this parameter range from 1 to 100, representing percentages. The default shrink factor of 10 results in cells that are shrunk by 10 percent. If **External Edges** mode is selected, the **shrink factor** parameter changes its meaning and becomes an **Edge Angle** parameter that controls the accuracy of the boundary representation on the base of the angle between two adjoining faces.

Geometry Display Mode

A radio button that selects **External Faces**, **External Edges** or **All Faces**. When **External Faces 1** selected, **ucd to geom** only creates exterior, visible cell faces in the output geometry. This makes converting to a geometry and rendering much faster than when **All Faces** is selected. When **External Edges** selected, **ucd to geom** only creates exterior visible edges, representing the "wireframe boundary" of the ucd structure. This renders faster than **All Faces** or **External Faces**. It also allows any interior geometry, such as a cropped ucd structure or streamlines to show through without being obscured by the faces. When **All Faces** selected, all faces of all cells will be displayed.

Explode Materials

If the **Explode Materials** parameter is **on**, the module will create different geometry objects for each of the materials. Each of the geometry objects can be manipulated separately using the Geometry Viewer.

Explode Factor

This parameter controls how far apart geometry objects with different materials are initially displayed.

Save Geometry

This parameter allows you to store the geometry object in the module and only update the geometry's colors when the input data or "Color Field" changes. This mode makes rendering faster but requires additional memory. **Save Geometry** is on by default.

Color Cells

When both the node and cell color field input ports are connected, **ucd to geom** defaults to coloring by nodes. Press this boolean toggle to color by cells or material ids instead.

OUTPUTS

Geometry The geometry that **ucd to geom** outputs represents the cells of the input UCD structure colored according to the values of the input color field.

EXAMPLE

```
GENERATE COLORMAP      READ UCD
       |                  |
       |     |------------|
     UCD CONTOUR          |
             |            |
             |----------| |
                   UCD TO GEOM
                          |
                          |
                  GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD Structure** input:

read ucd
field to ucd
ucd extract
Any module that outputs a UCD Structure.

Modules that could provide the **Color Field** input:

- ucd contour
- ucd legend
- ucd cell color

Modules that can process **ucd to geom**'s output:

- geometry viewer
- *Any module that takes an AVS geometry*.

SEE ALSO

The example scripts READ UCD, UCD THRESHOLD, UCD CROP, as well as others, demonstrate the **ucd to geom** module.

ucd tracer

NAME

ucd tracer - perform ray-traced volumetric rendering on a UCD structure

SUMMARY

Name	ucd tracer
Availability	UCD module library
Type	mapper
Inputs	ucd structure tracker info (field 2D scalar float) colormap (required)
Outputs	field 2D 4-vector byte (image)

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Size	integer	128	0	1024
alpha scale	float	1.0	0.0	10.0
exterior	boolean	on		

DESCRIPTION

ucd tracer belongs to a family of modules that render volumetric data. **ucd tracer** takes a UCD structure, consisting of tetrahedral cells, and generates a 2D image using ray-tracing. Each cell in the structure has data values associated with its nodes. These values are used to assign a color and opacity value to every node in the structure. Note that, by default, **ucd tracer** "exterior" parameter is **on**, and therefore only an object's surface is ray-traced.

The ray tracing method is as follows. For each pixel in the output image a ray is "shot" into the UCD structure. Each cell the ray passes through makes some contribution to the color of the pixel. The color is calculated by interpolating between the color of the point at which the ray enters the cell and the color of the exit point. How much color a cell contributes depends on its opacity. The ray travels through the volume until the opacity of all the cells it has passed through adds up to 1.0. This is an "additive light model", because the rays accumulate cell color contributions as they travel through a volume.

For example, if a ray hits a completely opaque red tetrahedron then it travels no further, and the pixel associated with that ray is colored red. On the other hand, if the tetrahedron is nearly transparent, then it confers only a fraction of its color to the pixel, and the ray passes deeper into the volume, summing the color values of the other cells it intersects.

Volumetric rendering such as this allows you to penetrate beneath the surface of 3D unstructured cell data, and see depths surrounded by "translucent" outer layers. The degree of opacity of the volume can be controlled by changing the alpha scale parameter, or by using **generate colormap**'s widget to edit the opacity values in the input colormap.

ucd tracer only works with UCD structures that have tetrahedral cells. You can convert hexahedral data to tetrahedral using the module **ucd hex to tet**.

INPUTS

UCD structure (required)

The input structure is in AVS unstructured cell data (UCD) format. The structure's cells must be tetrahedrons.

tracker info *(field 2D scalar float)*

The middle input port on the module **ucd tracer** can receive a 4x4 transformation matrix describing rotations and translations to apply to the UCD structure. The matrix (field 2D scalar float) can come from the module **euler transformation** or **display tracker**. This allows you to rotate the structure in 3-space.

colormap (required; colormap)

An AVS colormap which is used by **ucd tracer** to associate colors with UCD node values. Note that this is a regular AVS colormap, and not the **color field** output by **ucd contour** and **ucd legend**.

PARAMETERS

Size (integer)

Value which determines the height and width of the output image measured in pixels. Another way of thinking of this is that the width determines the number of rays that are projected into the volume along the x and y directions. This changes the size of the square window through which you view the volume,.

alpha scale (float)

A floating point value between 0.0 and 10.0 which is multiplied by the alpha value of every node in the structure. This determines how transparent the the structure will seem. As the value of alpha scale approaches 0.0 the volume becomes more transparent, allowing rays to penetrate deeper into the volume, and making inner regions visible.

exterior (boolean)

If **exterior** is selected, then only the surface of the UCD structure is ray-traced. Note that this is the default.

OUTPUTS

Field (field 2D 4-vector byte)

The output field is an AVS image.

EXAMPLE

The following network reads in a UCD structure, which is converted from hexahedral cells to tetrahedal cells. This structure is then passed to **ucd tracer**. The module **display tracker** allows you to rotate the volume to produce views from any angle. Objects are manipulated using the usual mouse buttons.

```
                           READ UCD
                               |
                               |
                         UCD HEX TO TET
GENERATE COLORMAP              |
             |-----------|     |
                         |     |
                       UCD TRACER
                             |
                             |
                      DISPLAY TRACKER
```

RELATED MODULES

Modules that could provide the **ucd structure** input:

read ucd
ucd hex to tet
any other module which outputs a tetrahedral UCD structure.

Modules that can process **ucd tracer**'s output:

display tracker

display image
image viewer
any other module which takes an AVS image as input.

SEE ALSO

Garrity, M., "Raytracing Irregular Volume Data," (Proceedings of the 1990 San Diego Workshop on Volume Visualization), *Computer Graphics*, Volume 24, Number 5, November 1990, pp. 35-40. ACM SIGGRAPH.

The example script UCD TRACER demonstrates the **ucd tracer** module.

NAME

ucd vecmag – compute the magnitude of a vector ucd

SUMMARY

Name	ucd vecmag
Availability	UCD module library
Type	filter
Inputs	ucd structure
Outputs	ucd structure
Parameters	none

DESCRIPTION

The **ucd vecmag** module accepts a ucd structure having one 3-element vector component (for example, x-momentum, y-momentum, z-momentum) as an input and computes the magnitude of each vector data value. The output is a single scalar component consisting of the vector magnitude.

INPUTS

UCD structure (required)

The input structure is a 3D AVS unstructured cell data (UCD) structure with a single component that is a 3-element vector of floating point data values for each node. (If your data consists of three scalar components, you can convert them to the required format with the **ucd extract vector** module.)

OUTPUTS

UCD structure

The output ucd structure has a single floating point value of a vector magnitude for each input ucd node.

EXAMPLE

The following network reads in a 3D vector ucd and computes the magnitude of the vectors:

```
                     READ UCD
                         |
                 UCD EXTRACT VECTOR
                         |
                   UCD VECMAG
                  _______|____
                  |           |
GENERATE COLORMAP |           |
              |   |           |
          UCD LEGEND          |
              |               |
              |   ____________|
              |   |
          UCD TO GEOM
               |
               |
        GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd

ucd extract vector

Modules that can process **ucd vecmag**'s output:

ucd to geom, ucd crop, ucd threshold, ucd hex to tet, ucd anno,
ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d,
ucd legend, ucd probe, ucd streamline, write ucd.

Other UCD vector modules:

ucd curl
ucd div
ucd grad

SEE ALSO

The example script UCD VECMAG demonstrates the **ucd vecmag** module.

NAME

ucd vol integral – calculate the volume of a UCD structure and the volume integral of a scalar data component

SUMMARY

Name	ucd vol integral		
Availability	UCD module library		
Type	data output		
Inputs	ucd structure		
Outputs	none		
Parameters	*Name*	*Type*	*Default*
	Node Data	choice	<data 1>
	Output File	typein	none

DESCRIPTION

ucd vol integral performs two functions:

1. It calculates the total volume enclosed by the UCD structure's cells.
2. It calculates the volume integral of any one of the scalar node data components. Volume integrals are often useful in UCD analysis. For example, the volume integral of a UCD structure with density node data is equal to the mass of the UCD structure.

ucd vol integral writes its output to both a screen text browser, and to a user-specified file.

INPUTS

UCD structure (required)
The input structure is in AVS unstructured cell data (UCD) format.

PARAMETERS

Node Data Selects which of the node's data components to volume integrate. A set of radio buttons shows the label attached to each cell data component. Before the module receives data, the default "<data 1>, <data 2>, ..." is displayed. Each data component must be scalar. (Convert vector components to scalar with **ucd extract scalars**.)

Output File
A typein to specify where the integrated volume should be written. This output is also always written to the **Output Browser**.

EXAMPLE 1

The following network reads in a UCD structure then calculates and displays its volume and volume integral.

```
   READ UCD
       |
       |
UCD VOL INTEGRAL
```

RELATED MODULES

read ucd
any module that outputs a UCD structure

SEE ALSO

The example script UCD VOL INTEGRAL demonstrates this module.

vector curl

NAME

vector curl – compute the curl of a vector field

SUMMARY

Name	vector curl
Availability	FiniteDiff module library
Type	filter
Inputs	field 3D 3-vector float *any-coordinates*
Outputs	field 3D 3-vector float *any-coordinates*
Parameters	none

DESCRIPTION

The **vector curl** module accepts a vector field as input and computes the curl of that field as output. Computation is a finite difference approximation based on a central difference scheme.

Where the input is the vector function:

$\{F_x, F_y, F_z\}(i, j, k)$

The equation used to compute the curl is:

$$curl = \left\{ \left[\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} \right], \left[\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \right], \left[\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right] \right\}$$

INPUTS

Data Field (required; field 3D 3-vector float *any-coordinates*)

The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

EXAMPLE

The following network reads in a 3D vector field and computes its curl, then displays the field vectors using **hedgehog**:

```
READ FIELD
     |
     |----------------------|
     |                      |
VECTOR CURL                 |
     |               VOLUME BOUNDS
 HEDGEHOG                   |
     |                      |
     |----------------------|
     |
GEOMETRY VIEWER
```

OUTPUTS

Data Field (field 3D 3-vector float *any-coordinates*)

The output field is in the same format as the input field.

The **min_val** and **max_val** attributes of the output field are invalidated.

RELATED MODULES

gradient shade
tracer

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR CURL demonstrates the **vector curl** module.

vector div

NAME

vector div – compute the divergence of a vector field

SUMMARY

Name	vector div
Availability	FiniteDiff module library
Type	filter
Inputs	field 3D 3-vector float *any-coordinates*
Outputs	field 3D scalar float *any-coordinates*
Parameters	none

DESCRIPTION

The **vector div** module accepts a vector field as input and computes the divergence of that field as output. This is related to the curl as follows:

$$curl = (DEL \times F)$$
$$div = (DEL \bullet F)$$

F the vector input field is:

$\{F_x, F_y, F_z\}(i, j, k)$

The equation used to compute the divergence is:

$$divergence = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}$$

INPUTS

Data Field (required; field 3D 3-vector float *any-coordinates*)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

OUTPUTS

Data Field (field 3D scalar float *any-coordinates*)
The output field has a single floating-point value for each input field element.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE

The following network reads in a 3D vector field and computes its divergence:

```
  READ VOLUME
       |
  VECTOR DIV
       |
ARBITRARY SLICER
       |
GEOMETRY VIEWER
```

RELATED MODULES

vector curl, vector div, vector norm, vector mag,
hedgehog, stream lines, stream mesh

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR DIV demonstrates the **vector div** module.

NAME

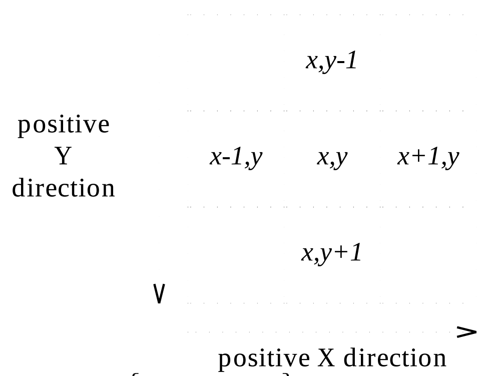
vector grad – compute the vector gradient of a 3D scalar field

SUMMARY

Name	vector grad
Availability	FiniteDiff module library
Type	filter
Inputs	field 3D scalar float *any-coordinates*
Outputs	field 3D 3-vector float *any-coordinates*
Parameters	none

DESCRIPTION

The **vector grad** module computes the gradient of a 3D field. The gradient is treated by some other modules as a "pseudo-normal" to the "surface" for each data element. A "nearest neighbor" algorithm is used to compute the gradient: the difference between the next data value (in each direction) and the previous data value. In two dimensions, this can be represented as follows:



$$gradient(F) = \left\{ \frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right\}$$

The **min_val** and **max_val** attributes of the output field are invalidated.

This module is identical to the **compute gradient** module, except that it does *not* normalize the output. **compute gradient** is designed for gradient shading fields, whereas this module is designed for input into the other vector field modules: **vector curl**, **vector div**, **vector mag**, and **vector norm**. Note that **vector grad** followed by **vector norm** produces the same results as **compute gradient**.

INPUTS

Data Field (field 3D scalar float *any-coordinates*)

The input field must represent a volume of elements, with a single floating-point value for each input field element.

OUTPUTS

Data Field (required; field 3D 3-vector float *any-coordinates*)

The output field has a 3D vector of floating-point data values for each element.

EXAMPLE

The following network reads a 3D scalar field, computes its gradient and then uses the **hedgehog** module to display the resulting vector field:

```
READ FIELD
     |
     |-----------------------|
     |                       |
VECTOR GRAD                  |
     |                       |
VECTOR NORM            VOLUME BOUNDS
     |                       |
 HEDGEHOG                    |
     |                       |
     |-----------------------|
     |
GEOMETRY VIEWER
```

RELATED MODULES

vector curl, vector div, vector norm, vector mag,
hedgehog, particle advector, stream lines, stream mesh

LIMITATIONS

There may be algorithms better than "nearest-neighbor" for computing the gradient.

This module produces 12 bytes per pixel (voxel). For example, a 128 x 128 x 128 byte volume is about 2.1 MB before the gradient is computed. The **compute gradient** module produces a 25.2 MB internal data set from this data. This will have an adverse performance effect on systems whose physical memory is 32 MB or less.

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR GRAD demonstrates the **vector grad** module.

vector mag

NAME

vector mag – compute the magnitude of a vector field

SUMMARY

Name	vector mag
Availability	FiniteDiff module library
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D scalar float uniform
Parameters	none

DESCRIPTION

The **vector mag** module accepts a vector field as input and computes the magnitude of each vector data value. The output is a scalar field consisting of the magnitudes.

The magnitude equation is:

```
Magnitude[X][Y][Z] = sqrt((dx[X][Y][Z]*dx[X][Y][Z]) +
                          (dy[X][Y][Z]*dy[X][Y][Z]) +
                          (dz[X][Y][Z]*dz[X][Y][Z]) )
```

INPUTS

Data Field (required; field 3D 3-vector float uniform)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

OUTPUTS

Data Field (field 3D scalar float uniform)
The output field has a single floating-point value for each input field element.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE

The following network reads in a 3D vector field and computes the magnitude of the vectors:

```
   READ VOLUME
       |
   VECTOR MAG
       |
 ARBITRARY SLICER
       |
 GEOMETRY VIEWER
```

RELATED MODULES

vector curl, vector div, vector norm, vector mag. hedgehog, particle advector, gradient shade, stream lines, stream mesh

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR MAG demonstrates the **vector mag** module.

NAME

vector norm – normalize a vector field

SUMMARY

Name	vector norm
Availability	FiniteDiff module library
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D 3-vector float uniform
Parameters	none

DESCRIPTION

The **vector norm** module accepts a vector field as input, and produces a normalized version of that vector field as output. The normalization equation looks like:

```
Magnitude = sqrt((dx*dx) +  (dy*dy) + (dz*dz))
New_dx      = dx / Magnitude
New_dy      = dy / Magnitude
New_dz      = dz / Magnitude
```

INPUTS

Data Field (required; field 3D 3-vector float uniform)
The input field must represent a volume of elements, with a 3D vector of floating-point data values for each element.

OUTPUTS

Data Field (field 3D 3-vector float uniform)
The output field is in the same format as the input field.

The **min_val** and **max_val** attributes of the output field are invalidated.

EXAMPLE

The following network reads a 3D scalar field, computes its gradient and then uses the **hedgehog** module to display the resulting vector field:

```
       READ FIELD
           |
           |------------------------|
           |                        |
       VECTOR GRAD                  |
           |                        |
       VECTOR NORM            VOLUME BOUNDS
           |                        |
        HEDGEHOG                    |
           |                        |
           |------------------------|
           |
    GEOMETRY VIEWER
```

RELATED MODULES

vector curl, vector div, vector norm, vector mag, hedgehog, particle advector, gradient shade, stream lines, stream mesh

LIMITATIONS

This module works only with 3D 3-vector float fields. This data type is widely used in flow analysis, where each 3-vector of floats represents the components of a velocity or a gradient.

SEE ALSO

The example script VECTOR NORM demonstrates the **vector norm** module.

NAME

volume bounds – generate bounding box of 3D 3-vector field

SUMMARY

Name	volume bounds
Availability	Volume, FiniteDiff module libraries
Type	mapper
Inputs	field 3D *n-vector any-data any-coordinates*
Outputs	geometry

Parameters

Name	*Type*
Hull	toggle
Min I	toggle
Max I	toggle
Min J	toggle
Max J	toggle
Min K	toggle
Max K	toggle
Colored Bounds	toggle

DESCRIPTION

The **volume bounds** module generates lines that indicate the "bounding box" of a 3D data set (field). It is frequently used in conjunction with other geometry-based volume-visualization modules (e.g. **bubbleviz**, **isosurface**, **hedgehog**, **arbitrary slicer**), since it provides some volumetric context for the data.

Normally, the axes are colored Red for X (or I), Green for Y (or J), and Blue for Z (or K). This can be disabled using the **Colored Bounds** toggle. When this button is turned off, **volume bounds** produces uncolored (white) lines which can then be colored using the Property Editor in the Geometry Viewer.

INPUTS

Data Field (required; field 3D *n-vector any-data any-coordinates*) The input data must be a 3D field, but may have any kind of data at each location in the field.

OUTPUTS

Bounds (geometry) The output *geometry* consists of the lines that form the bounding box.

PARAMETERS

Hull Draws lines for the perimeter of the data set.

Min I
Max I
Min J
Max J
Min K
Max K These toggle switches provide further help is visualizing the way the computational space is mapped into physical space. Each one fills in one of the six faces of the hull. For example, turning on **Min I** draws a mesh showing the 2D slice of field elements with the minimum index value in the first dimension; turning on **Max K** draws a mesh showing the 2D slice of field elements with the maximum index value in the third dimension.

Colored Bounds

The default behavior for this module is to produce Red, Green, and Blue bounding lines corresponding to the X, Y, and Z axes for uniform and rectilinear field data, or the I, J, and K bounds of irregular data. When the **Colored Bounds** toggle is turned off, the lines are left uncolored (they show up as white in the Geometry Viewer). They can now be set to whatever color you like using the Geometry Viewer's Property Editor.

EXAMPLE

The following network showing a typical usage of **volume bounds**:

```
               READ VOLUME
       ______________|___________________
       |             |                  |
 VOLUME BOUNDS   BUBBLEVIZ      ARBITRARY SLICER
       |_____________|__________________|
                     |
              GEOMETRY VIEWER
```

RELATED MODULES

read volume, read field, geometry viewer

SEE ALSO

The example scripts BRICK, HEDGEHOG, PROBE, as well as others demonstrate the **volume bounds** module.

NAME

volume manager – share volumes among subnetworks

SUMMARY

Name	volume manager		
Unsupported	this module is in the unsupported library		
Type	data		
Inputs	none		
Outputs	field 3D scalar byte		
Parameters	*Name*	*Type*	*Choices*
	VOLUMGR select	choice	Select, Replace
	Volume Manager	browser	
	Volume Choices	choice	

DESCRIPTION

The **volume manager** module reads an volume file from disk and outputs the volume as a "field 3D scalar byte". It works like the **read volume** module, except that it has both a caching mechanism and a way of sharing data among **volume manager** modules in separate subnetworks.

See the **read volume** manual page for a description of the volume format.

PARAMETERS

VOLUMGR Select

A choice that determines how newly-read volumes will be placed to the list of currently active volumes:

- If **Select** is chosen, a new volume is added to the end of the list.
- If **Replace** is chosen, a new volume replaces the currently selected member on this list.

  In either case, the change is reflected in *all* the *volume manager* modules in all active subnetworks.

Volume Manager

A file browser that allows you to select an volume file to read.

Volume Choices

A set of choices, listing each of the currently active volumes.

OUTPUT

Data Field (field 3D scalar byte)

The output is the byte data cast as the scalar data in a 3D *field*.

EXAMPLE

The following subnetworks might be used to display two volumes:

```
 VOLUME MANAGER          VOLUME MANAGER
       |                       |
   COLORIZER               COLORIZER
       |                       |
ORTHOGONAL SLICER       ORTHOGONAL SLICER
       |                       |
 DISPLAY IMAGE           DISPLAY IMAGE
```

In this case, both **volume manager** modules would contain "select/replace" choice buttons, a file browser, and an area below the browser:

```
+-------------------------+    +-------------------------+
| Active Volumes          |    | Active Volumes          |
+-------------------------+    +-------------------------+
| (no volumes)            |    | (no volumes)            |
+-------------------------+    +-------------------------+
```

Once a volume (e.g. *hydrogen.dat*) was selected from the browser in the **volume manager** on the left, these buttons would look like this:

```
+-------------------------+    +-------------------------+
| * hydrogen.dat          |    |      hydrogen.dat       |
+-------------------------+    +-------------------------+
```

If a different file (e.g. *benzene.dat*) is chosen from the browser in the **volume manager** on the right, the buttons would look like this:

```
+-------------------------+    +-------------------------+
| * hydrogen.dat          |    |   hydrogen.dat          |
|   benzene.dat           |    | * benzene.dat           |
+-------------------------+    *-------------------------+
```

By selecting the same active volume, you can have both networks display the same volume:

```
+-------------------------+    +-------------------------+
| * hydrogen.dat          |    | * hydrogen.dat          |
|   benzene.dat           |    |   benzene.dat           |
+-------------------------+    +-------------------------+
```

Now, if you want to replace this volume with a new one, click on the **Replace** buttons above the browser, then select a new file (e.g. *methane.dat*) in just one of the **volume manager** browsers. The result is that all **volume manager** modules with the old volume (*hydrogen*) selected as their active volume will be automatically updated with the new volume (*methane.dat*).

RELATED MODULES

Same as for **read volume**.

LIMITATIONS

The cached volumes are not freed until all **volume manager** modules are destroyed. Because volume data can be large, caching multiple volume datasets can use a lot of memory.

NAME

volume render – volume render a uniform volume with geometry

SUMMARY

Name	volume render
Availability	Volume, FiniteDiff module libraries requires 3D texture mapping, alpha transparency, and volume rendering support
Type	mapper
Inputs	field 3D uniform *n-vector any-data*
Outputs	geometry
Parameters	*none*

DESCRIPTION

The **volume render** module is another way of visualizing 3D uniform volume data. In this technique, the user assigns a color and an opacity for each volume cell (or voxel) in the volume, usually using the **generate colormap** and **colorizer** modules. The data is then rendered in the Geometry Viewer such that each voxel in the scene occupies a particular 3D region. If the voxel is totally transparent, it will not be displayed at all in the scene. If the voxel is completely opaque, it will be drawn with its designated color and it will obscure all voxels (or fractions of voxels) that might be behind it given the current viewing angle.

Other volume visualization techniques can be combined with the **volume render** module, such as **isosurface** and **arbitrary slicer**. These objects will be properly combined with the volume rendered objects.

The **volume render** module, when connected to the **geometry viewer** module produces an object in the Geometry Viewer that is set up to display the volume rendered object. Volume rendering is based on the 3D texture mapping functionality. In order to get the volume rendering to occur, a colorized version of the same input that is connected to the **volume render** module's input port should also be connected to the left input port on the **geometry viewer** module.

This module can be effectively used in conjunction with the **clip geom** module to allow the user to slice through the volume as it is being rendered to reveal detail on the inside of the volume.

AVAILABILITY

volume render requires that the underlying graphics renderer support: 3D texture mapping, alpha transparency, and volume rendering. Not all hardware renderers support these rendering techniques (see the release note information that accompanies AVS on your platform). The AVS software renderer does support 3D texture mapping, alpha transparency, and volume rendering.

If a renderer does not support 3D texture mapping, the object will appear a featureless white. If it does not support alpha transparency, the opacity settings on the **generate colormap** Colormap Editor will have no effect on the transparency of the object. If a renderer does not support volume rendering in this narrow sense (a specific option to the **GEOMedit_texture_options** call), the object will likely simply not be drawn or will appear black.

On multi-renderer platforms, you can turn on the **Software Renderer** button under the Geometry Viewer's **Cameras** submenu. If no such choice appears, it is likely that the software renderer is the only renderer available.

volume render

INPUTS

Data Field (required; field 3D uniform)

The input field is a 3D uniform volume. A version of this volume that is colors should be passed to the field input of the **geometry viewer** module.

OUTPUTS

Geometry (geometry)

This module creates a volume render object through the geometry port.

EXAMPLE

The following network reads a byte volume. The volume is fed to **colorizer** to paint the byte values as colors, to **volume render** to create the volume render object, and to **volume bounds** to draw a box around the limits of the volume. The **generate colormap**, **colorizer**, and **geometry viewer** parts of the network are vital; they create the 3D texture map that is needed for the volume rendering to work. All in turn feed into **geometry viewer**.

```
                          READ VOLUME
                              |
GENERATE COLORMAP             |
        |                     |
        |  |------------------|--------------------|
        |  |                  |                    |
     COLORIZER          VOLUME RENDER        VOLUME BOUNDS
        |                     |                    |
        |                     |                    |
        |--------------|      |--------------------|
                       |      |
                    GEOMETRY VIEWER
```

RELATED MODULES

Modules that could provide the **Data Field** input:

read volume
read field
Any module that outputs a 3D uniform field

Modules that could be used in place of **volume render**:

tracer
arbitrary slicer
orthogonal slicer
thresholded slicer

Modules that can process **volume render** output:

geometry viewer

LIMITATIONS

The rendering process does not perform any lighting on the object. In order to get lighting affects, you will need to do this by hand using the **gradient shade** module.

The rendering process is fairly slow when the volume rendered object is made large on the screen. It is best to experiment with a small version of the object and only zoom in on it when you have the proper view.

NAME

wireframe – convert object from surface to wireframe representation

SUMMARY

Name	wireframe
Availability	Volume, FiniteDiff module libraries
Type	filter
Inputs	geometry
Outputs	geometry
Parameters	none

DESCRIPTION

The **wireframe** module transforms an AVS *geometry*, replacing all surfaces defined as polytriangle strips with wireframe representations. This is useful for constructing a wireframe version of an object that has been defined as a shaded surface.

INPUTS

Geometry (required; geometry) Any AVS *geometry*, created with the *libgeom* library or produced by another AVS module.

OUTPUTS

Geometry A geometry that represents the same object as the input data.

EXAMPLE

This example shows the use of the **wireframe** module to generate a wireframe version of a polygonal object:

```
    READ GEOM
        |
    WIREFRAME
        |
 GEOMETRY VIEWER
```

EXAMPLE 2

This example uses the **wireframe** and **tube** modules to have a geometry involving spheres drawn with cylinders instead of lines:

```
    READ GEOM
        |
    WIREFRAME
        |
      TUBE
        |
 GEOMETRY VIEWER
```

RELATED MODULES

read geom, offset, shrink, flip normal, tube, geometry viewer, render geometry

LIMITATIONS

The **wireframe** module generates lines based on the order of the vertices of a polytriangle strip. Sometimes, the resulting object is not exactly what you want. It may have "cobwebs" and other (usually invisible) data inconsistencies of the original polytriangle strip. You may need to regenerate the original data in order to produce the desired wireframe representation.

SEE ALSO

The example scripts TUBE, and WIREFRAME demonstrate the **wireframe** module.

write field

NAME

write field – write a field description to disk

SUMMARY

Name	write field
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field *any-dimension n-vector any-data any-coordinates*
Outputs	none

Parameters

Name	*Type*	*Default*
Write Field	browser	
Native / Portable(XDR)	choice	Native
Compute Extent	boolean	Off
Compute Min / Max	boolean	Off

DESCRIPTION

The **write field** module writes an AVS *field* description to disk. The field format on disk includes two parts, an *ASCII header* and a *binary area.* This format is described in detail in the manual page for **read field**.

write field will include any information present about the field such as labels and units, as well as its dimensions, vector length, data type, etc.

By default, **write field** will write out the field structure exactly as it is. That is, if no minimum and maximum extent information or minimum and maximum values are computed for the field, **write field** will not compute and write them out. However, if values for the minimum and maximum extents for the field, and the minimum and maximum data values for each vector component in the field are not already present, **write field** can calculate them and store them in the output ASCII header. This is controlled with the **Compute Extent** and **Compute Min/Max** buttons. This is useful when you are importing data into AVS format with **read field**'s data parsing input mode, and you do not know these correct values. You should let **write field** calculate them rather than try to guess them.

After the field file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

INPUTS

Data Field (field *any-dimension n-vector any-data any-coordinates*)
The input can be any AVS *field.*

PARAMETERS

Write Field

A file browser that allows you to specify the name of the field file to be created. The file suffix *.fld* is appended to the name automatically. If the file already exists, **write field** issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

Native/Portable(XDR)

Controls the format of the binary portion of the output field file.

Native

The binary portion of the output field file will be written in the same format as the platform on which the **write field** module is executing. A comment stating this platform will be added to the end of the "data=" line in the ASCII header.

Portable(XDR)

The binary portion of the output field file will be written in Sun's XDR (external data representation) format. This option is useful for transporting field files between machines with different binary architectures ("big-endian" vs "little-endian"). The "data=" line in the ASCII header will specify **xdr_integer**, **xdr_float**, or **xdr_double** rather than the simple data type.

See the "Binary Compatibility on Different Hardware Platforms" section of the **read field** man page for more information on the purpose of this feature.

Compute Extent

If the extents are not already computed, turning this button 'ON' will cause them to be computed and written out with the field. This feature is off by default.

Compute Min/Max

If the minimum and maximum values for each vector component are not already computed, turning this button 'ON' will cause them to be computed and written out with the field. This feature is off by default.

EXAMPLE 1

Following is an example of a native-format file produced by **write field**. The "data=" line indicates that the field file was written on an DEC workstation.

```
# AVS field file (@(#)write_field.c     5.10 Stellar 91/06/28)
# creation date: Thu Jul 18 16:03:36 1991
#
ndim=3                          # number of dimensions in the field
dim1=40                         # dimension of axis 1
dim2=32                         # dimension of axis 2
dim3=32                         # dimension of axis 3
nspace=3                        # number of physical coordinates per point
veclen=5                        # number of components at each point
data=float                      # native format of dec3100
field=irregular                 # field type (uniform, rectilinear, irregular)
min_ext=-7.815747 0.000000 0.000000        # coordinate space extent
max_ext=14.362204 8.327559 5.724251        # coordinate space extent
label= density x-momentum y-momentum z-momentum stagnation
min_val=0.192600 -2.183500 -0.325250 -3.733900 0.768957   # minimum data values
max_val=4.977500 5.790300 3.545400 1.502900 25.160999     # maximum data values
```

The field has three dimensions, 40x32x32. There is a vector of 5 floating point values at each point, and the field is irregular.

EXAMPLE 2

The following is an example of an XDR format file produced by **write field** of a 3D uniform field with a vector of 3 values at each point. This field had no labels or units.

```
# AVS field file (@(#)write_field.c     5.10 Stellar 91/06/28)
# creation date: Fri Aug 23 14:25:54 1991
#
ndim=3                          # number of dimensions in the field
dim1=27                         # dimension of axis 1
dim2=25                         # dimension of axis 2
```

```
dim3=32                          # dimension of axis 3
nspace=3                         # number of physical coordinates per point
veclen=3                         # number of components at each point
data=xdr_float                   # portable data format
field=uniform                    # field type (uniform, rectilinear, irregular)
min_ext=0.000000 0.000000 0.000000        # coordinate space extent
max_ext=26.000000 24.000000 31.000000     # coordinate space extent
min_val=-29.381140 -33.578682 -10.565389    # minimum data values
max_val=42.604145 24.940878 29.761003       # maximum data values
```

EXAMPLE 3

The following network reads in a field, crops it and then writes the resultant field to a file:

```
READ FIELD
     |
   CROP
     |
WRITE FIELD
```

RELATED MODULES

read field
print field
compare field

write field writes *any* AVS field file.

ERRORS

Write field complains if it can't open the file, or if there isn't enough space to write the complete file.

SEE ALSO

The example script WRITE FIELD demonstrates the **write field** module.

NAME

write image – store image data in a file

SUMMARY

Name	write image
Availability	Imaging, Volume, FiniteDiff module libraries
Type	data output
Inputs	field 2D 4-vector byte uniform
Outputs	none
Parameters	*Name* *Type* Write Image browser

DESCRIPTION

The **write image** module writes an AVS *image* data structure to a file. This structure takes the form of a "field 2D 4-vector byte". See the **read image** manual page for a detailed description of the image format.

INPUTS

Data Field (required; field 2D 4-vector byte uniform)
The input can be any AVS *image*.

PARAMETERS

Write image

A file browser that allows you to specify the name of the image file to be created. The file suffix *.x* is appended to the name automatically. If the file already exists, **write_image** issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

After the image file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

EXAMPLE

```
         READ GEOM
             |
      RENDER GEOMETRY
             |
             |-------------------|
             |                   |
      DISPLAY PIXMAP      PIXMAP TO IMAGE
                                 |
                                 |
                            WRITE IMAGE
```

RELATED MODULES

Image processing:

contrast, threshold, histogram stretch, clamp, interpolate

Decompose/compose images from separate bands:

extract scalar, combine scalars

Show image:

display image
image viewer

write image

Take output from data output module, and write the data out as an image:

geometry viewer, image to postscript

SEE ALSO

read image
image viewer

The example script WRITE IMAGE demonstrates the **write image** module.

NAME

write ucd – write unstructured cell data to disk

SUMMARY

Name	write ucd		
Availability	UCD module library		
Type	data output		
Inputs	ucd structure		
Outputs	none		
Parameters	*Name*	*Type*	*Default*
	Write UCD	browser	
	File Format	choice	Binary

DESCRIPTION

The **write ucd** module writes a UCD structure to disk.

write ucd outputs a binary or ASCII file. Both binary and ASCII file formats are read by the module **read ucd**.

The format of UCD structure, as well as the format of ASCII and binary UCD files is described in detail in the manual page for **read ucd**, and in the "Unstructured Cell Data" section of the *AVS Developer's Guide*.

After the UCD file is written, the filename is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering the rewriting of the file. A new file is written only when you enter a filename.

INPUTS

ucd structure

The input can be any UCD structure.

PARAMETERS

Write UCD

A file browser that allows you to specify the name of the ucd file to be created.

File Format

A pair of radio buttons that specify **Binary** or **ASCII** file output.

EXAMPLE

The following network reads in a UCD structure, crops it, and writes the resulting structure to disk:

```
READ UCD
    |
UCD CROP
    |
WRITE UCD
```

RELATED MODULES

Modules that could provide the **UCD structure** input:

read ucd
field to ucd
Any module that outputs a UCD structure.

ERRORS

write ucd will complain if it can't open the file, or if there isn't enough space to write the complete file.

SEE ALSO

The example script WRITE UCD demonstrates the **write ucd** module.

NAME

write volume – write volume data to a file

SUMMARY

Name	write volume
Availability	Volume, FiniteDiff module libraries
Type	data output
Inputs	field 3D scalar byte uniform
Outputs	none
Parameters	*Name* *Type* Write Volume browser

DESCRIPTION

The **write volume** module writes volume data to a file. The volume is in the AVS format "field 3D scalar byte". The data format on disk is:

1 byte: number of voxels in X 1 byte: number of voxels in Y 1 byte: number of voxels in Z nx * ny * nz * 1 byte: voxel data

Each time the file is written, the filename is reset to NULL. This prevents successive changes upstream in the network to automatically trigger a volume data file to be written. A new filename must be entered each time the file is to be written out.

If the file to be written exists, the following warning appears:

```
File FILENAME
  already exists.  Do you want to overwrite it?
```

Two choices are presented. If you select **Cancel**, the write operation is aborted. If you select **Overwrite**, the existing file on disk is replaced with the new volume data.

This module is commonly used to pre-process a volume database for later use. For example, the input data might be very low-contrast. You could construct a network that includes the **contrast** module and the **write volume** module. Once you select appropriate settings for the contrast, the data could be written to a file, and used later for other types of processing.

INPUTS

Data Field (required; field 3D scalar byte uniform)

The input data must be a 3D field, with a byte value at each location in the field.

PARAMETERS

Write Volume

A file browser that allows you to specify the name of the volume data file to be created. The file suffix *.dat* is appended to the name automatically. If the file already exists, **write_volume** issues a warning message and has you confirm the operation ("Overwrite") or cancel it ("Cancel").

RELATED MODULES

read volume, clamp, contrast, crop, downsize, histogram stretch, interpolate, mirror, threshold, transpose

EXAMPLE

The following network writes a volume-format output file to disk.

write volume

```
  READ VOLUME
       |
     CROP
       |-----------------|
       |                 |
   COLORIZER             |
       |                 |
ORTHOGONAL SLICE     WRITE VOLUME
       |
 DISPLAY IMAGE
```

LIMITATIONS

The format of volume databases on disk is severely limiting. The dimensions are restricted to a maximum of 255 in x, y and z. The data also must be in the range 0 - 255.

SEE ALSO

read volume

The example script WRITE VOLUME demonstrates the **write volume** module.

NAME

x-ray – perform simple orthographic volume visualization

SUMMARY

Name	x-ray			
Availability	Volume, FiniteDiff module libraries			
Type	filter			
Inputs	field 3D uniform scalar *any-data*			
Outputs	field 2D uniform scalar *same-data*			
Parameters	*Name*	*Type*	*Default*	*Choices*
	Axis	choice	K	I,J,K
	Operation	choice	mean	sum, mean, median, min, max

DESCRIPTION

x-ray peforms simple, orthogonal volume visualization on 3D uniform fields. It outputs a 2D field that can be colorized and displayed as an image.

Looking directly along the X, Y, or Z axis, the module looks at the row of voxels "behind" each screen pixel and, depending on the selected operation, creates a new pixel based on those voxels. The 2D result resembles an x-ray.

x-ray is a fast volume visualization technique. It is useful to quickly get a sense of the contents of an unfamiliar dataset.

INPUTS

Data Field (required; field 3D uniform scalar *any-data*)
An input field. Note that the field may have any data type.

PARAMETERS

Axis (choice)
The choices are **I**, **J**, and **K**. The default is **K**. If you choose **I**, you look down the X axis into the YZ plane. If you choose **J**, you look down the Y axis into the XZ plane. If you choose **K**, you look down the Z axis into the XY plane.

Operation (choice)
The choices are sum, mean, median, min, and max. The default is mean.

sum Each screen pixel is the sum of the stack of voxels.

mean Each screen pixel is the sum of the stack of voxels divided by the number of voxels in each stack.

median The stack of voxels is sorted by value and the screen pixel gets the center value in the sorted stack.

min The screen pixel gets the smallest value in the stack.

max The screen pixel gets the largest value in the stack.

median is very slow to compute.

sum and **mean** produce the same visual results if you normalize the colormap to the data, but **mean** takes a little longer to compute.

mean and **max** are the best techniques for most operations.

OUTPUTS

Data Field (field 2D uniform scalar *same-data*)
x-ray outputs an field with the same data type as the input field. This field must be colorized in order to be displayed. Note that the **sum**

operation could cause data values to overflow their data type. Byte input fields should probably be converted to integer (**field to int** module) if the **sum** operation is used.

EXAMPLE

Here is the typical **x-ray** network:

```
GENERATE COLORMAP                READ FIELD
         |                           |
         |                         X-RAY
         |_____________    __________|
                      |    |         |
                    COLOR RANGE      |
                        |_________   |
                                 |   |
                               COLORIZER
                                   |
                             DISPLAY IMAGE
```

RELATED MODULES

tracer, orthoslicer

SEE ALSO

The example script X-RAY demonstrates this module.