ConvexAVS Module Reference

First Edition

CONVEX COMPUTER CORPORATION

CONVEX Computer Corporation
3000 Waterview Parkway
P.O. Box 833851
Richardson, TX 75083-3851
United States of America
(214)497-4000

ConvexAVS Module Reference

Order No. DSW-305

First Edition
February 1992

CONVEX Press
Richardson, Texas
United States of America

ConvexAVS Module Reference

Order No. DSW-305

Copyright ©1992 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OR ANY THIRD PARTY.

Copyright ©1990 by Auto-trol Technology Corporation, Denver, Colorado.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears on all copies and that both the copyright and this permission notice appear in supporting documentation and that the name of Auto-trol not be used in advertising or publicity pertaining to distribution of the software without specific, prior written permission.

Redistribution and use in source and binary forms are permitted provided that the above copyright notice and this paragraph are duplicated in all such forms and that any documentation, advertising materials, and other materials related to such distribution and use acknowledge that the software was developed by David E. Smyth. The name of David E. Smyth may not be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Copyright ©1987-1991 Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies. The University of California makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.

ConvexAVS is a trademark of CONVEX Computer Corporation.

ImageNode is a trademark of Diaquest Inc.

IRIS is a trademark of Silicon Graphics, Inc.

LaserWriter is a trademark of Apple Computer, Inc.

Mathematica is a trademark of Wolfram Research, Inc.

MOPAC is a trademark of Quantum Chemistry Program Exchange.

PostScript is a trademark of Adobe Systems, Inc.

VideoCreator is a trademark of Silicon Graphics, Inc.

X Window System is a trademark of the Massachusetts Institute of Technology.

AVS was created and developed by, and is a trademark of, Advanced Visual Systems, Inc.

Printed in the United States of America

Revision information for

ConvexAVS Module Reference

Edition	Document No.	Description
First	710-013030-003	Released February 1992. Initial release of this book. Contains module reference information related to the ConvexAVS V3.0/3+ software.

Contents

ConvexAVS modules 1
ConvexAVS module groups 9
avs 13
AVS Animator 25
animate lines 29
animated float 33
animated integer 37
antialias 41
arbitrary slicer 43
background 47
boolean 51
bubbleviz 53
cfd values 57
character string 61
clamp 63
color range 65
colorizer 69
colormap manager 71
combine scalars 73
compare field 75
composite 79
compute gradient 81
contour to geom 85
contrast 87
convolve 91
crop 95
density PLOT3D 99
display image 101
display pixmap 105
display tracker 109
dot surface 113
downsize 117
euler transformation 121
extract scalar 123
extract vector 125
field legend 129

field math 133
field to byte 137
field to double 139
field to float 141
field to int 143
field to mesh 145
field to ucd 149
file browser 153
flip normal 155
float 157
generate colormap 159
generate filters 165
generate histogram 169
gradient shade 173
graph viewer 177
hedgehog 183
histogram stretch 187
hq display image 189
image compare 191
image manager 195
image to pixmap 197
image to postscript 199
image viewer 201
integer 205
interpolate 207
isosurface 211
local area ops 215
luminence 219
mirror 221
momentum PLOT3D 223
offset 225
oneshot 227
orbital tiler 229
orthogonal slicer 233
output ImageNode 237
output VideoCreator 241
output postscript 245
paint mesh 247
particle advector 251
pdb to geom 257
pdb viewer 259
pixmap to image 265
prepare video 267
print field 271
probe 275
read PLOT3D 279
read field 283
read frame seq 299

read gaussian 303
read gaussian CHK 307
read geom 313
read hdf field 315
read hdf image 319
read image 323
read mopac 325
read plot3d 331
read rle image 335
read ucd 337
read volume 341
render geometry 343
render manager 349
replace alpha 353
samplers 355
scalar PLOT3D 359
scatter dots 363
shrink 367
sobel 369
stagnation PLOT3D 371
statistics 373
stream lines 377
surface mapper 381
threshold 383
thresholded slicer 385
tracer 389
transpose 395
tristate 397
tube 399
ucd anno 401
ucd cell to node 405
ucd contour 407
ucd crop 411
ucd extract 415
ucd hex to tet 417
ucd hog 419
ucd iso 423
ucd legend 427
ucd offset 431
ucd print 435
ucd probe 439
ucd rslice 445
ucd slice 2d 449
ucd streamline 453
ucd threshold 457
ucd to geom 461
ucd tracer 465
ucd vecmag 469

vbuffer ... 471
vector PLOT3D ... 475
vector curl ... 477
vector div ... 479
vector grad ... 481
vector mag ... 485
vector norm ... 487
volume bounds ... 489
volume manager ... 491
wireframe ... 493
write field ... 495
write frame seq ... 499
write hdf field ... 503
write hdf image ... 505
write image ... 507
write ucd ... 509
write volume ... 511

Figures

Figure 1 animate lines module ... 31
Figure 2 animated float module ... 35
Figure 3 animated integer module ... 39
Figure 4 antialias module ... 41
Figure 5 arbitrary slicer module ... 46
Figure 6 background module ... 49
Figure 7 background module ... 50
Figure 8 boolean module ... 52
Figure 9 bubbleviz module ... 55
Figure 10 cfd values module ... 60
Figure 11 character string module ... 62
Figure 12 clamp module ... 64
Figure 13 color range module ... 67
Figure 14 colorizer module ... 70
Figure 15 colormap manager module ... 72
Figure 16 combine scalars module ... 74
Figure 17 compare field module ... 76
Figure 18 composite module ... 80
Figure 19 Computing the gradient ... 81
Figure 20 compute gradient module ... 83
Figure 21 compute gradient module ... 83
Figure 22 contour to geom module ... 86
Figure 23 contour to geom module ... 86
Figure 24 Transforming field values ... 89
Figure 25 contrast module ... 89
Figure 26 convolve module ... 93
Figure 27 crop module ... 97
Figure 28 density PLOT3D module ... 100
Figure 29 display image module ... 104
Figure 30 display pixmap module ... 108
Figure 31 display pixmap module ... 108
Figure 32 display tracker module ... 110
Figure 33 dot surface module ... 114
Figure 34 Downsize factor of 4 on a 2D field ... 118
Figure 35 downsize module ... 119
Figure 36 euler transformation module ... 122

Figure 37 extract scalar module 124
Figure 38 extract vector module 126
Figure 39 field legend module 131
Figure 40 field math module 135
Figure 41 field math module 136
Figure 42 field to byte module 138
Figure 43 field to double module 140
Figure 44 field to float module 142
Figure 45 field to int module 144
Figure 46 field to mesh module 146
Figure 47 field to mesh module 147
Figure 48 field to ucd module 150
Figure 49 file browser module 154
Figure 50 flip normal module 155
Figure 51 float module 158
Figure 52 Editing panel organization 162
Figure 53 generate colormap module 163
Figure 54 generate filters module 167
Figure 55 generate filters module 167
Figure 56 generate histogram module 171
Figure 57 gradient shade module 175
Figure 58 gradient shade module 176
Figure 59 graph viewer module 180
Figure 60 graph viewer module 180
Figure 61 graph viewer module 181
Figure 62 graph viewer module 182
Figure 63 hedgehog module 185
Figure 64 histogram stretch module 188
Figure 65 hq display image module 190
Figure 66 image compare module 193
Figure 67 image manager module 196
Figure 68 image to pixmap module 198
Figure 69 image to postscript module 200
Figure 70 image viewer module 203
Figure 71 integer module 206
Figure 72 interpolate module 208
Figure 73 isosurface module 213
Figure 74 local area ops module 217
Figure 75 luminence module 220
Figure 76 mirror module 222
Figure 77 momentum PLOT3D module 224
Figure 78 offset module 226
Figure 79 oneshot module 228
Figure 80 orbital tiler module 231
Figure 81 orthogonal slicer module 234
Figure 82 orthogonal slicer module 235
Figure 83 orthogonal slicer module 235
Figure 84 output VideoCreator module 244

Figure 85 output postscript module 246
Figure 86 paint mesh module 249
Figure 87 particle advector module 255
Figure 88 pdb to geom module 258
Figure 89 pdb viewer module 263
Figure 90 pixmap to image module 266
Figure 91 Gamma correction curves 268
Figure 92 prepare video module 269
Figure 93 print field module 273
Figure 94 probe module 278
Figure 95 read PLOT3D module 280
Figure 96 ASCII file for an image 284
Figure 97 ASCII file for irregular data 287
Figure 98 ASCII file for a volume 294
Figure 99 ASCII file for a volume 294
Figure 100 ASCII file for an irregular volume 295
Figure 101 ASCII file decay.fld 296
Figure 102 ASCII file converting hydrogen.dat 297
Figure 103 ASCII file converting mandrill.x 297
Figure 104 read field module 298
Figure 105 read frame seq control panel 299
Figure 106 read frame seq module 301
Figure 107 read gaussian module 305
Figure 108 read gaussian CHK module 311
Figure 109 read geom module 314
Figure 110 read hdf field module 316
Figure 111 read hdf image module 321
Figure 112 read image module 324
Figure 113 read mopac module 329
Figure 114 read plot3d module 334
Figure 115 read rle image module 336
Figure 116 UCD file format 339
Figure 117 UCD file example 340
Figure 118 read ucd module 340
Figure 119 read volume module 342
Figure 120 render geometry module 346
Figure 121 render manager module 351
Figure 122 replace alpha module 354
Figure 123 samplers module 357
Figure 124 scalar PLOT3D module 360
Figure 125 scatter dots module 364
Figure 126 scatter dots module 365
Figure 127 shrink module 368
Figure 128 sobel module 370
Figure 129 stagnation PLOT3D module 372
Figure 130 statistics module 375
Figure 131 statistics module 375
Figure 132 stream lines module 380

Figure 133 surface mapper module 382
Figure 134 threshold module 384
Figure 135 thresholded slicer module 388
Figure 136 tracer module 392
Figure 137 tracer module 393
Figure 138 transpose module 396
Figure 139 tube module 400
Figure 140 ucd anno module 403
Figure 141 ucd cell to node module 406
Figure 142 ucd contour module 409
Figure 143 ucd crop module 413
Figure 144 ucd extract module 416
Figure 145 ucd hex to tet module 418
Figure 146 ucd hog module 422
Figure 147 ucd iso module 425
Figure 148 ucd legend module 430
Figure 149 ucd offset module 432
Figure 150 ucd print module 437
Figure 151 ucd probe module 442
Figure 152 ucd rslice module 447
Figure 153 ucd slice 2d module 451
Figure 154 ucd streamline module 456
Figure 155 ucd threshold module 459
Figure 156 ucd to geom module 463
Figure 157 ucd tracer module 467
Figure 158 ucd vecmag module 470
Figure 159 vector PLOT3D module 476
Figure 160 vector curl module 478
Figure 161 vector div module 480
Figure 162 Computing the gradient 481
Figure 163 vector grad module 483
Figure 164 vector mag module 486
Figure 165 vector norm module 488
Figure 166 volume bounds module 490
Figure 167 volume manager module 492
Figure 168 wireframe module 493
Figure 169 wireframe module 494
Figure 170 Example write field file 496
Figure 171 write field module 496
Figure 172 write frame seq control panel 500
Figure 173 write frame seq module 502
Figure 174 write hdf field module 504
Figure 175 write hdf image module 506
Figure 176 write image module 508
Figure 177 write ucd module 510
Figure 178 write volume module 512

Tables

Table 1 ImageNode_VCR_type values 239
Table 2 ASCII file organization 1 293
Table 3 ASCII file organization 2 293
Table 4 ASCII file organization 3 293
Table 5 ASCII file organization 4 293

How to use this manual

Purpose and audience

ConvexAVS Module Reference describes modules and topics related to them used in ConvexAVS, the CONVEX adaptation of the Application Visualization System (AVS).

Each reference includes the following information:

- An overall summary of the module.
- Description of the module.
- Details of the module's inputs, outputs, and parameters.
- An example network using the module.
- Any modules related to it.
- Any limitations the module has.

This reference addresses scientists and engineers who want to visualize their data using ConvexAVS and need to know specific information about modules.

Organization

The modules are arranged alphabetically. Several related topics on module listings and groupings appear first followed by the modules. Online information is available through the `avs_help` browser or by clicking on a module's dimple while in the Network Editor.

Notational conventions

In general, the following conventions are used in this guide:

- **`Bold constant-width font`** identifies user input in examples.
- *Italics*
  - Designate user-supplied variables in a command-line example
  - Introduce new and important terms
  - Identify variables in mathematical equations
  - Indicate document titles
- `Constant-width font` designates input and output, including:
  - Command names and options
  - Data structures and types
  - Directives, program statements, display examples, printout examples, and error messages returned
- Horizontal ellipsis (...) shows repetition of the preceding item(s).
- Words and abbreviations that indicate keyboard keys you press are identified in a distinctive bold type. For example, **RETURN** refers to the carriage return key. Words separated by a hyphen indicate two keys that you must press simultaneously. For example, **CTRL-X** indicates that you must press and hold down the **CTRL** key and then press the **X** key.
- Button names that appear on the screen are also identified in the same distinctive bold type as keyboard keys. For example, the **prepare video** module uses an **interlace** button to merge two fields together.
- Module names appear in **bold** type to distinguish them from surrounding text. For example, **integer** or **particle advector**.
- The word "enter" in a phrase such as "enter **`ls`**" means that you type the command and then press **RETURN**.
- References to the *ConvexOS Programmer's Reference* appear in the form `malloc`(1), where the name of the man page is followed by its section number enclosed in parentheses.

Note

A Note highlights supplemental information.

Associated documents

Using this software may require information not specific to the tasks described in this document. Additional help may be found in the following manuals:

- *Using ConvexAVS to Visualize Data* (DSW-304). This book contains information for users to visualize their data using ConvexAVS.
- *Animating AVS Data Visualizations* (DSW-306). This book contains information about creating animations with ConvexAVS.

Ordering documentation

To order the current edition of this or any other CONVEX document, send requests to the following address:

CONVEX Computer Corporation
Customer Service
P.O. Box 833851
Richardson TX 75083-3851 USA

Include order number or exact title, as listed on the front cover.

Technical assistance

If you have questions that are not answered in this book, contact the CONVEX Technical Assistance Center (TAC).

- Within the continental U.S., call (800)952-0379.
- From Canada, call 1(800)345-2384.
- Outside continental U.S., contact your local CONVEX office.

Acknowledgments

I would like to thank the following groups for their contributions to this document. This book would not have been possible without their help.

- Development—technical content
- Testing—technical accuracy
- Documentation and Training—technical support
- Editorial Services—technical quality

Neal W. Johnston
Technical Communication Leader

ConvexAVS modules

Introduction to reference pages for ConvexAVS modules

Description

This section includes a manual page for each module in the ConvexAVS distribution. There are four types of modules:

- Data input
- Filters
- Mappers
- Data output

Using the right mouse button, click on the small square (dimple) in any module icon to open its Module Editor window. Then click the **Show Module Documentation** button to view the complete manual page for the module.

Throughout the reference pages for ConvexAVS modules, a number of terms are used to describe these data types:

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.
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, 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 that have uniform, rectilinear, or irregular coordinates. If a module cannot operate on one of these types of fields, this will be declared.

ConvexAVS modules

Module listing

The modules included in this release of ConvexAVS are:

ConvexAVS modules	Introduction to reference pages for ConvexAVS modules
module groups	Types of ConvexAVS modules
avs	Application Visualization System
AVS Animator	Create and edit animation scripts
animate lines	Animate stream lines for a vector field
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
antialias	Antialias an image
arbitrary slicer	Map 3D scalar field to 3D mesh
background	Create a shaded backdrop image
boolean	Send a boolean value to one or more module(s) boolean parameter port(s)
bubbleviz	Generate spheres to represent values of 3D field
cfd values	Calculate values for a field containing **read plot3d** data (unsupported)
character string	Send a string to one or more module(s) string parameter port(s)
clamp	Restrict values in data field
color range	Scale colormap to the range of data in a field
colorizer	Convert field of data values to color values
colormap manager	Share colormaps among subnetworks (unsupported)
combine scalars	Combine scalar fields into a vector field
compare field	Compare two fields, display and write data difference
composite	Blend two images using alpha transparency
compute gradient	Compute gradient vectors for 2D or 3D data set
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
crop	Extract subset of elements from a field
density PLOT3D	Strip out the PLOT3D density
display image	Show image in a display window
display pixmap	Show pixmap in a display window
display tracker	Display and directly manipulate the **tracer** module's output
dot surface	Generate points that define an isosurface
downsize	Reduce size of data set by sampling
euler transformation	Send object transformation matrix to other modules
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 a 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 ucd	Convert field to unstructured cell data format
file browser	Send a file name to one or more module(s) file name parameter port(s)
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 colormap	Output colormap
generate filters	Generate 2D filters for image processing

generate histogram	Plot distribution of data values in a scalar field
gradient shade	Apply lighting and shading to colored data set
graph viewer	Create contour and XY-plots of data (Graph Viewer subsystem)
hedgehog	Show vectors in a 3D 3-vector field
histogram stretch	Balance the histogram of a data set
hq display image	Display a high quality image
image compare	Display two images together
image manager	Share images among subnetworks (unsupported)
image to pixmap	Convert image to pixmap
image to postscript	Convert image to PostScript and store in file
image viewer	Display and manipulate collections of images (Image Viewer subsystem)
integer	Send an integer to the integer parameter port of one or more module(s)
interpolate	Compute intermediate values to change the size of a field
isosurface	Generate an isosurface for a volume of data
local area ops	Image processing based on pixel neighborhoods
luminence	Compute the luminance of an image
mirror	Reverse array indices in a 2D or 3D data set
momentum PLOT3D	Strip out the PLOT3D momentum vector
offset	Translate vertices along vertex normals
oneshot	Send a oneshot value to one or more module(s) oneshot parameter port(s)
orbital tiler	Generate an isosurface for a volume of data
orthogonal slicer	Slice through 2D or 3D field with plane perpendicular to coordinate axis
output ImageNode	Write images to video tape using a Diaquest ImageNode
output VideoCreator	Write images to video tape using a Silicon Graphics VideoCreator
output postscript	Convert pixmap to PostScript and store in file

paint mesh	Paint an image onto a surface in 3D space
particle advector	Release grid of particles into velocity field
pdb to geom	Create molecule geometry from PDB file
pdb viewer	View and manipulate information in Brookhaven PDB format
pixmap to image	Transform pixmap to image
prepare video	Perform interlacing, low-pass filtering, and gamma correction for video output
print field	Create an ASCII printable and readable version of a field
probe	Interactively show numeric data values in a geometry rendered field
read PLOT3D	Read PLOT3D files
read field	Read field from a file or import data files into field format
read frame seq	Read a sequence of frames from a file generated by the **write frame seq** module
read gaussian	Extract structural and volume data from a Gaussian output file
read gaussian CHK	Extract structural and volume data from a Gaussian checkpoint file
read geom	Reads a data file containing a geometry
read hdf field	Read a scientific data set from an HDF file into a field
read hdf image	Read 8-bit or 24-bit raster images from an HDF file into an image
read image	Read file into an image
read mopac	Extract structural and volume data from a MOPAC 6.0 output file
read plot3d	Read a PLOT3D format file into a field (unsupported)
read rle image	Read image file in RLE format from disk into a field
read ucd	Read UCD structure from a file
read volume	Read volume file from disk into a field

render geometry	Convert geometric description to image (Geometry Viewer subsystem)
render manager	Share geometries among subnetworks (unsupported)
replace alpha	Replace the alpha channel in an image
samplers	Extract a subset of locations from a 3-vector 3D field
scalar PLOT3D	Calculate derived PLOT3D scalar functions
scatter dots	Generate spheres at points in 3D space
shrink	Make polygons of a geometry object smaller
sobel	Apply an edge detecting filter to 2D field
stagnation PLOT3D	Strip out the PLOT3D stagnation energy
statistics	Display statistics on field contents
stream lines	Generate stream lines for a vector field
surface mapper	Represent a list of points as a surface of dots or spheres
threshold	Restrict values in data field
thresholded slicer	Slice through volume data with high and low values invisible
tracer	Perform ray-traced volumetric rendering on 3D field
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 to node	Convert UCD structure cell data into node data
ucd contour	Generate list of color values associated with unstructured cell data
ucd crop	Subset UCD structure data using slice plane or box
ucd extract	Extract single node component from a UCD structure

ucd hex to tet	Convert a UCD structure from hexahedral cells to tetrahedral cells
ucd hog	Show UCD structure node vector values as line segments in 3D space
ucd iso	Generate an isosurface for a UCD structure with scalar node data
ucd legend	Create a color legend relating UCD structure data to a color scale
ucd offset	Deform a UCD structure based on vector values at each node
ucd print	Create an ASCII readable version of a UCD structure for debugging purposes
ucd probe	Interactively show numeric data values in a geometry-rendered UCD structure
ucd rslice	Slice away portions of a UCD structure
ucd slice 2d	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 a geometry
ucd tracer	Perform ray-traced volumetric rendering on a UCD structure
ucd vecmag	Compute the magnitude of a vector UCD structure
vbuffer	Perform volumetric rendering on volume data (unsupported)
vector PLOT3D	Calculate derived PLOT3D vector functions
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 3D 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)

wireframe	Convert object from surface to wireframe representation
write field	Write a field description to a file
write frame seq	Write image sequences to a file
write hdf field	Write a field as a scientific data set to an HDF file
write hdf image	Write image data to an HDF file
write image	Write image data to a file
write ucd	Write unstructured cell data to disk
write volume	Write volume data to a file

ConvexAVS module groups

Types of ConvexAVS modules

Description

The ConvexAVS 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 which modules take fields and convert them to geometries or which modules save data to disk. The following is a division of 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	pdb viewer	read rle image

DISPLAYING DATA

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

SAVING AND PRINTING DATA

output postscript	write field	write image
write volume	write ucd	print field
image to postscript		

COLORING DATA

colorizer	color range	generate colormap
field legend	ucd contour	ucd legend

GENERATING VALUES TO PARAMETER PORTS

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

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

CONVERTING FIELDS TO GEOMETRIES

bubbleviz	field to mesh	isosurface
contour to geom	hedgehog	probe
stream lines	volume bounds	thresholded slicer
arbitrary slicer	scatter dots	particle advector
scatter dots	paint mesh	

CONVERTING VOLUMES TO IMAGES

orthogonal slicer	tracer

CONVERTING GEOMETRIES TO PIXMAPS

render geometry

CONVERTING PIXMAPS AND IMAGES

pixmap to image	image to pixmap

CONVERTING FIELD TO UCD

field to ucd

CONVERTING UCD STRUCTURES TO GEOMETRIES

ucd to geom	ucd hog	ucd streamline

FIELD PROCESSING AND FILTERING

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

IMAGE PROCESSING

contrast	crop	mirror
generate filters	convolve	luminence
background	sobel	interpolate
threshold	clamp	antialias
composite	image compare	local area ops
transpose	replace alpha	

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		

GEOMETRY UTILITIES

flip normal	offset	shrink
tube	wireframe	

UCD UTILITIES

ucd anno	ucd extract	ucd hex to tet
ucd contour	ucd legend	write ucd
ucd cell to node	ucd crop	ucd hog
ucd iso	ucd offset	ucd probe
ucd rslice	ucd streamline	ucd threshold
ucd to geom	ucd tracer	ucd print
ucd vecmag	read ucd	field to ucd
ucd slice 2d		

CFD UTILITIES

read plot3d	cfd values

ANIMATION

AVS Animator	read frame seq	write frame seq
prepare video	output ImageNode	output VideoCreator

avs

Application Visualization System

Description

The Application Visualization System includes the following subsystems:

- **Image Viewer**—High-level tool for processing and viewing images
- **Graph Viewer**—High-level tool for graphing data
- **Geometry Viewer**—Tool for composing scenes that contain geometrically-defined objects. The objects must have been created by programs or ConvexAVS modules that use the geometry programming library. You can transform the objects, change the viewing parameters, and control the way in which the graphical images are rendered.
- **Network Editor**—Visual programming interface for connecting computational modules together into networks that perform visualization functions.

ConvexAVS can be run from any workstation or X terminal with color display hardware and an X11 server that supports at least an 8-plane PseudoColor visual.

ConvexAVS is a large application that makes heavy demands on your X server. ConvexAVS makes use of a variety of fonts in different sizes if they are available. This requires that a substantial amount of memory be available for use by the X server. With X terminals that do not have virtual memory access, it is possible to crash the server by making requests that exceed available memory. As a rule of thumb, a minimum of 12 megabytes for workstations and 4 megabytes for X terminals should be available.

To conserve screen space, you may prefer to minimize the border decorations added by many window managers.

ConvexAVS can usually automatically determine your screen size. You can also use initialization options set in a .avsrc file to change the window size. Smaller windows imply smaller fonts. However, some X servers do not have a large set of fonts sizes and may degrade the displayed menus and text windows.

There are three ways to affect how ConvexAVS starts:

- Command line options
- The .avsrc start-up file
- Environment variables

Options

To start ConvexAVS, enter

% **avs** [*options*]

The following command line options are recognized:

-appsfile *file_specification*

(Start-up file equivalent: **ApplicationsFile**)

Use this option to specify the location of the ConvexAVS applications file (AVS.applns). This file specifies the buttons that appear in the Applications menu.

-class *string*

(Start-up file equivalent: none)

This is the command line option equivalent of the DISPLAYCLASS environment variable. You can use it to make ConvexAVS behave in different ways when it is started from different types of display hardware. **-class** has two effects:

- An Xdefaults file specifies the look of the ConvexAVS interface; what shades of grey are used for command buttons, what fonts to use, the background, and so on. When **-class** *string* is given, ConvexAVS does not use the default avs.Xdefaults file, located in the /usr/avs/runtime directory. Instead, it looks for an Xdefaults.*string* file in the /usr/avs/runtime directory and uses it. You can specify an alternate location by setting the AVSXDEFAULTS environment variable.
- ConvexAVS will look for a .avsrc.*string* file in your home directory and use it instead of your usual .avsrc file.

-class x is used when running ConvexAVS from an X terminal or workstation.

-cli [*argument*]

(Start-up file equivalent: none)

Run ConvexAVS with the Command Language Interpreter functioning in the terminal emulator window from which ConvexAVS was invoked. This takes an optional argument, which is a CLI command string, to be executed after ConvexAVS starts up.

Note

Demo scripts cannot be run from the avs_help browser if the -cli option is used.

-data *directory*

(Start-up 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. This redirects ConvexAVS's default data input directory to your own data files. The default data directory is /usr/avs/data.

-display *display-name*

(Start-up file equivalent: none)

Specifies the X Window System display on which ConvexAVS is to execute. This overrides the current setting of the **DISPLAY** environment variable.

-geometry *[geom-option(s)]*

(Start-up file equivalent: none)

Automatically invokes the Geometry Viewer subsystem. You can include the following options that are specific to this subsystem. This option and any associated sub-options must be entered as the last arguments on the command line:

-defaults *file-name*

Specifies a Geometry Viewer defaults file.

-dir *path-name*

Specifies *path-name* as the default directory used by the functions Read Object, Save Object, Read Scene, Save Scene, and the Read and Save functions in the Edit Property window.

The default data directory is /usr/avs/data.

-filter *path-name*

Specifies *path-name* as the directory to search for geometry conversion utilities called *name*_to_geom.

The default directory for these programs is /usr/avs/bin.

-geometry *geom_spec*

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

-noroll

Disables auto-roll. Use this option to disable the ability to spin an object in the pixmap display window.

-scene *scene-file*

Automatically loads a scene from disk storage. This option executes the Geometry Viewer's Read Scene function, using the *scene-file*.scene file.

-usage

Displays a list of Geometry Viewer start-up options.

Note

The -geometry option and any associated sub-options must be entered as the last argument on the command line.

-graph

(Start-up file equivalent: none)

Automatically invokes the ConvexAVS Graph Viewer subsystem.

-image

(Start-up file equivalent: none)

Automatically invokes the ConvexAVS Image Viewer subsystem.

-library *filespec*

(Start-up file equivalent: **ModuleLibraries**)

Specifies which ConvexAVS module library files to load into the Network Editor. Module library files are ASCII files describing sets of modules. The Supported library in /usr/avs/avs_library is one example. This option allows you to load your own set of modules instead of relying on the system default module libraries. It is equivalent to using the Read Module Library function. This option causes ConvexAVS to load only the libraries specified on the command line. Be sure to give the name of a valid module library file; not a directory or a module binary.

-modules *directory*

(Start-up file equivalent: none)

Specifies the directory in which the Network Editor subsystem initially looks for executable modules. All executable files in the directory are examined to determine whether they contain one or more modules.

This option differs from **-library** 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 this more than once to have ConvexAVS search through multiple directories for modules. This is a tool for loading individual modules that you have not yet put 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 /usr/avs/avs_library.

-netdir *directory*

(Start-up file equivalent: **NetworkDirectory**)

Specifies the directory in which the Network Editor subsystem initially looks for network files (Read Network and Write Network functions). Use this tool to redirect ConvexAVS's default network focus away from the samples provided and onto your own network files.

The default network directory is /usr/avs/networks.

-network *network-file*

(Start-up file equivalent: none)

Automatically invokes the Network Editor subsystem and loads the specified network file.

-path *directory*

(Start-up file equivalent: **Path**)

Specifies the directory tree in which ConvexAVS is installed.

-separate

(Start-up file equivalent: none)

This option disables ConvexAVS'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.

-server

(Start-up file equivalent: none)

This option opens a connection that an external process can use to connect to ConvexAVS and exchange with it a stream of CLI commands and their output.

Note

Demo scripts cannot be run from the avs_help browser if the -server option is used.

-shm/noshm

(Start-up file equivalent: **SharedMemory**)

This turns the ConvexAVS shared memory option on and off. When shared memory is on, ConvexAVS keeps only one copy of field and UCD data that all modules in a network share. (Geometry-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, ConvexAVS's use of the finite shared memory area is interfering with other applications. Shared memory is on by default.

-size *Xdim* x *Ydim*

(Start-up file equivalent: **ScreenSize**)

Specifies size, in pixels, to use for ConvexAVS's virtual display screen size. ConvexAVS automatically resizes its interface to fit into the virtual screen. You could use this to confine ConvexAVS to run within one section of your screen instead of across the whole screen. The upper-left portion of the screen is used. The aspect ratio should be 5 by 4 for proper results. A minimum pixel size of 400 by 400 is allowed.

-usage

Displays a usage message for ConvexAVS. No ConvexAVS session is started.

-version

Displays the ConvexAVS version number. No ConvexAVS session is started.

-viewer *viewer-file*

(Start-up file equivalent: none)

Automatically creates a viewer that provides turnkey access to a group of existing networks. The Image and Volume Viewer systems under Applications in the main menu are implemented in this way. You must enclose the complete path specification within quotation marks. By default, ConvexAVS looks in the current directory, then looks in the path specified in $PATH.

-volume

Automatically invokes the Volume Viewer application.

Start-up file

When it begins execution, ConvexAVS searches for a start-up file in the following order:

./.avsrc	Current directory
$HOME/.avsrc	Home directory
/usr/avs/runtime/avsrc	System directory

Only one of these files is read. A default /usr/avs/runtime/avsrc file is included on the ConvexAVS distribution tape.

Each line of the start-up file consists of keyword-value pair, with white space separating the keyword and the value. For example:

```
ModuleLibraries      /avs_library/Supported
NetworkWindow        867x567+407+2
NetworkDirectory     /usr/henry/avs/nets
DataDirectory        /usr/neal/avs/data
```

Often, the keyword corresponds to a command line option. If you use a command line option, it overrides the specification in the start-up file.

The start-up file keywords that have command line equivalents are:

ApplicationsFile

> (Command line equivalent: **`-appsfile`**)
>
> Specifies the complete file specification for the ConvexAVS applications file (AVS.applns). This file is used to specify the applications available through the Applications menu.

DataDirectory

> (Command line equivalent: **`-data`**)
>
> Specifies the directory in which the various ConvexAVS data input file browsers used in the subsystems (Image Viewer, Graph Viewer, and Geometry Viewer) and Network Editor modules read data modules initially look for data files. This is the main tool to refocus ConvexAVS's data input attention off the sample data files and onto your own.

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 with the Network Editor's Select Module Library function under Module Tools. There is no way to continue the list of module libraries to a new line; the list must be on one line. Line length is limited to 255 characters.

NetworkDirectory

(Command line equivalent: **-netdir**)

Specifies the directory in which the Network Editor subsystem initially looks for network files.

Path

(Command line equivalent: **-path**)

Specifies the directory tree in which ConvexAVS is installed.

ScreenSize *XDIMxYDIM*

(Command line equivalent: **-size**)

Specifies the size of ConvexAVS's virtual display in pixels, confining ConvexAVS to run within this area. ConvexAVS scales its interface to fit the virtual screen.

SharedMemory *switch*

(Command line equivalent: **-shm/noshm**)

Specifying **SharedMemory off** turns off shared memory.

The start-up file keywords that have no command line equivalents are:

BoundingBox (1 or 0)

If **BoundingBox 1** is set, then the ConvexAVS Image Viewer and Geometry Viewer 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 in realtime, it only moves a wirebox representation of the object. Only when you release the mouse button is the object or subimage rendered at its new location.

Colors *red green blue gray*

This option controls how many cells of a system colormap ConvexAVS will attempt to allocate to itself when it starts on a pseudo-color system. The *red green blue* and *gray* variables represent integers for red, green, blue, and gray.

DisplayGeometryWindow *Xgeometry*

Specifies the X Window System geometry of the Geometry Viewer display window.

DisplayPixmapWindow *Xgeometry*

Controls the default X Window System geometry of the **display pixmap** module's window.

ForceXRender (1 or 0)

Force ConvexAVS to always use the X renderer. By default, ConvexAVS uses the supported available renderer. Set this variable to 1 to enable.

Gamma *value*

Used by your display to make the screen lighter or darker. The *value* ranges from 1.0 (darker) to 3.0 (lighter) intensities. The default is 1.5. **AVS_GAMMA** is the environment variable equivalent.

GridSize *size*

Controls the size in pixels of the Layout Editor's alignment squares when the **Snap to Grid** button is on. The default is 10.

HelpPath *directory ...*

Expands the list of directories that ConvexAVS will search to find a module's documentation when you click the **Show Module Documentation** button in the module's Module Editor window. This is useful when you are using modules other than the set provided with ConvexAVS.

Hosts *fullfilespec*

Gives the name of a hosts file that lists machines, access methods, and directories of remote modules. It provides a personal override to the /usr/avs/runtime/hosts default file when you click on the Network Editor's **Read Remote Module(s)** button under Module Tools.

ImageAutomagnify (0 or 1)

Causes the magnification factor that is selected for an image to be based on the window's size. The default is off (0).

ImageScrollbars (0 or 1)

If set to 0 (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.)

ModulePanelHeight *height*

Controls the proportion of the Network Construction window devoted to the module palette as opposed to the Workspace. Specify the palette size as height in pixels.

NetworkWindow *Xgeometry*

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.

NetWriteAllParams *boolean*

Save all parameter values when writing out a network with the Network Editor's **Write Network** button, not just those changed since the network was created. The default is to save only the changed parameters.

PrintNetwork *command*

The Network Editor's **Print Network** button normally sends output to your default printer. This option lets you specify an alternate print command to run. The output file name is appended to the *command* string you provide.

StackSelector *option*

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

VisualType *visualtype*

ConvexAVS attempts to choose the best visual for your workstation. Use this option to specify a visual type of **PseudoColor**, **Direct**, or **TrueColor**. ConvexAVS searches the X server's visual list until it finds the first visual with the given visual type and uses it. You can also specify the visual identification number if your X server supports multiple visuals:

```
VisualType VisualID 080064
```

WindowMgr *mgr*

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 /usr/avs/runtime/avs.Xdefaults file. The recognized values are: **awm**, **mwm**, **twm**, **uwm**, **olwm**, and **dxwm**.

XWarpPtr (on or off)

Causes the mouse cursor to be moved (warped) into dialog boxes when they appear. This option is off by default.

Variables

ConvexAVS uses the following environment variables:

AVS_FORCEX

Forces the use of the X renderer on PEX- and GL-capable workstations. By default, ConvexAVS checks your terminal type and uses the fast graphics of PEX and GL geometry renderers when possible.

If you overflow the memory capabilities on your workstation, you should use this option and take advantage of the memory capabilities of your CONVEX supercomputer.

AVS_GAMMA *value*

Used by your display to make the screen lighter at its lowest value (1.0) or darker at the highest value (3.0). The default is 1.5. The start-up file equivalent is **Gamma**.

AVS_HELP_PATH

Specifies one or more locations in the file system for ConvexAVS to use when searching for online help files. This is a colon–separated list of complete path names.

AVS_VISUAL *visualtype*

Specifies a *visualtype* of **PseudoColor**, **DirectColor**, or **TrueColor**, or "**VisualID** *number*." The start-up file equivalent is **VisualType**.

For example:

```
setenv AVS_VISUAL "VisualID 080064"
```

AVSXDEFAULTS

Specifies the complete path for the alternate Xdefaults file to be used by ConvexAVS.

DISPLAY (*host:server.screen*)

Used by the X Window System to indicate the display screen at which you are working. For example, **big-dawg:0.0**.

DISPLAYCLASS X

Specifies the display class. Refer to the discussion on the **-class** command line option.

Other environment variables may be required to configure the system for your window manager.

Accessing help

To access the online help facility from your command line, you must be using a color X terminal. Enter

`% avs_help` [*module*]

where *module* is the optional name of a module.

AVS Animator

Create and edit animation scripts

Summary

Name AVS Animator

Type data input

Inputs none

Outputs frame number
frames/second
current time
active channel

Parameters

Name	*Type*
Frames/Second	integer
Keyframe Increment	typein
New Keyframe Time	typein
Key Edit	radio
Smooth	radio
Key Advance	radio
Wireframe	radio
WYSIWYG	radio
State	radio
Modules	radio
Cameras	radio
Lights	radio
Objects	radio
Expand List	radio
Show Active Keys	radio
Set Play List	radio
Clear Play List	radio
Move Keyframe	radio
Add New Key	radio
Mode Toggle	radio
Exit Animator	radio
String Value	typein
Integer Value	typein integer
Float Value	typein real
Delete Key	oneshot
Set Values	oneshot
Minutes	typein integer
Seconds	typein integer
Frames	typein integer
Set New Time	oneshot

Reset	oneshot
Cancel	oneshot
Reverse	oneshot
Single Reverse	oneshot
Stop	oneshot
Single Forward	oneshot
Forward	oneshot
Continuous	oneshot
Single Play	oneshot
Bounce	oneshot

Description

The **AVS Animator** module is part of the **AVS Animation Application** for creating, reading, writing, and editing animation scripts.

The application consists of (but is not limited to) the following modules:

- AVS Animator
- write frame seq
- read frame seq
- prepare video
- output VideoCreator
- output ImageNode

The **AVS Animator** can record the state of both the Geometry Viewer subsystem and the module networks and can play back previously recorded animation scripts. Intermediate frames are generated automatically by interpolating the object/camera/light settings (position, scale, rotation, colors, and so on) and module parameters between recorded frames.

Parameters

Refer to *Animating AVS Data Visualizations* for details about this module and its parameters.

Outputs

Integer (frame number)

This outputs the frame number.

Integer (frames/second)

This outputs the frames per second.

Real (current time)

This outputs the current time position in seconds.

Active Channel (field 2D scalar real uniform)

This outputs the value of the current active channel. A valid channel must be selected before a value is output. You must click on the name of the desired channel in the play list to highlight it. If no channel is selected, then there is no output. The channel selected cannot be a module name. Valid channels output integer, floating (real) or character values.

This output can be connected to the **graph viewer** module's Linear input port to view a single channel's value as a function of time.

Example

The **AVS Animator** module can be used in conjunction with most networks and the Geometry Viewer to generate animation scripts. Refer to *Animating AVS Data Visualizations* for specific examples.

Note

If you save a network that contains the **AVS Animator**, this module will not be written to that network file.

Related modules

prepare video, read frame seq, write frame seq, output VideoCreator, output ImageNode

See also

Refer to the book, *Animating AVS Data Visualizations* for detailed information about using this module.

animate lines

Animate stream lines for a vector field

Summary

Name animate lines

Type filter

Inputs geometry
upstream transform

Outputs geometry
upstream transform

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Objects	text			
Maxlen	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 ConvexAVS 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. But you can store frames in **display pixmap** and play them back faster.

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 **render geometry** in a network, **render geometry** 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 invisible. This gives direct mouse-manipulation control over **stream lines'** sample probe.

Outputs

Animated Lines (geometry)

Successive portions of the input streamlines are output sequentially.

Upstream Transform (optional; invisible, autoconnect)

Parameters

Objects

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

Maxlen

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

A oneshot button that initiates the animation of the streamlines.

Example

The network in Figure 1 reads in a 3D vector field and calculates streamlines for the field. **animate lines** is used to dynamically represent the output of **stream lines**.

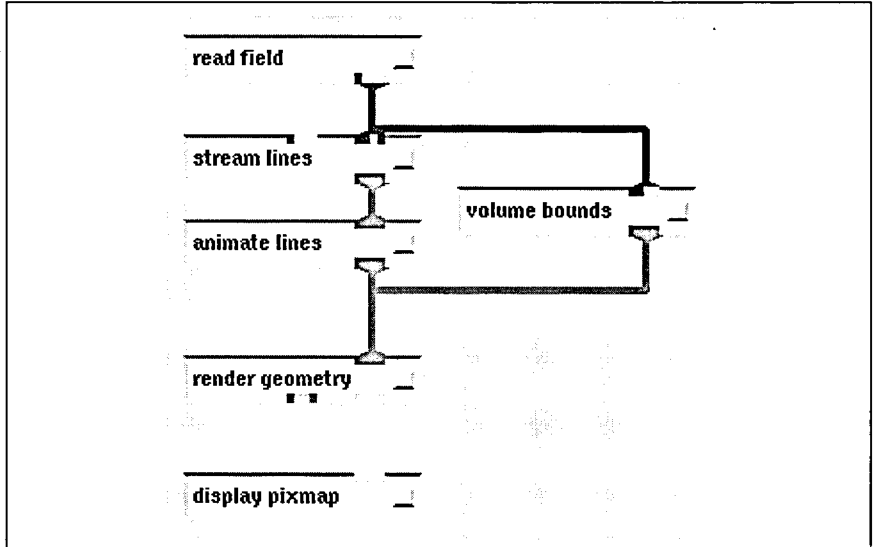


Figure 1
animate lines module in an example network

Related modules

hedgehog, particle advector, stream lines

animated float

Send a sequence of floating-point numbers to a module's parameter port

Summary

Name animated float

Type data input

Inputs none

Outputs real

Parameters

Name	*Type*	*Default*	*Min*	*Max*
min value	float typein	0.0	none	none
max value	float typein	0.0	none	none
steps	int typein	10	2	none
sleep	switch	on		
choice	choice	one time		

Description

The **animated float** module automatically modifies floating-point parameters. It is used to create simple animations or to drive your 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 ConvexAVS 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.

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.

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 flip book cycle of images.

Parameters

min value

A typein to specify the lowest value in the floating point number sequence. It is typed in as a real number. There are no upper or lower bound restrictions. The default is 0.0.

max value

A typein to specify the maximum value in the floating point number sequence. It is typed in as a real number. 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 selecting **sleep**, some number of additional values may continue to flow through the network before **animated float** actually goes to sleep.

choice

A set of choices that determine what **animated float** does when it reaches its maximum value:

one time With **one time** on (the default), the values are sent only once (for example, 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 (for example, 1 2 3 4 5 1 2 3 4 5...).

bounce When **bounce** is selected, the values count up and down repeatedly (for example, 1 2 3 4 5 4 3 2 1...).

Outputs

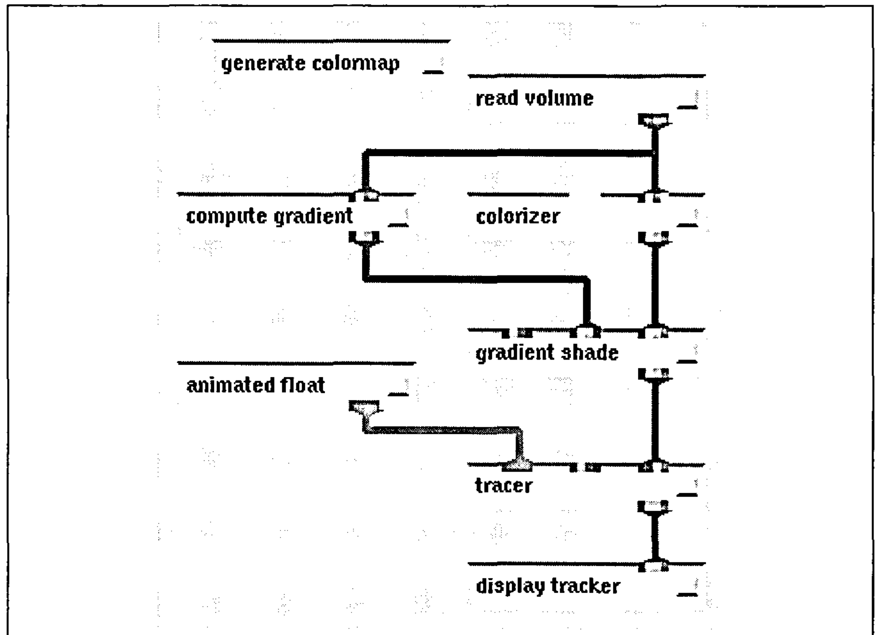
Float Value (real)

A floating-point number intended to be input into a floating-point parameter port of another module.

Example

The network in Figure 2 animates the alpha value (transparency) of a volume that has been gradient shaded, then rendered with **tracer. display tracker** sends an upstream transform to the **tracer** module.

Figure 2
animated float module in an example network



Related modules

Modules that can process **animated float** output are those 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 **animated float** module.

animated integer

Send a sequence of integers to a module's parameter port

Summary

Name animated integer

Type data input

Inputs none

Outputs integer

Parameters

Name	*Type*	*Default*	*Min*	*Max*
min value	int typein	0	none	none
max value	int typein	0	none	none
steps	int typein	10	2	none
sleep	switch	on		
choice	choice	one time		

Description

The **animated integer** module automatically modifies integer parameters. This can be used to create simple animations or to drive your 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)/steps), 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 ConvexAVS 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.

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.

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 flip book cycle of images.

Parameters

min value

A typein to specify the lowest value in the integer number sequence. It is typed in as a whole number. This parameter has no upper or lower bounds. The default is 0.

max value

A typein to specify the maximum value in the integer number sequence. It is typed in as a whole number. 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)/steps delta calculation produces real values, each value is rounded down to the nearest whole integer value. **steps** 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 selecting **sleep,** some number of additional values may continue to flow through the network before **animated integer** actually goes to sleep.

choice

A set of choices that determine what **animated float** does when it reaches its maximum value. **choice** descriptions follow:

one time With **one time** on (the default), the values are sent only once (for example, 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 (for example, 1 2 3 4 5 1 2 3 4 5...).

bounce When **bounce** is selected, the values count up and down again repeatedly (for example, 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

The network in Figure 3 animates slices through a volume.

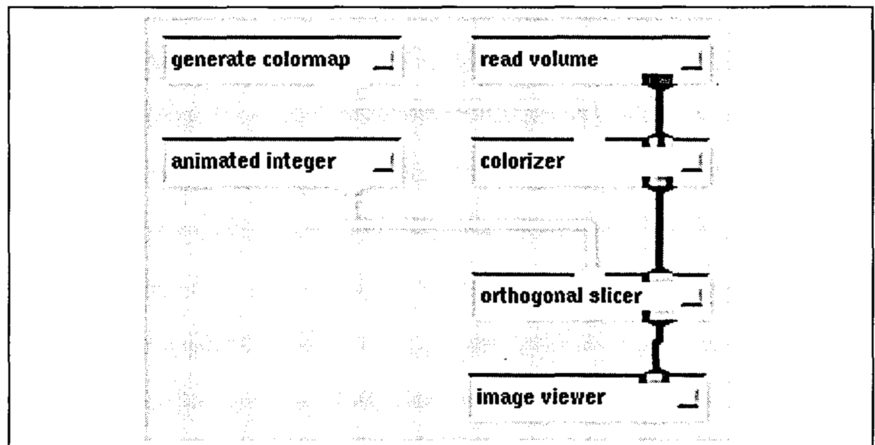


Figure 3 animated integer module in an example network

Related modules

Modules that can process **animated integer** output are those 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 **animated integer** module.

antialias

Antialias an image

Summary

Name	antialias
Type	filter
Inputs	field 2D uniform 4-vector byte
Outputs	field 2D uniform 4-vector byte
Parameters	none

Description

The **antialias** module down samples an image using a Gaussian 3 by 3 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 512 by 512 image becomes a 256 by 256 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

The network in Figure 4 reads an image, antialiases it, and displays it through the **display image** module.

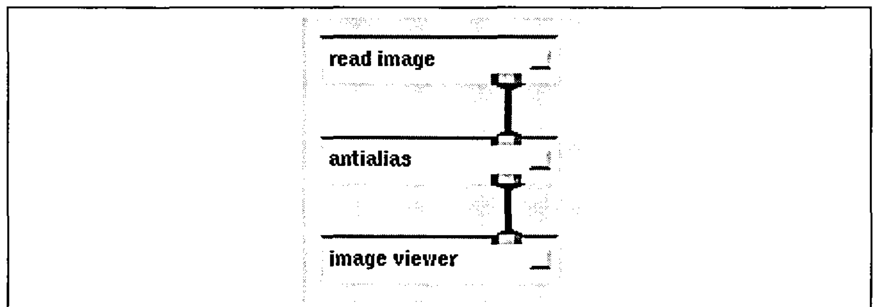


Figure 4
antialias module in an example network

Related modules

Modules that could provide the **Image** input are colorizer, composite, convolve, field math, local area ops, read image, and replace alpha.

Modules that can process **antialias** output are extract scaler, image viewer, and display image.

See also

downsize and **interpolate**

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

arbitrary slicer

Map 3D scalar field to 3D mesh

Summary

Name arbitrary slicer

Type mapper

Inputs field 3D scalar *any-data any-coordinates*
colormap
upstream transform

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. Because the mesh vertices do not coincide with the original lattice points, an interpolation method can be used.

By default, the volume is placed at the origin and the slice plane is the XY-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 **render geometry** module that tells it how the slice plane has been 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.

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 irregular.

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 **render geometry** module in a network, and the slice plane object has been picked, **render geometry** 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 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 that 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 8 by 8.

Sampling Style

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 a geometry.

Example

Figure 5 shows a usage of the **arbitrary slicer** module. The **volume bounds** modules gives a reference frame for orienting the slice plane.

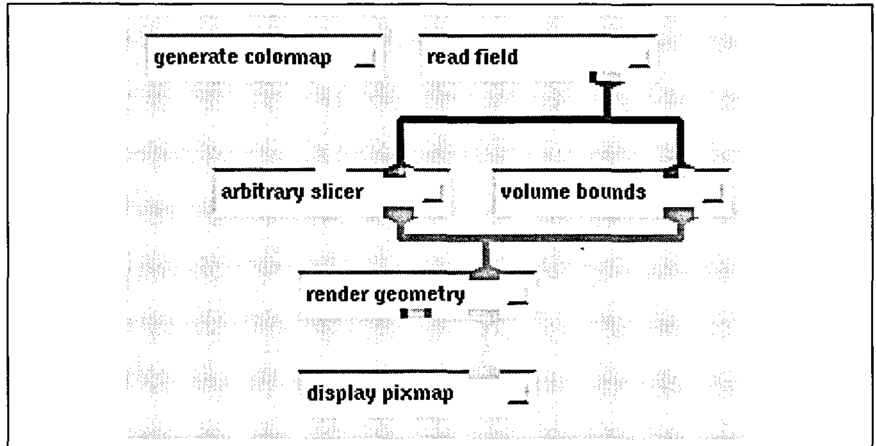


Figure 5
arbitrary slicer module in an example network

Related modules

Modules that could provide the **Data Field** input are read field and read volume.

Modules that can replace **arbitrary slicer** are orthogonal slicer and thresholded slicer.

Module that can process **arbitrary slicer**'s output is render geometry.

See also

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

background

Create a shaded backdrop image

Summary

Name	background
Type	data input
Inputs	field 2D 4-vector byte uniform
Outputs	field 2D 4-vector byte uniform

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 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 128 by 128 pixel-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 0.670 (blue); the defaults for the lower left and right are 0.0. Other values are:

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

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) and value (how much black is mixed in with the hue). 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 128 by 128. 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.

Examples

1.

The network in Figure 6 creates a shaded image and writes the image to disk.

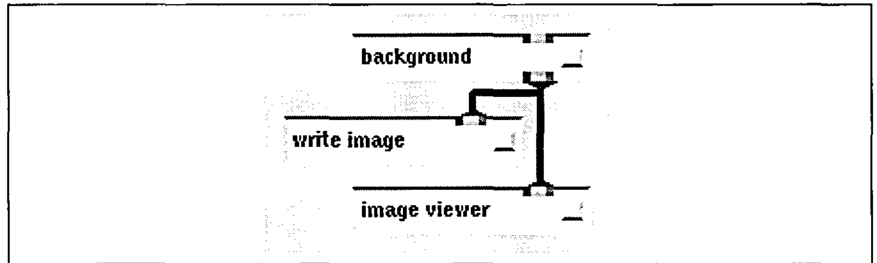
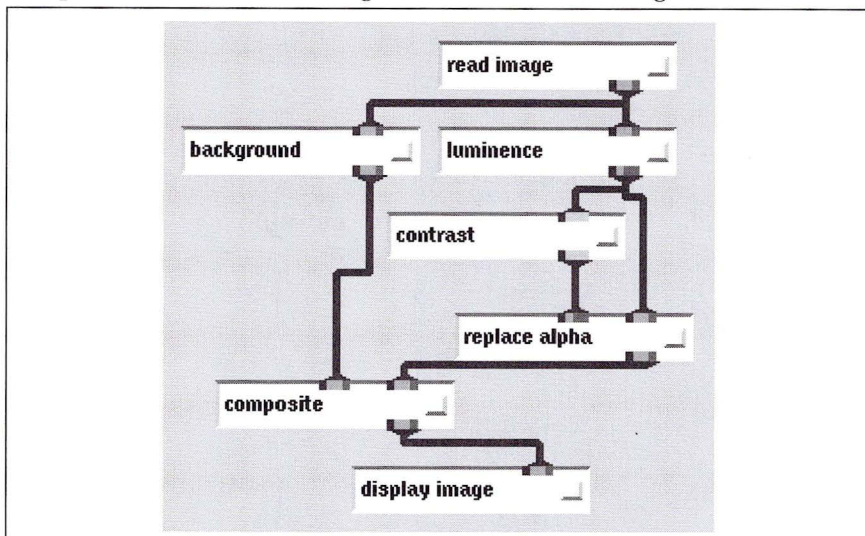


Figure 6
background module in an example network

2.

The network in Figure 7 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.

Figure 7
background module in an example network



Related modules

Modules that could provide the **Image** input are read image and pixmap to image.

Modules that can process **background** output are image viewer and composite.

See also

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

boolean

Send a boolean value to boolean parameter port(s)

Summary

Name	boolean		
Type	data input		
Inputs	none		
Outputs	boolean		
Parameters	*Name*	*Type*	*Default*
	boolean	boolean	off

Description

The **boolean** module sends a single 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.

Parameters

boolean (boolean)

The single 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

In Figure 8, the **boolean** module has been connected to **isosurface's flip normals** parameter.

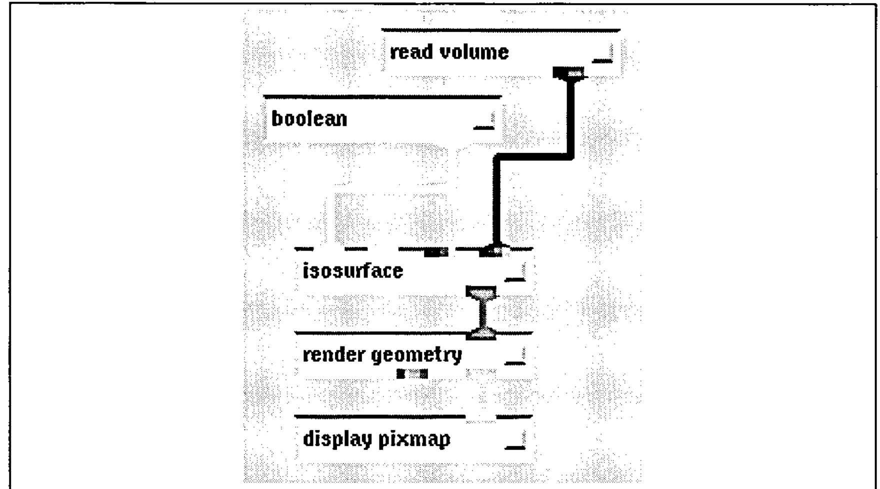


Figure 8
boolean module in an example network

Related modules

Modules that can process **boolean** output are those with boolean type parameters.

bubbleviz

Generate spheres to represent values of 3D field

Summary

Name	bubbleviz
Type	mapper
Inputs	field 2D/3D scalar *any-data any-coordinates* colormap
Outputs	field irregular 1D 3-coord 4-vector real

Parameters

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

Description

The **bubbleviz** module generates spheres of various radii and colors at the element locations of a 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** module's opacity panel.

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

Systems that 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 is in the range 1-8; using a low value can improve the performance of **bubbleviz** considerably.

Inputs

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

The principal input data for the **bubbleviz** module is a 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.

Colormap (optional; colormap)

The optional colormap may be of any size. Because each input data 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 data.

Outputs

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

The output is a list of points in 3D space, with a 4-vector real 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 Scale** parameter.
- The second, third, and fourth elements of the lookup value specify the red-green-blue components of the sphere's color (0.0 = no color; 1.0 = maximum color).

Parameters

Radius Scale

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 Scale** is used to scale the opacity element in the input colormap.

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

Example

A network using **bubbleviz** appears in Figure 9.

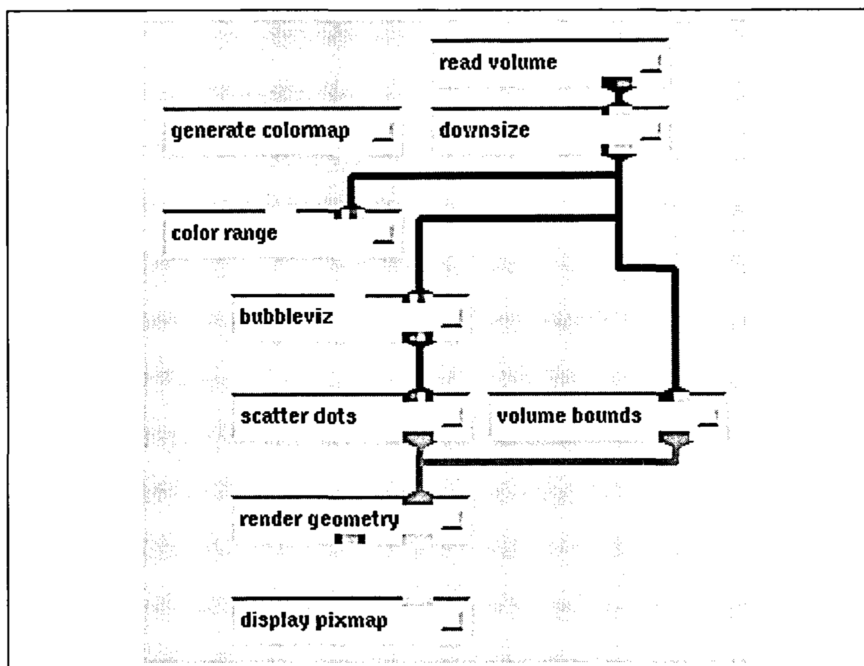


Figure 9
bubbleviz module in an example network

Related modules

Modules that could provide the **Data Field** input are read volume and read field.

Modules that could be used in place of **bubbleviz** are colorizer and gradient shade.

Module that can process **bubbleviz** output is scatter dots.

Limitations

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

In ConvexAVS, the **bubbleviz** module only produces spheres if your renderer is using X and not PEX or GL.

See also

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

cfd values

Calculate values for a field containing read plot3d data

Summary

Name	cfd values
Type	filter
Inputs	field irregular 5-vector float
Outputs	field 1- to 12-vector irregular float

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Gamma	float	1.4	1	5
Gas constant	float	1	0	5
density	boolean	true		
x momentum	boolean	true		
y momentum	boolean	true		
z momentum	boolean	true		
stagnation	boolean	true		
energy	boolean	true		
pressure	boolean	true		
enthalpy	boolean	true		
mach number	boolean	true		
temperature	boolean	true		
total pressure	boolean	true		
total temp	boolean	true		
Vector Length	integer	12	1	12

Description

cfd values takes the 5-vector irregular field, which **read plot3d** outputs, and derives seven 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. The input field must have a 5-vector at each location.

The field that **cfd values** receives from **read plot3d** has the following five values: density, x momentum, y momentum, z momentum, and stagnation.

From these five values, **cfd values** computes seven new values: energy, pressure, enthalpy, mach number, temperature, total pressure, and total temp. The gamma constant(γ) and the gas constant (R) are controllable parameters while 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:

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

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

$$machnumber\,(M) = \frac{\sqrt{(U_2^2 + U_3^2 + U_4^2)}}{U_1\sqrt{\frac{\gamma p}{U_1}}}$$

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

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

$$totaltemperature\,(T_0) = T\left(1 + \frac{\gamma - 1}{2}M^2\right)$$

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

In calculating the seven 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. The **cfd values** module only computes the values required by your selections.

This module is in the unsupported library.

Inputs

Field Input (required; field 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 three to five values at each field location.

Parameters

Gamma

A floating-point value between one and five, which determines the value of the gamma constant. The previous equations assume an ideal gas with a constant ratio of specific heats.

Gas Constant

A floating-point value between zero and five, which determines the value of the gas constant.

values

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 one and 12 values at each field location.

Vector Length

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

Outputs

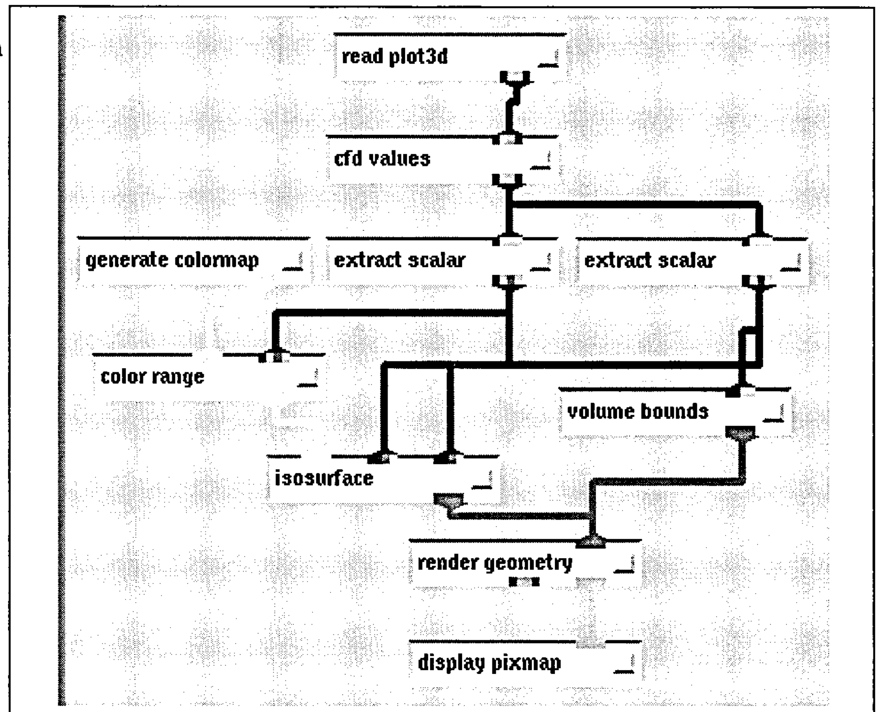
Field Output (field 1- to 12-vector irregular float)

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

Example

The network in Figure 10 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 generates an isosurface of the density in a field and then colors this isosurface based on the temperature values at each point on the isosurface.

Figure 10
cfd values module in an example network



Related modules

Module that could provide the **Data Field** input is read plot3d.

Modules that can process **cfd values**'s output are isosurface, orthogonal slicer, hedgehog, bubbleviz, and tracer.

See also

Pieter Buening, *PLOT3D Reference Manual.*

The example script READ PLOT3D and CFD VALUES demonstrates the **cfd values** module.

character string

Send a string to string parameter port(s)

Summary

Name	character string		
Type	data input		
Inputs	none		
Outputs	string		
Parameters	*Name*	*Type*	*Default*
	character	typein	off

Description

The **character string** module sends a single string to one or more string parameter ports on one or more receiving modules. Its purpose is to make it possible for you 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.

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

Parameters

character

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

Outputs

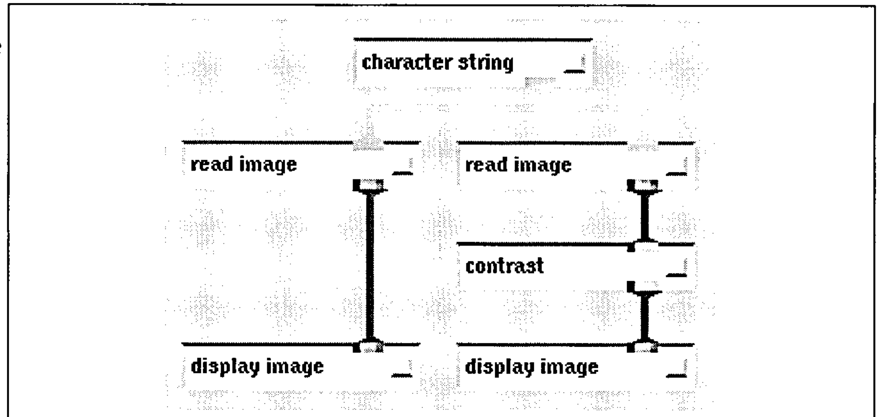
string (string)

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

Example

The network in Figure 11 shows an example of how the **character string** module can be used to send a string constant to two different modules.

Figure 11
character string module in an example network



Related modules

Modules that can process **character string**'s output are those with string type parameters.

See also

The example script CHARACTER STRING demonstrates the **character string** module.

clamp

Restrict values in data field

Summary

Name clamp

Type filter

Inputs field *any-dimension n*-vector *any-data any-coordinates*

Outputs field *any-dimension n*-vector *any-data any-coordinates*

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 clamped, 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.

Difference between the **clamp** and **threshold** modules are:

- **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.

clamp

Inputs

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

The input data may be any 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*-vector *same-data same-coordinates*)

The output field has the same dimensionality and type as the input field.

Example

The network in Figure 12 reads in a field. The **statistics** module is used to display the field contents with and without clamping.

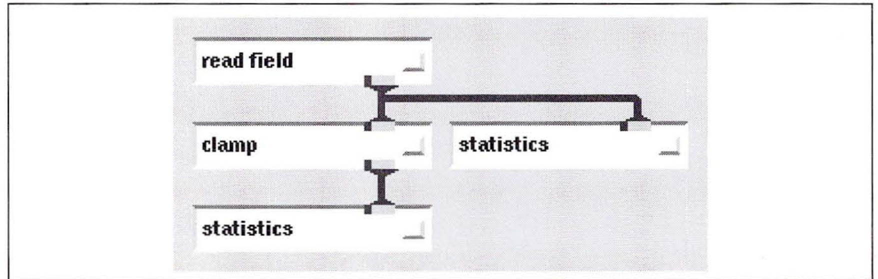


Figure 12
clamp module in an example network

Related modules

Module that could provide the **Data Field** input is read volume.

Module that could be used in place of **clamp** is threshold.

Module that can process **clamp** output is colorizer.

Modules that tell you the range of data in the field are statistics, print field, and generate histogram.

See also

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

color range

Scale colormap to the range of data in a field

Summary

Name	color range
Type	data input
Inputs	field *any-dimension* scalar *any-data any-coordinates* colormap
Outputs	colormap
Parameters	none

Description

color range adjusts the minimum and maximum values of a colormap to those of a field, thus normalizing the colormap to the range of the data in the field. To do this, **color range** examines a scalar 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 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 (for example, **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 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 colormap 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. This 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 field whose data structure is 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 field's data structure and the output colormap. Because **color range** can modify the original field, data passing through this module is not shared.

Colormap (required; colormap)

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

Outputs

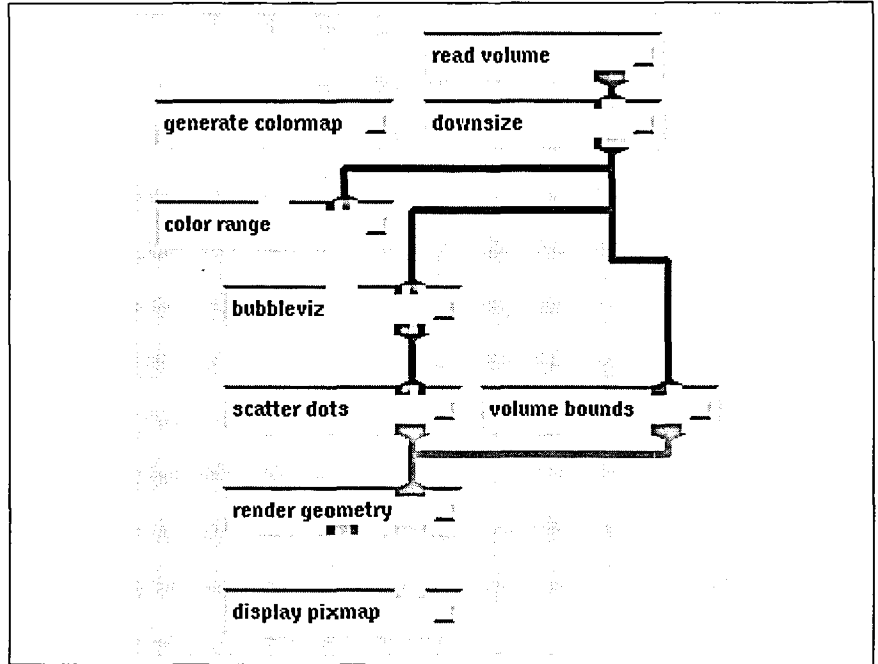
Colormap (colormap)

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 network in Figure 13 reads in a 3-vector field (that is, every field location has three values associated with it). The **extract scalar** module selects one of the field's 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.

Figure 13
color range module in an example network



Related modules

Modules that could provide the **Data Field** input are read field and extract scalar (for fields with vectors).

Module that could provide the **Color Map** input is generate colormap.

Modules that can process **color range** output are arbitrary slicer, bubbleviz, colorizer, field legend, field to mesh, isosurface, and probe.

See also

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

colorizer

Convert field of data values to color values

Summary

Name	colorizer
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:

1	opacity	red value	green value	blue value
2	opacity	red value	green value	blue value
3	opacity	red value	green value	blue value
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
146	opacity	red value	green value	blue value
147	opacity	red value	green value	blue value
148	opacity	red value	green value	blue value
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

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.

Colormap (optional; colormap)

The optional colormap may be of any size, but any entries beyond the 256th are unused. If this input is omitted, a gray-scale colormap is used (0 = black; 255 = white).

Outputs

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

Each input value is transformed into a color value, which is structured as four bytes. The red, green, and blue bytes specify a true-color pixel value. The auxiliary byte is used to specify an opacity value (0 = completely transparent; 255 = completely opaque).

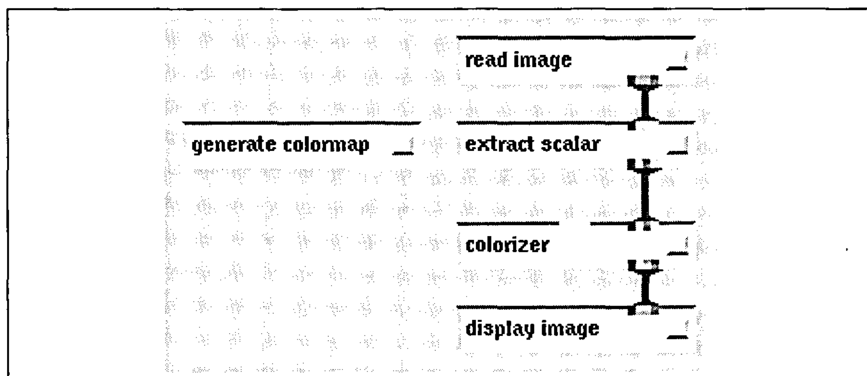
The dimensionality of the output field is the same as that of the input field. For byte input, the output field is four times as large as the input field because each byte (8 bits) is converted to a color value (32 bits).

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 network in Figure 14 reads in an 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**.

Figure 14
colorizer module in an example network



Related modules

Modules that could provide the **Data Field** input are read volume and field to byte.

Module that could provide the **Colormap** input is generate colormap.

Module that could be used in place of **colorizer** is arbitrary slicer.

Modules that can process **colorizer** output are gradient shade, display image, and tracer.

colormap manager

Share colormaps among subnetworks

Summary

Name	colormap manager	
Type	data input	
Inputs	none	
Outputs	colormap	
Parameters	*Name*	*Type*
	Colormap Manager	colormap
	Colormap Choices	choice

Description

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

- **arbitrary slicer**
- **bubbleviz**
- **colorizer**
- **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.

This module is in the unsupported library.

Parameters

Colormap Manager

A colormap generator widget. Refer to the **generate colormap** module reference for details on using this widget.

Colormap Choices

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

Outputs

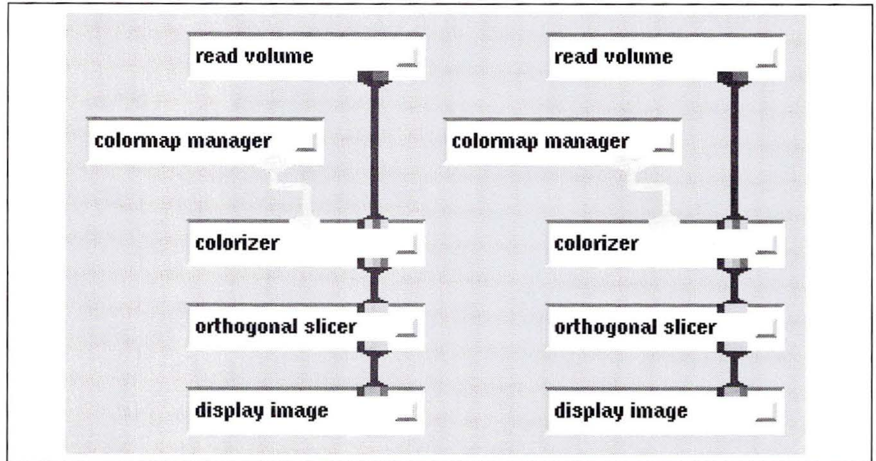
Colormap (colormap)

The output is a colormap.

Example

Suppose the two subnetworks in Figure 15 are active, created to slice through two different databases.

Figure 15
colormap manager module in example networks



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

See also

colormap manager modules are used in both the Image Viewer and Volume Viewer applications.

combine scalars

Combine scalar fields into a vector field

Summary

Name	combine scalars
Type	filter
Inputs	field *any-dimension* scalar *any-data any-coordinates* field *any-dimension* scalar *any-data any-coordinates* field *any-dimension* scalar *any-data any-coordinates* field *any-dimension* scalar *any-data any-coordinates*
Outputs	field *same-dimension n*-vector *same-data*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
vector length	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 useful for constructing images or gradient fields from separately computed components.

The inputs ports on this module's 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

No inputs are absolutely required, but at least one must be present. If an input field is omitted, zero values are output in the corresponding element of each output vector.

Channel 0 (optional; field *any-dimension* scalar *any-data any-coordinates*)

> A set of values to be output in the first vector element 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 vector element 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 vector element of the output vectors.

Channel 3 (optional; field *any-dimension* scalar *any-data any-coordinates*)

A set of values to be output in the fourth vector element of the output vectors.

Parameters

vector length

Specifies the output field's vector length. The number of input ports that appear on the module's icon varies according to the value of this parameter.

Outputs

Vector Field (field *same-dimension n*-vector *same-data*)

The scalar input streams are assembled into a single output stream consisting of vectors, whose length is specified by **vector length**. The coordinate type (uniform, rectilinear, or irregular) of the output field is the same as the rightmost, nonempty input field.

Example

The network in Figure 16 shows **combine scalars**.

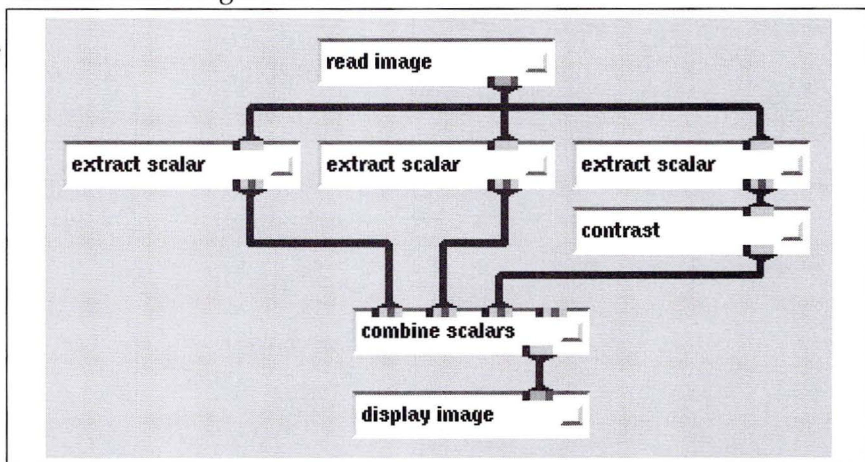


Figure 16
combine scalars module in an example network

Related modules

extract scalar

See also

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

compare field

Compare two fields, display and write data difference

Summary

Name	compare field
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			

Description

The **compare field** module compares any two identically-structured 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 Layout Editor.

compare field was originally written to ensure 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.

Inputs

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

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

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

The second 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, and so on), and the type of coordinate system (uniform, rectilinear, irregular), if a comparison of the two fields' data is to be done.

Parameters

Do Compare

A oneshot 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.

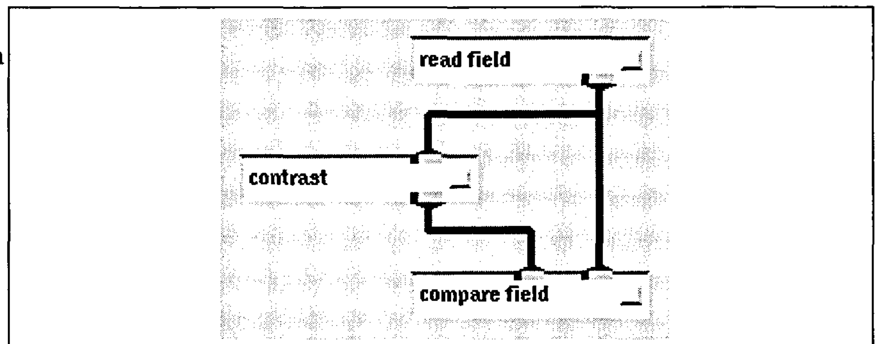
Output File

An ASCII typein for specifying the output file name. By default, **compare field** writes to /tmp/cfield_*nnn* file (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. Because the Output Browser is limited in size, this output file can be useful to examine directly, using a conventional text editor.

Example

The network in Figure 17 reads an image into a 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.

Figure 17
compare field module in an example network



Related modules

print field

Limitations

compare field writes to the /tmp directory by default. This can cause problems if:

- There is no /tmp directory on your system.
- 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

Blend two images using alpha transparency

Summary

Name	composite
Type	filter
Inputs	field 2D uniform 4-vector byte field 2D uniform 4-vector byte
Outputs	field 2D uniform 4-vector byte
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 equations for this blending are:

$$red = (Foreground(red) \bullet ALPHA) + (Background(red) \bullet (1.0 - ALPHA))$$

$$green = (Foreground(green) \bullet ALPHA) + (Background(green) \bullet (1.0 - ALPHA))$$

$$blue = (Foreground(blue) \bullet ALPHA) + (Background(blue) \bullet (1.0 - ALPHA))$$

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

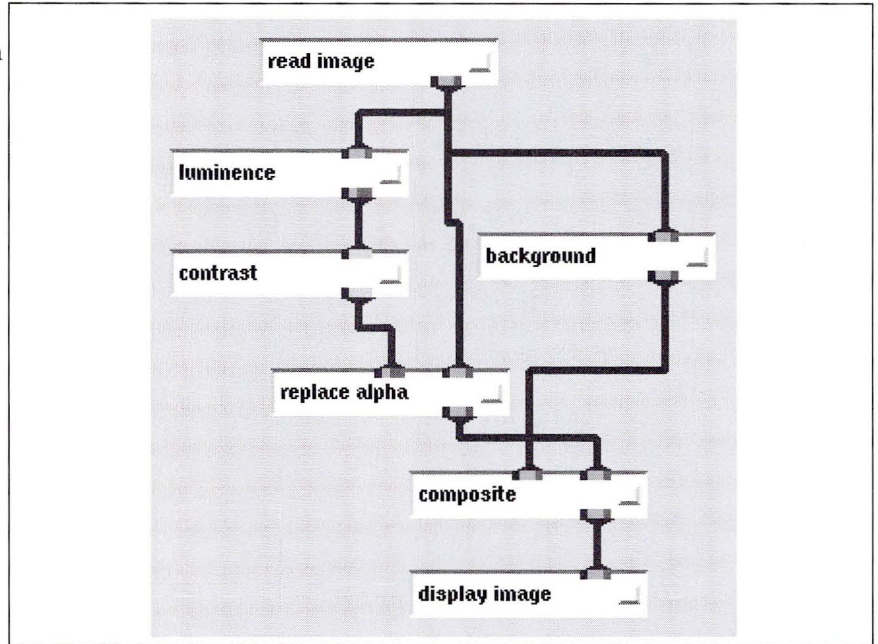
Image (field 2D uniform 4-vector byte)

The blended image of the two input images.

Example

The network in Figure 18 reads an image, computes its luminance (gray scale intensities), uses that to create an alpha mask, generates a shaded background, and blends the rendered image with the shaded background image.

Figure 18
composite module in an example network



Related modules

Modules that could provide the foreground **Image** input are read image and replace alpha.

Module that could provide the background **Image** input is background.

Modules that can process **composite** output are image viewer and display image.

See also

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

compute gradient

Compute gradient vectors for 2D or 3D data set

Summary

Name	compute gradient
Type	filter
Inputs	field 2D/3D scalar byte *any-coordinates*
Outputs	field *same-dimension* 3-vector real *same-coordinates*

Parameters

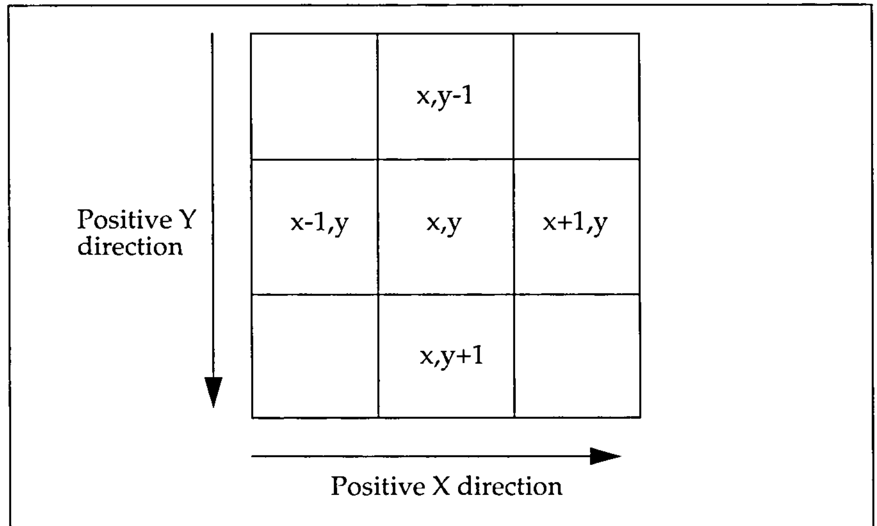
Name	*Type*	*Default*	*Min*	*Max*
2D Height	float	0.5	0.0	1.0

Description

The **compute gradient** module computes the gradient vector at each point in a 2D or 3D field of data. The gradient can be used (for example, 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 represented as shown in Figure 19.

Figure 19 Computing the gradient



$$\Delta x(x, y, z) = data(x+1, y, z) - data(x-1, y, z)$$

$$\Delta y(x, y, z) = data(x, y+1, z) - data(x, y-1, z)$$

$$\Delta z(x, y) = 2Dheight \quad \text{(for 2D data)}$$

$$\Delta z(x, y, z) = data(x, y, z+1) - data(x, y, z-1) \quad \text{(for 3D data)}$$

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 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.

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.

Examples

1.

The network in Figure 20 shades a 2D image.

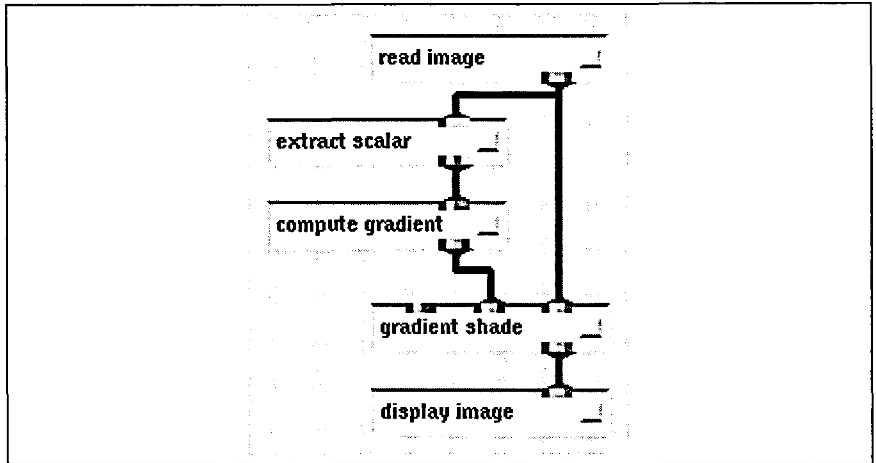


Figure 20
compute gradient module in an example network

2.

The network in Figure 21 shades a 3D image.

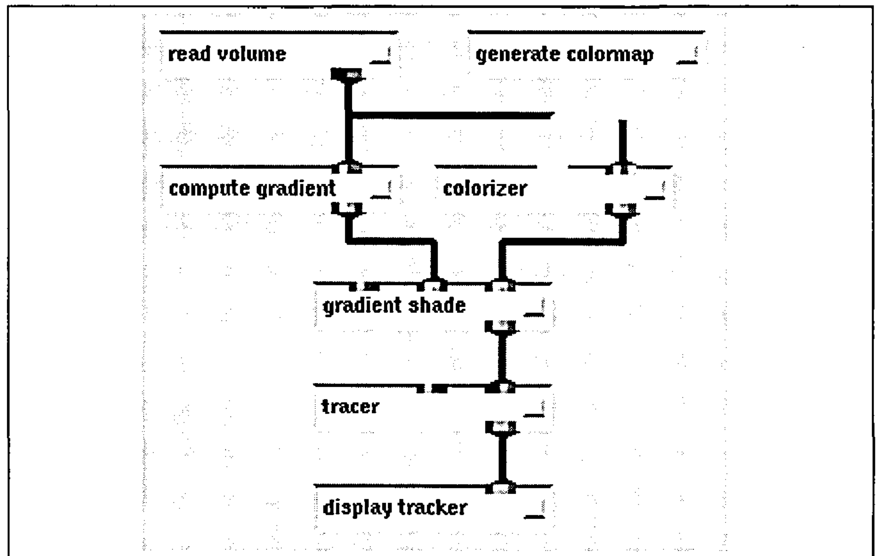


Figure 21
compute gradient module in an example network

Related modules

gradient shade, display image, and extract scalar

Limitations

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

See also

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

contour to geom

Create geometry of 2D or 3D scalar field contour slices

Summary

Name	contour to geom				
Type	mapper				
Inputs	field 2D/3D scalar *any-data any-coordinates*				
Outputs	geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	threshold	float dial	128.0	none	none

Description

The **contour to geom** module finds and creates contour lines of similar value in a scalar field, then outputs the result as a ConvexAVS geometry. The contour lines can be disjoint. The **threshold** parameter controls the contour level. **contour to geom** handles 2D and 3D data sets, 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 has no minimum or maximum.

Outputs

Geometry (geometry)

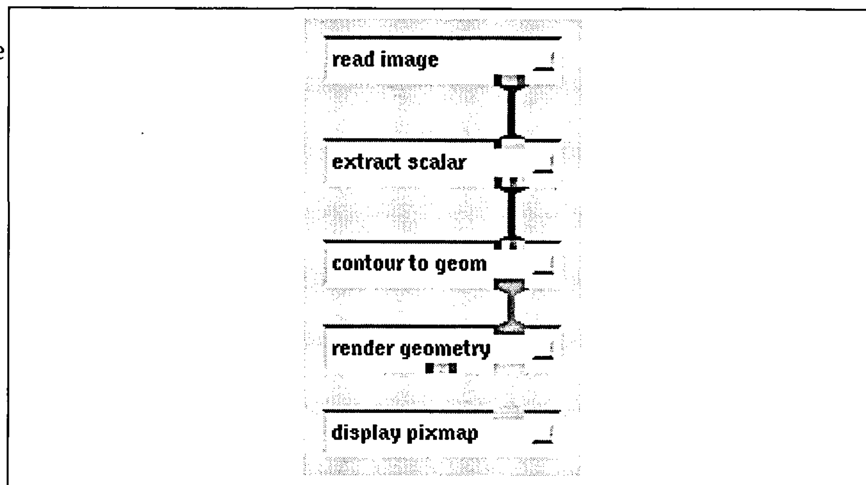
The contour lines are represented as a ConvexAVS geometry.

Examples

1.

The network in Figure 22 finds a contour on the red channel of the mandrill.x image.

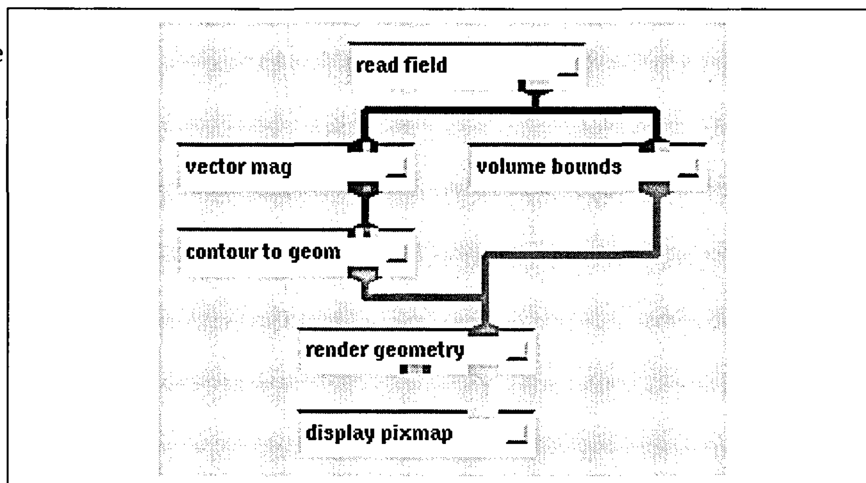
Figure 22
contour to geom module in an example network



2.

The network in Figure 23 finds the magnitude of a vector field and contours it.

Figure 23
contour to geom module in an example network



Related modules

Modules that can process **contour to geom** output are render geometry and render manager.

See also

Two CONTOUR GEOMETRY example scripts demonstrate the **contour to geom** module.

contrast

Perform linear transformation on range of field values

Summary

Name	contrast
Type	filter
Inputs	field *any-dimension n*-vector *any-data any-coordinates*
Outputs	field *same-dimension n*-vector *same-data same-coordinates*

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:

- 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** through **cont_out_max**:

$$new_value = \frac{(cont_out_max - cont_out_min) * (value - cont_in_min)}{(cont_in_max - cont_in_min)}$$

  More precisely, this is an *affine transformation*. 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 is used 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 a 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 less than or equal to **cont_in_min** will be transformed to this value.

cont_out_max

Specifies the top of the range of output values. All values greater than or equal to **cont_in_max** will be transformed to this value.

Outputs

Data Field (field *same-dimension n*-vector *same-data same-coordinates*)

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.

Examples

1.

The diagram in Figure 24 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.

2.

The network in Figure 25 reads in an image, extracts the red, green, and blue channels, contrast stretches only the red channel, then uses **combine scalars** to pack the separate channels back into an image.

Related modules

Module that could provide the **Data Field** input is read volume.

See also

The example script CONTRAST demonstrates the **contrast** module.

Figure 24
Transforming field values

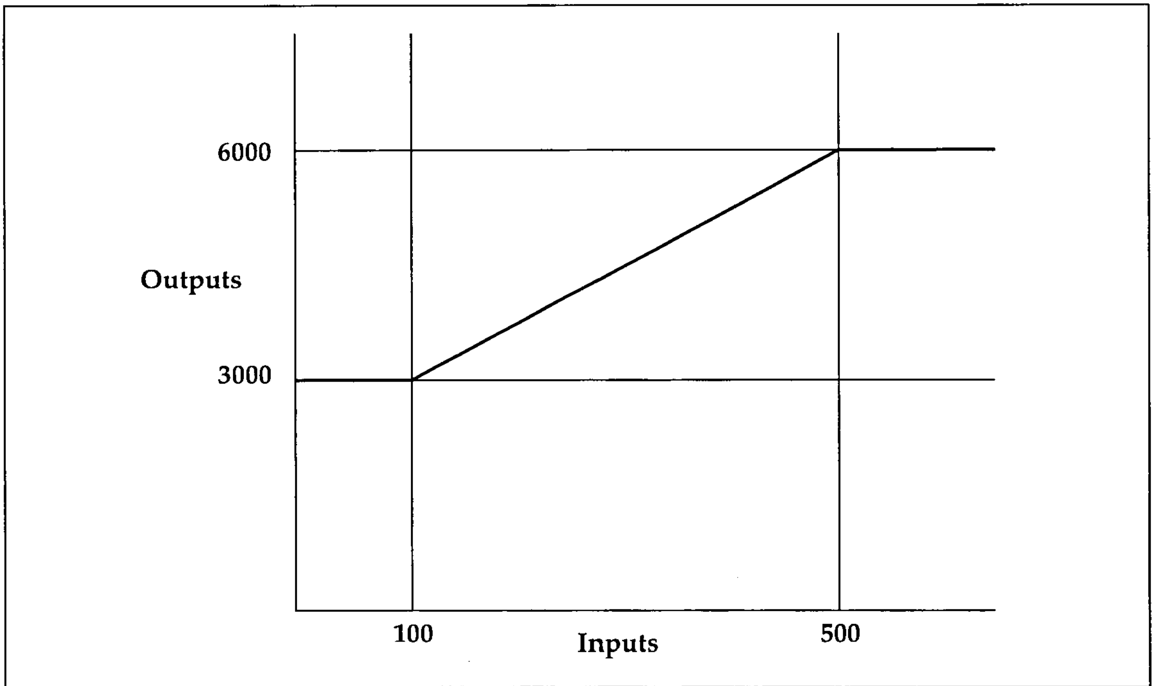
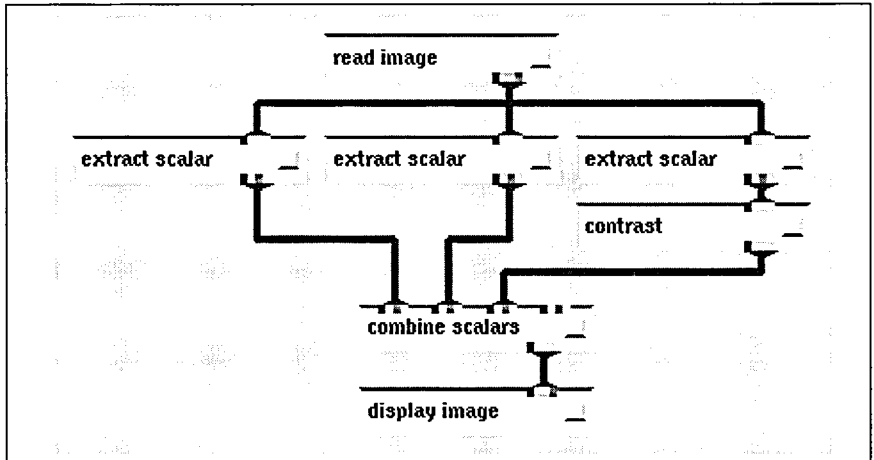


Figure 25
contrast module in an example network



convolve

Apply a signal processing filter to 2D field

Summary

Name	convolve		
Type	filter		
Inputs	field 2D *n*-vector *any-data any-coordinates* field 2D scalar float uniform		
Outputs	field *n*-vector *same-data same-coordinates*		
Parameters	*Name*	*Type*	*Default*
	Normalize Filter	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 images, but they might also be 2D slices of 3D fields. Filters can be produced by the module **generate filters** or by your 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 0–1 before applying them to an image.

Filters are applied as follows, taking a typical case in which a small, 10 by 10 filter is applied to a larger, 256 by 256 image. Imagine the filter sitting on top of the source image centered on one pixel in the image, for example, (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 by 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 (those above 12 by 12) 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.

Inputs

Data Field (required; field 2D *n*-vector *any-data any-coordinates*)

A 2D 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 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 Filter

If **Normalize Filter** is selected, filters are normalized to the range 0–1 before they are convolved with the input field.

Outputs

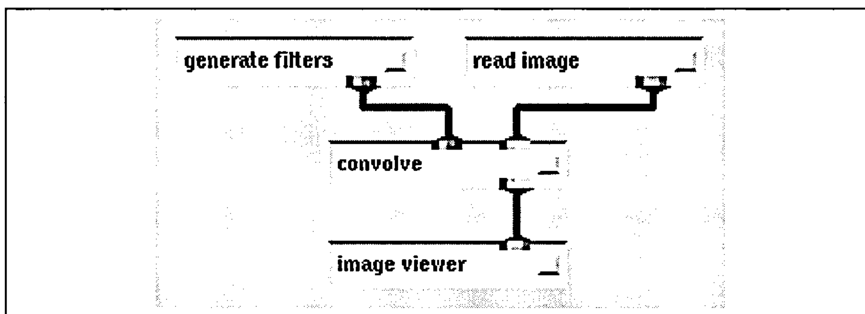
Data Field (field *n*-vector *same-data same-coordinates*)

The output field is the same type as the input data field.

Example

The network in Figure 26 reads in an image and a filter, convolves the two, and displays the resulting image.

Figure 26
convolve module in an example network



Related modules

Modules that could provide the **Data Field** input are read image, pixmap to image, and orthogonal slicer.

Module that could provide the **Filter Field** input is generate filters.

Modules that can process **convolve** output are display image and image viewer.

See also

The example scripts CONTRAST, GENERATE FILTERS, and SOBEL demonstrate the **convolve** module.

crop

Extract subset of elements from a field

Summary

Name crop

Type filter

Inputs field *any-dimension any-data any-coordinates*

Outputs field *same-dimension same-data same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
min x	int	1st index	1st index	last index
max x	int	last index	1st index	last index
min y	int	1st index	1st index	last index
max y	int	last index	1st index	last index
min z	int	1st index	1st index	last index
max z	int	last index	1st index	last index

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 (for example, by interpolation). It preserves the resolution of the data but may change its aspect ratio. One use is to eliminate uninteresting portions of the data and another is 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. Then, the dials cannot be turned lower than the min index or higher than the max index. If you use the Dial Editor to change these values, they would return to their original values.

If the value of the min dial is set greater than the value of the max dial, the two dials will swap values. In addition, the min and max dial cannot be set to the same value.

Inputs

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

The input data may be a field with any dimension.

Parameters

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.

Outputs

Data Field (field *same-dimension same-data same-coordinates*)

The output field has the same dimensionality as the input field, but the number of elements in each dimension might be reduced.

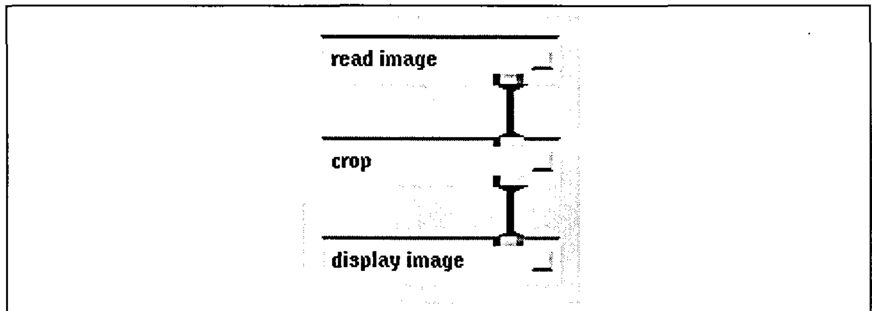
The **min_val** and **max_val** attributes of the output field are invalidated. Appropriate new **min_ext** and **max_ext** values are written to the output field.

Examples

1.

The network in Figure 27 reads a 2D field (image), crops it, and displays the result.

Figure 27
crop module in an example network



2.

Suppose you want to process the middle third of a field that contains a 500 by 300 pixel image (one-ninth of the original field). Set the X-axis and Y-axis parameters as follows:

min x = 167

max x = 333

min y = 100

max y = 200

Related modules

Module that could provide the **Data Field** input is read volume.

Modules that could be used in place of **crop** are downsize and interpolate.

Modules that can process **crop** output are arbitrary slicer, gradient shade, orthogonal slicer, and colorizer.

Limitations

crop works for 2D and 3D data sets only.

See also

The example script CROP demonstrates the **crop** module.

density PLOT3D

Strip out the PLOT3D density

Summary

Name	density PLOT3D
Type	filter
Inputs	field 3D 5-vector irregular 3-space float
Outputs	field 3D scalar irregular 3-space float
Parameters	none

Description

This module allows you to visualize the density field in the PLOT3D file. It converts the 5-vector into the density scalar field.

Inputs

Data Field (required; field 3D 5-vector irregular 3-space float)

This input data is the 5-vector field output by **read PLOT3D**. It is the most important field because it contains the mesh and solution data.

Outputs

Scalar Field (field 3D scalar irregular 3-space float)

A scalar field representing the density field from the PLOT3D file read by **read PLOT3D**.

Example

The network in Figure 28 reads in a PLOT3D data set and does an isosurface on the density field. Notice how only the PLOT3D field itself is used. We do not use the blanking records because we are extracting a part of the raw PLOT3D field.

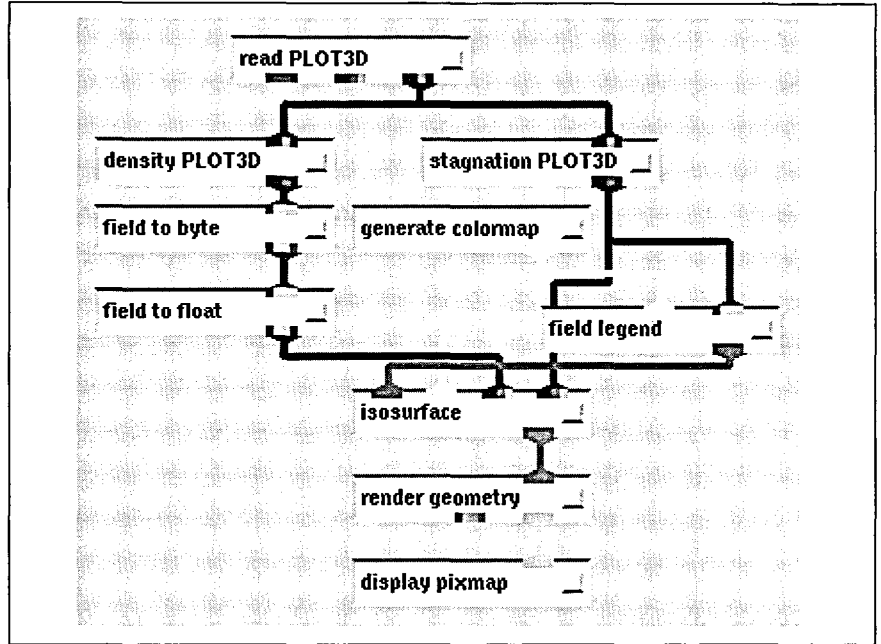


Figure 28
density PLOT3D module in an example network

Related modules

momentum PLOT3D, stagnation PLOT3D, read PLOT3D, scalar PLOT3D, and vector PLOT3D

Related programs

export_PLOT3D and import_PLOT3D

display image

Show image in a display window

Summary

Name	display image
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	integer	256	50	1024
Max Image Dimension	integer	1280	100	4096
approx technique	choice	dither		

Description

The **display image** modules takes an input image and displays it in a display window. This window has a pull-down 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 pull-down menu.

When running ConvexAVS as a remote client on a pseudocolor X terminal, **display image** has an additional choice parameter for selecting the dithering method.

The **display image** window can be reparented to page and stack widgets using the Layout Editor.

Inputs

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

The input field must be in the ConvexAVS image format.

Parameters

Magnification

A choice to specify a power of two 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. It has been kept solely for the purpose of backward compatibility.

approx technique

A choice of four 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.

random Uses a 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 grey scale equivalent for each pixel. Selecting monochrome converts the color image into a monochrome image, resembling a black and white photograph.

floyd steinberg Uses a slower dithering technique that produces better results for computer-generated imagery.

none Has each color in the input image approximated by the closest color in the spectrum of colors actually available on a pseudocolor terminal.

This parameter only appears when running ConvexAVS on pseudocolor X terminals.

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 pull-down menu.

Because **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 your 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 **Automag_Size** option is enabled.

The **Automag_Size** parameter reenables the automatic magnification of the image to at least fill the window when the window is resized. By default, this is disabled because the combination of autofit and automagnification can produce unexpected window behavior.

The pull-down 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 by 1024) and magnifies the image to fit. If the window is embedded in a page or stack, it becomes a top-level window that can be freely resized and moved using the X window manager.
- 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).
- 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.

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.

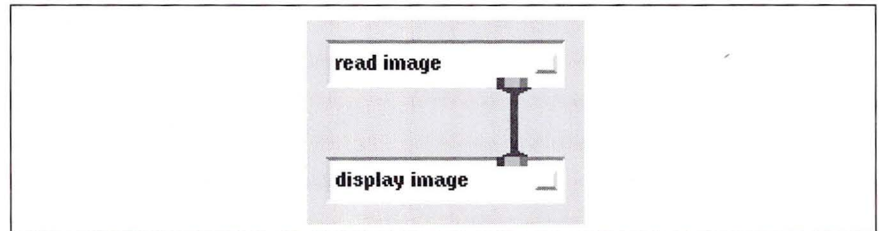
The Scrollbars - Turn On/Off selection on the pull-down menu allows you to disable or reenable the appearance of scrollbars along the right and bottom edges of the display window. (The drag-the-image method is always enabled.) You may want to suppress the scrollbars to reduce distraction or to provide additional viewing space.

The ImageScrollbars keyword in the start-up file determines whether image windows get scrollbars by default when they contain oversize images. If you do not use this start-up parameter, scrollbars are initially enabled.

Example

The network in Figure 29 shows **display image**.

Figure 29
display image module in an example network



Related modules

display pixmap and image viewer

See also

The example scripts ANIMATED INTEGER, FIELD MATH, and GENERATE FILTERS demonstrate the **display image** module.

display pixmap

Show pixmap in a display window

Summary

Name	display pixmap
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 slider		
Max Frames	integer typein		
Replay Speed	integer slider		
Save Image	string typein		

Description

The **display pixmap** module displays its input pixmap in a display window. It automatically sizes the pixmap to fit the window. **display pixmap** is frequently used in conjunction with the **render geometry** module to display geometry output. You can also:

- Save the pixmap as an image in a file.
- Create and play back a flip book of consecutive images.

These capabilities are invoked using the module's input parameters.

The **display pixmap** window can be reparented to page and stack widgets using the Layout Editor.

Inputs

Pixmap (required; pixmap)

The input data must be a pixmap, typically created by the **render geometry** module.

Parameters

The following parameters control a pixmap-animation capability. This is independent of the animation facility in the Geometry Viewer (**render geometry** module) and works somewhat differently.

Store Frames

Controls whether all new frames are automatically added to the animation sequence.

Append Frame

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

Controls how the animation sequence is to be played back. The choices are **Continuous**, **Bounce**, and **Off**.

Current Frame

Indicates the number of the current frame in the animation sequence (first frame = 0).

Max Frames

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

Saves the current pixmap to a file.

Resizing

display pixmap's pull-down menu, which is accessed by clicking on the dimple in the upper left 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 by 1024) and magnifies the image to fit. If the window is embedded in a page or stack, it becomes a top-level window that can be freely resized and moved using your 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 file name 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, you can have the **display pixmap** window show a sequence of images. You can create a flip book by designating certain images to be frames. Then, you can play back the images and adjust the speed with a control widget.

Because each of the images in a flip book takes up a significant amount of X server memory, there is a **Max Frames** parameter. Be sure that its value is low enough so that your X server can comfortably keep all of the images in memory at the same time. ConvexAVS requires three bytes of memory per pixel for 24-bit true color displays and 1 byte of memory per pixel for 8-bit pseudocolor displays. 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), 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 oneshot **Append Frame**. This appends the image to the end of the existing flip book. This method allows you to carefully construct a flip-book 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 playback facility.

The **Replay Speed** parameter controls the rate at which flip-book frames are displayed.

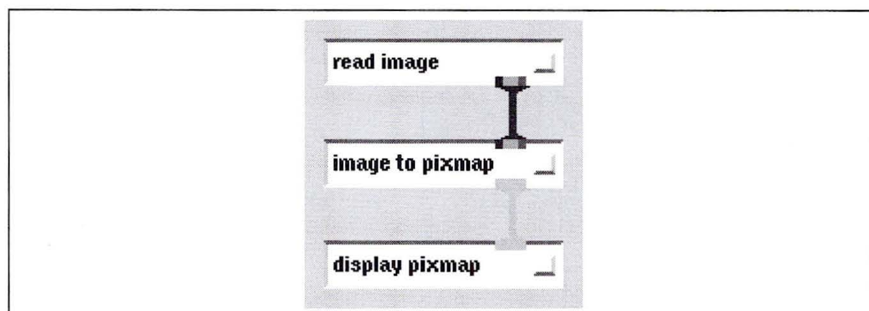
The **Current Frame** parameter allows you to select a particular frame manually. It is normally updated to display the current frame, but for cases in which such updating would impact animation performance, it is not updated. Because only the last **Max Frames** frames are stored, the animation can begin at a frame other than 0. After you select a particular frame, you can delete it with the oneshot **Delete Frame**.

Examples

1.

The network Figure 30 in reads in an image, converts it to a pixmap, then displays the image using **display pixmap**.

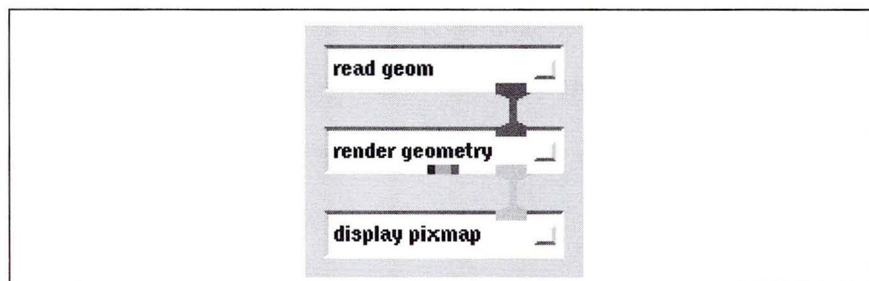
Figure 30 display pixmap module in an example network



2.

The network in Figure 31 reads in a geometry object, renders it, then displays the rendered object using **display pixmap**.

Figure 31 display pixmap module in an example network



Related modules

render geometry

Limitations

There is no way to store the "first **Max Frames"** frames of an animation loop.

See also

The example scripts FIELD LEGEND and PROBE demonstrate the **display pixmap** module.

display tracker

Display and directly manipulate the tracer module's output

Summary

Name	display tracker
Type	data output
Inputs	field 2D 4-vector byte uniform
Outputs	field 2D scalar float
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 the SHIFT key in combination with the middle mouse button.

display tracker takes 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)

A ConvexAVS image, typically output by the module **tracer.**

Parameters

Scale

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

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 (field 2D scalar float)

The output port on the module **display tracker**, which is usually invisible, sends out a 4 by 4 transformation matrix describing rotations, scalings, 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 input port. This allows you to directly rotate and translate an image by moving the mouse in **display tracker**'s window.

Example

The network in Figure 32 shows how **display tracker** displays the output of **tracer**. **display tracker** also sends data upstream to **tracer**.

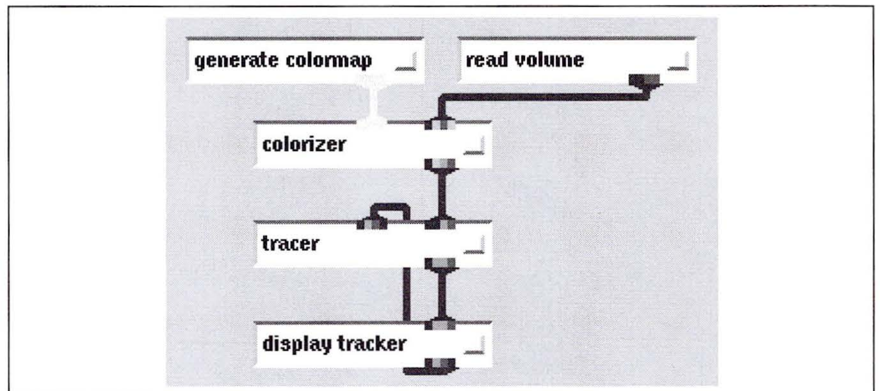


Figure 32
display tracker module in an example network

Related modules

Module that could provide the **Data Field** input is tracer.

Modules that can receive **display tracker**'s upstream transform are tracer and gradient shade.

See also

The example script ANIMATED FLOAT demonstrates the **display tracker** module.

dot surface

Generate points that define an isosurface

Summary

Name	dot surface
Type	filter
Inputs	field 3D scalar 3-space
Outputs	field 1D 0-vector real irregular 3-space

Parameters

Name	*Type*	*Default*	*Min*	*Max*
stepsize	real	0.5	1.0E-5	2.0
thresh	real	0.02	none	none

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 less than or equal to 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,+0.2]. 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 3-space)

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 the value, the smoother the surface.

thresh

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 **thresh** value.

Outputs

Point List (field 1D 0-vector real irregular 3-space)

The scalar data value for each output field element is unused. The only useful information is the 3D coordinate data.

Example

The network in Figure 33 shows **dot surface**.

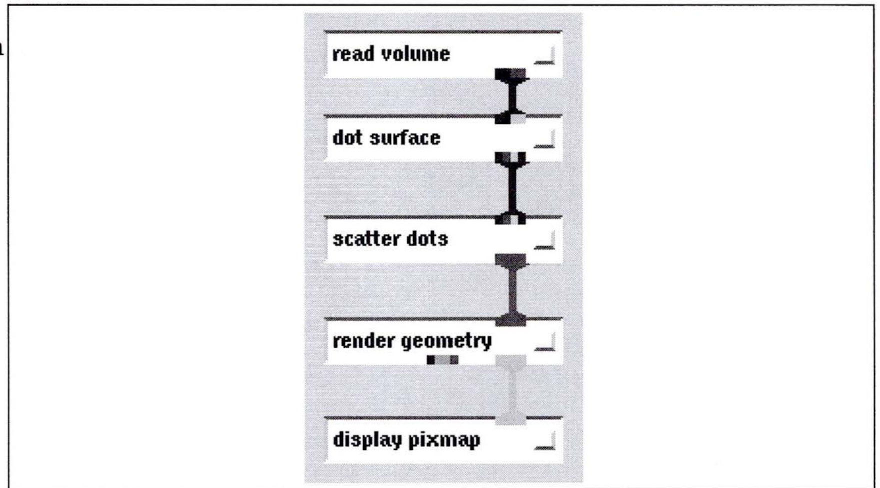


Figure 33
dot surface module in an example network

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 1,000,000 points will be generated. Once the module calculates this number of points, it returns, leaving all other cells unprocessed. Use the **downsize** module to avoid this if possible.

Related modules

Modules that could provide the **Data Field** input are read volume and combine scalars.

Modules that could be used in place of **dot surface** are isosurface and tracer.

Module that can process **dot surface** output is scatter dots.

See also

The example script DOT SURFACE demonstrates the **dot surface** module.

downsize

Reduce size of data set by sampling

Summary

Name	downsize
Type	filter
Inputs	field *any-dimension any-data any-coordinates*
Outputs	field *same-dimension same-data same-coordinates*

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** 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** = 1 for highest-resolution data) and execution speed (**downsize** > 1 for lower-resolution data).

Inputs

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

The input data may be any ConvexAVS 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 (field *same-dimension same-data same-coordinates*)

The output field has the same dimensionality as the input field, but the number of elements in each dimension might be reduced by the **downsize** parameter.

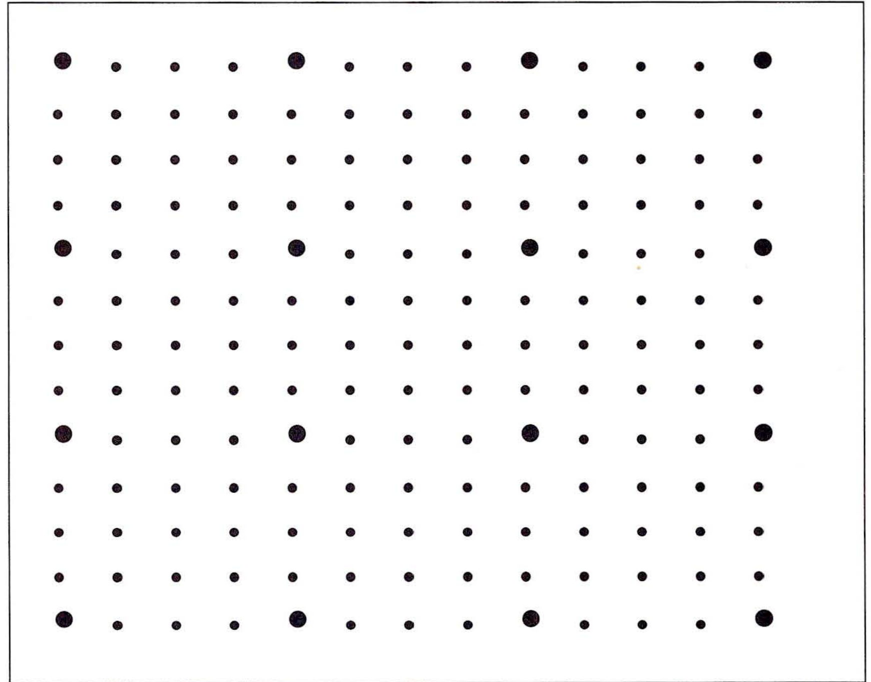
The **min_val** and **max_val** attributes of the output field are invalidated. Appropriate new **min_ext** and **max_ext** values are written to the output field.

Examples

1.

The diagram in Figure 34 shows how a **downsize** 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.

Figure 34
Downsize factor of 4 on a 2D field



2.

The network in Figure 35 shows an example of **downsize**.

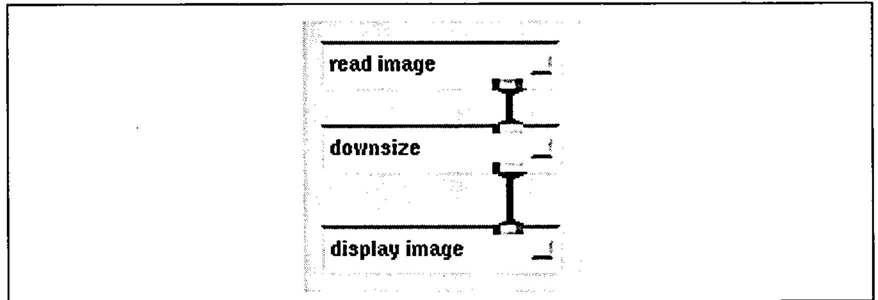


Figure 35
downsize module in an example network

Limitations

downsize works for 2D and 3D data sets only.

Related modules

Modules that could provide the **Data Field** input are read volume and any filter module.

Modules that could be used in place of **downsize** are interpolate and crop.

Modules that can process **downsize** output are colorizer, gradient shade, arbitrary slicer, and orthogonal slicer.

See also

The example scripts FIELD MATH and GRAPH VIEWER demonstrate the **downsize** module.

euler transformation

Send object transformation matrix to other modules

Summary

Name euler transformation

Type data input

Inputs none

Outputs field 2D uniform scalar float

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 4 by 4 transformation matrix specifying scaling and rotations in x, y, and z.

Use **euler transformation** 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 transformation** does not supply the full upstream transform accepted by such modules as **thresholded slicer**. Currently, **euler transformation** only works with the **gradient shade** and **tracer** modules.

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. That is, transformations are not remembered and accumulated.

Parameters

theta

A floating-point dial widget that 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 that controls rotation of the object's y-axis. The y-axis initially runs vertically from negative at the bottom to positive at the top.

rho

A floating-point dial widget that 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 that controls the scaling coefficient of the transformation matrix. This makes the data object look larger or smaller.

Outputs

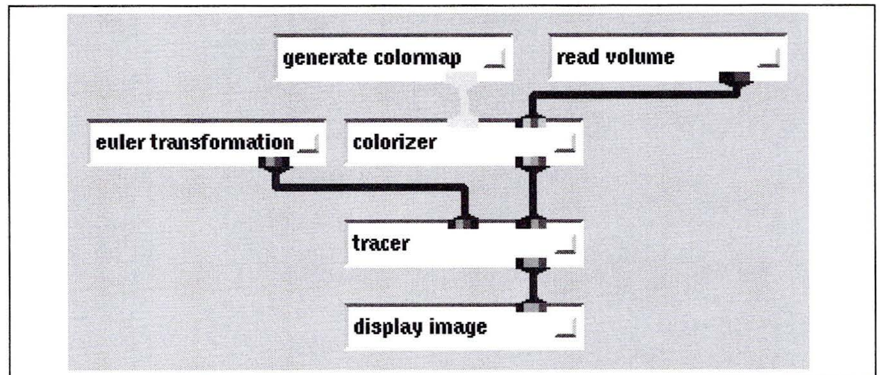
Transformation Matrix (field 2D uniform scalar float)

The output is a 4 by 4 array of floating-point values that specifies rotations and scaling operations that can be applied to transform an object around the origin of its own coordinate system.

Example

The network in Figure 36 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.

Figure 36
euler transformation module in an example network



Related modules

Modules that accept **euler transformation**'s output are tracer and gradient shade.

See also

The example script EULER TRANSFORMATION demonstrates the **euler transformation** module.

extract scalar

Extract a scalar field from a vector field

Summary

Name	extract scalar
Type	filter
Inputs	field *any-dimension n*-vector *any-data any-coordinates*
Outputs	field *same-dimension* scalar *same-data same-coordinates*
Parameters	*Name* *Type* Channel radio buttons

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 because their data values are considered to be 1D vectors.

Parameters

Channel

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

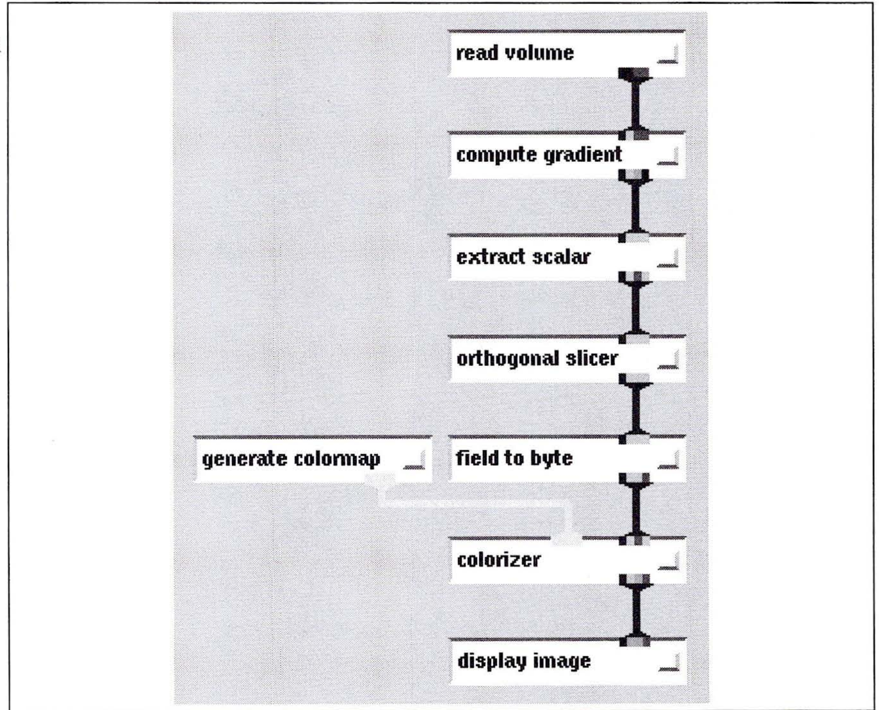
Data Field (*same-dimension* scalar *same-data same-coordinates*)

The output field has the same dimensionality as the input field. The data for each element is reduced from a vector to a scalar.

Example

Figure 37 displays a slice of the Y-component of the gradient field of a volume.

Figure 37
extract scalar module in an example network



Related modules

extract vector and combine scalars

See also

The example scripts CONTOUR GEOMETRY and CONTRAST demonstrate the **extract scalar** module.

extract vector

Subset of field vector elements as new field

Summary

Name	extract vector
Type	filter
Inputs	field *any-dimension n*-vector *any-data any-coordinates*
Outputs	field *same-dimension n*-vector *same-data same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Channel 0	boolean	off		
Channel 1	boolean	off		
Channel 2	boolean	off		
.	.	.		
.	.	.		
.	.	.		
Channel 24	boolean	off		
Vector Length	integer dial	3	1	25

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 than (and essentially equivalent to) using **extract scalar** modules to extract individual vector elements and then pasting them together with **combine scalar**.

extract vector can handle up to 25 vector elements. You can extract any subset of those elements.

Inputs

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

A 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

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 labeled, 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**).

Vector Length

An integer dial that specifies the vector length of the output field.

Outputs

Data Field (field *same-dimension n*-vector *same-data same-coordinates*)

The output field has the same form as the input field, except that its vectors are shorter.

Example

The network in Figure 38 extracts the X-, Y-, and Z-momentum vector elements from a field data set, then plots their sum vector using **hedgehog**. The data set operated on is bluntfin.fld, which contains PLOT3D data in field format.

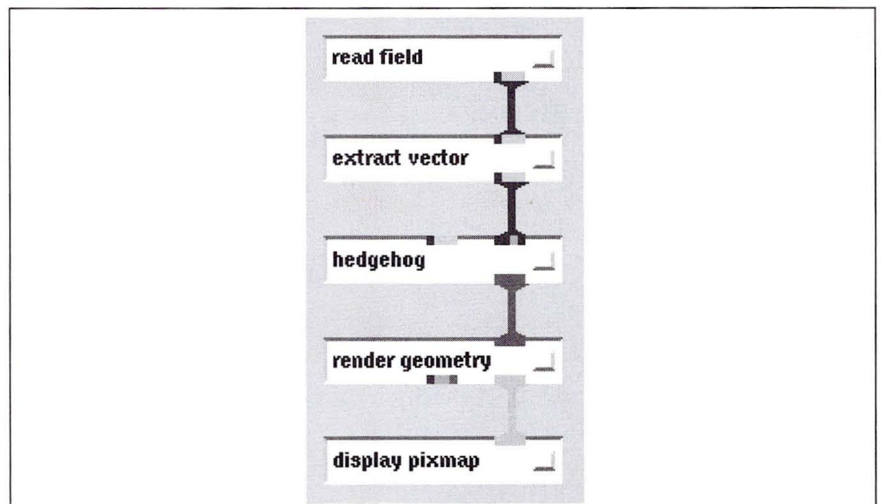


Figure 38
extract vector module in an example network

Related modules

Modules that could provide the **Data Field** input are those that produce a vector field output.

Modules that could be used in place of **extract vector** are extract scalar and combine scalars.

Modules that can process **extract vector** output are those that can process vector fields.

See also

The example script STREAMLINES demonstrates the **extract vector** module.

field legend

Select value from scalar field using color legend

Summary

Name	field legend
Type	mapper
Inputs	field *n-dimensions n*-vector *any-data any-coordinates* colormap
Outputs	real

Parameters

Name	*Type*	*Default*
node data	radio buttons	<data 1>
value	dial	0

Description

field legend takes an *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 (that is, momentum, m/sec) **field legend** will print these as the title of the color legend.

By moving a 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.

field legend

The module **field legend** accepts *n*-dimensional *n*-vector fields. Use the **node data** choices 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.

Inputs

Data Field (required; field *n-dimensions n*-vector *any-data any-coordinates*)

A field that supplies the range of values displayed by **field legend**.

Colormap (required; colormap)

A colormap that is used as the legend for selecting values from the data field.

Parameters

node data

Selects the vector 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

Selects the color legend data value.

Outputs

Value (real)

A single floating-point value selected from the range of values in the field.

Example

The network in Figure 39 displays isosurfaces of a 3D scalar field. **field legend** allows you to select what level of values should be displayed as a surface. **field legend** performs the equivalent of **extract scalar** and **color range**, but these two modules still need to filter the field that **isosurface** receives. Also, **generate colormap** sends the same colormap to both **field legend** and **color range.**

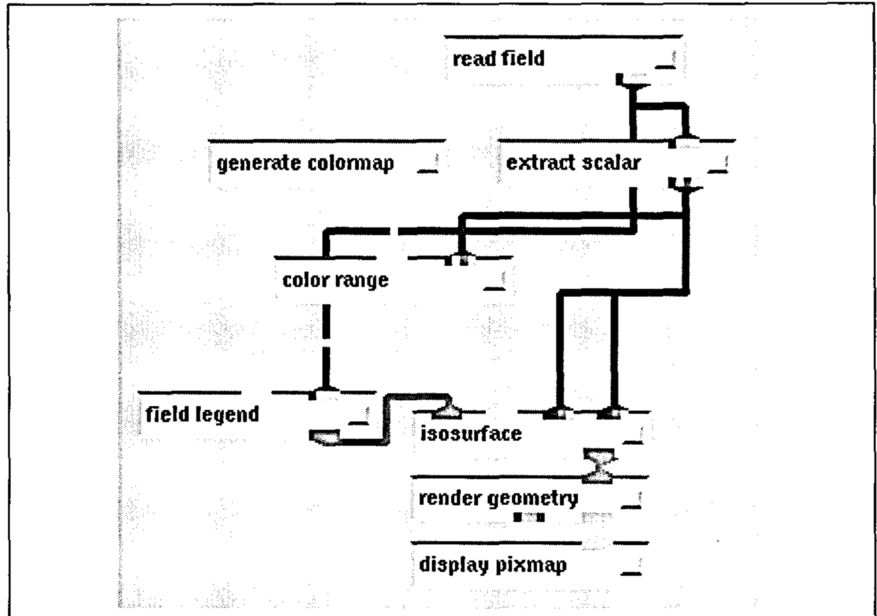


Figure 39
field legend module in an example network

Related modules

Module that could provide the **Data Field** input is read field.

Modules that could provide the **Colormap** input are generate colormap and color range.

Modules that can process **field legend**'s output are isosurface, thresholded slicer, and contour to geom.

See also

The example script FIELD LEGEND demonstrates the **field legend** module.

field math

Perform math operations between fields

Summary

Name	field math
Type	filter
Inputs	field *any-dimension n*-vector *any-data any-coordinates* field *same-dimension same*-vector *any-data same-coordinates*
Outputs	field *same-dimension same*-vector *any-data same-coordinates*

Parameters

Name	*Type*	*Default*
choice	choice	+
Normalize	boolean	off
Constant	float typein	0.0

Description

The **field math** module performs unary and binary operations upon fields.

The unary operations are addition, subtraction, multiplication, division, logical bitwise And, Or, Xor, Not, and Square, Sqrt, and Root Mean Square (RMS). Unary operations are performed against the right-most port field when there is only one field attached. They use the value supplied by the **Constant** input parameter. Even when a second field is supplied, the Not, Square, and Sqrt operations are performed only on the first field.

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 have the same dimension, size, vector length, and coordinate type (that is, uniform, rectilinear, or irregular). When the fields contain different data types, they are cast to the higher order. In other words, if one field is a byte field and the other is a float, the math is done as floating point and the output field is floating point.

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 and as the first operand in binary operations.

Data Field (optional; field *same-dimension same*-vector *any-data same-coordinates*)

The leftmost field is the second operand in binary operations. It must have the same dimension, size, vector length, and coordinate type as the first input field.

Parameters

choice

A choice of operations.

With only one field, the unary operations performed are:

+	field_value + Constant
-	field_value - Constant
*	field_value * Constant
/	field_value / Constant
And	field_value AND Constant
Or	field_value OR Constant
Xor	field_value XOR Constant
Not	field_value
Square	field_value * field_value
Sqrt	sqrt (field_value)
RMS	sqrt (field**2 + constant**2)

With two fields, the binary operations performed are:

+	field1 + field2
-	field1 - field2
*	field1 * field2
/	field1 / field2
And	field1 AND field2
Or	field1 OR field2
Xor	field1 XOR field2
Not	NOT field1
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 reals, 0 and 255 for bytes, and 0 and 65535 for integers. **Normalize** is off by default.

Constant

A floating-point typein to specify the constant value to use in unary 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*)

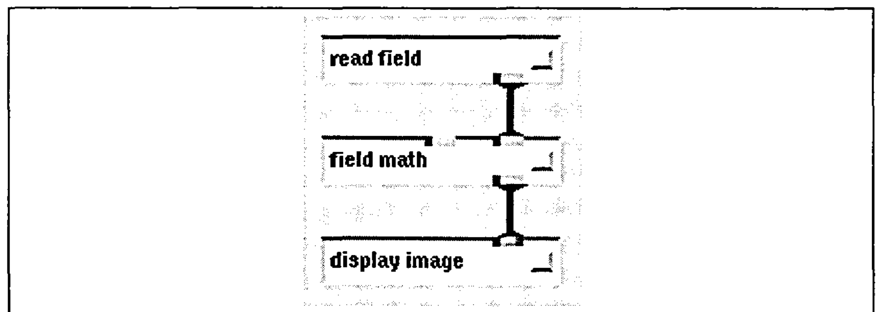
The output field has the same form as the input field. If both input fields are used but differ in their data type, the output field will have the more inclusive data type.

Examples

1.

The network in Figure 40 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.

Figure 40
field math module in an example network



2.

The network in Figure 41 does a logical AND on a volume against 128 (0 by 80), which produces a volume with only 0s and 255s based on whether the source voxel was greater or less than 128.

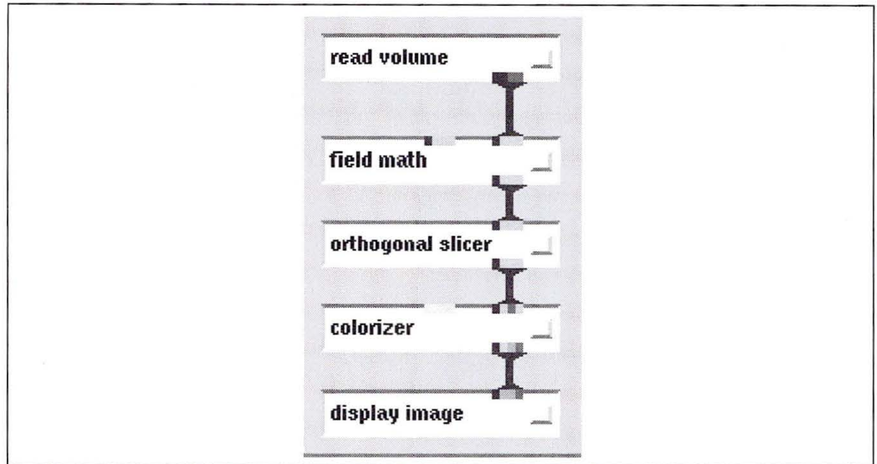


Figure 41
field math module in an example network

Related modules

Modules that could provide the **Data Field** inputs are those that output a field.

Modules that can process **field math** output are those that input a field.

See also

Two FIELD MATH example scripts demonstrate the **field math** module.

field to byte

Transform any field to a byte-valued field

Summary

Name	field to byte			
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 a byte field. It can be used in conjunction with volume visualization modules that have a bias towards byte fields (for example, **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.

Inputs

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

The input data may be any field.

Parameters

byte_normalize

This is a toggle parameter:

- If on, the data is transformed linearly into the range 0-255:

$$newvalue = \frac{(value - min) \cdot 255}{max - min}$$

- If off, the data is clamped so that no value falls outside the range 0-255:

If $value < 0$	$newvalue = 0$
If $0 <= value <= 255$	$newvalue = value$
If $value > 255$	$newvalue = 255$

Outputs

Data Field (field *same-dimension same*-vector byte *same-coordinates*)

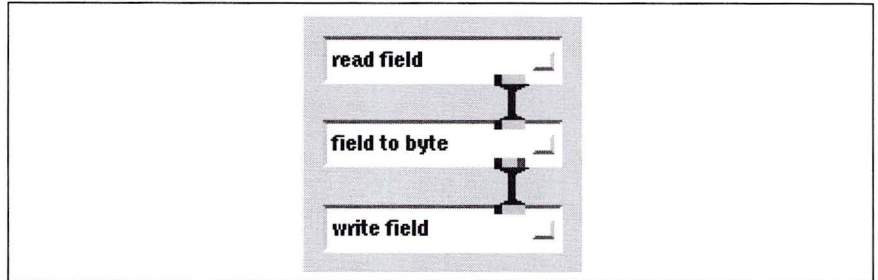
The output field has the same dimensionality as the input field, but each scalar value is forced to be a byte.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

Example

The network in Figure 42 shows **field to byte**.

Figure 42
field to byte module in an example network



Related modules

Module that could provide the **Data Field** input is read volume.

Modules that could be used in place of **field to byte** are field to int, field to float, and field to double.

Module that can process **field to byte** output is read volume.

See also

The example scripts FIELD TO BYTE and FIELD TO INTEGER demonstrate the **field to byte** module.

field to double

Transform any field to a field of double-precision floating-point values

Summary

Name	field to double
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 a double field. This may be useful for computing fields at greater data resolutions.

By default, the input data is simply cast to double-precision floating point. If the toggle parameter **double normalize** is turned on, the data is also normalized to the range 0-1.

Inputs

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

The input data may be any field.

Parameters

double normalize

This is a toggle parameter:

- If on, the data is transformed linearly into the range 0-1:

$$newvalue = \frac{value - min}{max - min}$$

- If off, the data is converted to double-precision floating point format.

Outputs

Data Field (field *same-dimension same*-vector double *same-coordinates*)

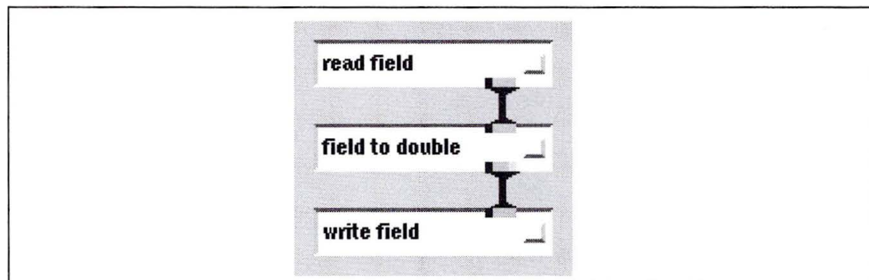
The output field has the same dimensionality as the input field, but each scalar value is forced to be a double-precision number.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

Example

The network in Figure 43 shows **field to double**.

Figure 43
field to double module in an example network



Related modules

read volume, field to byte, field to int, and field to float

See also

The example script FIELD TO INTEGER demonstrates the **field to double** module.

field to float

Transform any field to a field of single-precision floating-point values

Summary

Name	field to float
Type	filter
Inputs	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, real, double, or integer) and converts it to a 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 to single-precision floating point. If the toggle parameter **float normalize** is turned on, the data is also normalized to the range 0-1.

Inputs

Data Field (required; *any-dimension n*-vector *any-data any-coordinates*)

The input data may be any field.

Parameters

float normalize

This is a toggle parameter:

- If on, the data is transformed linearly into the range 0-1:

$$newvalue = \frac{value - min}{max - min}$$

- If off, the data is converted to single-precision floating point format (IEEE 754).

Outputs

Data Field (field *same-dimension same-vector* float *same-coordinates*)

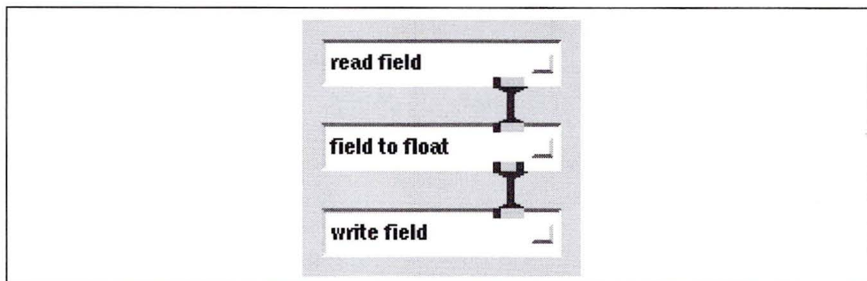
The output field has the same dimensionality as the input field, but each scalar value is forced to be a single-precision number.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

Example

The network in Figure 44 shows **field to float**.

Figure 44
field to float module in an example network



Related modules

read volume, particle advector, samplers, field to byte, field to int, and field to double

See also

The example script FIELD TO INTEGER demonstrates the **field to float** module.

field to int

Transform any field to an integer-valued field

Summary

Name	field to int
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, real, double, or int) and converts it to an integer field. This may be useful for performing integer math with greater precision (-2,147,483,648 through 2,147,483,647) than that offered by byte fields (0 through 255).

By default, the input data is converted directly into 32-bit, two's-complement integer format. If the toggle parameter **int normalize** is turned on, the data is normalized to that range instead.

Inputs

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

The input data may be any field.

Parameters

int_normalize

This is a toggle parameter:

- If on, the data is transformed linearly into the range 0-65535:

$$newvalue = \frac{(value - min) \cdot 65535}{max - min}$$

- If off, the data is simply converted into an integer that has 32-bit, two's-complement representation. No clamping occurs.

Outputs

Data Field (field *same-dimension same*-vector integer *same-coordinates*)

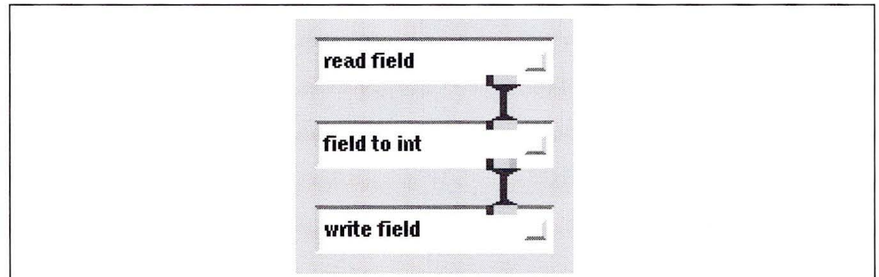
The output field has the same dimensionality as the input field, but each scalar value is forced to be an integer.

Appropriate new values of the **min_val** and **max_val** attributes are written to the output field.

Example

The network in Figure 45 shows **field to int**.

Figure 45
field to int module in an example network



Related modules

read volume, field to byte, field to float, and field to double

See also

The example script FIELD TO INTEGER demonstrates the **field to int** module.

field to mesh

Transform a 2D scalar field to a surface in 3D space

Summary

Name	field to mesh
Type	mapper
Inputs	field 2D scalar *any-data any-coordinates* colormap
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Z scale	float	1.0	-25.0	25.0

Description

The **field to mesh** module converts a 2D field into a surface in 3D space, represented as a geometry-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; colormap)

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

With uniform input fields, determines the height of the mesh. With rectilinear and irregular input fields, this parameter is unused.

Outputs

Geometry (geometry)

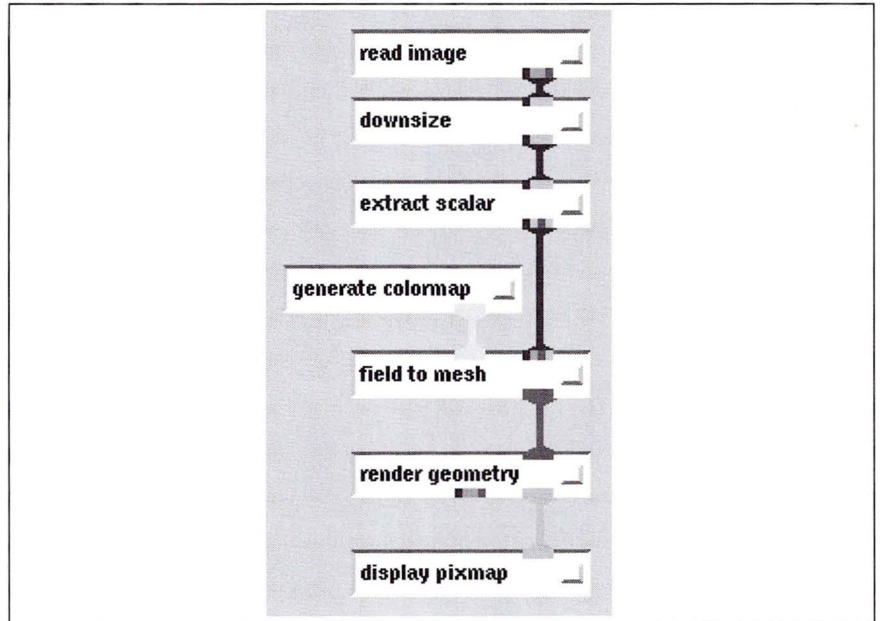
The output is a geometry.

Examples

1.

The network in Figure 46 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.

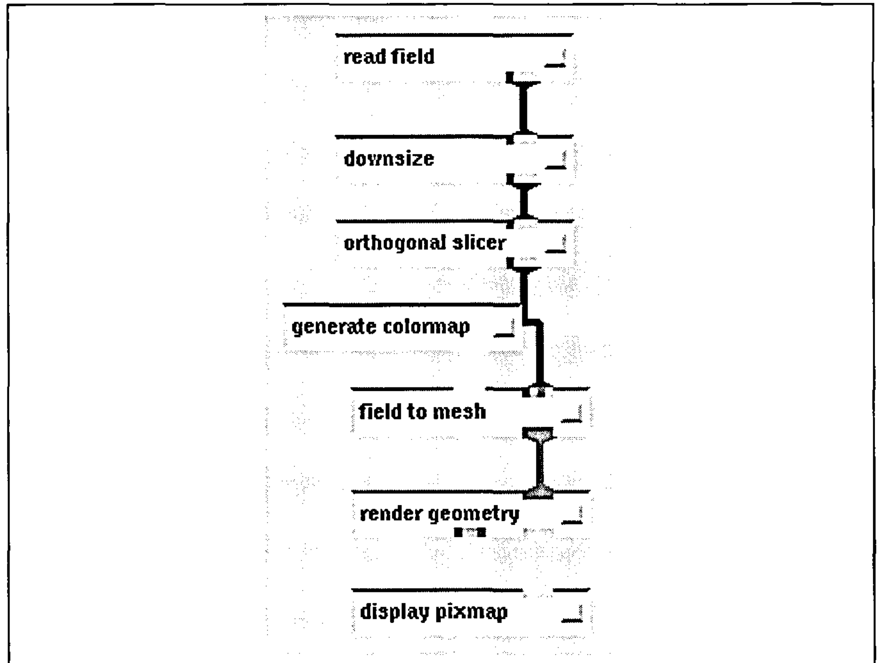
Figure 46
field to mesh module in an example network



2.

The example in Figure 47 shows how to take orthographic slices through a curvilinear data set, showing them as XYZ meshes.

Figure 47
field to mesh module in an example network



Limitations

This module can output meshes that are too big for the **render geometry** module to handle, causing ConvexAVS 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 to mesh** module.

field to ucd

Convert field to unstructured cell data format

Summary

Name	field to ucd		
Type	filter		
Inputs	field 3D *n*-vector *any-data any-coordinates*		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	Field Comp	choice	<data 1>

Description

field to ucd converts a 3D field into a UCD structure. To initiate the conversion, you must select the **Field Comp** parameter. This is the case even if the parameter is highlighted from a previous operation.

field to ucd converts a scalar value at each location in the input field into the value of a node in the UCD structure. **field to ucd** can receive an *n*-vector field, but its output UCD structure can only have a scalar value at each node. The cells of the output structure will be hexahedral.

A field is an array with a vector of values at each location. On the other hand, a UCD structure has a hierarchical structure consisting of structure data, cell data, and node data. Both structure and cell data are optional. Node data is required.

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 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 be either a vector or a scalar. Adjacent cells may share the same nodes. Conversely, nodes will tend to belong to several cells. For each node, there is a list of cells it belongs to. This is called the node connectivity list.

field to ucd computes the minimum and maximum extents of the structure.

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.

Parameters

Field Comp

Selects the vector component of the input field to extract and initiates the conversion of the field to a UCD structure.

Outputs

UCD Structure (ucd structure)

The output structure is in UCD format.

Example

The network in Figure 48 reads in a field, converts it into a UCD structure, then into a geometry and renders it.

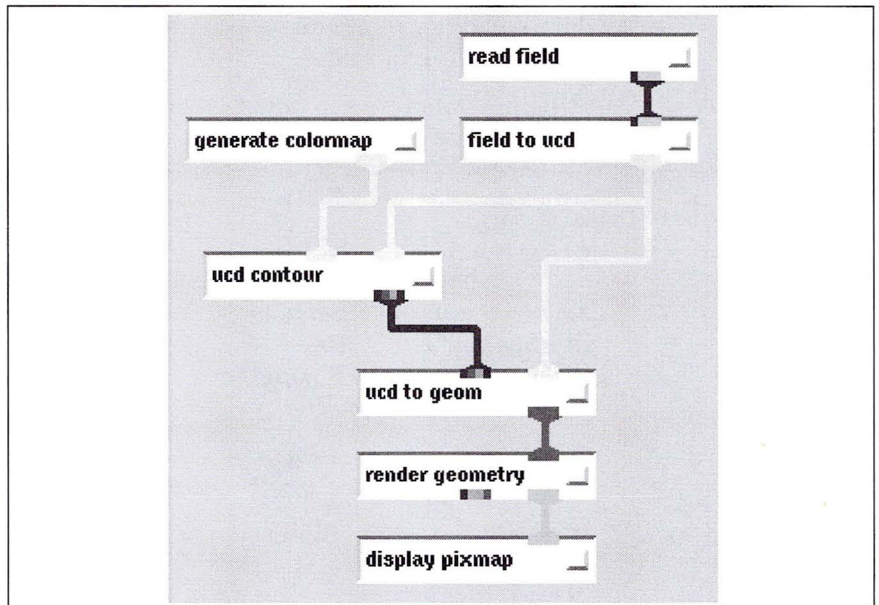


Figure 48
field to ucd module in an example network

Related modules

Module that could provide the **Data Field** input is read field.

Modules that can process **field to ucd**'s output are 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 slice 2d, ucd legend, ucd probe, ucd streamline, and write ucd.

See also

The example script FIELD TO UCD demonstrates the **field to ucd** module.

file browser

Send a file name to one or more module(s) file name parameter port(s)

Summary

Name	file browser		
Type	data input		
Inputs	none		
Outputs	string		
Parameters	*Name*	*Type*	*Default*
	File Browser	browser	NULL

Description

The **file browser** module sends a single file name string to one or more string parameter ports on one or more receiving modules. It allows you to simultaneously control file name parameter input to more than one module using only a single **File Browser** parameter.

Before you can connect **file browser** to the receiving module, you must make that receiving module's parameter port visible.

Parameters

File Browser

The single file name string to be sent to the receiving module(s) file name string parameter port(s). The default value is NULL.

Outputs

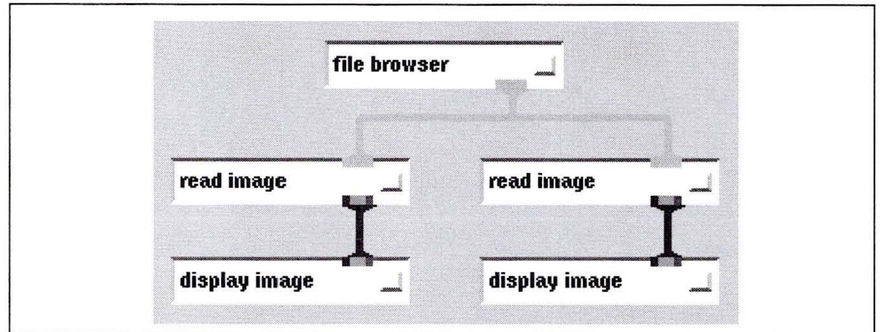
File Name (string)

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

Example

The network in Figure 49 shows the **file browser** module. To construct this network, the file name parameter port on each **read image** module was made visible.

Figure 49
file browser module in an example network



Related modules

Modules that can process **file browser** output are those with file name string parameters.

See also

The example script FILE BROWSER demonstrates the **file browser** module.

flip normal

Change direction of each vertex normal for a geometry object

Summary

Name	flip normal
Type	filter
Inputs	geometry
Outputs	geometry
Parameters	none

Description

The **flip normal** module transforms a 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 (required; geometry)

The input can be any geometry.

Outputs

Geometry (geometry)

The output is a geometry that represents the same object.

Example

The network in Figure 50 shows **flip normal.**

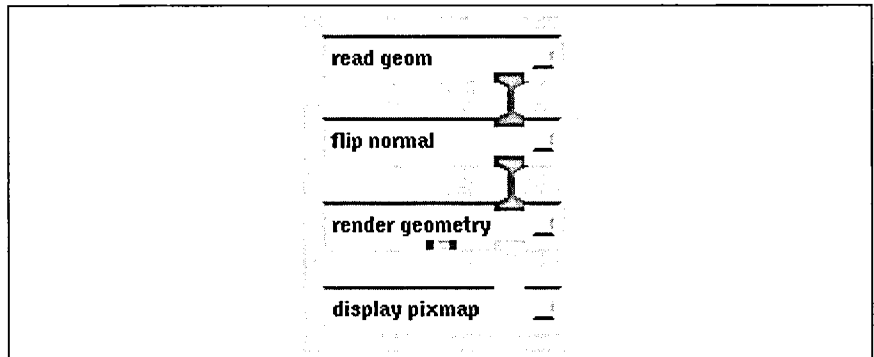


Figure 50
flip normal module in an example network

flip normal

Related modules

read geom, offset, shrink, tube, and render geometry

Note

Some filter modules (for example, **offset**) sometimes produce bad normals that can be corrected with **flip normal**.

See also

The example script FLIP NORMALS demonstrates the **flip normal** module.

float

Send a floating-point number to one or more module(s) floating-point parameter port(s)

Summary

Name	float				
Type	data input				
Inputs	none				
Outputs	real				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	float value	dial	0.0	none	none

Description

The **float** module sends a single 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 you 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.

Parameters

float value

The single 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 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 (for example, 0.55 or -100.25).

Outputs

Float Output (real)

The floating-point value is sent to all modules with float type parameter ports connected to the **float** module.

Example

The network in Figure 51 reads a field, then produces both a contour and an isosurface for the same floating-point value with both outputs composited in the **render geometry** display window.

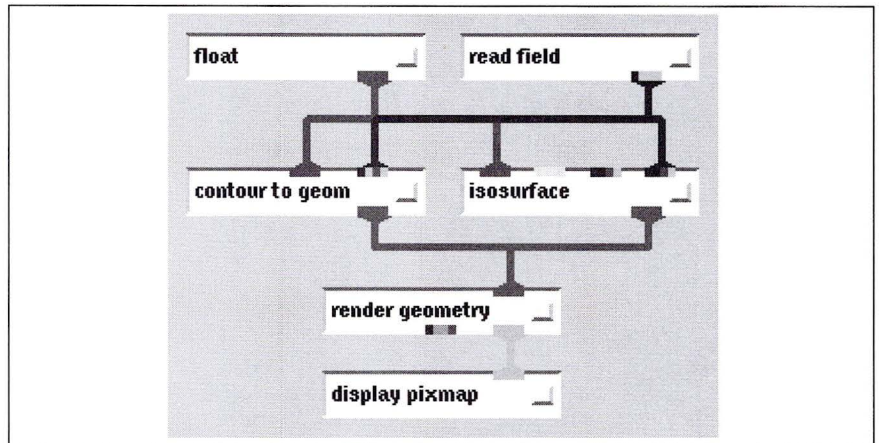


Figure 51
float module in an example network

Related modules

Modules that can process **float** output are those with float type parameters.

generate colormap

Output a colormap

Summary

Name	generate colormap				
Type	data input				
Inputs	none				
Outputs	colormap				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	colormap	color editor			
	lo value	float	0	none	none
	hi value	float	255	none	none

Description

The **generate colormap** module produces a 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**

When the range of values in the input field is not evenly distributed between 0 and 255, or if much of the data lies outside the 0-255 range, you can use the **color range** module to effectively scale the output colormap to the range of your data.

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).

Parameters

colormap

The state of the colormap editor control widget specifies the colormap to be generated:

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.

composite

This is a toggle—when ON, the editing panel becomes a composite of the hue, saturation, and brightness panels. A line through the composite panel display indicates the status of the currently-selected panel: hue, saturation, brightness, or opacity.

edit

Selecting this pops 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 240.0 around this circle. 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 0-175, 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** controls to numbers that produce a contrasting hue (values are 0.0-1.0). Then select **do interpolation** 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 **lo value** becomes assigned to **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 (for example, an opacity of 0.667 is becomes 0.333).

- **cycle**

  Performs a circular shift on the current editing panel. For example, with a value of 10, selecting **cycle** effectively moves the image in the editing panel down by 10 slots. Subsequent selections 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. Selecting this pops up a File Browser widget, allowing you to specify a file name. You can also change the working directory.

write

Writes the current colormap to a disk file. Selecting this pops up a File Browser widget, allowing you to specify a file name. You can also change the working directory.

lo value

A floating-point dial that specifies the minimum data value that can be used as input to the colormap (the value at the top of the editing panel).

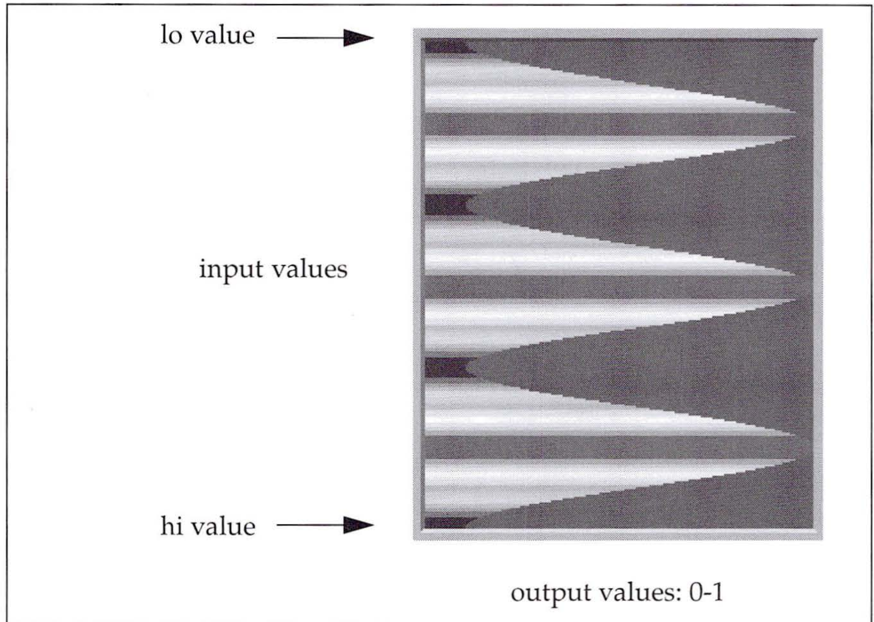
hi value

A floating-point dial that specifies the maximum data value that can be used as input to the colormap (the value at the bottom of the editing panel).

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 shown in Figure 52.

Figure 52
Editing panel organization



Outputs

Colormap (colormap)

The output is a 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 256 entries, with the hue, saturation, brightness, and opacity default values as described above.

Example

The network in Figure 53 reads in a 3-vector field. The **extract scalar** module selects one of the fields values. **color range** stores the field's minimum and maximum values so that the colormap can be scaled to the range of data in the field.

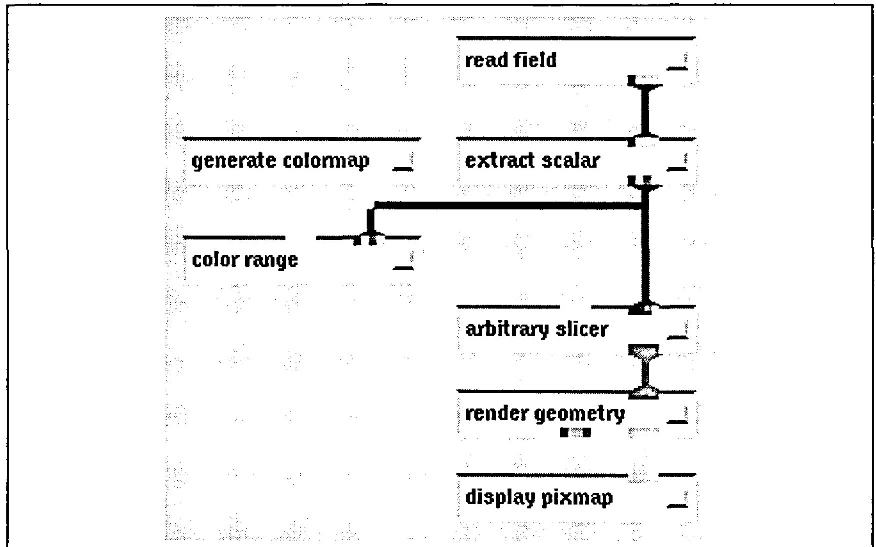


Figure 53
generate colormap module in an example network

Limitations

The **generate colormap** module can only generate colormaps with 256 entries.

See also

The example scripts COLOR RANGE and PROBE demonstrate the **generate colormap** module.

generate filters

Generate 2D filters for image processing

Summary

Name	generate filters
Type	data input
Inputs	none
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.

Parameters

selection

Sets the function used to produce the image processing filter. Each function 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 0.0-1.0.

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 uniformly distributed random filter that is not normalized.
dx	Generates the x-component of the sobel operator, which detects changes in the image in the x-direction. This can be used to locate vertical edges in images. The dx filter is 3 by 3 and cannot be resized.
dy	Generates the y-component of the sobel operator, which detects changes in an image in the y-direction. This can be used to locate horizontal edges in images. The dy filter is 3 by 3 and cannot be resized.

Size

Determines the length of the filter's sides. Filters are squares. Convolving a filter with an image is a N by 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 (field 2D scalar float)

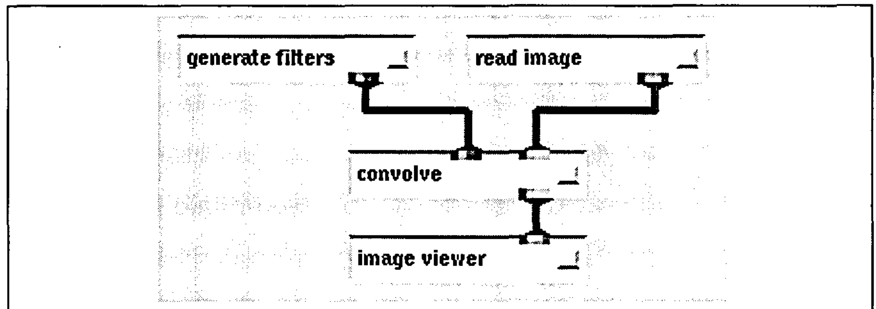
The output is a 2D field of scalar floats (that is, a grid where every location contains one floating point value).

Examples

1.

The network in Figure 54 generates a filter, convolves it with an image, then displays the result.

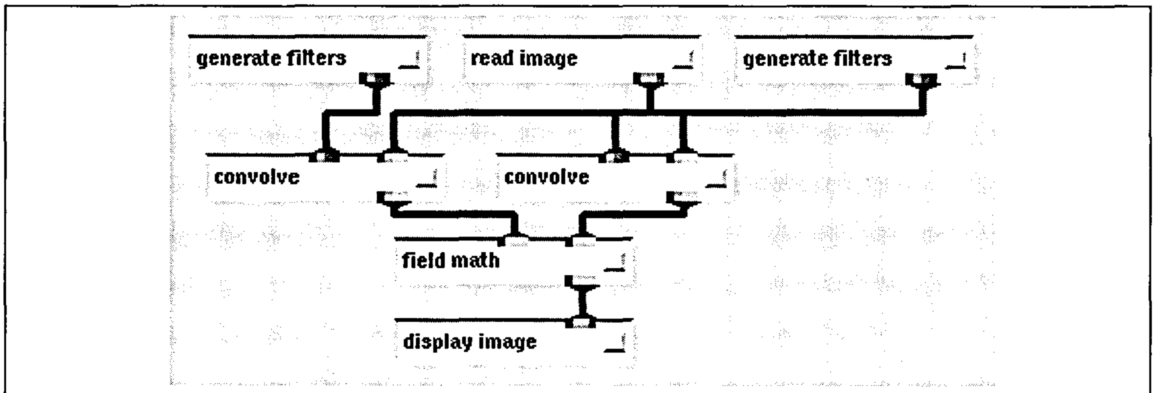
Figure 54
generate filters module in an example network



2.

The network in Figure 55 shows how you can combine the dx and dy filters into the equivalent of a sobel edge detecting operator.

Figure 55
generate filters module in an example network



Related modules

Modules that can process **generate filter**'s output are convolve, colorizer, and orthogonal slicer.

See also

The example script GENERATE FILTERS demonstrates the **generate filter** module.

generate histogram

Plot distribution of data values in a scalar field

Summary

Name generate histogram

Type filter

Inputs field *any-dimension* scalar *any-data any-coordinates*

Outputs field 1D scalar integer uniform

Parameters

Name	*Type*	*Default*	*Min*	*Max*
number of bins	integer dial	256	1	1024
min bin	float dial	0.0	none	none
max bin	float dial	255.0	none	none
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. The range of each chunk is determined by **(max bin - min bin) / number of bins.**

generate histogram reads the input field and examines each value. It decides which subdata range bin the value would fit into and increments the integer count for that bin by one. If the value is below **min bin** or above **max bin,** it is discarded.

The result of this is a 1D field of **number of bins** elements, with each element an integer count of the number of original data values that fell into that range. Connect the output field to the **graph viewer** module's rightmost input port.

Inputs

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

A scalar 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** 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.

choice

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 subrange. If **cumulative** is selected, each bin contains a count of the number of data values that fell into its subrange plus the total of all bins preceding it.

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 cannot be used; if they are moved, they will snap back to their original values.

Normalize is on by default.

Outputs

Data Field (field 1D scalar integer uniform)

The output field is a 1D field, **number of bins** long, with each element an integer count of the number of data values that fell into its range. It is used as One Column input to the **graph viewer** module's rightmost input port.

Example

The network in Figure 56 reads in a volume (byte data in the range 0-256), calculates the distribution of values, and graphs the result.

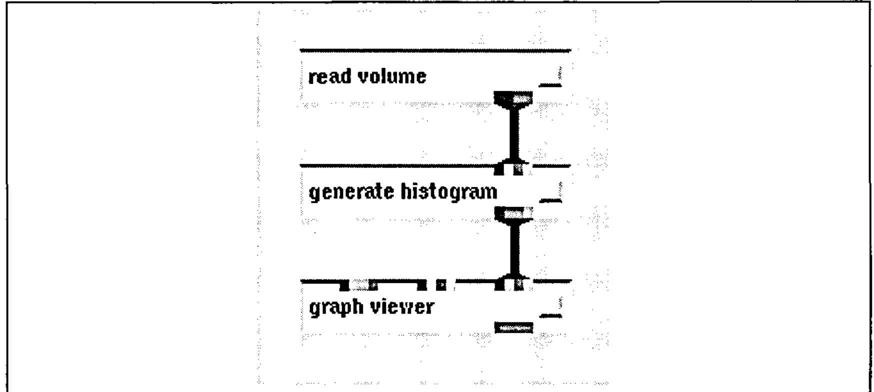


Figure 56
generate histogram module in an example network

Related modules

Modules that could provide the **Data Field** input are those that output a scalar field.

Modules that can process **generate histogram** output are graph viewer and print field.

See also

The example scripts GENERATE HISTOGRAM and GRAPH VIEWER demonstrate the **generate histogram** module.

gradient shade

Apply lighting and shading to colored data set

Summary

Name gradient shade

Type filter

Inputs field 4-vector byte uniform
field 3-vector real uniform
field 2D scalar real

Outputs field *same-dimension* 4-vector byte uniform

Parameters

Name	*Type*	*Default*	*Min*	*Max*
ambient	float	0.1	0.0	1.0
diffuse	float	0.8	0.0	1.0
specular	float	0.0	0.0	1.0
gloss	float	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. These include creating shiny and matte surfaces and controlling the location of the light source.

Inputs

Data Field (required; field 4-vector byte uniform)

The input field is an image (2D pixel array) or a block of voxels (3D pixel array).

Gradient (required; field 3-vector real uniform)

This field is the gradient of the **Data Field**. The gradient is supplied by **compute gradient**.

Transformation Matrix (optional; field 2D scalar real)

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

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 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.

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.

lt off-ctr

The angle between the light source and the positive Z-axis (which comes out of the screen at a right angle).

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

$$(int_{amb} \cdot ambient) + (int_{diff} \cdot diffuse \cdot \cos(phi)) + (int_{diff} \cdot specular \cdot \cos^{gloss}(ltoff-ctr))$$

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.

Examples

1.

The network in Figure 57 shades a 2D image.

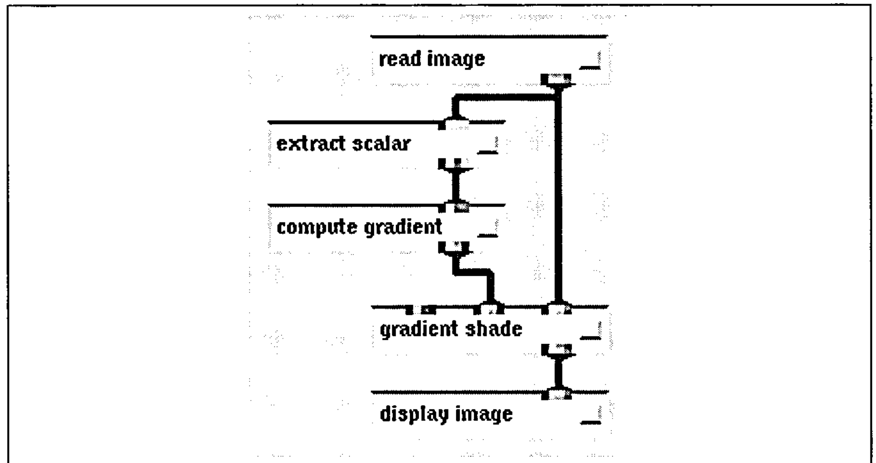


Figure 57
gradient shade module in an example network

2.

The network in Figure 58 shades a 3D image.

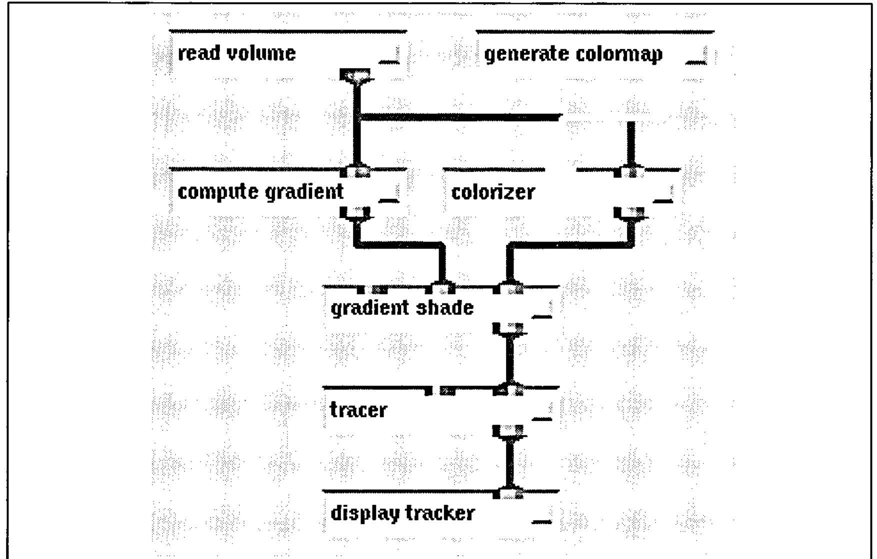


Figure 58
gradient shade module in an example network

Related modules

Module that could provide the **Data Field** input is read volume.

Module that could provide the **Gradient** input is compute gradient.

Modules that could provide the **Transformation Matrix** input are display tracker and euler transformation.

Module that could be used in place of **gradient shade** is colorizer.

Module that can process **gradient shade** output is display image (for 2D data).

See also

The example script ANIMATED FLOAT demonstrates the **gradient shade** module.

graph viewer

Create contour and XY-plots of data (Graph Viewer)

Summary

Name	graph viewer
Type	data output
Inputs	field *any-dimension* scalar *any-data any-coordinates* field *any-dimension* scalar *any-data any-coordinates* field 2D 4-vector byte uniform
Outputs	geometry
Parameters	none

Description

The **graph viewer** module provides access within a network to the complete Graph Viewer subsystem. Many modules can supply input data. That is, many field-format outputs can be connected to **graph viewer**'s input ports. Depending 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 PostScript image of the plot can be saved with the Write PostScript selection.

The **graph viewer** window can be reparented to page and stack widgets using the Layout Editor.

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 multilevel menu of the Graph Viewer subsystem. Thus, when you add the **graph viewer** 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** 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 any mouse button. When the AVS Data Viewers pop-up menu appears, move the cursor to Graph Viewer and release the mouse button.

In some circumstances, it is useful 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 your 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**, ConvexAVS 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 pull-down menu, which is accessed by clicking on the dimple in the upper left 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 and magnifies the image to fit. If the window is embedded in a page or stack, it becomes a top-level window that can be freely resized and moved using your 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 only graphs the first two dimensions and ignores the rest. Many modules can create 2D subsets of fields (**orthogonal slicer** is an example). If such a module is used twice in succession, a 1D subset of the field is created. The simplest example is **orthogonal slicer.** 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. 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 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 (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 geometry-format files to the format of another type of device. You could also use the **render geometry** module followed by the **pixmap to image** module to produce an image version of a **graph viewer** plot for the **image viewer.**

Examples

1.

The network in Figure 59 reads a volume, then uses **orthogonal slicer** to section out a 2D slice of the volume for plotting as X and Y data. 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.

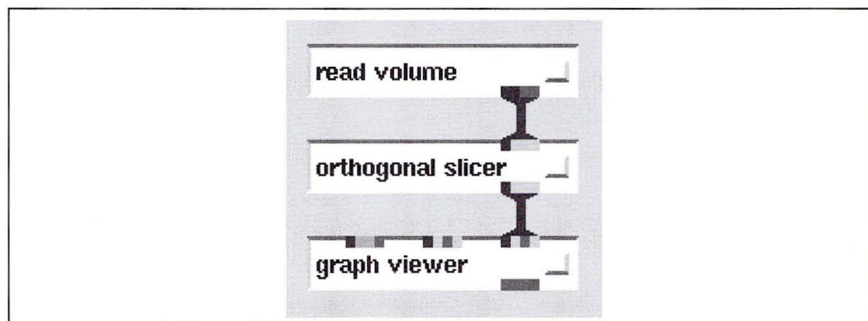


Figure 59
graph viewer module in an example network

2.

The network in Figure 60 reads a volume, then uses the **orthogonal slicer** module twice to extract a 1D slice through the volume data.

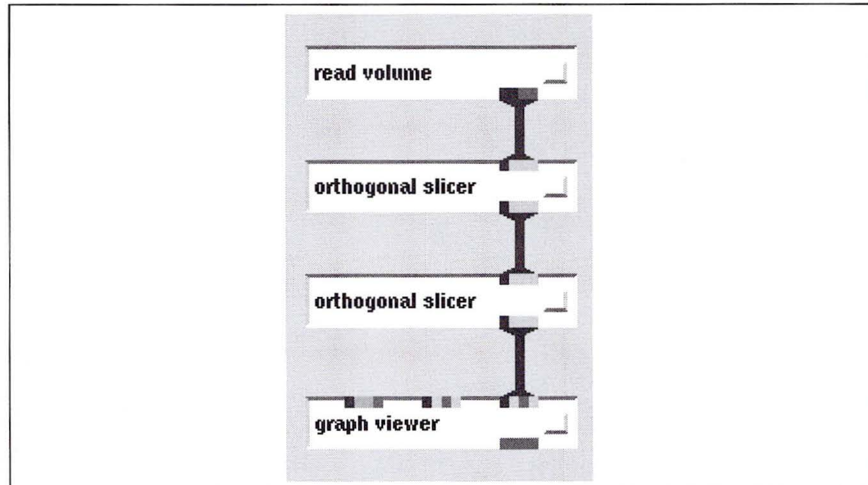


Figure 60
graph viewer module in an example network

3.

The network in Figure 61 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.

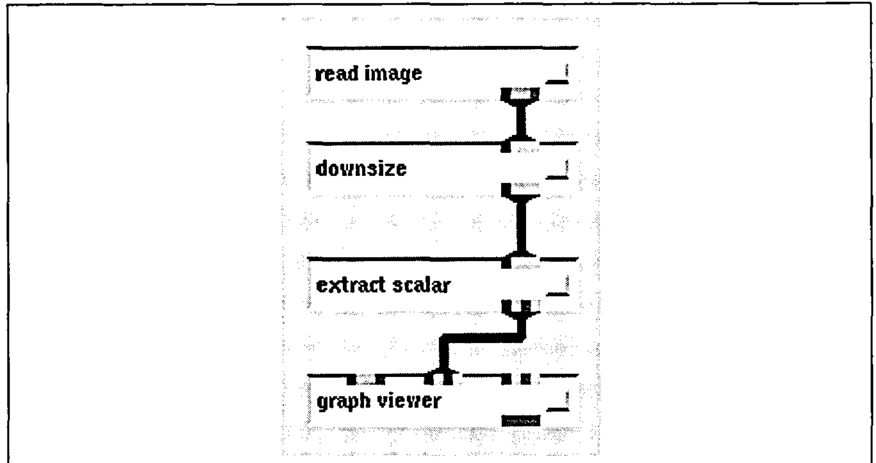


Figure 61
graph viewer module in an example network

4.

The network in Figure 62 does the same as in Figure 61, but displays the contour plot on top of the image it is a contour of. As with the previous network, downsize the image to some reasonable size, and extract either the red, green, or blue bytes from it. The **mirror** module is necessary 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. To flip the image top to bottom like this, select Y on the **mirror** module. 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.

Related modules

generate histogram

See also

Two example GRAPH VIEWER scripts demonstrate the **graph viewer** module.

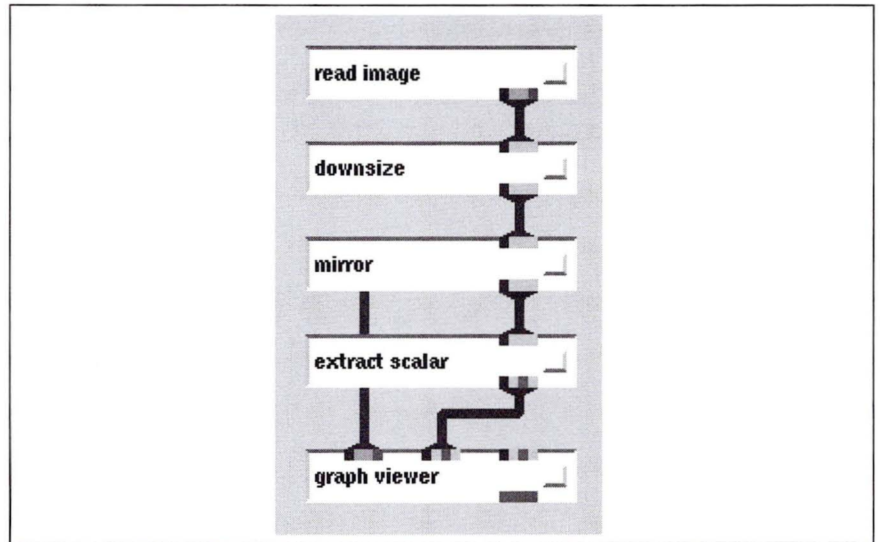


Figure 62
graph viewer module in an example network

hedgehog

Show vectors in a 3D 3-vector field

Summary

Name hedgehog

Type mapper

Inputs
field 3D 3-vector float uniform
field irregular 3-space
upstream transform

Outputs
geometry
upstream transform

Parameters

Name	*Type*	*Default*	*Min*	*Max*
sample	radio	Point		
Vector Scale	float	1.0	0.01	10.0
N Segment	integer	16	2	64
choice	radio	point		

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 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

The module outputs the line segment(s) representing the values of the vector field at the sample location(s). The lines have 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.

Because arbitrarily-oriented sample locations 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 **choice** selections disappear from **hedgehog**'s control panel.

Inputs

Volume Data (required; field 3D 3-vector float uniform)

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 floats.

Sample (optional; 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 **choice** parameter.

Upstream Transform (optional; invisible, autoconnect)

When the **hedgehog** and **render geometry** 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, **render geometry** informs the **hedgehog** module what the sample's new location is, and the **hedgehog** module recalculates the location and data it is displaying accordingly.

Parameters

sample

Controls the way in which the field value is determined at each sample location:

- 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.

Vector Scale

The lengths of the line segments output by this module are proportional to this value.

N Segment

An integer value that 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**.

choice

Specifies the type of sample taken from the vector field: point, line, circle, plane, or space.

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.

Upstream Transform (`struct upstream_transform`)

When the **samplers** module is connected to the leftmost input, this port is used to pass upstream transformation data, originating from **render geometry**, to the **samplers** module.

Example

The network in Figure 63 visualizes the output of the **compute gradient** module.

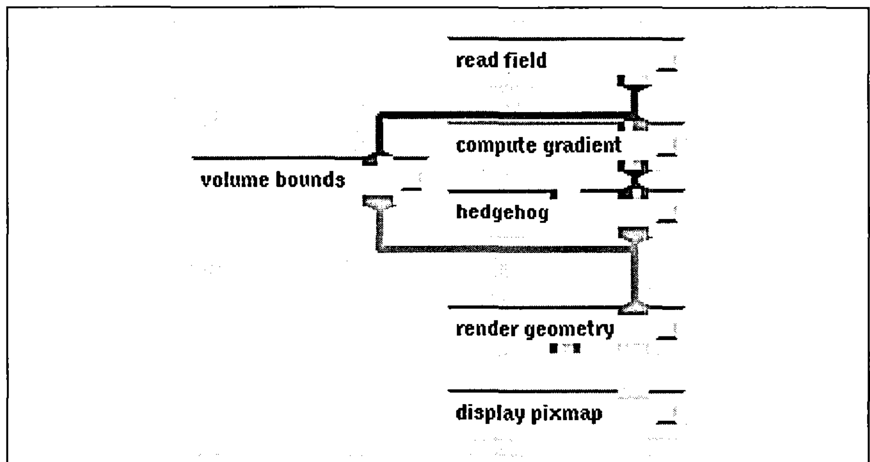


Figure 63
hedgehog module in an example network

Related modules

Modules that could provide data input are read volume and volume manager.

Module that could provide gradient computation is compute gradient.

Modules that could provide vector operations are vector curl, vector div, vector grad, vector mag, and vector norm.

Modules that could provide additional geometries are volume bounds and isosurface.

Modules that can provide geometric rendering are render manager, render geometry, and display pixmap.

Module that could provide the **Sample** input is samplers.

See also

The example script HEDGEHOG demonstrates the **hedgehog** module.

histogram stretch

Balance the histogram of a data set

Summary

Name histogram stretch

Type filter

Inputs field *any-dimension* scalar byte *any-coordinates*

Outputs field *same-dimension* scalar byte *same-coordinates*

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 look-up table. This has the effect of creating an even distribution of values throughout the data set. It is 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 a 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 (field *same-dimension* scalar byte *same-coordinates*)

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.

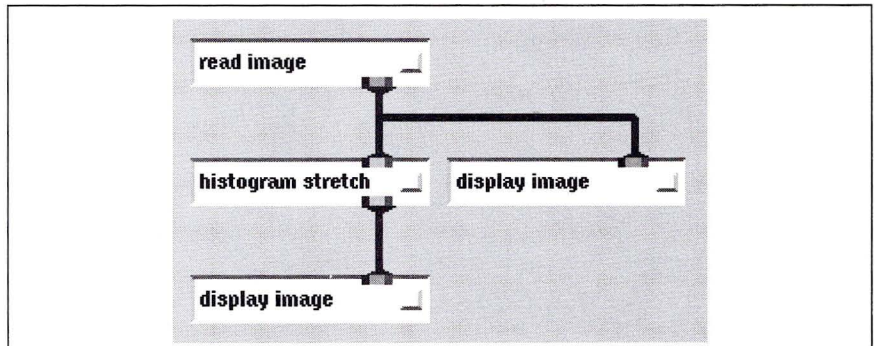
Limitations

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 preprocess the data.

Example

The network in Figure 64 operates on the alpha channel of the image as well as the color channels.

Figure 64 histogram stretch module in an example network



Related modules

Modules that could provide the **Data Field** input are read volume and field to byte.

Modules that can process **histogram stretch** output are field to integer, field to float, and field to double.

See also

The example script HISTOGRAM STRETCH demonstrates the **histogram stretch** module.

hq display image

Display a high-quality image

Summary

Name hq display image

Type data output

Inputs field 2D 4-vector byte

Outputs field 2D 4-vector byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Display Name	string			
Colors	integer	255	2	255
Precision	integer	5	3	8
Dither	boolean	0	0	1
Gamma	float	3	0.01	10.0

Description

The **hq display image** module displays an image in a window on a 8-bitplane pseudocolor X terminal. It performs color quantization on the image and chooses the best set (one that has the minimum sum-of-squares error) of colors for the colormap.

Inputs

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

The input field must be in the ConvexAVS image format.

Parameters

Display Name

A text typein that is used to specify the name of the X server on which the window displaying the image will appear.

Colors

The number of colors the image is quantized to.

Precision

The number of bits of the original color that are used when quantizing the image. The more bits that are used, the better the quantized image will look, but the longer the module will take to compute.

Dither

When set, the quantized image is dithered by the Floyd-Steinberg method. This helps to reduce false contours that sometimes appear in images with smoothly varying intensities.

Gamma

The amount of gamma correction applied to the image. Changing this parameter only effects how the image is displayed in the window, not the output image.

Outputs

Image (field 2D 4-vector byte)

The quantized image containing only the number of unique colors specified by the **Colors** parameter.

Example

The example in Figure 65 shows **hq display image** in an example network.

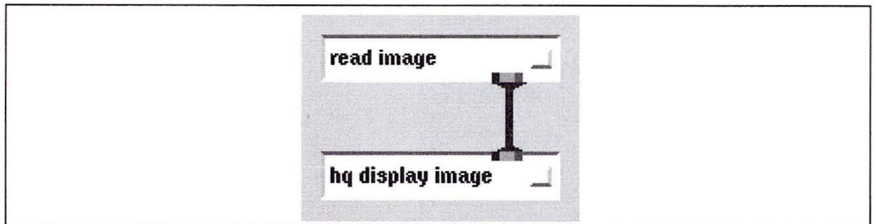


Figure 65
hq display image module in an example network

Related modules

display image and display pixmap

Limitations

This module only works on X terminals that support an 8-bitplane pseudocolor visual.

image compare

Display two images together

Summary

Name	image compare
Type	filter
Inputs	field 2D 4-vector byte field 2D 4-vector byte
Outputs	field 2D 4-vector byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
choice	radio	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 (for example, 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 and the other the secondary image. You can flip back and forth between the primary and secondary image using the **switch** parameter. In most cases, the **valuator** parameter controls the ratio of primary to secondary image appearing in the rectangle.

Inputs

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

The first of two images to compare.

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

The second of two images to compare.

Parameters

choice

Sets the way the two images are displayed together:

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** parameter. This lets you flicker between the images using the **switch** parameter.

circle

Transforms the **valuator** parameter 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 the primary and secondary images.

random

Randomly dithers between one image and the other based on the probability assigned by the **valuator** parameter.

switch

Exchanges the proportions of the screen given to each image.

valuator

Controls the proportion of the rectangle viewing space that each image occupies. Allowable values range from 0.0-1.0, with a default of 0.5 (meaning show half of one image and half of the other). As you move the dial, one image or the other gets more rectangle space.

Outputs

Image (field 2D 4-vector byte)

The output image is the patchwork combination of the primary and secondary images with the same dimensions.

Example

The network in Figure 66 compares an image with a contrasted version of itself.

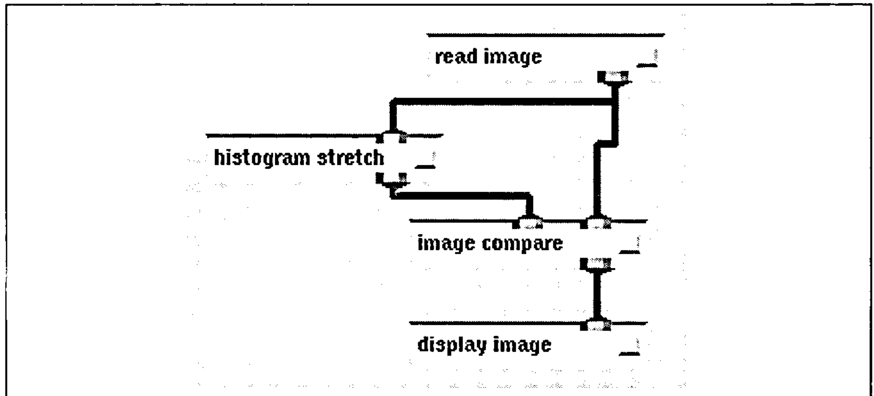


Figure 66
image compare module in an example network

Related modules

Module that could provide the **image compare** inputs is read image.

Modules that can process **image compare** output are image viewer and display image.

Limitations

Both input images must have the same dimensions. Use the **crop** or **interpolate** module to change the size of an image.

See also

The example script IMAGE COMPARE demonstrate the **image compare** module.

image manager

Share images among subnetworks

Summary

Name	image manager	
Type	data input	
Inputs	none	
Outputs	field 2D 4-vector byte	
Parameters	*Name*	*Type*
	IMAGMGR Select	choice
	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 caching mechanism and a way of sharing data among **image manager** modules in separate subnetworks.

This module is in the unsupported library.

Parameters

IMAGMGR Select

A choice that determines how newly-read images will be placed into the list of currently active images:

- If **select** is chosen (the default), 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

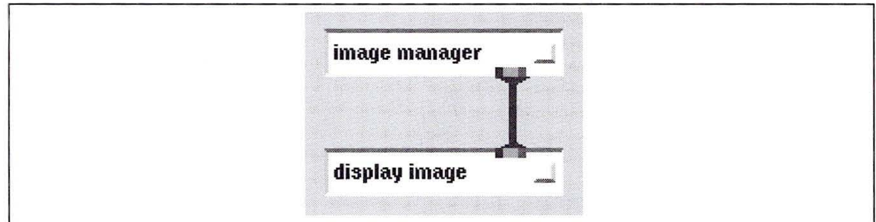
Data Field (field 2D 4-vector byte)

The output is an image.

Example

The network in Figure 67 is used to display an image.

Figure 67
image manager module in an example network



Related modules

Modules that could provide additional image processing are contrast, threshold, histogram stretch, clamp, interpolate, luminence, generate filters, sobel, convolve, and local area ops.

Modules that could decompose or compose images from separate bands are extract scalar and combine scalars.

Module that can display results is display image.

See also

The **read image** module reference contains a description of the image format.

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 to pixmap

Convert image to pixmap

Summary

Name	image to pixmap		
Type	mapper		
Inputs	field 2D 4-vector byte		
Outputs	pixmap		
Parameters	*Name*	*Type*	*Default*
	approx technique	choice	dither

Description

The **image to pixmap** module takes as input an image 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 pseudocolor systems.
- A pixmap is represented by an X Window System resource ID. This means that transferring a pixmap from one module to another is more efficient than transferring all the data that defines an image.

Inputs

Data Field (required; field 2D 4-vector byte)

The input field must be an image.

Parameters

approx technique

Controls the conversion of color values to pixel values. The approximation techniques are:

dither	Uses a dither matrix to approximate each color
floyd steinberg	Slower dithering technique that produces better results for computer-generated imagery
random	Uses a random number dither to approximate each color
monochrome	Uses the luminance of the color as an index into a grey scale ramp
none	Takes the closest approximation for each color

Outputs

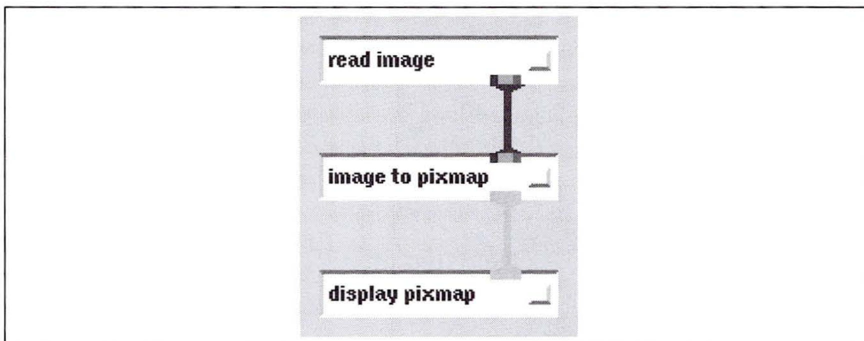
Pixmap (pixmap)

The output is a pixmap.

Example

The network in Figure 68 allows an image to be displayed in an arbitrary-sized window.

Figure 68
image to pixmap module in an example network



Related modules

pixmap to image, transform pixmap, and display pixmap

See also

The example script OUTPUT POSTSCRIPT demonstrates the **image to pixmap** module.

image to postscript

Convert image to PostScript and store in file

Summary

Name	image to postscript	
Type	data output	
Inputs	field	
Outputs	none	
Parameters	*Name*	*Type*
	filename	typein
	mode	radio
	monochrome	toggle
	8 bit	toggle
	compress	toggle
	dither	toggle

Description

The **image to postscript** module converts its input image to the PostScript page description language and stores it in a file.

Two types of PostScript output are supported:

- Suitable for sending to a PostScript-compatible laser printer
- Mathematica compatible

Inputs

Image (required; field)

A field of any type can be input to **image to postscript.**

Parameters

filename

A typein that allows you to enter the name of the PostScript file to be created. After the file is written, the file name 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 file name.

mode

Selects the type of PostScript output:

- laserwriter
- mathematica

The following parameters apply to Mathematica options:

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.

compress

If **ON**, produces compressed output.

If **OFF**, produces uncompressed output.

dither

If **ON**, produces dithered output.

If **OFF**, produces undithered output.

Example

The example in Figure 69 converts an image to a PostScript file.

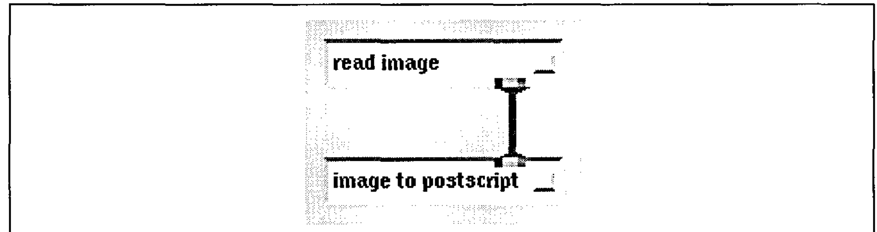


Figure 69
image to postscript module in an example network

Related modules

image to pixmap, output postscript, and render geometry

Limitations

This module does not work on a 16-plane system.

The **compress** option is not supported in any released version of Mathematica.

The dither option produces visual artifacts on some images.

image viewer

Display and manipulate collections of images

Summary

Name	image viewer
Type	data output
Inputs	field 2D 4-vector byte
Outputs	none
Parameters	none

Description

The **image viewer** module provides access within a ConvexAVS 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.

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** button on the **image viewer** control panel. Images sent by upstream modules can be saved to files using the **Write Image** and **Save Scene** buttons. In this way, you can save visualization results and retrieve them later with the **Read Scene** or **Read Image** buttons.

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.

Resizing

The **image viewer**'s pull-down menu, which is accessed by clicking on the dimple in the upper left 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 by 1024), and magnifies the image to fit. If the window is embedded in a page or stack, it becomes a top-level window that can be freely resized and moved using your 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.

Considerations

Instead of having a few control widgets organized onto a single control panel page, its control panel is the entirely separate multilevel menu of the Image Viewer subsystem. Thus, when you add the **image viewer** 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, move the cursor to Image Viewer and release the mouse button.

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 your window manager to move one of these menus.

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**, ConvexAVS 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.

Inputs

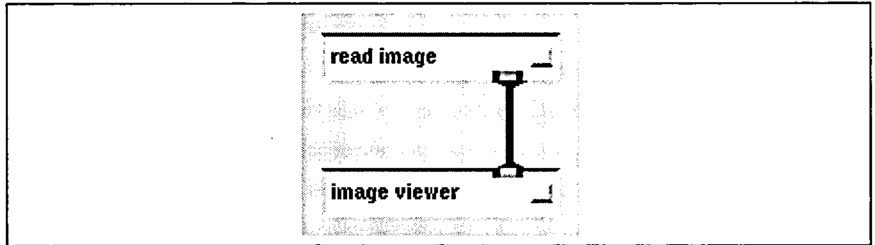
Image (optional; field 2D 4-vector byte)

The input data can be any image. Multiple images can be input to this port. All the images will be combined into the same scene.

Example

The network in Figure 70 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.

Figure 70
image viewer module in an example network



Related modules

display image, read image, and pixmap to image

integer

Send an integer to the integer parameter port of one or more module(s)

Summary

Name	integer				
Type	data input				
Inputs	none				
Outputs	integer				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	integer value	dial	0	none	none

Description

The **integer** module sends a single integer value to one or more integer 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.

Parameters

integer value

The single integer value to be sent to the module(s) integer parameter port(s). The default value is 0. You should be aware of the range of numbers that it is reasonable to send to the receiving modules.

Outputs

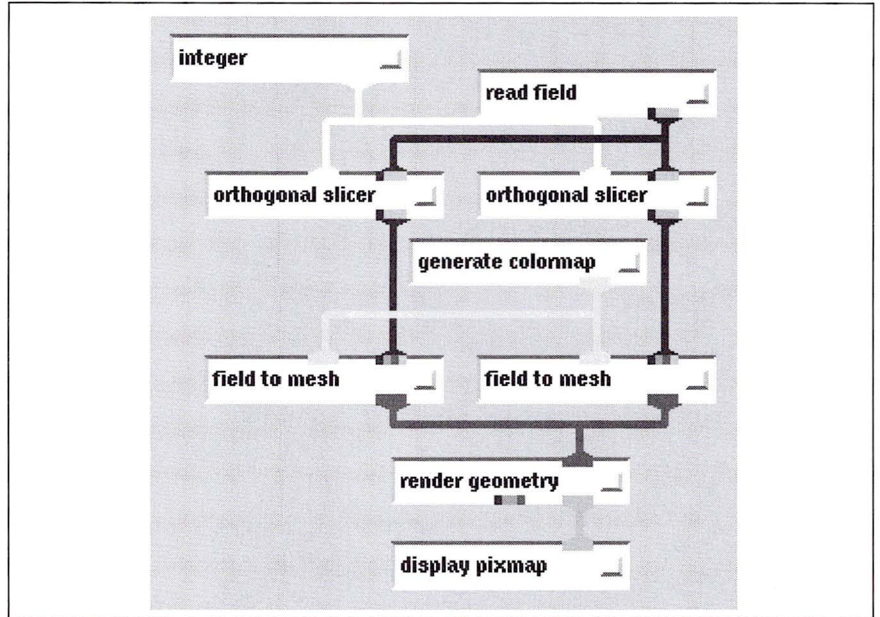
Integer (integer)

The integer value is sent to all modules with integer parameter ports connected to the **integer** module.

Example

The network in Figure 71 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 render geometry window.

Figure 71
integer module in an example network



Related modules

oneshot and tristate

See also

The example scripts INTEGER and FIELD TO BYTE demonstrate the **integer** module.

interpolate

Compute intermediate values to change the size of a field

Summary

Name	interpolate
Type	filter
Inputs	field 2D/3D *n*-vector *any-data any-coordinates*
Outputs	field 2D/3D *n*-vector *same-data 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

The **Point** sampling mode is much quicker than the linear sampling and should be used when interactivity is more important than image quality.

Unlike the **crop** and **downsize** modules, **interpolate**'s advantages are that it can:

- Scale non-uniformly in each dimension
- Do high-quality linear sampling
- Scale data up instead of only down

Inputs

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

The input field to interpolate. The field can be uniform, rectilinear, or irregular.

Parameters

interp_sx
interp_sy
interp_sz

The interpolation factors for the coordinate dimensions.

Sampling

This choice determines the sampling method:

Point Selects the closest pixel (voxel) to the computed one

Bilinear Finds the four pixels (2D) around the computed point and does a linear sampling for in between pixels

Trilinear Finds the eight voxels (3D) around the computed point and does a linear sampling for in between pixels

Outputs

Data Field (field 2D/3D *n*-vector *same-data same-coordinates*)

The output field has the same form as the input field. Appropriate new **min_ext** and **max_ext** values are written to the output field.

Example

The network in Figure 72 shows **interpolate** in an example network.

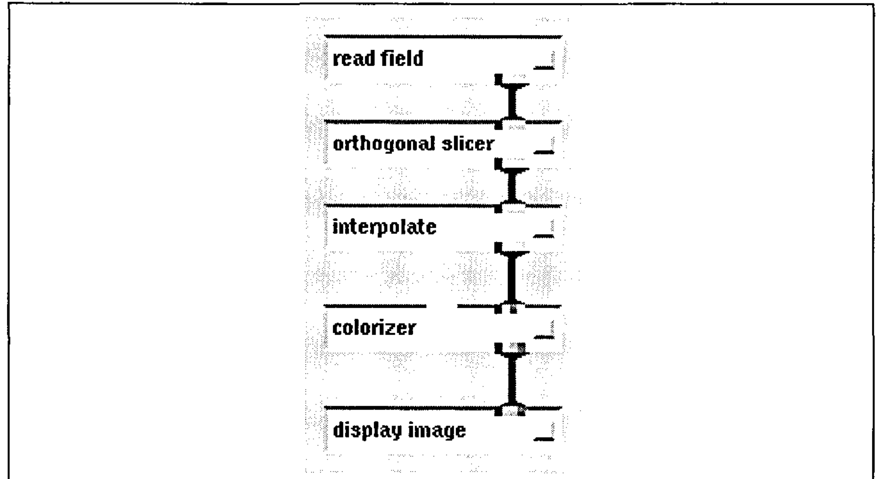


Figure 72
interpolate module in an example network

Related modules

This module is similar to **downsize** (which does uniform, stride-based point sampling) and **crop** (which selects a subset of the data but does not change the resolution).

Limitations

This module does the wrong thing when down-sampling (going from a large image to a small one) in the **Bi/Trilinear** mode. It should average appropriately chosen regions down to each pixel. Instead, it chooses the four pixels around the center of that region and interpolates among them.

See also

The example script INTERPOLATE demonstrates the **interpolate** module.

isosurface

Generate an isosurface for a volume of data

Summary

Name	isosurface	
Type	mapper	
Inputs	field 3D scalar *any-data any-coordinates* field 3D scalar *any-data* colormap	
Outputs	geometry	
Parameters	*Name*	*Type*
	level	float
	optimize surf	toggle
	optimize wire	toggle
	flip normals	toggle

Description

The **isosurface** module inputs a volume data set (3D field of values, 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. Because 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.

Parameters

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 **level** value.

optimize surf
optimize wire

These two toggle parameters allow you to control a trade-off between how efficiently the isosurface is computed and how efficiently it can be rendered. If you turn on **optimize surf**, extra time will be spent generating a more optimal surface description, containing fewer triangles.

Turn on **optimize wire** 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 **render geometry** 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 select the Bi-Directional lighting option in the Geometry Viewer's Lights menu.

Outputs

Isosurface (geometry)

A shaded surface, optionally with an associated wireframe representation.

Note

The most important parameter is **level** (threshold), which is defined in the unbounded floating-point data space of the volume. It is not always easy to know in what range the data is defined. Often, the data is defined as some well-known, real-world domain (for example, temperature in degrees). In some cases, the data has been converted from byte data and therefore must lie within the range 0.0-255.0.

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.

Example

The network in Figure 73 shows **isosurface** in an example network.

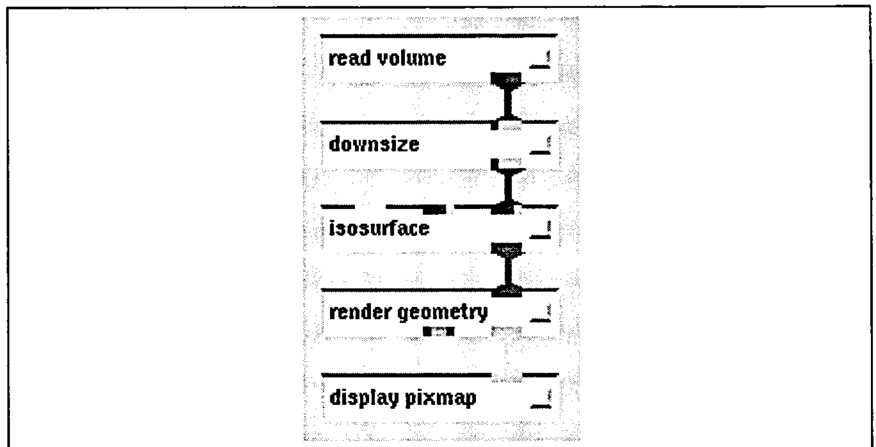


Figure 73
isosurface module in an example network

Related modules

render geometry, downsize, generate colormap, read field, and 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 bidirectional lighting may be helpful.

See also

The example script FIELD LEGEND demonstrates the **isosurface** module.

local area ops

Image processing based on pixel neighborhoods

Summary

Name	local area ops
Type	filter
Inputs	field 2D 4-vector byte uniform **OR** field 1-3D scalar *any-data any-coordinates*
Outputs	field of same type as input

Parameters

Name	*Type*	*Default*	*Min*	*Max*
choice	choice	Min		
kernel width	integer	3	3	31

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 *N* by *N* 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 three, the value of each pixel in the output image becomes the minimum value of the pixel and the eight 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 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)

Typically, the input will be an image, which is a 2D field of 4-vector bytes.

OR

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

The input may be any 1-3D field of scalar values of *any-data any-coordinates.*

Parameters

choice

Sets which local area operation to apply. There are four 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 referred 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 referred 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. Because the median calculation requires a sort, it is very compute intensive, especially when the filter width is large. ConvexAVS 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

Data Field

The output is the same type as the input.

Example

The network in Figure 74 reads in an image, applies the local area operations to it, and displays the resulting image.

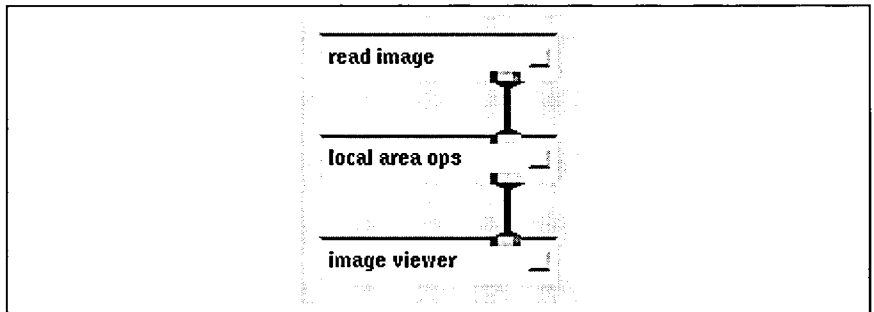


Figure 74
local area ops module in an example network

Related modules

Modules that could provide the **Data Field** input are read image, pixmap to image, and orthogonal slicer.

Modules that can process the output of **local area ops** are display image and image viewer.

See also

The example script LOCAL OPS demonstrates the **local area ops** module.

luminence

Compute the luminance of an image

Summary

Name	luminence
Type	filter
Inputs	field 2D 4-vector byte
Outputs	field 2D scalar byte
Parameters	none

Description

The **luminence** module computes the luminance (brightness) of an image, then outputs a 2D 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 * red) + (0.587 * green) + (0.114 * 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 4-vector byte)

The image whose luminance to calculate.

Outputs

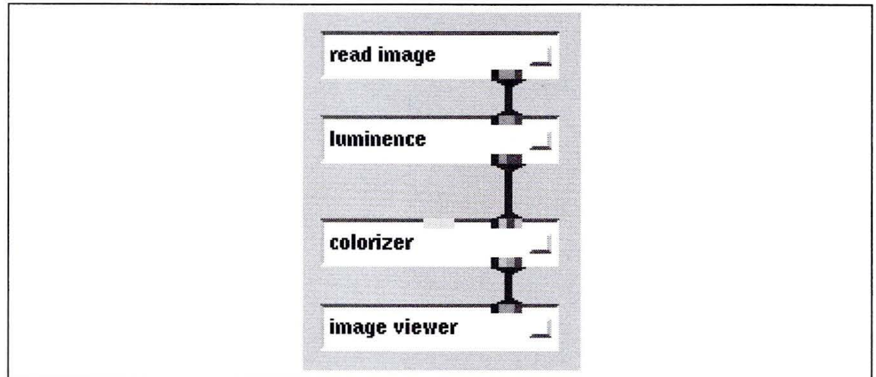
Data Field (field 2D scalar byte)

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

The network in Figure 75 reads an image, computes its luminance, colorizes the resulting field with the default black and white colormap, and produces a black and white version of the original image. The result is displayed through the **image viewer**.

Figure 75
luminence module in an example network



Related modules

Modules that could provide the **Image** input are those that produce an image as output.

Modules that can process **luminence** output are colorizer, contrast, and replace alpha.

See also

The example script LUMINENCE demonstrates the **luminence** module.

mirror

Reverse array indices in a 2D or 3D data set

Summary

Name	mirror
Type	filter
Inputs	field 2D/3D *n*-vector *any-data any-coordinates*
Outputs	field 2D/3D *n*-vector *same-data same-coordinates*

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. In a 50 by 100 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)

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 (required; field 2D/3D *n*-vector *any-data any-coordinates*)

The input may be any 2D/3D scalar field.

Parameters

Axis

The choices for exchanging the data are:

Original	Copies the input to the output; no transformation is performed.
X	Reverses the array indices in the X-dimension (first dimension).
Y	Reverses the array indices in the Y-dimension (second dimension).
Z	Reverses the array indices in the Z-dimension (third dimension). (Equivalent to Original for a 2D field.)

Outputs

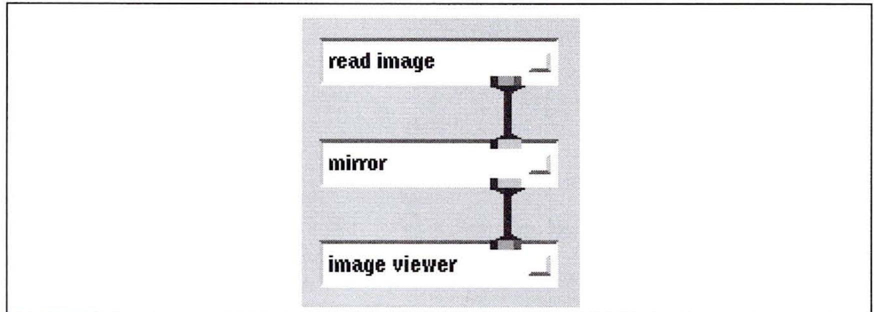
Data Field (field 2D/3D *n*-vector *same-data same-coordinates*)

The output field as the same form as the input field.

Example

The network in Figure 76 uses the **mirror** module.

Figure 76
mirror module in an example network



Related modules

This module combined with **transpose** can reorient the data in any way.

See also

The example script GRAPH VIEWER demonstrates the **mirror** module.

momentum PLOT3D

Strip out the PLOT3D momentum vector

Summary

Name	momentum PLOT3D
Type	filter
Inputs	field 3D 5-vector irregular 3-space float
Outputs	field 3D 3-vector irregular 3-space float
Parameters	none

Description

This module allows you to visualize the momentum field in the PLOT3D file. It merely converts the 5-vector into the momentum vector field by extracting the momentum 3-vector.

Inputs

Data Field (required; field 3D 5-vector irregular 3-space float)

This input data is the 5-vector field output by **read PLOT3D**. It is the most important field because it contains the mesh and solution data.

Outputs

Vector Field (field 3D 3-vector irregular 3-space float)

A vector field representing the momentum field from the PLOT3D file read by **read PLOT3D**.

Example

The example in Figure 77 reads in a PLOT3D data set and does an isosurface on the momentum field. Only the PLOT3D field itself is used. We do not use the blanking records because we are extracting a part of the raw PLOT3D field.

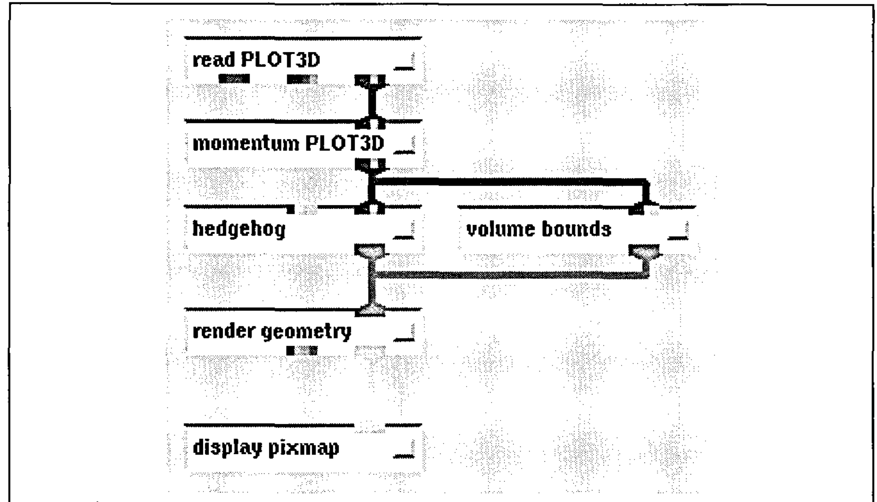


Figure 77
momentum PLOT3D module in an example network

Related modules

density PLOT3D, stagnation PLOT3D, read PLOT3D, scalar PLOT3D, and vector PLOT3D

Related programs

export_PLOT3D and import_PLOT3D

offset

Translate vertices along vertex normals

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 a geometry, so that each vertex of each polygon is translated along its vertex normal. It is useful for emphasizing surface discontinuities (for example, cusps) and enlarged objects.

Inputs

Geometry (required; geometry)

A geometry, created with the geometry library or by another module.

Parameters

offset

The amount by which each vertex is translated along its normal. Positive values enlarge the geometry. Negative values collapse it.

Outputs

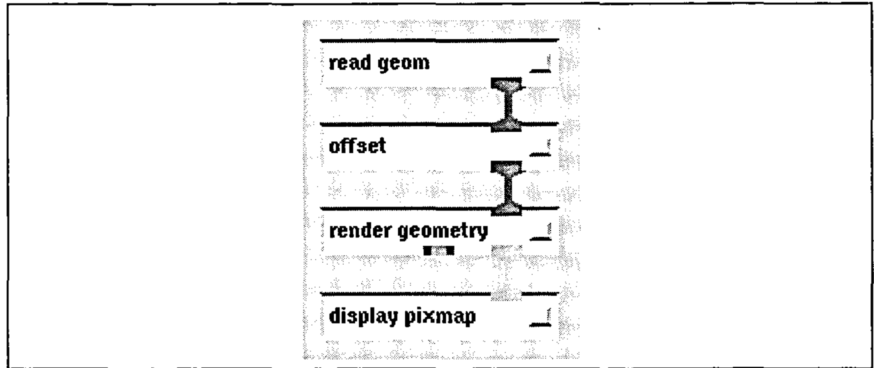
Geometry (geometry)

A geometry that represents that same object(s) as the input data.

Example

The network in Figure 78 shows the **offset** module.

Figure 78
offset module in an example network



Related modules

read geom, flip normal, tube, and render geometry

Limitations

This module works only for polytriangle strips and meshes but not for polyhedra, lines, or spheres. It has no effect on objects that do not have surface normals.

See also

The example script OFFSET demonstrates the **offset** module.

oneshot

Send a oneshot value to one or more module(s) oneshot parameter port(s)

Summary

Name	oneshot				
Type	data input				
Inputs	none				
Outputs	oneshot				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	oneshot	oneshot	0	0	none

Description

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

oneshot outputs an integer that 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 controllable, but depends on the speed with which ConvexAVS executes the network to which **oneshot** is connected. Thus, if ConvexAVS 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. **oneshot** is often used as a signal to perform some operation.

Because the 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.

Parameters

oneshot

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 parameter ports that are connected to the **oneshot** module.

Example

The network in Figure 79 shows the use of **oneshot**.

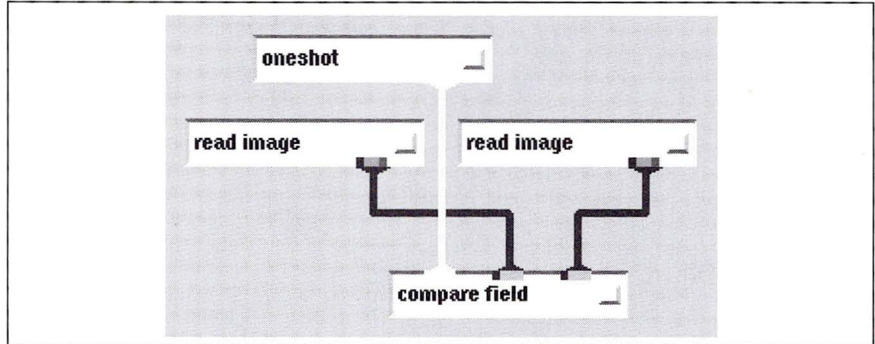


Figure 79
oneshot module in an example network

Related modules

integer and tristate

See also

The example scripts WRITE VOLUME and WRITE IMAGE demonstrate the **oneshot** module.

orbital tiler

Generate an isosurface for a volume of data

Summary

Name orbital tiler

Type mapper

Inputs field 3D scalar *any-data any-coordinates*
field 3D scalar *any-data*
colormap

Outputs geometry

Parameters

Name	*Type*	*Default*
level	float	0.002
tile +/–	toggle	off
save surface	toggle	off
optimize surf	toggle	off
optimize wire	toggle	off
flip normals	toggle	off

Description

The **orbital tiler** module inputs a volume data set (3D field of values, 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. If the **tile +/–** parameter is selected, isosurfaces for both the positive and negative values of level are created. If **save surface** is selected, the current surface will be saved, and all future operations will affect a new surface.

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. Because 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.

Parameters

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 **level** value. The default value is 0.002, which is common for viewing molecular orbitals.

tile +/−

Generate isosurfaces at the positive and negative values of level. If the level is zero, only one surface is generated.

save surface

Save the geometry of the current surface. Future operations will create a new surface. You may access the surfaces from the Geometry Viewer. The files are named orbital_tiler*x* and orbital_tiler_inverse*x*, where *x* is an integer that increments after each surface is saved.

optimize surf
optimize wire

These two toggle parameters allow you to control a trade-off between how efficiently the isosurface is computed and how efficiently it can be rendered. If you turn on **optimize surf**, extra time will be spent generating a more optimal surface description, containing fewer triangles.

Turn on **optimize wire** 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 **render geometry** 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 select the Bi-Directional lighting option in the Geometry Viewer's Lights menu.

Outputs

Isosurface (geometry)

A shaded surface, optionally with an associated wireframe representation.

Note

The most important parameter is the **level** (threshold), which is defined in the unbounded floating-point data space of the volume. It is not always easy to know in what range the data is defined. Often, the data is defined as some well-known, real-world domain (for example, temperature in degrees). In some cases, the data has been converted from byte data and therefore must lie within the range 0.0-255.0.

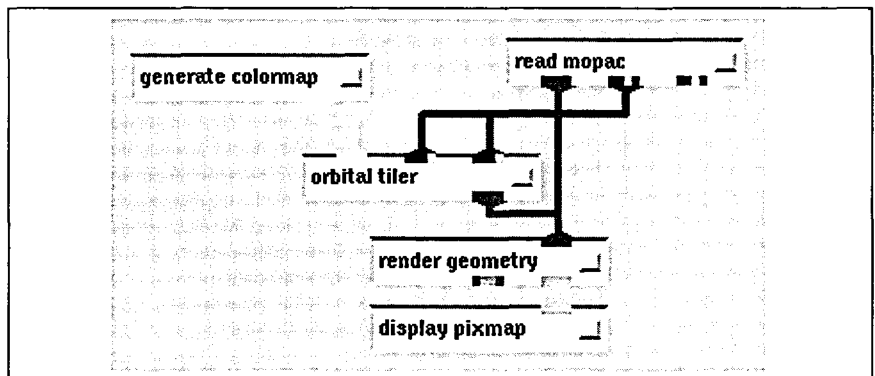
Because **orbital tiler** is compute-intensive, it is often advisable to include a **downsize** module in the network. This allows you to quickly select a proper isosurface level before generating one at full resolution.

Another technique is to use the **Action** capability of the Geometry Viewer (**render geometry** module) to save and play back a sequence of isosurfaces at different value levels.

Example

The network in Figure 80 shows **orbital tiler** in an example network.

Figure 80
orbital tiler module in an example network



Related modules

render geometry, downsize, generate colormap, read field, and 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 using bidirectional lighting may be helpful.

orthogonal slicer

Slice through 2D or 3D field with plane perpendicular to coordinate axis

Summary

Name	orthogonal slicer
Type	mapper
Inputs	field 2D/3D *n*-vector *any-data any-coordinates*
Outputs	field 1D/2D *n*-vector *same-data same-coordinates*

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Choices*
slice plane	int	0	0	1	
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). In fact, the **orthogonal slicer** module can takes slices of vector-valued fields, also. It passes through whatever data type is presented to it. For example, if the input is a "field 3D 3-vector float," the output is a "field 2D 3-vector float."

orthogonal slicer

Inputs

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

The input may be any 2D or 3D 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.

Examples

1.

The network in Figure 81 takes a slice from a scalar volume and displays it.

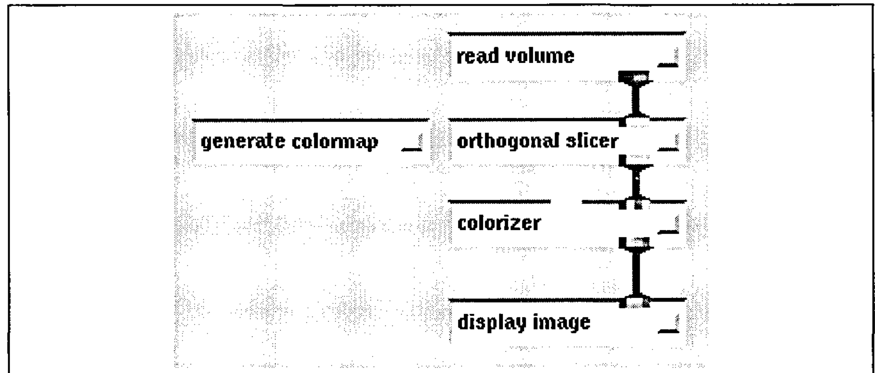
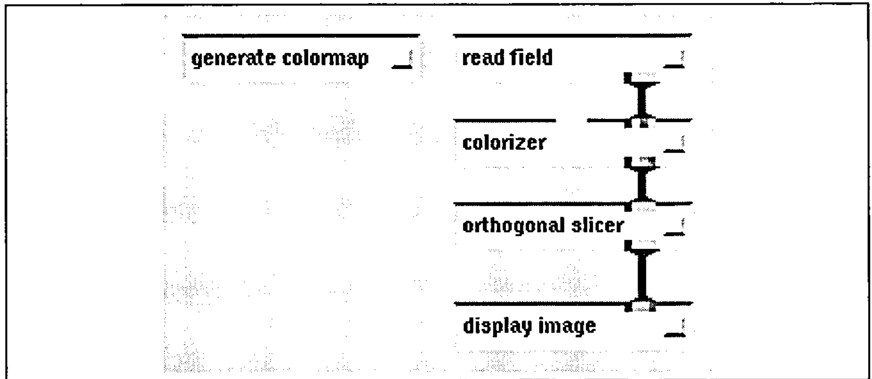


Figure 81
orthogonal slicer module in an example network

2.

orthogonal slicer supports the passing of points data for rectilinear and irregular data. This is an important module for visualizing curved data sets, as shown in Figure 82.

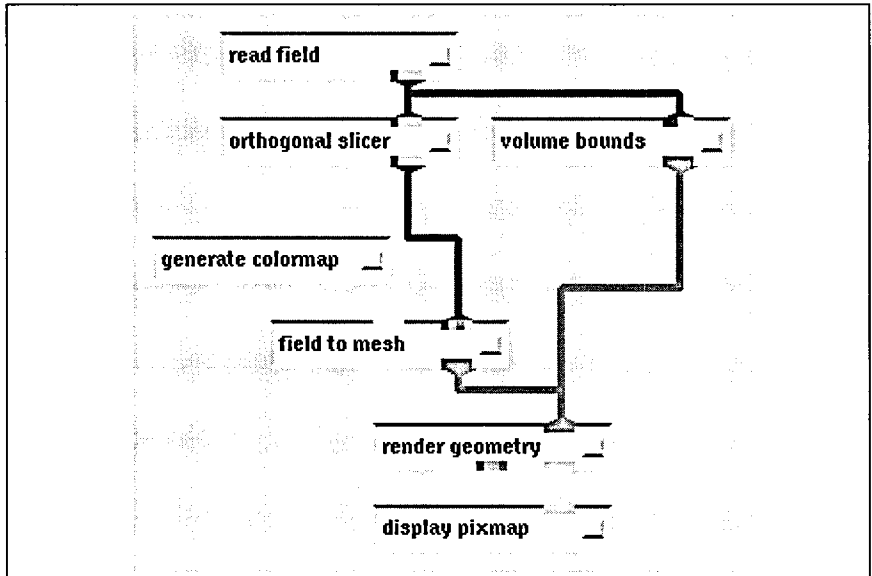
Figure 82
orthogonal slicer module in an example network



3.

The network in Figure 83 shows how to use **orthogonal slicer** to take a slice from an irregular field and convert it to a geometry.

Figure 83
orthogonal slicer module in an example network



Related modules

field to mesh and colorizer

Note

A slice plane has normals pointing in only one direction. If you rotate the view, turn on bidirectional lighting to see the other side.

See also

The example scripts ANIMATED INTEGER, COLOR RANGE, and VECTOR CURL demonstrate the **orthogonal slicer** module.

output ImageNode

Write images to video tape using a Diaquest ImageNode

Summary

Name	output ImageNode
Type	data output
Inputs	field 2D 4-vector uniform byte
Outputs	none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Insert Point	text	00:00:00		
Edit Length	integer	1	1	270000
Record	boolean	false		

Description

The **output ImageNode** module records animation onto video tape. The module displays each image it receives on the frame buffer of a Diaquest ImageNode.

Set the **Record** parameter to record the images on an attached VCR using single frame recording.

If the input image is smaller than the ImageNode's frame buffer, it is centered and the remaining parts of the frame buffer are filled with black. If the input image is larger than the frame buffer, the image is cropped, and the center part is displayed. When the ImageNode uses NTSC format, the frame buffer is 648 by 486 pixels. In PAL format, the frame buffer is 768 by 576.

The source code for **output ImageNode** is distributed in /usr/avs/examples/imagenode.

Inputs

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

The input image must be a uniform 2D field, with a 4-vector of byte data values at each location in the field. The image can be any size. It will be cropped and matted to fit.

Parameters

Insert Point

A text typein that may be used to specify the video tape location where the next image will be recorded. This may be specified either as *mm:ss:ff* where *mm*, *ss*, and *ff* are minutes, seconds, and frames or as an absolute frame number. The insert point is incremented by the edit length after every entry record.

Edit Length

An integer typein that specifies for how many frames the next image will be recorded.

Record

A boolean that enables frame recording.

Configuring

The **output ImageNode** module uses three parameters to configure itself. The module reads these parameters as CLI variables.

ImageNode_hostname

This is the Internet host name that **output ImageNode** uses to connect to the ImageNode. It may be either a name listed in the /etc/hosts file or a numeric Internet address of the form *nnn.nnn.nnn.nnn*. The default host name is "imagenode."

ImageNode_format

This may be either NTSC or PAL. NTSC is the default.

ImageNode_VCR_type

When **output ImageNode** is instantiated, it looks for this variable. If it exists, it is passed to the ImageNode followed by an "init" command. This can be used to tell the ImageNode what type of VCR is connected to it. The allowed values for this variable are shown in Table 1. Only "sony" and "simul" are supported by ConvexAVS.

If this variable is undefined, no initialization is done, which is correct for Sony VCRs.

You may set these variables by creating a CLI initialization file and referencing it in your .avsrc file. For example, if you wanted to use PAL as the default video format, you could put this into a file in your home directory called .avs_cli_init:

```
var_set ImageNode_format "PAL"
```

Then you would add this line to your .avsrc file:

```
CLIinit   /your home directory/.avs_cli_init
```

Table 1
ImageNode_VCR_type values

Value	VCR
abekas a60	Abekas A60
abekas a62	Abekas A62
abekas a64	Abekas A64
ampex	Ampex VPR series
lv-s100	Hitachi LV-S100
br-s810	JVC BR-S810
br-s811	JVC BR-S811
cr-850	JVC CR-850
kr-m800	JVC KR-M800
pr-900	JVC PR-900
tq-3031	Panasonic TQ-3031
simul	Software VCR simulation
lvr1	Sony LVR-5000/6000 at 1200 baud
lvr9	Sony LVR-5000/6000 at 9600 baud
sony	other Sony
teac9	TEAC LV-210A at 9600 baud

Related modules

Modules that could provide the **Image** input are prepare video and those that output an image.

Modules that can replace **output ImageNode** are output VideoCreator and display image.

See also

AVS Animator, prepare video, and output VideoCreator

Refer to *Animating AVS Data Visualizations* for detailed information about using this module.

Limitations

Only the ImageNode Professional model is supported.

output ImageNode

output VideoCreator

Write images to video tape using a Silicon Graphics VideoCreator

Summary

Name	output VideoCreator
Type	data output
Inputs	field 2D 4-vector uniform byte integer
Outputs	none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Insert Point	text	00:00:00		
Edit Length	integer	1	1	270000
Record	boolean	false		

Description

The output VideoCreator module writes images to video tape using a Silicon Graphics VideoCreator

The **output VideoCreator** module is used to record animation on video tape. The module displays each image it receives on the frame buffer of a Silicon Graphics VideoCreator.

Set the **Record** parameter to record the images on an attached VCR using single frame recording.

If the input image is smaller than the VideoCreator's frame buffer, it is centered and the remaining parts of the frame buffer are filled with black. If the input image is larger than the frame buffer, the image is cropped, and the center part is displayed. When the VideoCreator uses NTSC format, the frame buffer is 640 by 484 pixels. In PAL format, the frame buffer is 768 by 575.

The **output VideoCreator** module runs as a remote module on a Silicon Graphics Iris workstation, using the remote module mechanism. Refer to *Using ConvexAVS to Visualize Data* for a description of remote module execution.

output VideoCreator

Inputs

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

The input image must be a uniform 2D field, with a 4-vector of byte data values at each location in the field. The image can be any size. It will be cropped and matted to fit.

Frames/Second (optional; integer)

The integer input is unused.

Parameters

Insert Point

A text typein that may be used to specify the video tape location where the next image will be recorded. This may be specified either as *mm:ss:ff* where *mm*, *ss*, and *ff* are minutes, seconds, and frames or as an absolute frame number. The insert point is incremented by the edit length after every entry record.

Edit Length

An integer typein that specifies for how many frames the next image will be recorded.

Record

A boolean that enables frame recording.

Configuring

This module must be installed on a Silicon Graphics workstation that has the VideoCreator hardware attached. You can install the module for your own use, or a system administrator can install it in a public place for use by several people.

The **output VideoCreator** module executes as a remote module.

The host that remote modules runs on has a logical name, which is not necessarily the same as its real host name. The easiest way to use **output VideoCreator** is to refer to the video creator's host as *VideoCreatorHost*. That is the logical host name used in the module libraries distributed with ConvexAVS.

To install the module for public use:

1. Create the directory /usr/avs/avs_library on the SGI workstation.
2. Copy two files, list_dir and output_vcnt, from the /usr/avs/bin/SGI directory on your CONVEX supercomputer into /usr/avs/avs_library on the SGI. Use `rcp` or `ftp` binary mode to copy the files.

3. Make both files executable on the SGI using `chmod`.

4. Edit the /usr/avs/runtime/hosts file. Find the line describing VideoCreatorHost. Change that line to describe how to execute modules on your SGI workstation.

   For example, if your workstation's host name was *silicon* and you wanted to use `rsh`, you would use the following line:

```
VideoCreatorHost "/usr/ucb/rsh silicon -n" /usr/avs/avs_library /usr/avs/data
```

5. You will have to restart ConvexAVS before you can access the VideoCreator.

To install the module for individual use:

1. Create a new directory somewhere on the SGI workstation.

2. Copy two files, list_dir and output_vcnt, from the /usr/avs/bin/SGI directory on your CONVEX supercomputer into your new directory on the SGI. Use `rcp` or `ftp` binary mode to copy the files.

3. Make both files executable on the SGI using `chmod`.

4. Create a hosts file in your home directory. You can call this file whatever you want. Add a line to the hosts file describing how to execute modules on your SGI workstation.

   For example, if your workstation's host name was *silicon*, the directory you created in Step 1 was /usr/people/fred/vc, and you wanted to use `rsh`, you would use the following line:

```
VideoCreatorHost "/usr/ucb/rsh silicon -n" /usr/people/fred/vc /usr/people/fred
```

5. Edit your .avsrc file. If you do not have one, copy the /usr/avs/runtime/avsrc file into your home directory (and call it .avsrc). Add a line to your .avsrc file describing your hosts file.

   For example, if you called your hosts file /mnt/fred/avshosts, you would add the following line to your .avsrc file:

```
Hosts     /mnt/fred/avshosts
```

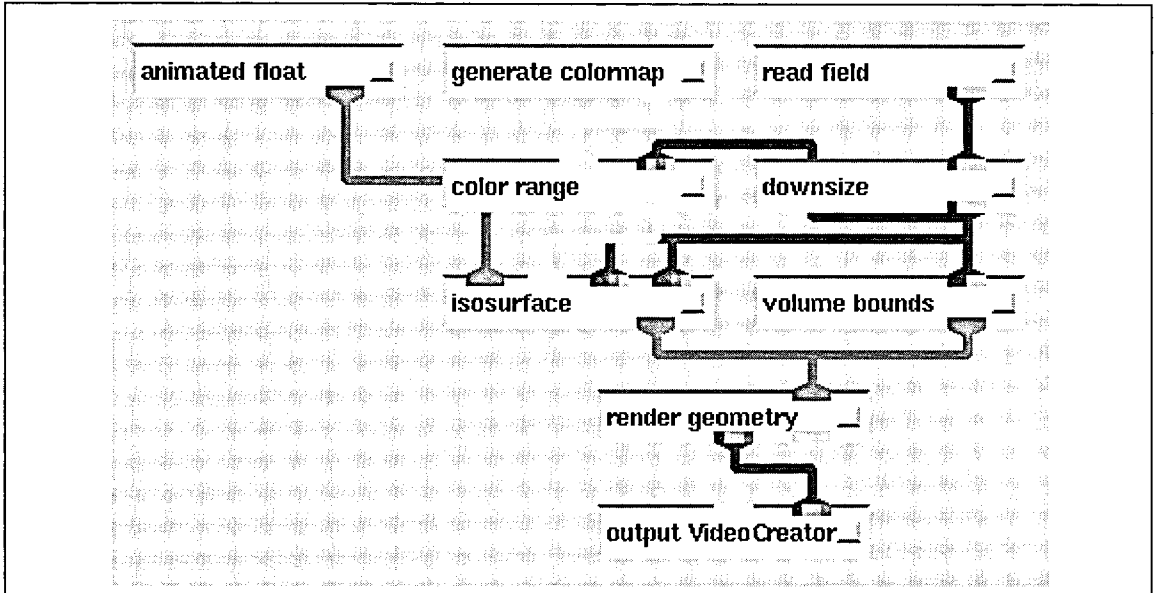
   Be sure to use the hosts file's absolute path name here.

6. You will have to restart ConvexAVS before you can access the VideoCreator.

Example

The network in Figure 84 shows **output VideoCreator**.

Figure 84
output VideoCreator module in an example network



Related modules

Modules that could provide the **Image** input are prepare video and those that output an image.

Module that can replace **output VideoCreator** is display image.

See also

AVS Animator, prepare video, output ImageNode

Refer to *Animating AVS Data Visualizations* for detailed information about using this module.

output postscript

Convert pixmap to PostScript and store in file

Summary

Name	output postscript	
Type	data output	
Inputs	pixmap	
Outputs	none	
Parameters	*Name*	*Type*
	filename	typein
	mode	radio
	monochrome	toggle
	8 bit	toggle
	compress	toggle
	dither	toggle

Description

The **output postscript** module converts its input pixmap to the PostScript page description language and stores it in a file.

Two types of PostScript output are supported:

- Suitable for sending to a PostScript-compatible laser printer
- Mathematica compatible

Inputs

Pixmap (required; pixmap)

Any pixmap can be input.

Parameters

filename

A typein that allows you to enter the name of the PostScript file to be created. After the file is written, the file name 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 file name.

mode

Selects the type of PostScript output: laserwriter or mathematica.

The following parameters apply to Mathematica options:

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.

compress

If **ON**, produces compressed output.

If **OFF**, produces uncompressed output.

dither

If **ON**, produces dithered output.

If **OFF**, produces undithered output.

Example

The example in Figure 85 converts an image to a PostScript file.

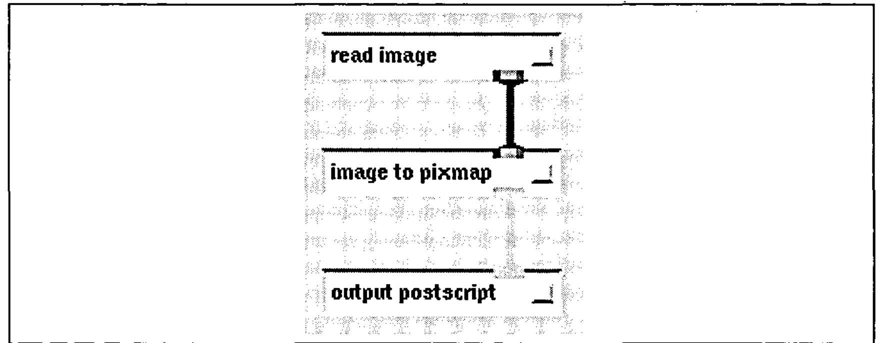


Figure 85
output postscript module in an example network

Related modules

image to pixmap, image to postscript, and render geometry

Limitations

This module does not work on a 16-plane system.

The **compress** option is not supported in any released version of Mathematica.

The dither option produces visual artifacts on some images.

See also

The example script OUTPUT POSTSCRIPT demonstrates the **output postscript** module.

paint mesh

Paint an image onto a surface in 3D space

Summary

Name	paint mesh
Type	mapper
Inputs	field 2D scalar field 2D uniform 4-vector byte
Outputs	geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Z scale	float	1.0	none	none

Description

paint mesh can be thought of as a field to mesh that takes an image instead of a color map for its second input. This allows a second field to be texture mapped onto a 2D mesh embedded in 3D physical space. This is useful for such things as plotting data on top of a topographical relief map of the surface of a planet.

The action of **paint mesh** can be broken into two cases: one in which the scalar field (input field 1) is a 2D 2-space field and the other where the scalar field is a 2D 3-space field (for example, an irregular field). In either case, the image field (input field 2) is treated the same. It is the scalar input field (field 1) that is treated differently in the two cases.

For 2D 2-space:

> The **paint mesh** module maps the 2D 2-space plane into 3-space by using the X- and Y- coordinates of the input field as the X- and Y-coordinates of a 3-space mesh and the value of the field at that point as the Z-coordinate of the mesh. It maps the X,Y-plane to a surface in X,Y,Z-space by using the values defined on the plane as the height of the surface above the plane. This is the usual way of graphing a two dimensional function. **paint mesh** will then texture map the image field onto the grid created from the scalar field.

For 2D 3-space:

The **paint mesh** module takes the input field and creates a mesh from it without using the value of the field. It is important to realize that the values of the field do not affect the geometry of the mesh. In **paint mesh,** the value of the scalar field is discarded. This is different from the operation of **field to mesh** because in this case, **field to mesh** will paint the 3-space mesh with the value of the scalar field at each vertex. **paint mesh** instead uses the value of the image field to paint the surface (mesh).

Inputs

Data Field (required; field 2D scalar)

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.

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

This input is the image to be texture mapped onto the mesh.

Outputs

Geometry (geometry)

The output is a geometry.

Parameters

Z scale

With uniform input fields, it determines the height of the mesh. With rectilinear and irregular input fields, this parameter is unused.

Example

The example in Figure 86 shows **paint mesh** in a network.

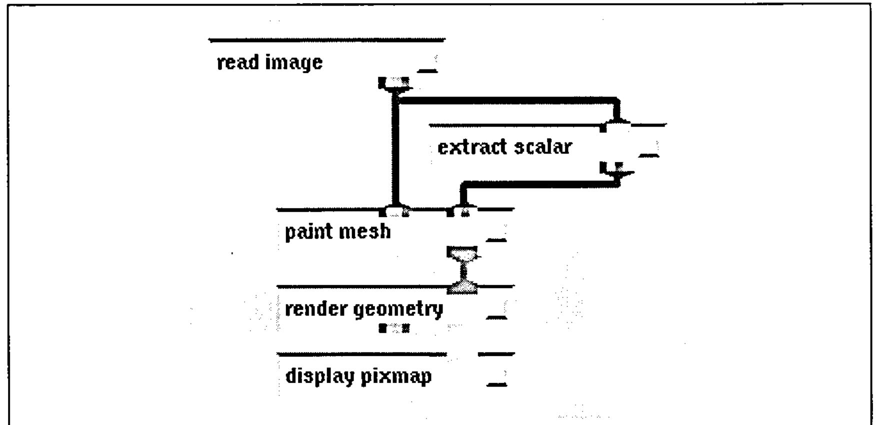


Figure 86
paint mesh module in an example network

Limitations

This module can output meshes that are too big for the **render geometry** module to handle, causing ConvexAVS to crash. Use the **downsize** module to reduce the size of the input data.

particle advector

Release grid of particles into velocity field

Summary

Name particle advector

Type mapper

Inputs field 3D 3-vector float *any-coordinates*
field irregular 3-space
upstream transform
integer

Outputs particles geometry
tracers geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Mesh Res	integer	5	2	100
Tracer Length	integer	0	0	100
Time Step	float	0.2	0.0	1.0
Size	float	0.0	0.0	1.0
Advect Batch	oneshot			
Stop Advection	toggle	off		
Replay Advection	oneshot			
Reset Particles	oneshot			
Color	toggle	off		
Show Bounds	toggle	on		
Surface	toggle	off		
Advection Method	radio	Euler		
Tracer Style	choice	cap		

Description

The **particle advector** module takes as input a 3D 3-vector field of floats (for example, fluid flow simulation data) and treats it as a velocity field. A batch of zero mass particles (the sample) 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.

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, 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, Show Bounds**, and **Surface** parameters 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.

By using the **Stop Advection** parameter, 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.

Inputs

Field Input (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 floats. The input field can be uniform, rectilinear, or irregular.

Scatter Input (optional; 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. **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 **render geometry** 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, **render geometry** 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. 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; integer)

Used in connection with the **AVS Animator**'s frame number output port. **particle advector** only updates the particles when there is a change to this input.

Parameters

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 that controls the length of the tracer output which shows the trajectory of each advected particle. The default is 0; higher numbers produce longer tracers. If the optional **Scatter Input** port is used, this parameter is ignored.

Time Step

Adjusts a scalar that multiplies the magnitude of the vector along which each particle is travelling. Increasing this parameter causes successive positions of particles to be more widely spaced.

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.

Color

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

Show Bounds

Controls the visibility of the mesh of particles.

Surface

Creates a solid shaded mesh. The coloring scheme is the same as that used with the **Color** parameter.

Advection Method

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 more accurate results.

Tracer Style

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 iterations of the flow. These lines appear to be tails attached to the advected particles.
add	Continuous lines that show the entire trajectories of the particles.

Outputs

Particles Geometry (geometry)

This output is a geometry that represents the batch of particles advected into the input vector field.

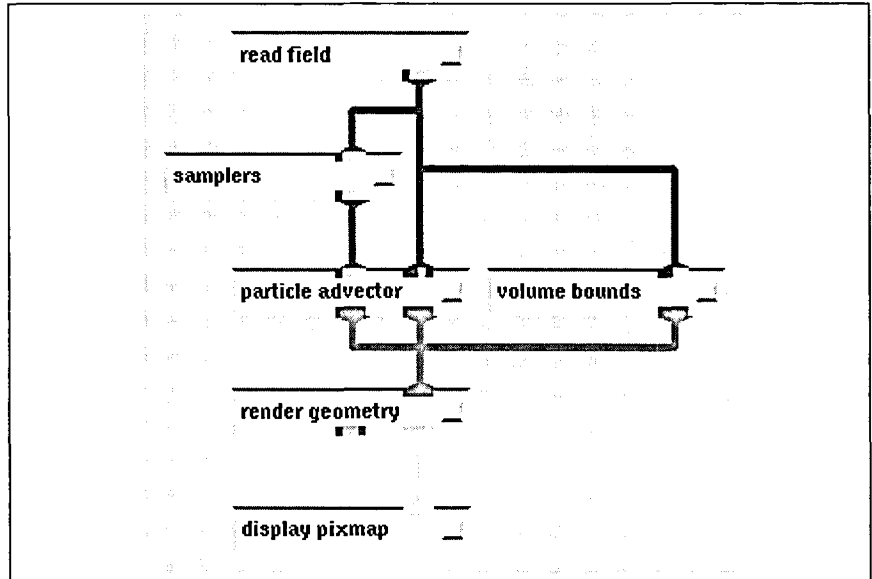
Tracers Geometry (geometry)

This output is a set of tracer lines (analogous to stream lines) produced by the sample particles. The **Tracer Style** and **Tracer Length** parameters control the form that these lines take.

Example

Figure 87 shows the **particle advector** module in an example network.

Figure 87
particle advector module in an example network



Related modules

Modules that could provide vector operations are vector curl, vector div, vector grad, vector mag, and vector norm.

Modules that could provide additional geometries are volume bounds, arbitrary slicer, and isosurface.

Modules that can render geometries are render manager, render geometry, and display pixmap.

See also

The example script PARTICLE ADVECTOR demonstrates the **particle advector** module.

pdb to geom

Create molecule geometry from PDB file

Summary

Name	pdb to geom		
Type	data input		
Inputs	none		
Outputs	geometry		
Parameters	*Name*	*Type*	*Choices*
	filename	browser	
	Representation	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 Protein Data Bank (PDB) data format. Such files have a .pdb file name suffix for Polygen or .ent for Brookhaven. The output is a geometry description of the molecule.

Parameters

Data File

A file browser allows you to specify the name of the file containing the molecule description.

Representation

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

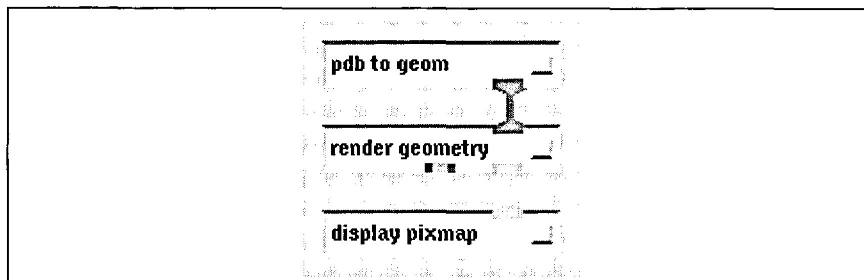
Molecule (geometry)

A geometry description of the molecule.

Example

The example in Figure 88 shows a simple application of **pdb to geom.**

Figure 88
pdb to geom module in an example network



Related modules

render geometry

Limitations

If you read in the same PDB file twice, you will get only one instance of the geometry.

pdb to geom does not read all available information contained in a PDB file, such as connectivity and motif. Consequently, **pdb to geom** determines connectivity (bonding) based on distances between atoms.

See also

The example script PDB TO GEOM demonstrates the **pdb to geom** module.

pdb viewer

View and manipulate information in Brookhaven PDB format

Summary

Name	pdb viewer
Type	data input
Inputs	none
Outputs	geometry
Parameters	none

Description

show control panel

Display the control panel for the PDB viewer.

File Browser

The file browser is set to a directory containing PDB data files. This directory may be set in your .avsrc file as follows.

`PdbDataDirectory` *lyour/directory/path*

The default is the /usr/avs/data/pdb data directory.

Clicking on the data file name causes it to be loaded into the **pdb viewer.** The **File Browser** button turns the file browser on and off. Its default is on.

If a data file is compressed using the `compress` command, it will be decompressed, loaded into the **pdb viewer**, and recompressed. You must have write permission in the directory the file is in for the decompression to occur.

Delete Active File

Cause the file that is currently active to be removed from the active select list. The previous file in the list is made active and visible. Ten files may be available at one time.

Active Selection

When a file is loaded, it becomes active and visible. The previous active file becomes invisible. The buttons on the **pdb viewer** panel control attributes of the active file. To change visibilities of files that are not active, use the Geometry Viewer. To rebuild an active molecule's geometry, click on its icon in the **Active Selection** menu.

Information

The textual data associated with a PDB file. The menu names and their corresponding PDB records follow:

Header - HEADER, COMPND, SOURCE, AUTHOR

Revision - OBSLTE, REVDAT, SPRSDE

Journal - JRNL

Remarks - REMARK

Footnote - FTNOTE

Nonstandard Group - HET, FORMUL

Symmetry - CRYST1, ORIGX, SCALE, MTRIX, TVECT

Statistics - Display counts of structural elements.

Data File - The entire data file in raw format is displayed.

Sequence

Display the residue sequence.

Atom

Select the manner in which atoms are displayed.

Atom Size

Atomic radius constants come from College Chemistry by Holtzclaw, Robinson, and Nebergall (1984).

Ball and Stick	Display atoms at 1/4 of their van der Waals radius. This is to produce a ball and stick representation.
Ionic	Display atoms according to their ionic radius.
Covalent	Display atoms according to their covalent radius.
Occupancy	Display atoms according to the occupancy field in the data file. Atom records with no occupancy field default to the value of one angstrom.
van der Waals	Display atoms according to their van der Waals radius. This is also known as CPK and is used to see a space filled representation.

Atom Color

The **Backbone Atoms** and **Side Chain Atoms** selections correspond to their respective atom colors with the exception of carbon whose buttons are white.

Residue Type	Color the atom according to which residue it belongs to and that residue's current color.
Atom Type	Color the atom according its type. Carbon is grey, oxygen is red, nitrogen is blue, sulphur is yellow, hydrogen is white, deuterium is light green, and phosphorus is light blue.
Residue Atom	The same as coloring by atom type, but with the carbons colored by their residue type.

Backbone Atoms

Control the visibility of atom types belonging to the backbone. In the case of oxygen, the visibility of the bond to the backbone is also toggled.

Side Chain Atoms

Control the visibility of atom types belonging to side chains.

Residue

Select the manner in which residues are displayed.

Residue Color

The **Display Amino Acid** selections correspond to each residue's respective color according to the currently selected color mode.

Unique	Each residue type has a unique color.
Hydro-icity	Color according to the residue hydrophobicity or hydrophilicity.

Display Amino Acid

Select which residues to display by their type.

The **ALL** button toggles the visibility of all of the residues. A useful technique is to turn all of the residues off, then turn on individual ones.

Affect Backbone

This determines whether residue backbone atoms and bonds are affected by changes in the display parameters. By default, the buttons are off and thus residue backbones are not affected. This allows entire backbones to be displayed with only selected side chains.

Affect Side Chain

This determines whether residue side chain atoms and bonds are affected by changes in the display parameters. By default, the buttons are on, so residue side chains are affected. This allows you to toggle the visibility of side chains of specific residue types.

Motif

Select display of the residues belonging to structural motifs defined in the data. The motif visibilities default to on when a new molecule is loaded.

Residues belonging to **Helix, Sheet**, and **Turn** motifs are mutually exclusive.

Sulphur Bridge and **Site** may share residues between themselves and among the previous three motifs.

Other displays the set of all residues not belonging to any other motif.

All toggles the visibility of all of the motifs. A useful technique is to turn this off and then turn on the individual motifs you want to view.

Affect Motif

This determines whether motif atoms and bonds are affected by changes in the motif display parameters. By default, the **Atoms** button is on and the **Bonds** button is off. This allows complete structures to be viewed with bonds and has the atoms of unselected motifs cut away.

Chain

Select the manner of displaying chains.

Backbone

Atoms	Toggle the visibility of all of the atoms in the backbone.
Bonds	Toggle the visibility of all of the bonds in the backbone.

Side Chain

Atoms	Toggle the visibility of all side chain atoms.
Bonds	Toggle the visibility of all side chain bonds.

Outputs

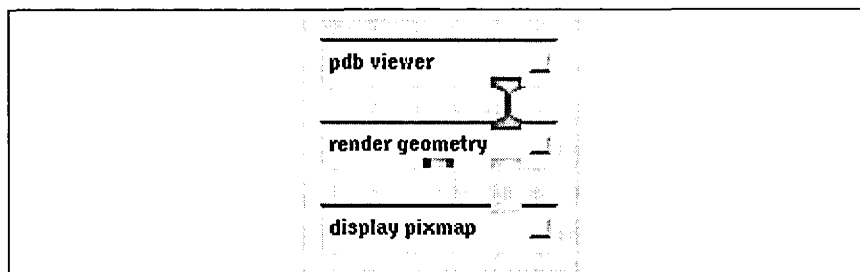
Geometry (geometry)

Geometry data that can be rendered by **geometry viewer.**

Example

The example in Figure 89 shows a simple application of **pdb viewer.**

Figure 89
pdb viewer module in an example network



Related modules

render geometry and display pixmap

Limitations

When switching between molecules with different display parameters, the display buttons get out of sync. To rebuild the active molecule according to the current display parameters, click on its icon in the **Active Selection** menu.

Nucleic Acid sequences are not fully supported. Bonds are not currently drawn nor are the residue and atom types available from the menus.

Some data files only contain side chain information. If you load a PDB file and do not see any geometry, select the **Side Chain** options under the **Chain** menu.

Data file records must all be the same length and contain valid PDB tokens. This may be a problem with some Amber and Charmm generated PDB files. If a file does not properly load, strip all records that are not ATOM or HETATOM, verify that all lines are of the same length, and try again.

pixmap to image

Transform pixmap to image

Summary

Name	pixmap to image
Type	mapper
Inputs	pixmap
Outputs	field 2D 4-vector byte
Parameters	none

Description

The **pixmap to image** module takes a pixmap as input and outputs an image. The pixmap is an X Window System resource used to store image data in the X server. This reduces the amount of data ConvexAVS 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
opacity value		color value	

The high-order byte field (auxiliary) is generally unused but sometimes contains alpha (opacity) information on a per-pixel basis.

Inputs

Pixmap (required; pixmap)

The input is any 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 image format.

Example

This module is useful for converting the output of data output modules into images for writing to a file. Figure 90 shows one use.

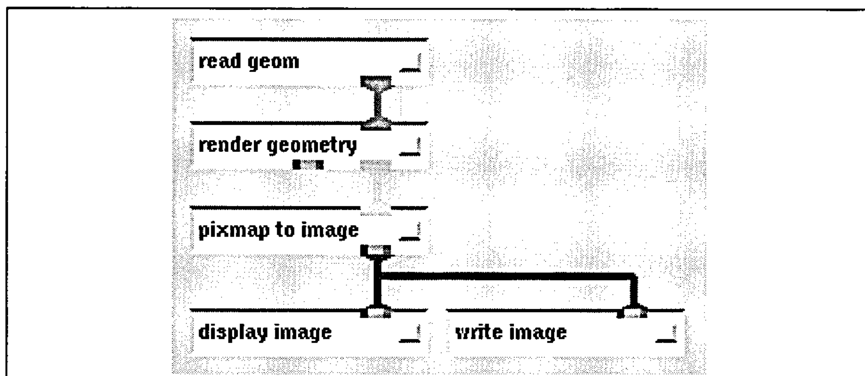


Figure 90
pixmap to image module in an example network

Related modules

Modules that can add to the image processing are contrast, threshold, histogram stretch, clamp, interpolate, colorizer, and generate colormap.

Module that generates pixmaps is render geometry.

Module that displays an image is display image.

Module that manipulates pixmaps and displays them is display pixmap.

Limitations

If your X display is less than 24 bits, the quality of the image produced by **pixmap to image** will be less than desired because *N* bits of data are being expanded to 24.

pixmap to image will only work with **render geometry** when the renderer is using X and not PEX or GL. In the PEX and GL cases, the pixmap is not available for conversion to an image.

See also

The example script BACKGROUND demonstrates the **pixmap to image** module.

prepare video

Perform interlacing, low-pass filtering, and gamma correction for video output

Summary

Name	prepare video				
Type	filter				
Inputs	field 2D 4-vector uniform byte				
Outputs	field 2D 4-vector uniform byte				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	interlace	boolean	false		
	low-pass filter	boolean	true		
	gamma correct	boolean	true		
	gamma	float	2.222	0	10

Description

The **prepare video** module prepares an image for output to a video device. It provides low pass filtering, field interlacing, and gamma correction.

NTSC and PAL broadcast video standards, are low resolution compared to computer graphics. Both use interlaced video, and both have insufficient bandwidth to permit abrupt horizontal color changes. A *low pass filter* helps to resolve both problems. A low pass filter smears images horizontally to eliminate abrupt transitions, and it smears vertically so features do not disappear between the scan lines when the picture is interlaced.

Gamma correction

The electron guns inside CRTs have nonlinear emissions with respect to their driving voltage. To compensate, **prepare video** multiplies each color component by a gamma correction curve. The equation for this curve is expressed as:

$$C = U^{1/\Upsilon}$$

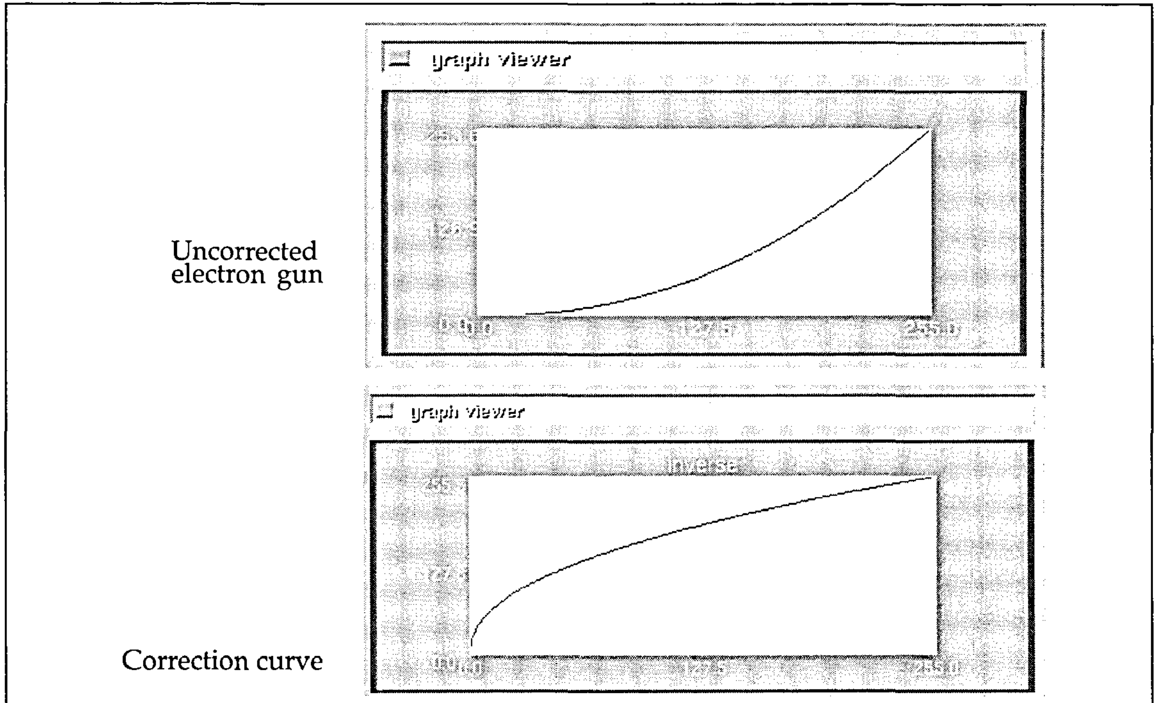
U is uncorrected intensity, and *C* is the corrected intensity.

For the NTSC standard, gamma is defined as 2 2/9 ($2.\overline{22}$....)

If your video system has gamma correction elsewhere in the system, you do not need to use the gamma correction described here.

The graphs in Figure 91 show an uncorrected electron gun and the NTSC gamma correction curve.

Figure 91
Gamma correction curves



The **prepare video** module implements low pass filtering, gamma correction, and interlacing for recording to videotape.

Interlacing

If the **interlace** button is set, two consecutive fields input to the **prepare video** module are merged together by interlacing the lines. The even scan lines of the output come from the first image, and the odd scan lines come from the second.

When the **prepare video interlace** button is set, it only produces output every other time it is called. This doubles the apparent frame rate from 30 to 60 when outputting animation to an NTSC video device, or from 25 to 50 for PAL video.

When interlacing, you have to take care to get the correct pairs of images interlaced together. One way to do that is to toggle the **interlace** button off, then on again. That will make **prepare video** output the current image, then the next two images in are interlaced together.

If two successive input images are different sizes, they are not interlaced. Instead, the first image is discarded, and the second is saved to interlace with the next image.

Inputs

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

The image to be prepared for output to a video device.

Parameters

interlace

If **interlace** is set, two consecutive fields input to **prepare video** are merged together by interlacing the lines. The even scan lines of the output come from the first image, and the odd scan lines come from the second.

low-pass filter

If **low-pass filter** is set, each input image is filtered using a filter optimized to reduce flicker in interlaced video. The filter uses an aperture four pixels high by two pixels wide, as described in [Amanatides 1990].

gamma correct

If **gamma correct** is set, the image is gamma corrected to the value specified by the **gamma** parameter.

gamma

This is the gamma correction value.

Outputs

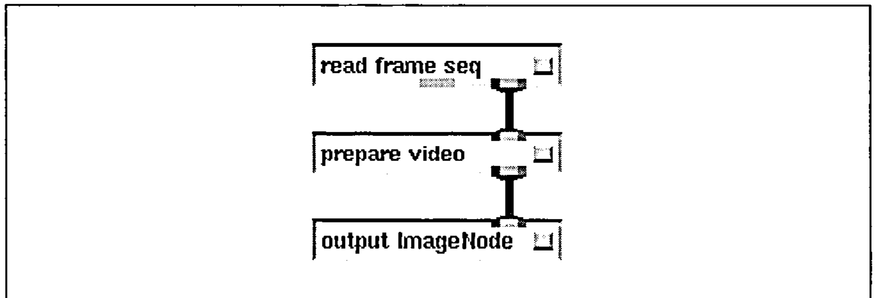
Image (field 2D 4-vector byte uniform)

The processed image ready for output to video.

Example

The example in Figure 92 shows a usage of the **prepare video** module.

Figure 92
prepare video module in an example network



Related modules

AVS Animator, read frame seq, write frame seq, output VideoCreator, and output ImageNode

Limitations

It is difficult to predict which pairs of input images are interlaced together.

See also

Amanatides and Mitchell, Antialiasing of Interlaced Video Animation, Computer Graphics, Volume 24, Number 4, pp. 77-85, August 1990.

Refer to *Animating AVS Data Visualizations* for detailed information about using this module.

print field

Create an ASCII printable/readable version of a field

Summary

Name print field

Type data output

Inputs field *any-dimension n*-vector *any-data any-coordinates*

Outputs none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Min X	typein	0	0	1000
Max X	typein	-1	-1	1000
Min Y	typein	0	0	1000
Max Y	typein	-1	-1	1000
Min Z	typein	0	0	1000
Max Z	typein	-1	-1	1000
Min W	typein	0	0	1000
Max W	typein	-1	-1	1000
Max Elements	integer	1	1	1000
Display Header	switch	on		
Display Data	switch	on		
Output File	typein	/tmp/pfield...		

Description

The **print field** module creates a human-readable version of the contents of a field. The information takes two forms: it is displayed in an Output Browser widget on the control panel, and it is written to a online file. **print field** is useful whenever you need to inspect the actual contents of a field.

By default, **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 and minimum or maximum data values associated with the field.

If the **Display Data** parameter 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 1000) 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 Z 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.

Inputs

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

The input field can be 1D, 2D, 3D, or 4D.

Parameters

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.

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 1000 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.

Display Header

A toggle switch that controls whether **print field** displays and writes the field's header information (dimensionality, type, etc.)

Display Data

A toggle switch that controls whether **print field** displays and writes the field's data and coordinate information.

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.

Example

The network in Figure 93 converts some data into a field, displays the contents of the new field, and gives you the option of writing the new field permanently to disk.

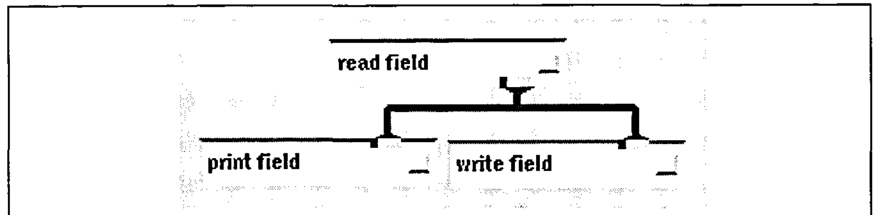


Figure 93
print field module in an example network

Related modules

compare field and write field

Limitations

print field writes to /tmp by default. This can cause problems if:

- There is no /tmp mounted on your system.
- The /tmp directory does not have much room in it or has inaccessible protections.

See also

The example scripts PRINT FIELD and FIELD MATH demonstrate the print field module.

probe

Interactively show numeric data values in a geometry rendered field

Summary

Name	probe
Type	mapper
Inputs	field 3D *n*-vector *any-data any-coordinates* colormap field irregular 0-vector upstream transform upstream geometry
Outputs	geometry upstream transform

Parameters

Name	*Type*	*Default*
sample_type	choice	Point
type	choice	Cursor
Pick Geometry	boolean	off

Description

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 a geometry. It works for uniform, rectilinear, and irregular coordinates and 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). 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.

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 is 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, rectilinear, or irregular, of any data type.

Colormap (optional; colormap)

If a 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. For example, if it is pointing at a low value with the default colormap from **generate colormap,** the probe object will be blue; it is pointing at a high value, it will be red.

Sample Field (optional; field irregular 0-vector)

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 only the first location and display its data value.

Upstream Transform (optional; invisible, autoconnect)

When the **probe** and **render geometry** 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, **render geometry** 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-and-click technique, this normally invisible port is what the **render geometry** 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

sample_type

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.

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

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. 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-and-click technique. Select probe as the current object in the Geometry Viewer, point at the object surface you want to sample with the 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.

Outputs

Geometry (geometry)

The output geometry has two parts:

- The rendering of the probe object.
- The rendering of the Text for Probe that lists the data value and coordinate position.

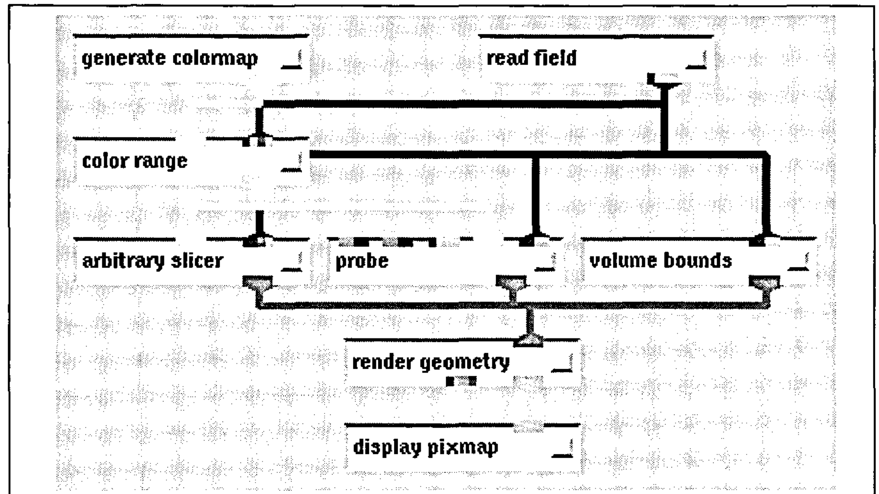
Upstream Transform (optional; invisible, autoconnect)

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

The network in Figure 94 inputs a curvilinear scalar field, scales the color values to the actual data range, and displays it through **arbitrary slicer** with a colorized probe object, surrounded by volume bounds.

Figure 94
probe module in an example network



Related modules

Modules that could provide the **Data Field** input are read volume, read field, and read plot3d.

Modules that could provide the **colormap** input are generate colormap and color range.

Module that could provide the **Sample Field** input is samplers.

Modules that can process **probe** output are render geometry and render manager.

See also

The example script PROBE demonstrates the **probe** module.

read PLOT3D

Read PLOT3D files

Summary

Name	read PLOT3D	
Type	data input	
Inputs	none	
Outputs	field 3D 5-vector irregular 3-space float field 3D scalar uniform integer field 1D scalar uniform float	
Parameters	*Name*	*Type*
	Read Plot3D Data Browser (q)	browser
	Read Plot3D XYZ Browser (x)	browser

Description

The **read PLOT3D** module reads a pair of PLOT3D files and builds three fields from it. The first field is a ConvexAVS floating point 5-vector irregular 3D 3-space field used to store the solution and mesh data itself. The second field is a 3D uniform scalar integer field used to store blanking records. The third field is a 4-element scalar real 1D uniform field used to store the four grid parameters (free stream Mach number, angle-of-attack (alpha), Reynolds number, and time from the solutions file). This last data is not used in all PLOT3D calculations but is made available for vector and scalar function routines.

This module is the most basic module in any PLOT3D network. It provides data for the system and other PLOT3D modules.

Outputs

Data Field (field 3D 5-vector irregular 3-space float)

This field combines the mesh and solution data into a single field.

Blanking Data Field (field 3D scalar uniform integer)

This field contains the blanking records for the plot3d data set in the form of a 3D scalar uniform integer field. This data can be used as an optional input for the **scalar PLOT3D** and **vector PLOT3D** modules.

Global Parameters (field 1D scalar uniform float)

This field contains the four grid parameters (free stream Mach number, angle-of-attack (alpha), Reynolds number, and the time) that are present in the solution data as a separate record.

Parameters

Read Plot3D Data Browser (q)

Selects the file name of the data file. The data file, or solutions file as it is sometimes called, contains the values of the five plot3d solution elements (density, x momentum, y momentum, z momentum, and stagnation energy) for each point at which they are defined. It also contains the four grid parameters.

Read Plot3D XYZ Browser (x)

Selects the file name of the mesh file. This file contains the location in physical space of the points at which the solution data, Q(1-5), is available. It also contains the blanking records if they are present.

Example

The example in Figure 95 reads in a PLOT3D data set and does an isosurface on the density field. Only the PLOT3D field itself is used. We do not use the blanking records because we are extracting a part of the raw PLOT3D field.

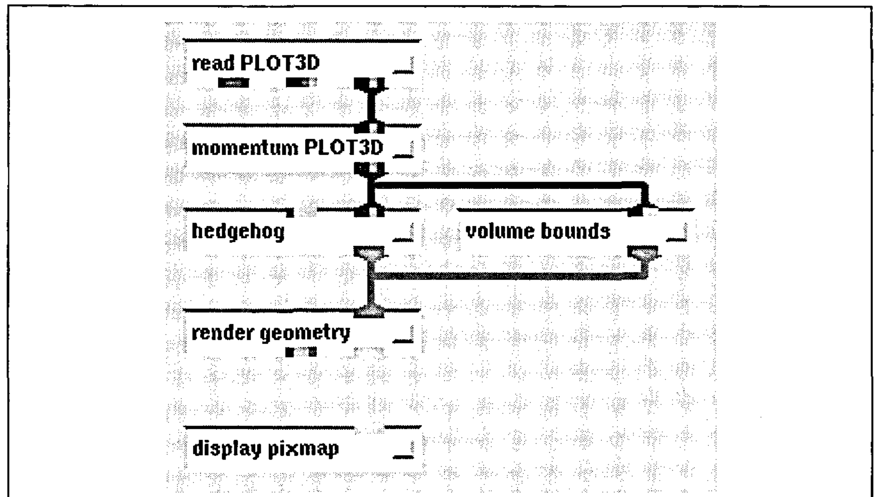


Figure 95
read PLOT3D module in an example network

Related modules

density PLOT3D, momentum PLOT3D, stagnation PLOT3D, vector PLOT3D, and scalar PLOT3D

Related programs

export_PLOT3D and import_PLOT3D

Limitations

The ConvexAVS **read PLOT3D** module only reads Convex FORTRAN unformatted PLOT3D records. This allows **read PLOT3D** to avoid overhead associated with ascii conversion at run time. It also allows the software to determine the file characteristics without user intervention. This is possible because Convex FORTRAN brackets the records it writes with record-byte-lengths when doing unformatted I/O. This record length information combined with the structure of a PLOT3D file allows the plt3dlib software to analyze the file to determine the existence of blanking records and whether the file is whole or plane. Unlike most PLOT3D software, you need not specify these characteristics to **read PLOT3D**.

read field

Read field from a file or import data files into field format

Summary

Name	read field	
Type	data input	
Inputs	none	
Outputs	field *same-dimension same*-vector *same-data same-coordinates*	
Parameters	*Name*	*Type*
	Read File Browser	browser
	Read Status	text block

Description

The **read field** module has two input modes:

- In its first input mode, it reads a field data structure from a file into a network.
- In its second input mode, it converts data stored in ASCII, FORTRAN unformatted, or pure binary data files into a field format.

Outputs

Data Field (field *same-dimension same*-vector *same-data same-coordinates*)

The output data is a field.

Parameters

Read File Browser

A file browser window to specify the name of the file to be read.

Read Status

A text block window to specify the status of the file to be read.

Native field input

read field can read files in the native field file format into an network. A 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, Figure 96 defines a field of type "field 2D 4-vector byte," which is the image format.

Figure 96
ASCII header description file for an image

```
# AVS field file
ndim=2 # number of computational dimensions
dim1=512
dim2=480
nspace=2 # number of physical dimensions
veclen=4
data=byte
field=uniform
```

Separator characters

The ASCII header must be followed by two form feed characters (that is, **CTRL-L**, octal 14, decimal 12, hex 0C) in order to separate it from the binary area. This scheme allows you to use the `more` command to examine just the header information without the binary data garbling your 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 (`nspace`) 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, maximum z.

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 `ndim`	(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` ... + `dim`*x*) * 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` ... * `dim`*x*) * `nspace` * 4

  As with rectilinear field, 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.

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 field format. To import data into ConvexAVS, you must create an ASCII description file that defines the structure of the field to make. The first part of this description file is identical in format and meaning to the ASCII header file. 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 follows this general scheme:

> `skip` *n* lines or bytes
> move over `offset` *m* columns on this line (ASCII only)
> read the value
> do until # *of values needed*
> {
> take *p* `stride`(s) to the next value
> }

The ASCII description file, data, and coordinate information for rectilinear and irregular data can all be read from different files. If the resulting 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 capability is meant to be used only once, in order to convert data into field format. The parsing activity makes **read field** run more slowly than when it reads a file that is already in 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 in a file.

ASCII description file

The ASCII description file contains a series of text lines that can be categorized as one of the following:

- A comment
- A required line in the form *token=value*
- An optional line in the form *token=value*
- A variable or coordinate parsing specification

The ASCII description file in Figure 97 imports 3D irregular data with a vector of values at each point into a field of type "field 3D 3-vector irregular float." This type of data occurs in computational fluid dynamics applications. The data and coordinate information are in separate files, both of which were written as 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 XYZ-coordinate file, all the X coordinate values appear, then all the Y's, then all the Z's.

Figure 97
ASCII description file for irregular data

```
# 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
```

The first five characters in the ASCII description file must be "`# AVS`" or **read field** will not recognize the file as valid.

Figure 97 shows all of the required *token=value* 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 field that **read field** produces as output. Optional *token=value* pairs are stored in the output field as well, if they are provided. `min_ext` and `max_ext` are stored in the output field even if they are not specified because **read field** calculates them. The `variable` and `coord` lines are not stored in the output field. They are only instructions to **read field**.

With the exception of file names, ASCII description file specifications are not case sensitive. You can surround the equal character (=) with any amount of white space (including none at all).

Value strings do not have to be padded out to 11 characters.

Token=Value pairs

`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 example, 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`).

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.

`veclen` = *value* (required)

The number of data values for each field element. All the data values must be of the same primitive type so that the collection of values is conceptually a `veclen`-dimensional vector. If `veclen` = 1, the single data value is a scalar. Thus, the term "scalar field" is often used to describe such a field.

`data` = *data type* (required)

The primitive data type of all the data values. *data type* must be byte, integer, float, or double.

`field` = *field type* (required)

The *field type* must be one of the following:

uniform	Has no computational-to-physical space mapping. The field implicitly takes its mapping from the organization of the computational array of field elements.
rectilinear	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.
irregular	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 field data structure. This value can be used by modules downstream, for example, to size the volume bounds drawn around the data in the Geometry Viewer or put minimum and maximum values on coordinate parameter manipulator dials. Values can be separated by blanks and/or commas.

If you do not know the extents, let **read field** calculate them. Most downstream modules use whatever values are supplied without checking their validity. If you specify the wrong numbers, 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 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." You can either use one label line or

separate label lines. 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 or separate unit lines. 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 field data structure. These are 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, leave these optional lines out. In this case, the **write field** module will compute these values when it creates a field file. Most downstream modules use whatever values are supplied without checking their validity. If you specify the wrong numbers, 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.

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.

The individual parameters are interpreted as follows:

file = *filespec*

The name of the file containing the data or coordinates. If only a file name is given with no directory information, the directory defaults to /usr/avs/data, the value given by the `DataDirectory` environment variable, or to the directory specified with the `-data` option on the command line (with increasing precedence). It does not default to the file browser widget's current directory.

filetype = *file type*

The *file type* can be one of the following:

ascii	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.
binary	The file is written in binary format.
unformatted	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, ignore size words in your calculation.

In each case, **read field** will use the data type specified in the `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 marked 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 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 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. `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 data. (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.

`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 * the size of the data type.* In a file where the data or coordinate values are sequential, the `stride` would be 1. 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 has been read.

The following tables show some examples for ASCII data. "As" are vector component 1 and "Bs" are vector component 2. Table 5 will not work unless the data labels and equal signs are removed.

Table 2
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`

Table 3
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`

Table 4
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`

Table 5
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

Examples

1.

The ASCII header file in Figure 98 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.

Figure 98
ASCII header description file for a volume

```
# 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 bytes

The coordinates area occupies (2 * 4) * 3 bytes. The total binary area occupies 262,168 bytes.

2.

The ASCII header file in Figure 99 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.

Figure 99
ASCII header description file for a volume

```
# 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=float # 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:

(27 * 25 * 32) * 4 * 3 = 259,200 bytes

The coordinates area occupies (2 * 4) * 3 bytes. The total binary area occupies 259,224 bytes.

3.

The ASCII header file in Figure 100 describes an irregular volume (3D irregular field) with one single-precision value for each field element. The binary area includes an XYZ-coordinate triple for each field element, indicating the corresponding point in physical space. This format might be used to represent fluid flow data.

Figure 100
ASCII header description file for an irregular volume

```
# 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

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 grain silo structure at a particular spot in its base. The data is in one ASCII file, decay.dat, as a long, sequentially-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. To read this data, you must use a text editor to globally edit out the commas and semicolons, changing them to spaces. The ASCII description file in Figure 101, decay.fld, would import the data into field format.

Figure 101
ASCII description file decay.fld

```
# 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=/name/Termites/where.coord filetype=ascii offset=3 stride=7
coord 2 file=/name/Termites/where.coord filetype=ascii offset=4 stride=7
coord 3 file=/name/Termites/where.coord filetype=ascii offset=5 stride=7
variable 1 file=/name/Termites/decay.dat filetype=ascii offset=1 stride=2
```

5.

The ASCII description file in Figure 102 specifies how to convert the volume data in the file /usr/avs/data/volume/hydrogen.dat into a 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` defaults to 1.

Figure 102
ASCII description file converting hydrogen.dat into a field

```
# 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
```

6.

The ASCII description file in Figure 103 specifies how to use **read field** to convert the image data in /usr/avs/data/image/mandrill.x into a 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 succession of 32-bit binary words, one word per pixel. In ConvexAVS, 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. The entire file is treated as a series of binary bytes.

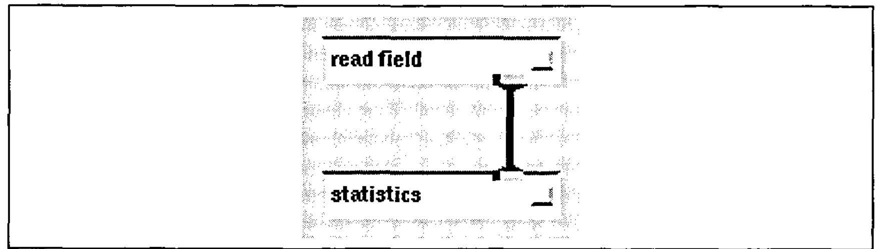
Figure 103
ASCII description file converting mandrill.x into a field

```
# 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=1 stride=4
variable 4 file=/usr/avs/data/image/mandrill.x filetype=binary skip=11 stride=4
```

7.

Figure 104 shows a simple network using **read field**.

Figure 104
read field module in an example network



Related modules

The **write field** module will take the field produced by **read field** and write it to a file as a permanent 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 a 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, and FIELD MATH demonstrate the **read field** module.

read frame seq

Read a frame sequence from a file

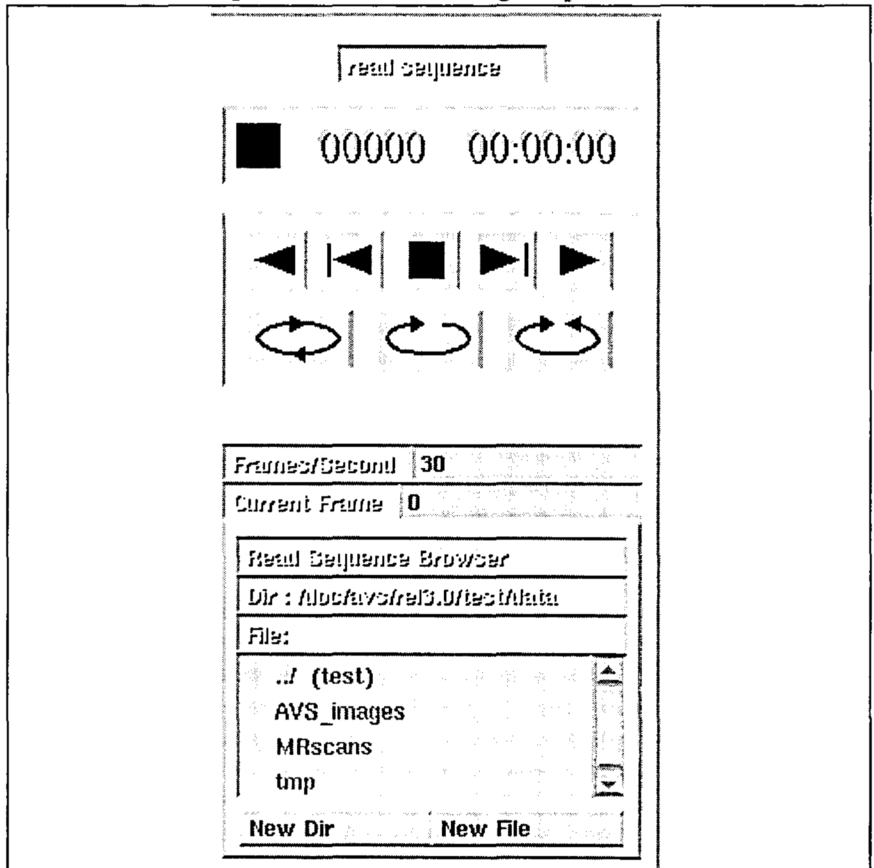
Summary

Name	read frame seq		
Type	data input		
Inputs	none		
Outputs	field 2D 4-vector byte integer		
Parameters	*Name*	*Type*	*Default*
	Frames/Second	integer	30
	Current Frame	integer	0
	Read sequence browser	browser	

Description

The **read frame seq** module reads an image sequence file.

Figure 105
read frame seq control panel



The playback controls act in the same way as the **AVS Animator** playback controls. The control panel has a row of status indicators and two rows of push-buttons.

The top row status indicators display the current playback direction, the current frame number, and the current frame expressed in minutes, seconds, and frames.

Playback controls

The following control buttons are available:

play reverse

This button causes **read frame seq** to read frames in descending order starting with the frame prior to the current frame. The sequence stops when the beginning of the current file is reached (frame zero). If the current frame is zero, the sequence starts with the last frame.

step reverse

This button generates the frame previous to the current frame. If the current frame is zero, the last frame is ignored.

stop

This button stops any sequence that is in progress.

step forward

This button generates the frame after the current frame. If the current frame is the last frame, frame zero is generated.

play forward

This button causes **read frame seq** to step forward towards the end of the script, one frame at a time, starting at the next frame after the current frame and stops when the last frame in file is reached. If the current frame is the last frame, the sequence starts at frame zero.

There are three cycle buttons:

play continuously

Run in a continuous loop, starting at the current frame, until the **Stop** button is pressed.

play once

Cycle once from the first frame to last frame.

bounce play

Cycle back and forth continuously.

Parameters

Frames/Second

This integer typein produces the *mm:ss:ff* (minutes, seconds, and frames) status display.

Current Frame

This integer typein is used to directly access any frame in the sequence. This parameter is automatically updated when the sequence is displayed.

Read Sequence Browser

A file browser allows you to select the sequence file that you wish to read. The file is automatically decompressed if previously compressed using the write sequence.

Outputs

Image (field 2D 4-vector byte integer)

The output is an image.

Example

The network in Figure 106 shows **read frame seq.**

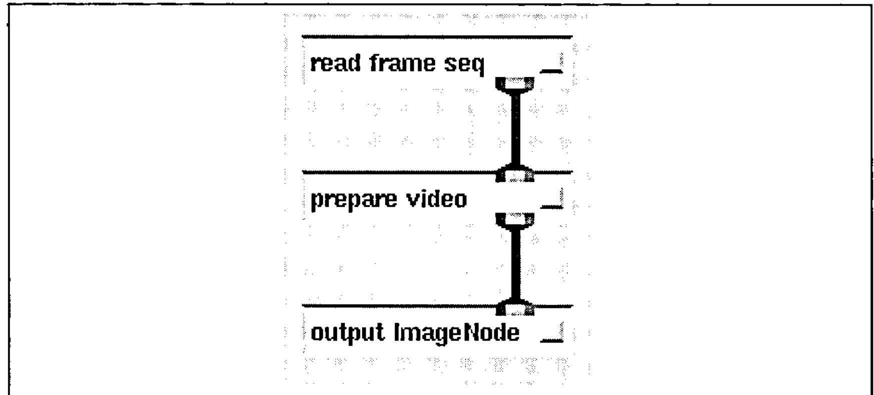


Figure 106
read frame seq module in an example network

Related modules

AVS Animator, write frame seq, output VideoCreator, and output ImageNode

See also

Refer to *Animating AVS Data Visualizations* for detailed information about using this module.

read gaussian

Extract structural and volume data from a Gaussian output file

Summary

Name	read gaussian		
Type	data input		
Inputs	none		
Outputs	field 3D real 3-space irregular geometry		
Parameters	*Name*	*Type*	*Choices*
	Density File	browser	
	Geom Representation	choice	stick, spoke, ball and stick, ball and spoke, Labels, CPK
	Title	toggle	
	Keywords	toggle	

Description

The **read gaussian** module reads the CubeDensity output from Gaussian and converts it into geometry and volume descriptions. CubeDensity files contain information about the molecular skeleton and one of the following combinations of information:

- Electron density or spin density
- Electron density and gradient
- Electron density, gradient, and divergence

The 3D field output by **read gaussian** contains all of the available density and density derivative information from the data file. To extract the individual values, use the **extract scalar** module. The scalar data will be present in the following channels of the **extract scalar** module:

Channel	Data
1	Density
2	X-gradient
3	Y-gradient
4	Z-gradient
5	Divergence

Parameters

Density File

A file browser allows you to specify the name of the file containing the Gaussian CubeDensity output.

Geom Representation

The type of geometry produced:

stick	Colored lines represent the bonds.
spoke	Colored cylinders represent the bonds.
ball and stick	Small spheres represent the atoms, and white lines represent the bonds.
ball and spoke	Small spheres represent the atoms, and white cylinders represent the bonds.
Labels	Atomic symbols followed by atom definition numbers represent the atoms, and white lines represent the bonds.
CPK	A space-filled representation with each large sphere representing the corresponding atom's van der Waals radius.

Title

Toggle the visibility of titles from the data file. The text is derived from the "title card" line of the Gaussian data file. The title is displayed as a geometry at the top of the screen.

Keywords

Toggle the visibility of keywords from the data file. The text is derived from the keywords supplied for the CubeDensity portion of the Gaussian input file. The keywords are displayed as a geometry at the bottom of the screen.

Outputs

Molecule Volume (field 3D real 3-space irregular)

A volume representation calculated by Gaussian.

Molecule Structure (geometry)

A description of the atoms and bonds in the molecule that includes title, keywords, and geometry.

Example

Figure 107 shows a network using **read gaussian**. The module on the right defines the shape of the surface, and the module on the left defines the colors on the surface. Typically, you look at the shape of densities and colors according to spin density or divergence.

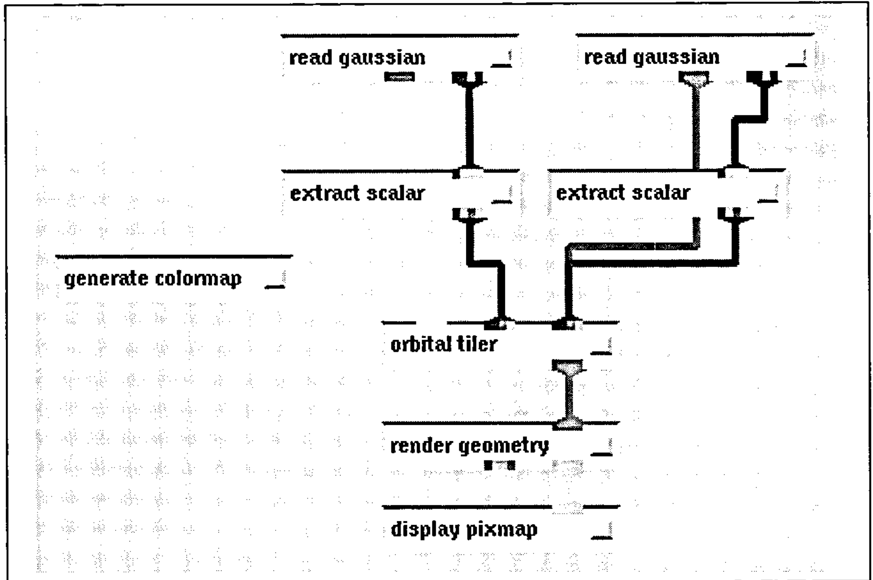


Figure 107
read gaussian module in an example network

Related modules

render geometry, isosurface, orbital tiler, extract scalar, volume bounds, orthogonal slicer, arbitrary slicer, field to mesh, colorizer, generate colormap, and color range

read gaussian CHK

Extract structural and volume data from a processed Gaussian checkpoint file

Summary

Name	read gaussian CHK		
Type	data input		
Inputs	none		
Outputs	field 3D real 3-space irregular geometry		
Parameters	*Name*	*Type*	*Choices*
	Checkpoint File	browser	
	Geom Representation	choice	stick, spoke, ball and stick, ball and spoke, Labels, CPK
	Title	toggle	
	Job info	toggle	
	MO	integer	
	Data Type	choice	Alpha MO, Beta MO, SCF Den, SCF Spin Den, MP2 Den, MP2 Spin Den, CI Den, CI Spin Den, MP Den, MP Spin Den, CC Den, CC Spin Den, Vib Mode
	Grid Spacing	real	
	Vib Mode	integer	
	Amplitude	real	

read gaussian CHK

Description

The **read gaussian CHK** module reads output from the Gaussian FormChk utility and converts it into structural and surface descriptions. The FormChk utility reads binary checkpoint files and outputs an equivalent ASCII file.

Your display choices depend upon the contents of the processed checkpoint file. The types of output that can be displayed are shown by the **Data Type** parameter.

Only the parameters that pertain to the current data set are visible. For example, if the data set contains no vibrational mode information, both the **Vib Mode** and **Amplitude** parameters are invisible.

Parameters

Density File

A file browser allows you to specify the name of the file containing the FormChk output.

Geom Representation

The type of geometry produced:

stick Colored lines represent the bonds.

spoke Colored cylinders represent the bonds.

ball and stick Small spheres represent the atoms, and white lines represent the bonds.

ball and spoke Small spheres represent the atoms, and white cylinders represent the bonds.

Labels Atomic symbols followed by atom definition numbers represent the atoms, and white lines represent the bonds.

CPK A space-filled representation with each large sphere representing the corresponding atom's van der Waals radius.

Title

Toggle the visibility of titles from the data file. The text is derived from the "title card" line of the Gaussian data file. The title is displayed as a geometry at the top of the screen.

Job Info

Toggle the visibility of information about the job. This information consists of the kind of job, the basis set, and the wave function. The job information is displayed as a geometry at the bottom of the screen.

Data Type

A list of possible operations that depends upon the wave function and job type of the Gaussian run:

Alpha MO A volume of expectation values ($e^{1/2}/bohr^{3/2}$) for a particular Molecular Orbital of alpha spin. The integer typein MO will determine which orbital is shown. Output will be generated for both output ports.

Beta MO A volume of expectation values ($e^{1/2}/bohr^{3/2}$) for a particular Molecular Orbital of beta spin. If the molecule is described by an RHF wave function, the alpha and beta molecular orbitals are described by the same function, so this option will not be shown. The integer typein MO will determine which orbital is shown. Output will be generated for both output ports.

SCF Den A volume of the electric charge density ($e/bohr^3$) for the SCF wave function. Output will be generated for both output ports.

SCF Spin Den A volume of the difference of the densities ($e/bohr^3$) of alpha and beta spin electrons for the SCF wave function. Output will be generated for both output ports.

MP2 Den A volume of the electric charge density ($e/bohr^3$) for the MP2 wave function. Output will be generated for both output ports.

MP2 Spin Den A volume of the difference of the densities ($e/bohr^3$) of alpha and beta spin electrons for the MP2 wave function. Output will be generated for both output ports.

CI Den A volume of the electric charge density ($e/bohr^3$) for the CI wave function. Output will be generated for both output ports.

CI Spin Den A volume of the difference of the densities ($e/bohr^3$) for the CI wave function of alpha and beta spin electrons. Output will be generated for both output ports.

MP Den A volume of the electric charge density ($e/bohr^3$) for the MP4 wave function. Output will be generated for both output ports.

MP Spin Den — A volume of the difference of the densities (e/bohr^3) of alpha and beta spin electrons for the MP4 wave function. Output will be generated for both output ports.

CC Den — A volume of the electric charge density (e/bohr^3) for the coupled cluster wave function. Output will be generated for both output ports.

CC Spin Den — A volume of the difference of the densities (e/bohr^3) of alpha and beta spin electrons for the coupled cluster wave function. Output will be generated for both output ports.

Vib Mode — The atomic structure is distorted along a normal mode of vibration. The normal mode of vibration is determined by the value of the **Vib Mode** parameter. The amplitude of the distortion is determined by the value of the **Amplitude** parameter. Output will be generated only for the geometry output port.

Grid Spacing

The spacing (bohr) between data points in the field 3D real 3-space irregular output.

Vib Mode

The number of the mass weighted vibrational mode to be displayed. The rotations and translations are the highest five or six vibrational modes. The other vibrations are ordered from lowest to highest energy.

Amplitude

The amplitude of the vibrational displacements.

Outputs

Molecule Volume (field 3D real 3-space irregular)

A volume representation calculated by **read gaussian CHK**.

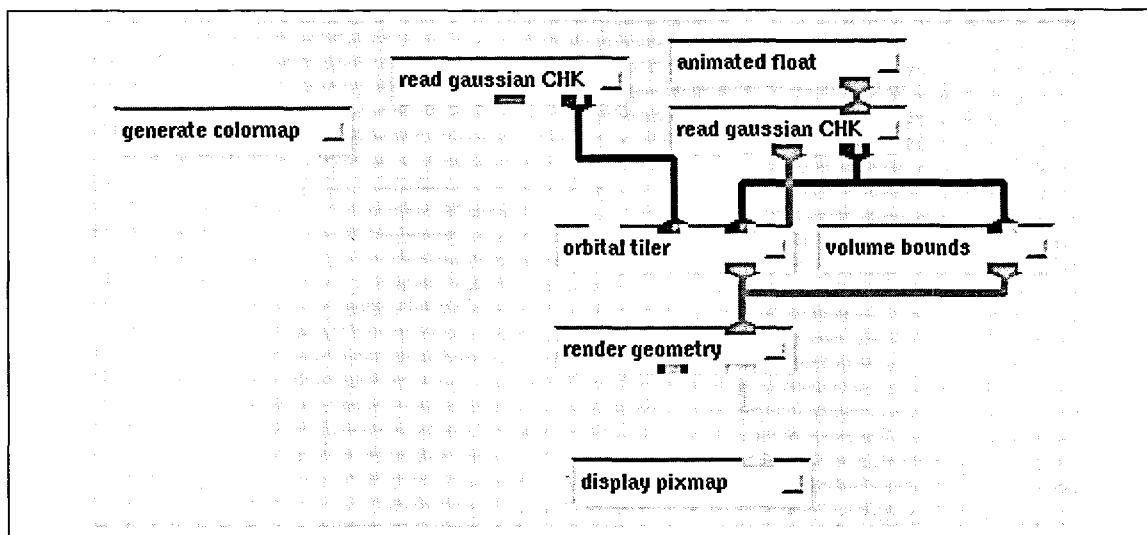
Molecule Structure (geometry)

A title, job information, and geometric description of the atoms and bonds in the molecule.

Example

The network in Figure 108 shows a simple application of **read gaussian CHK**. The module on the right can be used to display vibrational modes, molecular orbitals, or densities; it defines a shape of the surface. The **read gaussian CHK** module on the left defines the color of the surface.

Figure 108
read gaussian CHK module in an example network



Related modules

render geometry, isosurface, orbital tiler, orthogonal slicer, arbitrary slicer, field to mesh, color range, generate colormap, colorizer, and volume bounds

read geom

Reads a data file containing a geometry

Summary

Name	read geom	
Type	data input	
Inputs	none	
Outputs	geometry	
Parameters	*Name*	*Type*
	filename	browser

Description

The **read geom** module reads a file containing a 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.

Because the Geometry Viewer subsystem (also accessible as the **render geometry** 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 (for example, **shrink**).

Parameters

filename

A file browser allows you to choose the name of the file that contains a geometry.

Outputs

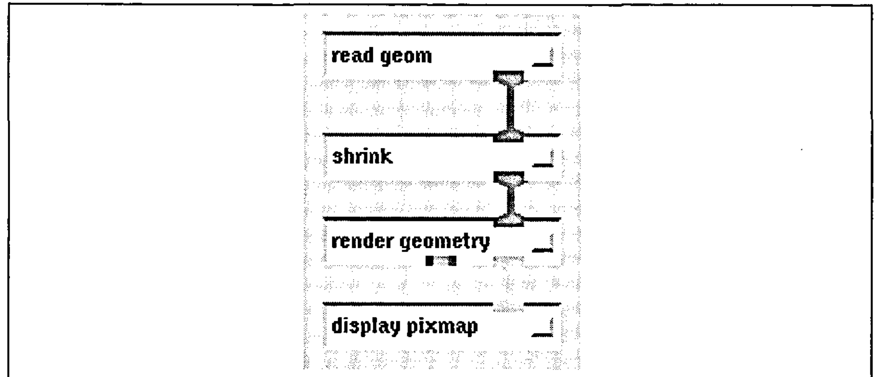
Geometry (geometry)

The output is the geometry that was read from the specified file.

Example

The network in Figure 109 shows **read geom.**

Figure 109
read geom module in an example network



Related modules

shrink, offset, render geometry, wireframe, and tube

Limitations

This module reads .geom files only. It cannot read .obj script files or .scene files that can be created with the Geometry Viewer Script Language.

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.

read geom throws away all hierarchical information in a .geom file and combines all objects into one. This prevents geometries from containing multiple properties for distinct subobjects.

See also

The example scripts CONTRAST, OFFSET, and PROBE demonstrate the **read geom** module.

read hdf field

Read a scientific data set from an HDF file into a field

Summary

Name read hdf field

Type data input

Inputs none

Outputs field *same-dimension* scalar float uniform
string

Parameters

Name	*Type*
Read HDF Field Browser	browser
Field	dial
Prev Field	oneshot
Next Field	oneshot

Description

The **read hdf field** module reads a Scientific Data Set (SDS) from an HDF file into a field. The Hierarchical Data Format (HDF) from the National Center for Supercomputing Applications (NCSA) facilitates the transfer of scientific data and images between computers.

Multiple data sets may reside in a single HDF file, and this module allows you to read all the data sets in sequence. When you first select a file, read hdf field reads the first data set from the file, and provides it on its output port. If there is more than one data set in the file, the Field, Prev Field, and Next Field parameters appear.

This module was developed using NCSA HDF Version 3.1.

Outputs

Data Field (field *same-dimension* scalar float uniform)

The output data is a field.

Label (string)

The current data set label (if set) is available as a string.

Parameters

Read HDF Field Browser

A file browser window to specify the name of the file to be read.

Field

An integer dial to select a data set by number.

Prev Field

A oneshot button to select the previous data set.

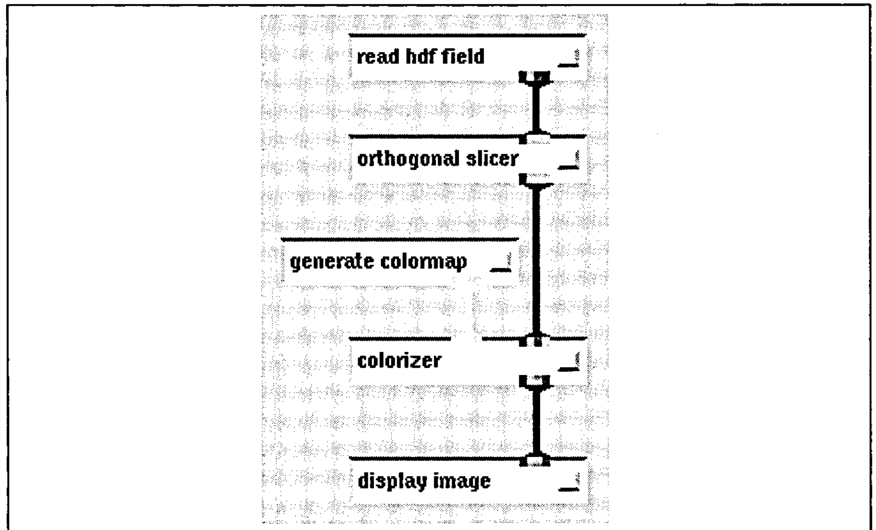
Next Field

A oneshot button to select the next data set.

Example

The network in Figure 110 shows **read hdf field**.

Figure 110
read hdf field module in an example network



Related modules

The **write hdf field** module will take the field produced by **read hdf field** and write it to disk as a scientific data set in an HDF file.

The **print field** module displays the ASCII header and contents of a field interactively on the screen. Connect it to **read hdf field**'s output port to verify that the data is being read correctly.

Error checking

read hdf 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 along with the type of error.

Limitations

It is possible to corrupt an HDF file if it is opened for writing by more than one module at a time.

See also

read field and write field

Acknowledgments

Portions developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.

read hdf image

Read 8-bit or 24-bit raster images from an HDF file into an image

Summary

Name	read hdf image	
Type	data input	
Inputs	none	
Outputs	field 2D 4-vector byte	
Parameters	*Name*	*Type*
	Read HDF Image Browser	browser
	Image	dial
	Use Palette	boolean
	Prev Image	oneshot
	Next Image	oneshot

Description

The **read hdf image** module reads 8-bit or 24-bit raster images from an HDF file and outputs each image as "field 2D 4-vector byte." The Hierarchical Data Format (HDF) from the National Center for Supercomputing Applications (NCSA) facilitates the transfer of scientific data and images between computers.

Multiple images may reside in a single HDF file, and this module allows you to read all the images in random order or in sequence. When you select a file, **read hdf image** reads the first image from the file and provides it on its output port. If there is more than one raster image in the file, you select a desired image by using the **Image**, **Prev Image**, or **Next Image** parameter.

HDF supports two kinds of raster images: 8-bit and 24-bit. The 8-bit images may have an associated palette (or colormap) while the 24-bit images do not. When the images are stored in the field structure, the auxiliary byte (alpha) is set to zero.

This module was developed using NCSA HDF Version 3.1.

Image layout

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
opacity value		color value	

The auxiliary field (alpha) is sometimes used to store opacity information on a per-pixel basis.

Parameters

Read HDF Image Browser

Allows you to specify the name of the image file to be read.

Image

Select a raster image by number.

Use Palette

Signifies whether to use the provided palette with 8-bit images.

Prev Image

Select the previous raster image.

Next Image

Select the next raster image.

Outputs

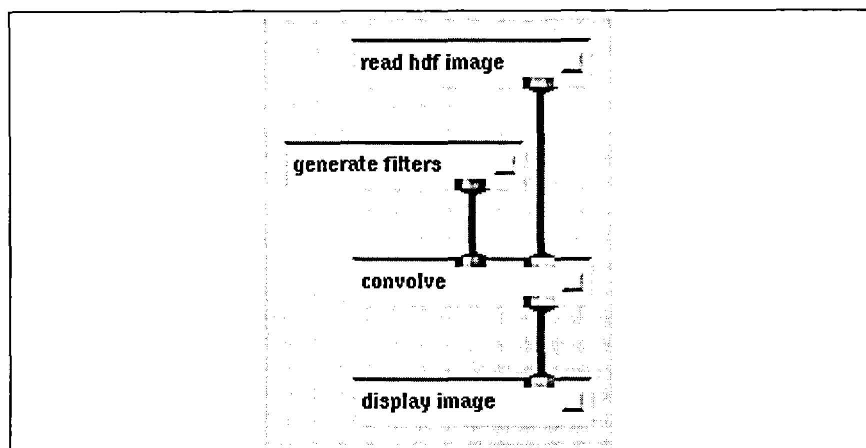
Data Field (field 2D 4-vector byte)

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 image format.

Example

The network in Figure 111 shows **read hdf image**.

Figure 111
read hdf image module in an example network



Related modules

The **write hdf image** module will take the image produced by **read hdf image** and write it to disk as a raster image in an HDF file.

Limitations

It is possible to corrupt an HDF file if it is opened for writing by more than one module at a time.

See also

read image and write image

Acknowledgments

Portions developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.

read image

Read file into an image

Summary

Name	read image	
Type	data input	
Inputs	none	
Outputs	field 2D 4-vector byte	
Parameters	*Name*	*Type*
	Read Image Browser	browser

Description

The **read image** module reads an image file from disk and outputs the image as "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
opacity value		color value	

The auxiliary field (alpha) is sometimes used to store opacity information on a per-pixel basis.

Parameters

Read Image Browser

A file browser window that allows you to specify the name of the image file to be read.

Outputs

Data Field (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 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)
```

Example

The network in Figure 112 shows **read image**.

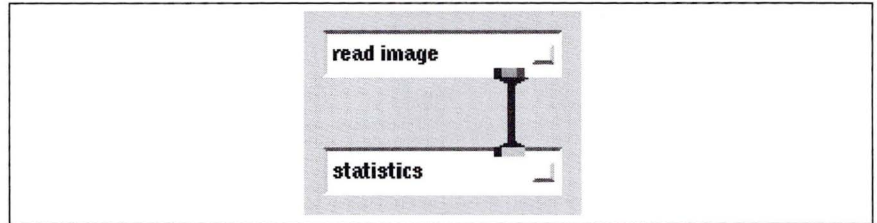


Figure 112
read image module in an example network

Related modules

Modules that can process images are contrast, threshold, histogram stretch, clamp, interpolate, luminence, generate filters, sobel, convolve, and local area ops.

Modules that decompose or compose images from separate bands are extract scalar and combine scalars.

Module that can display an image is display image.

See also

The example script CONTRAST demonstrates the **read image** module.

read mopac

Extract structural and volume data from a MOPAC 6.0 output file

Summary

Name	read mopac		
Type	data input		
Inputs	none		
Outputs	field 1D real 3-space irregular field 3D real 3-space irregular geometry		
Parameters	*Name*	*Type*	*Choices*
	MOPAC AVS File	browser	
	Geom Representation	choice	stick, spoke, ball and stick, ball and spoke, Labels, CPK
	Title	toggle	
	Keywords	toggle	
	MO	integer	
	Data Type	choice	Alpha MO, Beta MO, Density, Spin Den, Fesp Vol, Fesp Con, Fesp Will, Esp Con, Esp Will, Energy Surf, Vib Mode
	Grid Spacing	real	
	Vib Mode	integer	
	Amplitude	real	

Description

The **read mopac** module reads output from the CONVEX version of **MOPAC 6.0** and converts it into structural and surface descriptions. The surfaces may be generated by **MOPAC 6.0** or by **read mopac**.

The options available depend upon whether surface data or a wave function was written by **MOPAC 6.0** to the output file.

If surface data was generated with **MOPAC 6.0**, then the **MO, Data Type**, and **Grid Spacing** parameters will be set to reflect the contents of the data file. The **Data Type** parameter will be set to the type of surface contained in the input file.

If a wave function was written by **MOPAC 6.0**, then **read mopac** will generate the surfaces according to the wave function. Your display choices depend upon the type of wave function in the data file.

Only the parameters that pertain to the current data set are visible. For example, if the data set contains no vibrational mode information, both the **Vib Mode** and **Amplitude** parameters are invisible.

If you do not have the CONVEX version of **MOPAC 6.0**, contact your local CONVEX office.

Parameters

MOPAC AVS File

A file browser allows you to specify the name of the file containing the **MOPAC 6.0** output.

Geom Representation

The type of geometry produced:

stick	Colored lines represent the bonds.
spoke	Colored cylinders represent the bonds.
ball and stick	Small spheres represent the atoms, and white lines represent the bonds.
ball and spoke	Small spheres represent the atoms, and white cylinders represent the bonds.
Labels	Atomic symbols followed by atom definition numbers represent the atoms, and white lines represent the bonds.
CPK	A space-filled representation with each large sphere representing the corresponding atom's van der Waals radius.

Title

Toggle the visibility of titles from the data file. The text is derived from the two lines of "title cards" from the **MOPAC 6.0** data file. The title is displayed as a geometry at the top of the screen.

Keywords

Toggle the visibility of keywords from the data file. The text is derived from the keywords supplied for the **MOPAC 6.0** job. The keywords are displayed as a geometry at the bottom of the screen.

MO

Determines which Molecular Orbital is shown.

Data Type

A list of possible operations that depend upon the wave function supplied to **read mopac:**

Alpha MO	A volume of expectation values ($e^{1/2}/bohr^{3/2}$) for a particular Molecular Orbital of alpha spin. The integer typein MO will determine which orbital is shown. Output will be generated for the geometry and molecule volume output ports.
Beta MO	A volume of expectation values ($e^{1/2}/bohr^{3/2}$) for a particular Molecular Orbital of beta spin. If the molecule is described by an RHF wave function, the alpha and beta molecular orbitals are described by the same function, so this option will not be shown. The integer typein MO will determine which orbital is shown. Output will be generated for the geometry and molecule volume output ports.
Density	A volume of the electric charge density ($e/bohr^3$). Output will be generated for the geometry and molecule volume output ports.
Spin Den	A volume of the difference of the densities ($e/bohr^3$) of alpha and beta spin electrons. Output will be generated for the geometry and molecule volume output ports.
Fesp Vol	A volume of electrostatic points ($e/bohr^3$) generated from point charges. The point charges are determined by the Mulliken Population. Output will be generated for the geometry and molecule volume output ports.
Fesp Con	A Connolly surface of electrostatic points (e/bohr) generated from point charges. The point charges are determined by the Mulliken Population. Output will be generated for the geometry and molecule surface output ports.
Fesp Will	A Williams surface of electrostatic points (e/bohr) generated from point charges. The point charges are determined by the Mulliken Population. Output will be generated for the geometry and molecule surface output ports.

Esp Con A Connolly surface of electrostatic points (e/bohr) generated from the one electron operator method of K. M. Merz, B. H. Besler, and P. A. Kollman. Output will be generated for the geometry and molecule surface output ports.

Esp Will A Williams surface of electrostatic points (e/bohr) generated from the one electron operator method of K. M. Merz, B. H. Besler, and P. A. Kollman. Output will be generated for the geometry and molecule surface output ports.

Energy Surf A 1- or 2-dimensional grid of energy points (Kcal). Output will be generated for the geometry and molecule volume output ports.

Vib Mode The atomic structure is distorted along a normal mode of vibration. The normal mode of vibration is determined by the value of the **Vib Mode** parameter. The amplitude of the distortion is determined by the value of the **Amplitude** parameter. Output will be generated only for the geometry output port.

Grid Spacing

The spacing (bohr) between data points. If the surface is generated by **read mopac**, this parameter may be adjusted to control its quality.

Vib Mode

The number of the mass weighted vibrational mode to be displayed. The rotations and translations are the highest five or six vibrational modes. The other vibrations are ordered from lowest to highest energy.

Amplitude

The amplitude of the vibrational displacements.

Outputs

Molecule Surface (field 1D real 3-space irregular)

A Connolly or Williams surface for the molecule.

Molecule Volume (field 3D real 3-space irregular)

A volume representation calculated by **MOPAC 6.0** or **read mopac**.

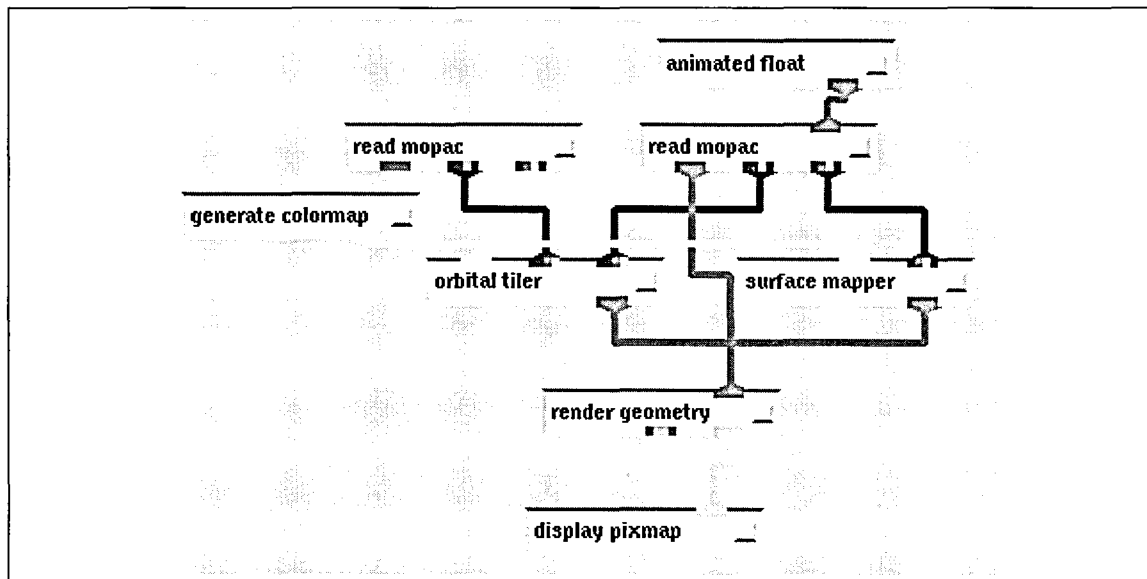
Molecule Structure (geometry)

A title, keywords, and geometric description of the atoms and bonds in the molecule.

Example

The network in Figure 113 shows a simple application of **read mopac**. The module on the right can be used to display vibrational modes, Connolly surfaces, Williams surfaces, density surfaces, molecular orbital surfaces, or electrostatic potential surfaces. The **read mopac** module on the left can be used to color the density, molecular orbital, or electrostatic potential surfaces.

Figure 113
read mopac module in an example network



Related modules

render geometry, isosurface, orbital tiler, surface mapper, orthogonal slicer, arbitrary slicer, field to mesh, color range, generate colormap, colorizer, and volume bounds

read plot3d

Read a PLOT3D format file into a field

Summary

Name read plot3d

Type data input

Inputs none

Outputs field irregular float

Parameters

Name	*Type*	*Default*
Multigrid	boolean	false
w/IBLANK	boolean	false
Data format	choice	binary
Q and X[YZ] organization	choice	3D/whole
Grid number	integer	1
Q (solution) File	browser	
X[YZ] File	browser	

Description

The **read plot3d** module reads computational fluid dynamics data files in the National Aeronautics and Space Administration's PLOT3D format and converts them into field format. There are two types of PLOT3D files:

- XYZ grid files that specify the irregular coordinate information.
- 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.
- Output data can have no more than 1,000,000,000 points in any one grid.
- 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.

This module is in the unsupported library.

Parameters

Multigrid

A toggle that specifies whether the file has a single grid or multiple grids.

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:

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.

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).

Q and X[YZ] 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 1D arrays of values.

2D

Input files are each a sequence of 2D arrays of values, stored in natural FORTRAN order.

3D/whole

Input files are each a sequence of 3D arrays of values, stored in natural FORTRAN order.

3D/planes

Input files are each a sequence of sets of 2D arrays of values, where each set of arrays corresponds to a single plane from the entire array.

Grid number

Which grid, in multigrid files, to use to produce the field.

Q (solution) File

A file browser widget for specifying the solution file.

X[YZ] File

A file browser widget for specifying the grid file.

Outputs

Data Field (field irregular float)

The field output will match the dimensionality of the original PLOT3D data set. At each point in the grid will be three to five floating point values: density, X-momentum (Y-momentum and Z-momentum, if appropriate), and stagnation, in that order. The output field represents only the one specified grid of multigrid parameter files. There is no way to pack multiple grids into an field.

Example

The network in Figure 114 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 **extract scalar** module on the left 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.

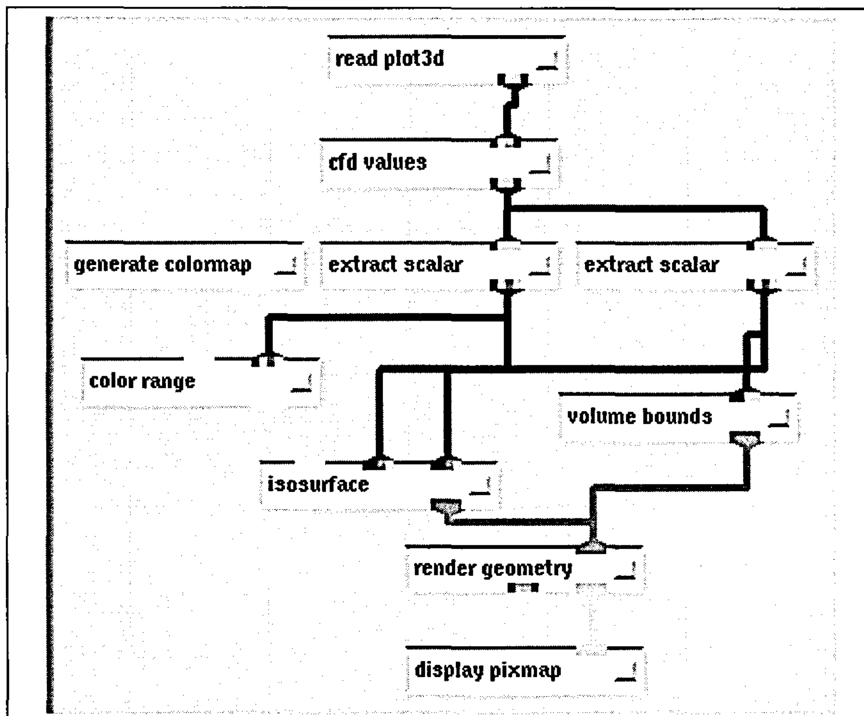


Figure 114
read plot3d module in an example network

Related modules

The **cfd values** modules is particularly designed to compute seven 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 are cfd values, extract scalar, extract vector, volume bounds, isosurface, and arbitrary slicer.

References

Pieter Buening, *PLOT3D Reference Manual.*

See also

The example scripts READ PLOT3D and CFD VALUES demonstrate the **read plot3D** module.

read rle image

Read image file in RLE format from disk into a field

Summary

Name	read rle image	
Type	data input	
Inputs	none	
Outputs	field 2D 4-vector byte	
Parameters	*Name*	*Type*
	Read RLE Image Browser	browser

Description

The **read rle image** module reads an image in the University of Utah's Run Length Encoded (RLE) image format from disk and outputs the image as "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:

0	RED	GREEN	BLUE
opacity value		color value	

Outputs

Data Field (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 image format.

Parameters

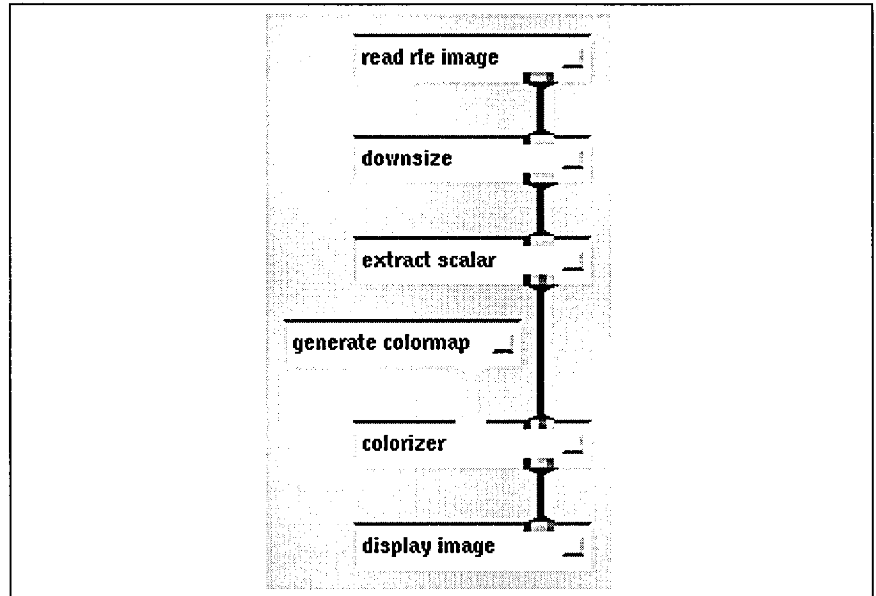
Read RLE Image Browser

A file browser window that allows you to specify the name of the RLE image file to be read.

Example

The network in Figure 115 shows **read rle image** in a network.

Figure 115
read rle image module in an example network



Related modules

clamp, combine scalars, contrast, display image, display pixmap, extract scalar, histogram stretch, image to pixmap, interpolate, read image, and threshold

See also

Information about the RLE file format can be found in the Utah Raster Toolkit available by anonymous FTP to cs.utah.edu (128.110.4.21) in the file pub/urt-3.0.tar.Z.

read ucd

Read UCD structure from a file

Summary

Name	read ucd	
Type	data input	
Inputs	none	
Outputs	ucd structure	
Parameters	*Name*	*Type*
	read file	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**.

Parameters

read file

A file browser window to specify the name of the UCD file to be read.

Outputs

UCD structure

The output structure is in unstructured cell data format.

ASCII file format

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.
3. For each cell: its cell ID, material, type (hexahedral, pyramid, etc.), and the list of node IDs that correspond to each of the cell's vertices.

4. For the data vector associated with nodes, how many components that vector is divided into (e.g., a vector of five floating-point numbers may be treated as three components: a scalar, a vector of three, and another scalar).
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. 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.

The input file cannot contain blank lines or lines with leading blanks. Refer to Figure 116 for the UCD file format.

The UCD structure and library will support either integer or character node, cell, and model IDs. However, the **read ucd** module only accepts integer node, cell, and model IDs.

At present, most of the provided 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>. Your modules can use the UCD library to manipulate cell- and model-based data.

Figure 117 is of a simple UCD file. This UCD structure has eight nodes in one hexahedral cell. Associated with each node is a single scalar data value, making up one component that is labeled "stress," and specifies a "lb/in**2" unit label. There is no cell- or model-based data.

Example

The network in Figure 118 reads in and displays a UCD ASCII file.

Related modules

Modules that can process **read ucd**'s output are 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 slice 2d, ucd legend, ucd probe, ucd streamline, write ucd, and ucd tracer.

See also

The example script READ UCD demonstrates the **read ucd** module.

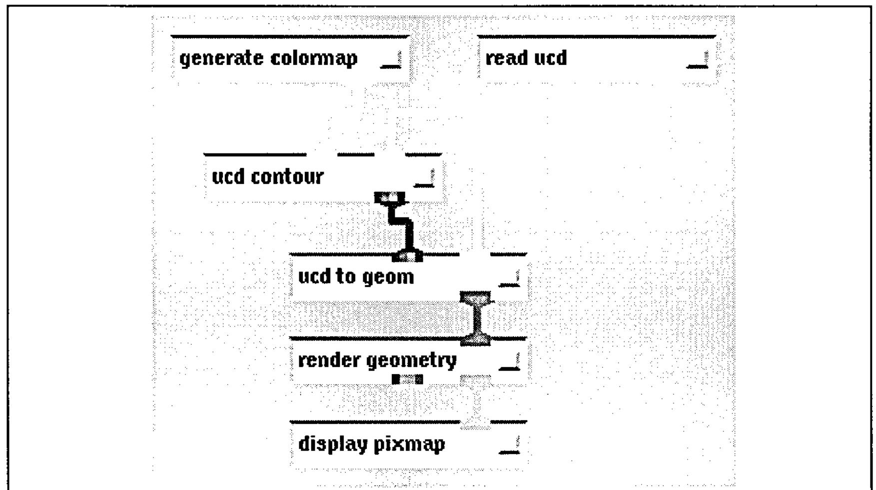
Figure 116
UCD file format

```
# <comment 1>
        .
        .
        .
# <comment n>
<num_nodes> <num_cells> <num_ndata> <num_cdata> <num_mdata>
<node_id 1> <x> <y> <z>
<node_id 2> <x> <y> <z>
        .
        .
        .
<node_id num_nodes> <x> <y> <z>
<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>
<num_comp for node data> <size comp 1> <size comp 2> ... <size comp n>
<node_comp_label 1> , <units_label 1>
<node_comp_label 2> , <units_label 2>
        .
        .
        .
<node_comp_label num_comp> , <units_label num_comp>
<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>
<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>
<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-is> <model-data 1> <model-data num_mdata>
```

Figure 117
UCD file example

```
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
```

Figure 118
read ucd module in an example network



read volume

Read volume file from disk into a field

Summary

Name	read volume	
Type	data input	
Inputs	none	
Outputs	field 3D scalar byte	
Parameters	*Name*	*Type*
	Read Volume Browser	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 (for example, CAT scan data or NMR data).

Parameters

Read Volume Browser

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

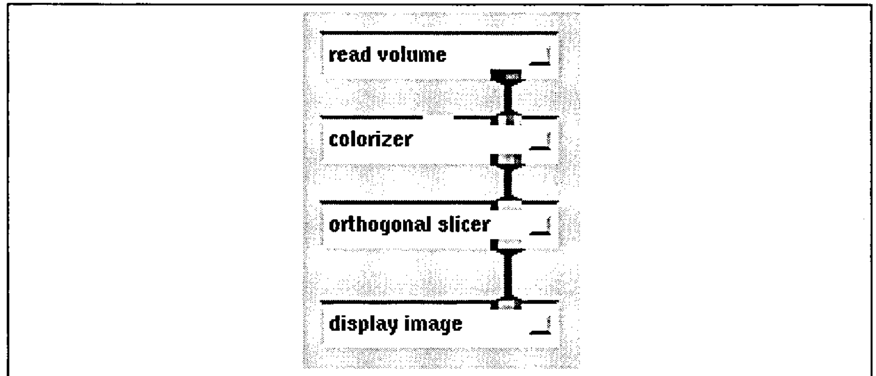
read volume expects its input file to be in the following format:

```
(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

The network in Figure 119 displays a volume data set.

Figure 119
read volume module in an example network



Related modules

Modules that can input colormaps are generate colormap and read colormap.

Modules that can filter data are 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, and gradient shade.

Modules that can map data are dot surface, arbitrary slicer, bubbleviz, orthogonal slicer, field to mesh, isosurface, and volume bounds.

Modules that can render data are display image and render geometry.

See also

The example scripts ANIMATED FLOAT and THRESHOLDED SLICER demonstrate the **read volume** module.

render geometry

Convert geometric description to image (Geometry Viewer)

Summary

Name	render geometry	
Type	data output	
Inputs	geometry field 4-vector byte	
Outputs	pixmap upstream transform upstream geometry field 2D 4-vector byte	
Parameters	*Name*	*Type*
	Output Image	boolean
	High Quality	boolean
	width	integer
	height	integer
	object transform	2D field uniform
	light transform	2D field uniform
	color	integer
	mode	string
	transparency	float
	object	string
	line radius	float
	point radius	float

Description

The **render geometry** module provides access within a 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.

Considerations

This module is special: in addition to having a few control widgets organized onto a single control panel page, it also uses the Geometry Viewer subsystem menu.

In some circumstances, it is useful 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 your window manager to move 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**, ConvexAVS 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; geometry)

The input data can be any geometry. More than one geometry can be input to this port. All the geometries will be combined into the same scene.

Texture (optional; field 4-vector byte)

This input port provides network compatibility with other ports that support texture mapping.

Parameters

Output Image

Enables the image output port. Only the first view of this scene (that is, camera 0) will be rendered, and the resulting image will be output.

High Quality

Enables the high-quality renderer on camera 0. The rendered image will output through the image port only if **Output Image** is enabled.

width

The width of the output image in pixels. Only used when **Output Image** is true.

height

The height of the output image in pixels. Only used when **Output Image** is true.

object transform

A field that represents a 4 by 4 transformation of an object to world coordinate space. The transform affects only the object specified by the **object** parameter.

light transform

A field that represents a 4 by 4 transformation of a light source. Only light source 1 is affected.

color

An integer that represents a packed color (0x00RRGGBB). The color affects only the object specified by the **object** parameter.

mode

Specifies the rendering mode of the object specified by the **object** parameter.

transparency

Specifies the transparency of the object specified by the **object** parameter.

object

Specifies which object will be transformed by the following parameters: **object transform**, **color**, **mode**, and **transparency**.

line radius

Half the thickness of lines that are rendered, in pixel units. Only used when **High Quality** is true.

point radius

The radius of points that are rendered, in pixel units. Only used when **High Quality** is true.

Outputs

Pixmap (pixmap)

The output is a pixmap containing a scene that includes all the input objects.

Transform (upstream transform)

This output informs upstream modules of the transformation applied to the input geometry. For example, **arbitrary slicer** uses this output to recalculate a new slice plane through a volume during your manipulations.

Geometry (upstream geometry)

This output informs upstream modules of the chosen vertex. For example, **probe** uses this output to determine where you have requested a data value.

Image (field 2D 4-vector byte)

The output is an image containing the first view of a scene. An image is output only when **Output Image** is enabled.

Example

The network in Figure 120 creates a tube version of an object.

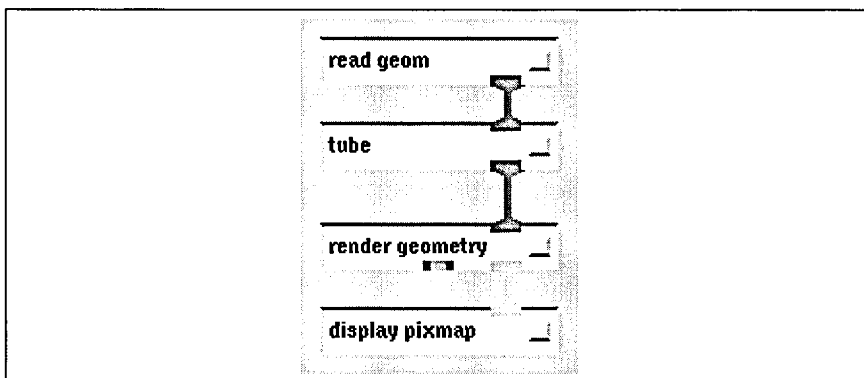


Figure 120
render geometry module in an example network

Related modules

display pixmap, read geom, pdb to geom, and render manager

Limitations

Labels do not appear in the image that is output from the image port. To generate visible labels, connect the pixmap output to the **pixmap to image** module.

If you are using ConvexAVS on a renderer that uses GL or PEX, the pixmap output should be connected only to **display pixmap**. Connecting it to other modules with pixmap inputs can cause undesirable behavior.

See also

The example scripts FLIP NORMALS and PDB TO GEOM demonstrate the **render geometry** module.

render manager

Share geometries among subnetworks

Summary

Name	render manager	
Type	data output	
Inputs	geometry field 4-vector byte	
Outputs	upstream transform upstream geometry	
Parameters	*Name* create new window Window Choices	*Type* oneshot radio buttons

Description

The **render manager** module takes geometries as input, uses the 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 data sets or different mappings of the same data set.

This module is in the unsupported library.

Inputs

Geometry (required; geometry)

Any geometry.

Texture (optional; field 4-vector byte)

This input port provides network compatibility with other ports that support texture mapping.

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.

Window Choices

Radio buttons that list 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.

Outputs

Transform (upstream transform)

This output informs upstream modules of the transformation applied to the input geometry. For example, **arbitrary slicer** uses this output to recalculate a new slice plane through a volume during your manipulations.

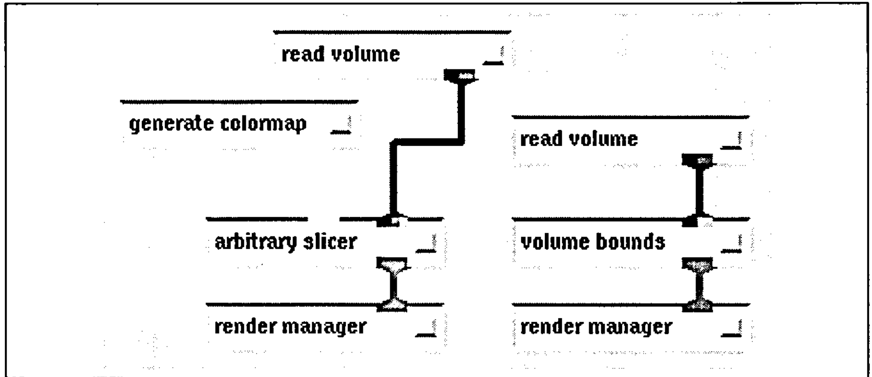
Geometry (upstream geometry)

This output informs upstream modules of the chosen vertex. For example, **probe** uses this output to determine where you have requested a data value.

Example

The networks in Figure 121 show **render manager**.

Figure 121
render manager module in example networks



Related modules

display pixmap, read geom, pdb to geom, and render geometry

Note

The output windows are not destroyed until all **render manager** modules are destroyed.

replace alpha

Replace the alpha channel in an image

Summary

Name	replace alpha
Type	filter
Inputs	field 2D uniform 4-vector byte field 2D uniform scalar byte
Outputs	field 2D uniform 4-vector byte
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 produced by passing the image through the **luminence** or **extract scalar** module, then performing further imaging techniques on the scalar value. 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 **replace alpha**.

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 **replace alpha**.

Outputs

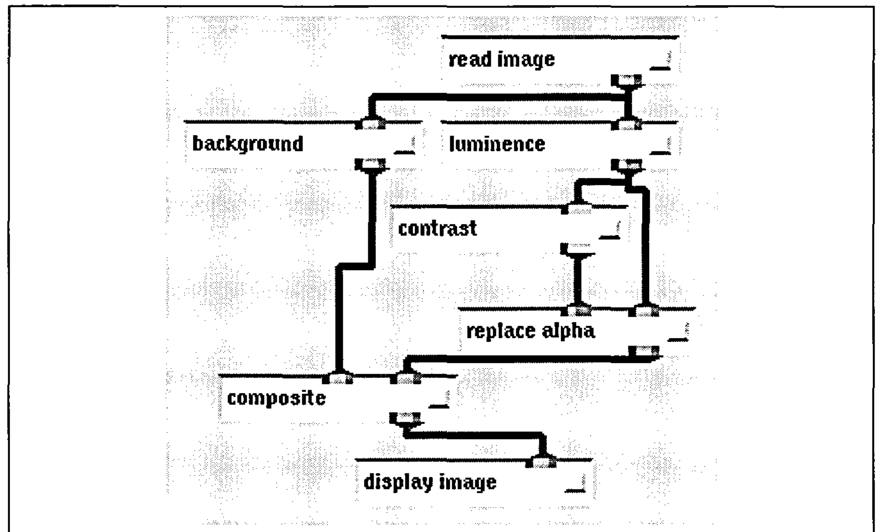
Image (field 2D uniform 4-vector byte)

> The output image has the same dimension as the input image.

Example

The network in Figure 122 reads an image, computes its luminence, uses that to create an alpha mask, generates a shaded background, and blends the rendered image over the shaded background image.

Figure 122
replace alpha module in an example network



Related modules

Modules that could provide the **Image** input are contrast, pixmap to image, read image, threshold, read rle image, and read hdf image.

Modules that could provide the **Data Field** input are luminence and extract scalar.

Modules that can process **replace alpha** output are composite, write image, image to pixmap, hq display image, image viewer, write frame seq, and write hdf image.

See also

The two example BACKGROUND scripts demonstrate the **replace alpha** module.

samplers

Extract a subset of locations from a 3-vector 3D field

Summary

Name	samplers
Type	data input
Inputs	field 3D float *any-data any-coordinates* upstream transform
Outputs	field 3-space 0-vector irregular
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 field of floating point data to produce an output field that is "3-space irregular" (that is, it contains a series of coordinates in 3-space 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
- 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 do not connect **samplers** to the left input port on **hedgehog**, **particle advector**, or **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.),

If you want less than a whole plane or 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 **render geometry** module in a network, **render geometry** 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 invisible. This gives direct mouse manipulation control over the **samplers** sample set.

Parameters

choice

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 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.

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**.

Outputs

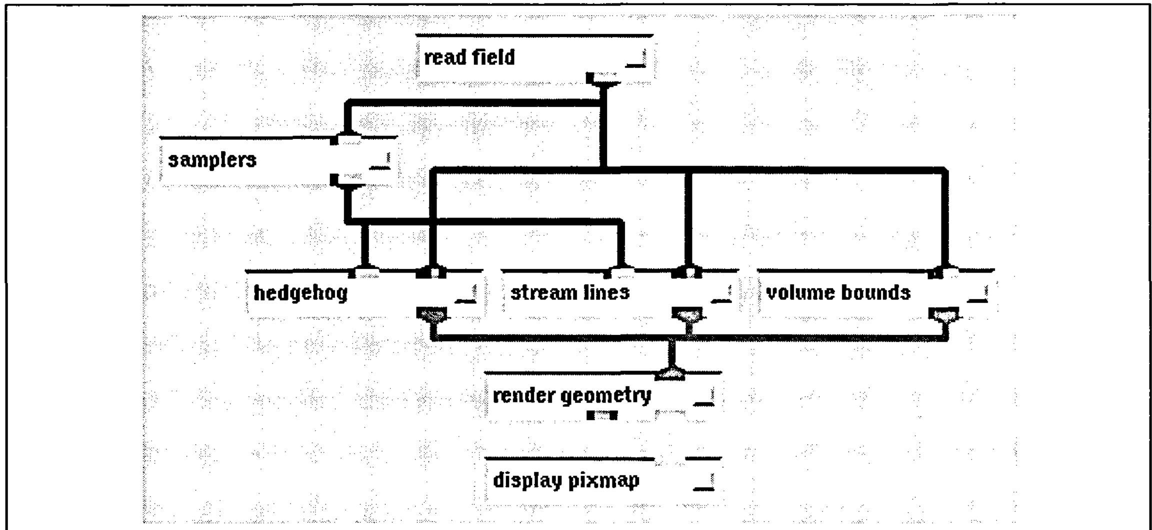
Data Field (field 3-space 0-vector irregular)

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

The network in Figure 123 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.

Figure 123
samplers module in an example network



Related modules

Module that could provide the **Data Field** input is read field.

Modules that can process **samplers** output hedgehog, particle advector, and stream lines.

See also

The example script PARTICLE ADVECTOR demonstrates the **samplers** module.

scalar PLOT3D

Calculate derived PLOT3D scalar functions

Summary

Name	scalar PLOT3D	
Type	filter	
Inputs	field 3D 5-vector irregular 3-space float field 3D scalar uniform integer field 1D scalar uniform float	
Outputs	field 3D scalar irregular 3-space float	
Parameters	*Name*	*Type*
	func name	choice

Description

This module allows you to visualize some of the scalar fields that can be derived from the basic PLOT3D data. The source code is designed so that each of the functions is implemented separately from the ConvexAVS flow. Each of the functions and their names (for the select button) are stored in a table. In this way, you can extend the functionality of this module beyond a list of functions implemented in the standard PLOT3D viewer by adding new functions and names. The source for this module is in the /usr/avs/examples/convex/plot3d/modules directory.

Inputs

Data Field (required; field 3D 5-vector irregular 3-space float)

This input data is the 5-vector field output by **read PLOT3D.** It is the most important field because it contains the mesh and solution data.

Blanking Data Field (required; field 3D scalar uniform integer)

This field contains the blanking records. If a PLOT3D data set contains blanking records and this is not connected, unpredictable results may occur.

Global Parameters (required; field 1D scalar uniform float)

This field contains the four global parameters (free stream mach number, angle-of-attack (alpha), Reynolds number, and the time). Not all derived scalar functions make use of this data, but it must be connected so that those derived scalar functions that do require the information can be calculated.

Outputs

Scalar Field (field 3D scalar irregular 3-space float)

A scalar field representing the function selected.

Parameters

func name

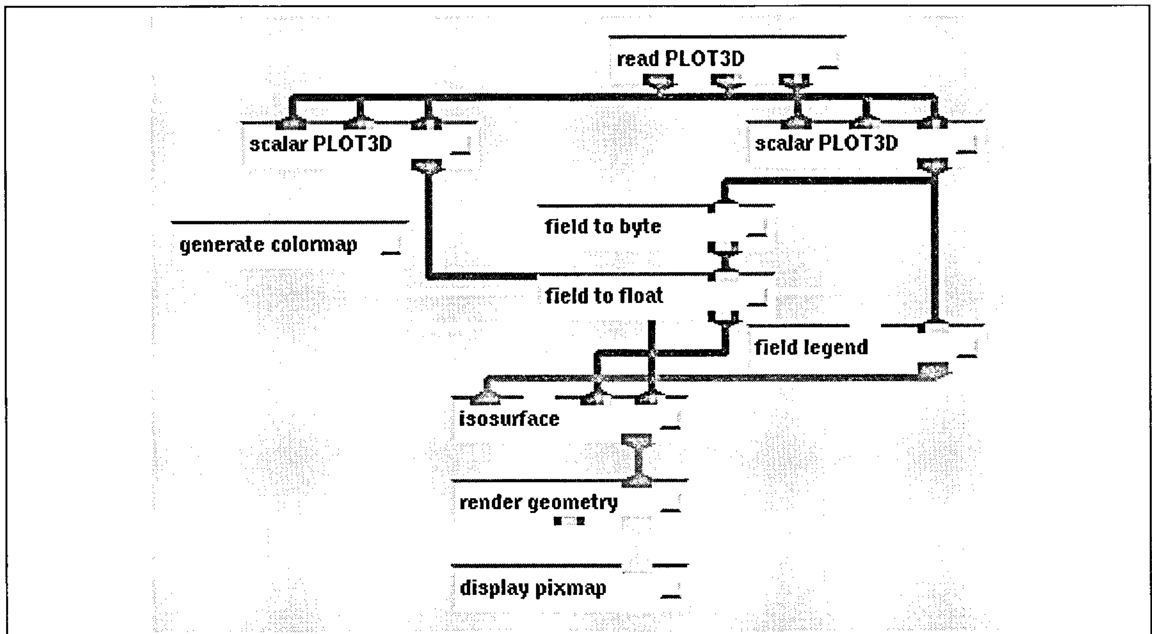
A choice parameter that selects the desired scalar function of the PLOT3D values to calculate. Implemented functions are:

- Pressure
- Temperature
- Enthalpy
- Internal energy
- Mach number
- Entropy

Example

The network in Figure 124 reads in a PLOT3D data set and does an isosurface on one of the derived scalar fields.

Figure 124
scalar PLOT3D module in an example network



Related modules

density PLOT3D, momentum PLOT3D, stagnation PLOT3D, read PLOT3D, and vector PLOT3D

Related programs

export_PLOT3D and import_PLOT3D

scatter dots

Generate spheres at points in 3D space

Summary

Name scatter dots

Type mapper

Inputs field 1D real 3-space irregular

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 the value of the radius parameter, 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).

Inputs

Point List (required; field 1D real 3-space irregular)

The input field must be a list of points in 3D space with a float value specified at each point (a scatter field).

Parameters

Connect the dots

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

A floating-point multiplier factor for the sphere radii.

Outputs

Geometry (geometry)

The output is a geometric mapping of data points to spheres.

Examples

1.

The **scatter dots** module can be used in combination with the **dot surface** module, as shown in Figure 125.

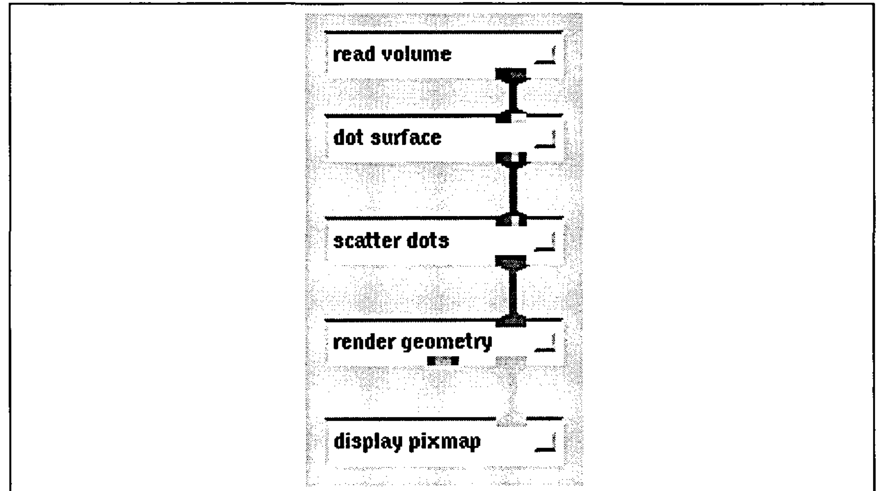


Figure 125
scatter dots module in an example network

2.

The **scatter dots** module is required to make **bubbleviz** work properly, as shown in Figure 126.

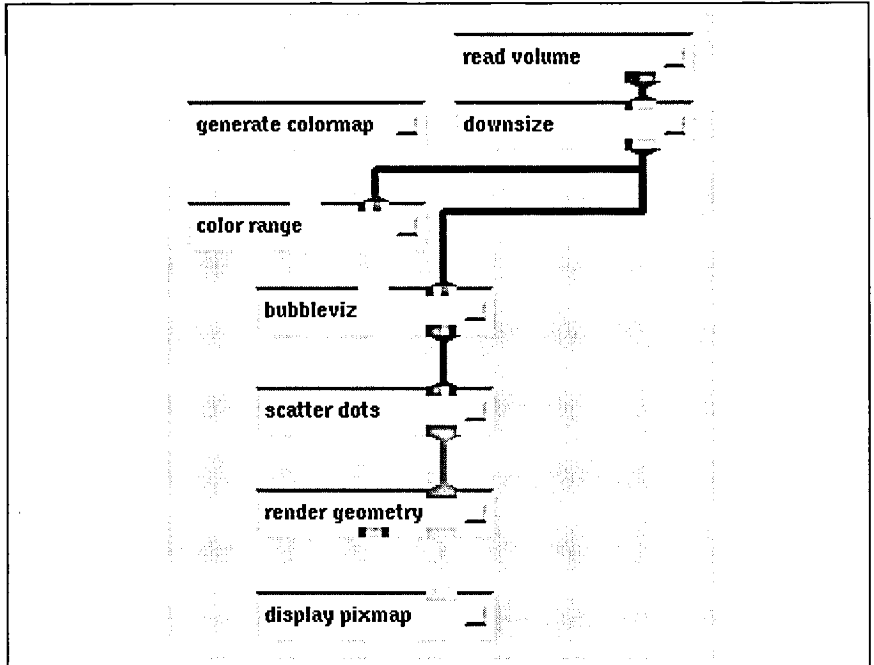


Figure 126
scatter dots module in an example network

Related modules

Modules that could provide the **Point List** input are read field, dot surface, and bubbleviz.

Modules that could process the **Geometry** output are wireframe and render geometry.

See also

The example scripts BUBBLEVIZ and DOT SURFACE demonstrate the **scatter dots** module.

shrink

Make polygons of a geometry object smaller

Summary

Name shrink

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 a 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)

A geometry created with the geometry library or by another module.

Parameters

offset

The amount by which each vertex is translated. The greater the value, the more the geometry is collapsed inward.

Outputs

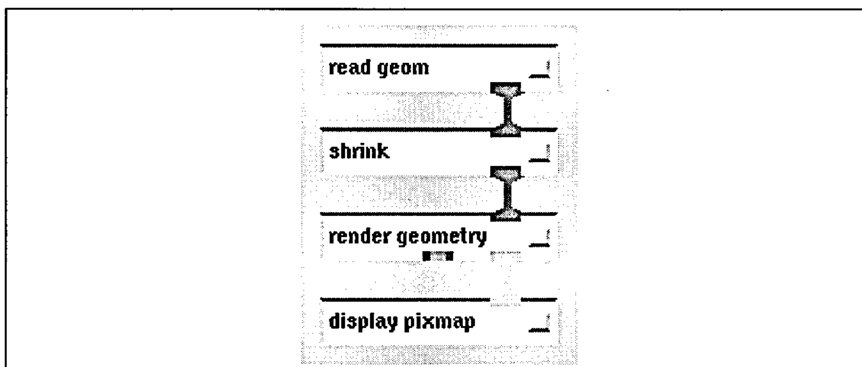
Geometry (geometry)

A geometry that represents the same object(s) as the input data.

Example

The network in Figure 127 shows **shrink** in a sample network.

Figure 127
shrink module in an example network



Related modules

Modules that could provide the **Geometry** input are read geom and isosurface.

Module that could display the **Geometry** output is render geometry.

Limitations

This module works only for polytriangle strips and meshes; it does not work for polyhedra, spheres, or lines.

This module does not copy UV data.

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.

sobel

Apply an edge detecting filter to 2D field

Summary

Name	sobel
Type	filter
Inputs	field 2D *n*-vector *any-data any-coordinates*
Outputs	field 2D *n*-vector *any-data any-coordinates*
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, 3 by 3 filters. One detects changes in an image in the x-direction, thus detecting vertical edges. The other detects changes in the y-direction, thus detecting 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 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 field to be operated on.

Outputs

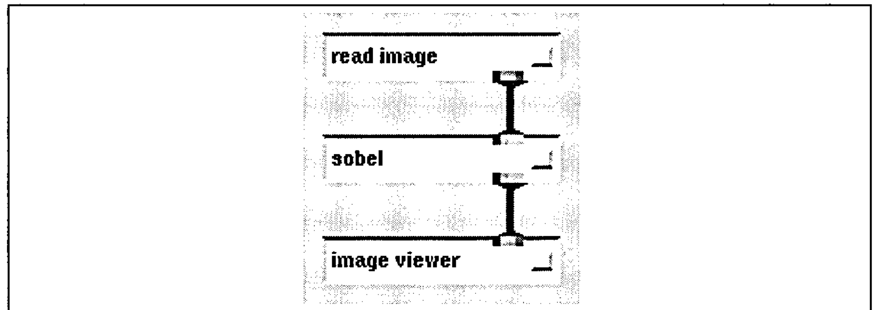
Data Field (field 2D *n*-vector *any-data any-coordinates*)

The output field is the same type as the input data field.

Example

The network in Figure 128 reads in an image, applies the sobel operation to it, and displays the resulting image.

Figure 128
sobel module in an example network



Related modules

Modules that could provide the **Data Field** input are read image, pixmap to image, and orthogonal slicer.

Modules that can process **sobel**'s output are display image, image viewer, and hq display image.

See also

The example script SOBEL demonstrates the **sobel** module.

stagnation PLOT3D

Strip out the PLOT3D stagnation energy

Summary

Name	stagnation PLOT3D
Type	filter
Inputs	field 3D 3-space irregular 5-vector real
Outputs	field 3D 3-space irregular scalar real
Parameters	none

Description

This module allows you to visualize the stagnation energy field in the PLOT3D file. It merely converts the 5-vector into the stagnation energy scalar field.

Inputs

Data Field (required; field 3D 3-space irregular 5-vector real)

This input data is the 5-vector field output by **read PLOT3D.** It is the most important field because it contains the mesh and solution data.

Outputs

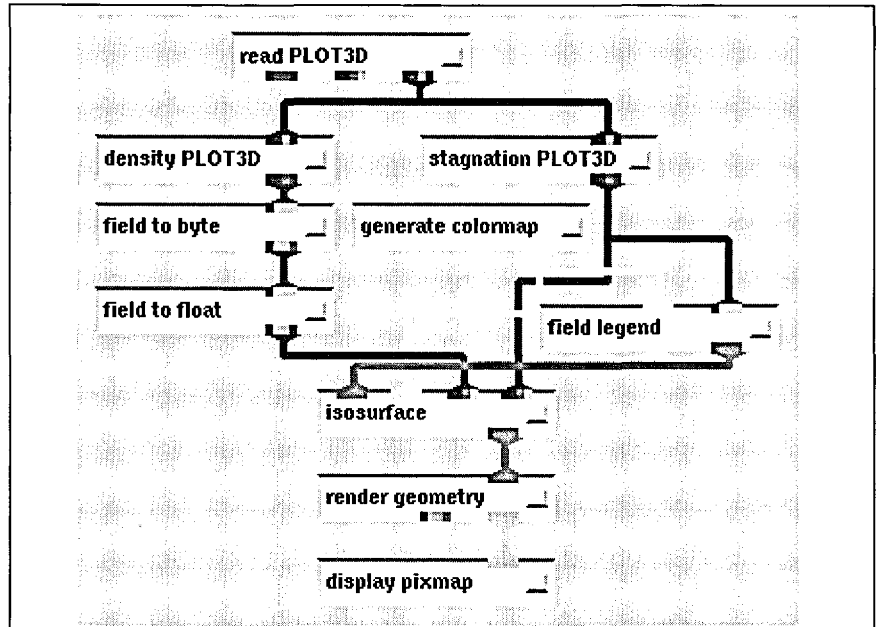
Scalar Field (field 3D 3-space irregular scalar real)

A scalar field representing the stagnation energy field from the PLOT3D file read by **read PLOT3D.**

Example

The network in Figure 129 reads in a PLOT3D data set and does an isosurface on the stagnation field. Only the PLOT3D field itself is used. We do not use the blanking records because we are extracting a part of the raw PLOT3D field.

Figure 129
stagnation PLOT3D module in an example network



Related modules

density PLOT3D, momentum PLOT3D, read PLOT3D, scalar PLOT3D, and vector PLOT3D

Related programs

export_PLOT3D and import_PLOT3D

statistics

Display statistics on field contents

Summary

Name	statistics
Type	data output
Inputs	field *any-dimension n*-vector *any-data any-coordinates*
Outputs	none

Parameters

Name	*Type*	*Default*
Compute Median	switch	off
Statistics	text	

Description

The **statistics** module displays global statistical information about field data. **statistics** scans the input field and produces output in the form shown below:

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

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 minimum and maximum are. This information is often useful if you wish to scale the dials in downstream modules that are operating on the same input field.

Inputs

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

The input 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.

Statistics

Text information about the field:

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.

Skewness

Third moment about the mean of the data set. Average of cubed variations about the mean.

When positive, the right side of the distribution curve is steeper than the left. When negative, the left side is steeper.

Kurtosis

Fourth moment about the mean of the data set. Average of the fourth power of variations about the mean.

When positive, the data has more spikes than a standard distribution. When negative, the data is more broadly-distributed than a standard distribution.

Examples

1.

The network in Figure 130 computes statistics on an image.

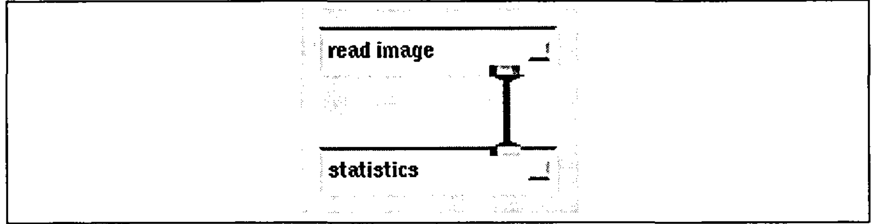


Figure 130
statistics module in an example network

2.

The network in Figure 131 shows how you might use the **statistics** module to determine the minimum and maximum values in a 3D field, so that you could scale the dials on the **thresholded slicer** module accordingly.

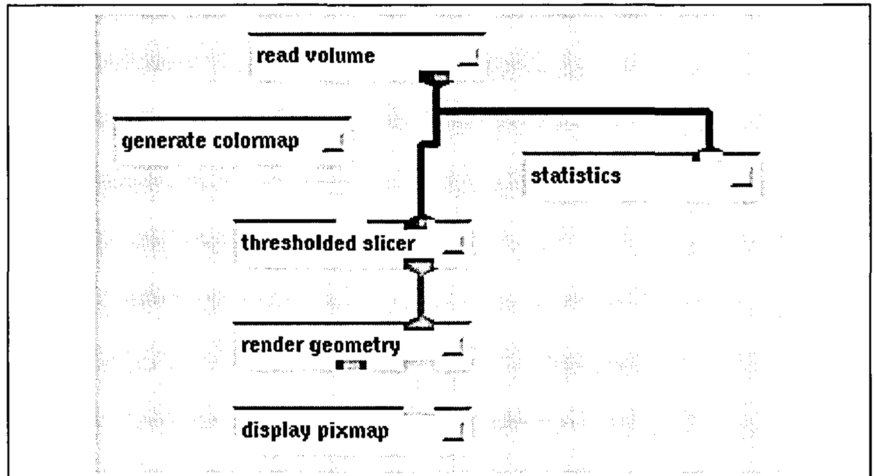


Figure 131
statistics module in an example network

Related modules

print field and compare field

See also

The example script STATISTICS demonstrates the **statistics** module.

stream lines

Generate stream lines for a vector field

Summary

Name stream lines

Type mapper

Inputs field 3D 3-vector *any-data any-coordinates* float
field irregular 3-space
upstream transform

Outputs geometry
upstream transform

Parameters

Name	*Type*	*Default*	*Min*	*Max*	*Values*
length	integer	12	4	128	
step	float	0.02	none	none	
mesh	radio	Lines			Lines, Mesh
method	radio	Euler			Euler, Runge-Kutta
N Segment	integer	16	2	64	
style	radio	point			point, line, circle, plane, space

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.

Inputs

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

The input field must be a 3D uniform field. The data for each field element must be a 3D vector of any primitive data type, representing the components of a velocity vector.

Sample Field (optional; field irregular 3-space)

This leftmost input port is used 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.

When the **stream lines** module receives input locations from **samplers**, **stream lines**'s **N Segments** dial and its **style** buttons disappear from the control panel.

Upstream Transform (optional; invisible, autoconnect)

When the **stream lines** and **render geometry** modules coexist in a network, they communicate through an invisible data port. Streamline shows up as an object in the Geometry Viewer. When you select the streamline object and move it, **render geometry** informs the **stream lines** module what the sample's new location is, and **stream lines** recalculates the location and streamlines. This module connection occurs automatically. The effect gives you direct mouse-manipulation control over the **stream lines** module's sample of locations.

Parameters

length

A scale factor that 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.

mesh

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

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 more accurate results.

N Segment

An integer value that determines the number of points for which stream lines are computed. This controls the density of the stream lines output by **stream lines**.

style

Specifies the configuration of points from which stream lines will be drawn: point, line, circle, plane, or space.

Outputs

Streamlines (geometry)

A set of disjoint lines.

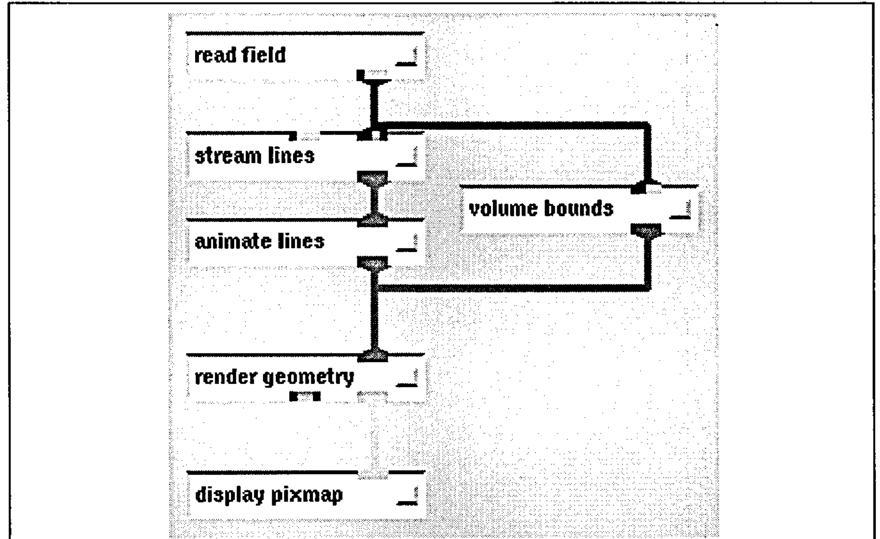
Upstream Transform (`struct upstream_transform`)

Automatically connects to the render geometry module.

Example

The network in Figure 132 reads in a 3D vector field and calculates streamlines for the field. **animate lines** is used to dynamically represent the output of **stream lines**.

Figure 132
stream lines module in an example network



Related modules

animate lines, hedgehog, particle advector, and samplers

See also

The example script STREAMLINES demonstrates the **stream lines** module.

surface mapper

Represent a list of points as a surface of dots or spheres

Summary

Name	surface mapper		
Type	mapper		
Inputs	field 1D real 3-space irregular colormap		
Outputs	geometry		
Parameters	*Name*	*Type*	*Choices*
	Geom Representation	radio	dots, line slice, sphere, uniform sphere
	Radius	typein real	
	Dot len	typein real	

Description

The **surface mapper** module renders 1D field output as dots, lines, or spheres. This module is useful when combined with **read mopac**.

Parameters

Geom Representation

The type of geometry produced:

dots	A short line (dot) is used to represent the surface values.
line slice	All points with equal Y-values are connected with an isoline.
sphere	A variable sized sphere is used to represent the surface values.
uniform sphere	A uniform sized sphere is used to represent the surface values.

Radius

The radius value for rendering uniform spheres.

Dot len

The length of the short lines used to represent the dots.

Inputs

Data Field (required; field 1D real 3-space irregular)

A 1D array of coordinates with a single floating-point value.

Colormap (optional; colormap)

This input colors each sphere at a position according to its floating-point value.

Outputs

Molecule Surface (geometry)

A geometric representation of the Connolly surface of the molecule.

Example

The network in Figure 133 shows a simple application of **surface mapper**.

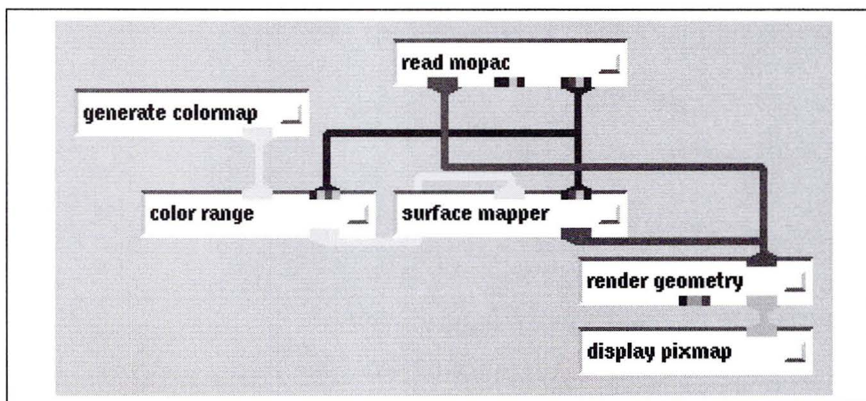


Figure 133
surface mapper module in an example network

Related modules

render geometry, isosurface, and orbital tiler

threshold

Restrict values in data field

Summary

Name	threshold
Type	filter
Inputs	field *any-dimension n*-vector *any-data any-coordinates*
Outputs	field *any-dimension n*-vector *any-data any-coordinates*

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 values of a field as follows:

- Any value less than the value of the **thresh_min** parameter is set to 0.
- Any value greater than the value of the **thresh_max** parameter is set to 0.
- All values within the **thresh_min** to **thresh_max** range are not changed.

The difference between the **clamp** and **threshold** modules follows:

- **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.

threshold operates on byte, integer, float, and double data types.

Inputs

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

The input data may be any field.

Parameters

thresh_min

The minimum threshold value.

thresh_max

The maximum threshold value.

Outputs

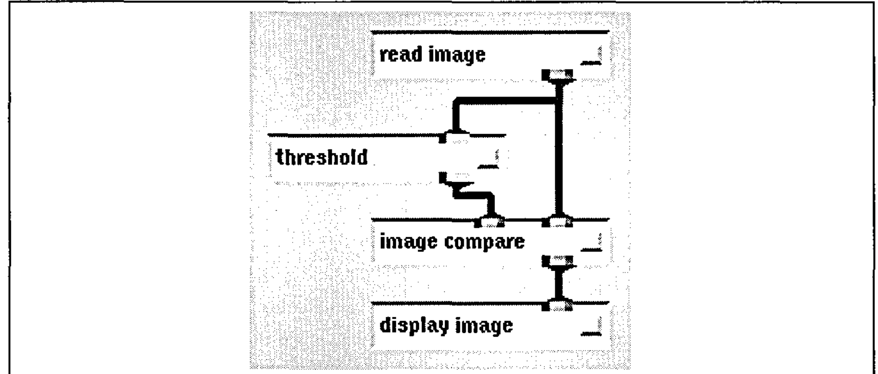
Field Data (*any-dimension n*-vector *any-data any-coordinates*)

The output field has the same dimensionality as the input field but with the values of out-of-range elements set to zero.

Example

The network in Figure 134 shows **threshold.**

Figure 134
threshold module in an example network



Related modules

Module that could provide the **Data Field** input is read volume.

Module that could be used in place of **threshold** is clamp.

Module that can process **threshold** output is colorizer.

See also

The example scripts CONTOUR GEOMETRY and THRESHOLDED SLICER demonstrate the **threshold** module.

thresholded slicer

Slice through volume data with high/low values invisible

Summary

Name thresholded slicer

Type mapper

Inputs field 3D scalar *any-data any-coordinates*
colormap
upstream transform

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
resolution	int	12	12	64
Distance	float	0.0	none	none
Low Threshold	float	0.0	none	none
High Threshold	float	255.0	none	none
choice	choice	coarse		
sample	choice	Point		

Description

The **thresholded slicer** module extracts a 2D slice from a 3D volume of data. It differs from the **arbitrary slicer** and **orthogonal slicer** 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. You can 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 **render geometry** 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. Because the mesh vertices do not coincide with the original lattice points, an interpolation method can be used.

By default, the volume is placed at the origin, and the slice plane is an XY-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 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.

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, then used as indexes into the colormap.

Upstream Transform (optional; invisible, autoconnect)

When the **thresholded slicer** module coexists with the **render geometry** module in a network, **render geometry** 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 invisible. This gives direct mouse-manipulation control over **thresholded slicer**'s slice plane.

Parameters

resolution

An integer dial that controls how many sampling points are taken through each dimension of the volume data. The default is low resolution.

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.

choice

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.

sample

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

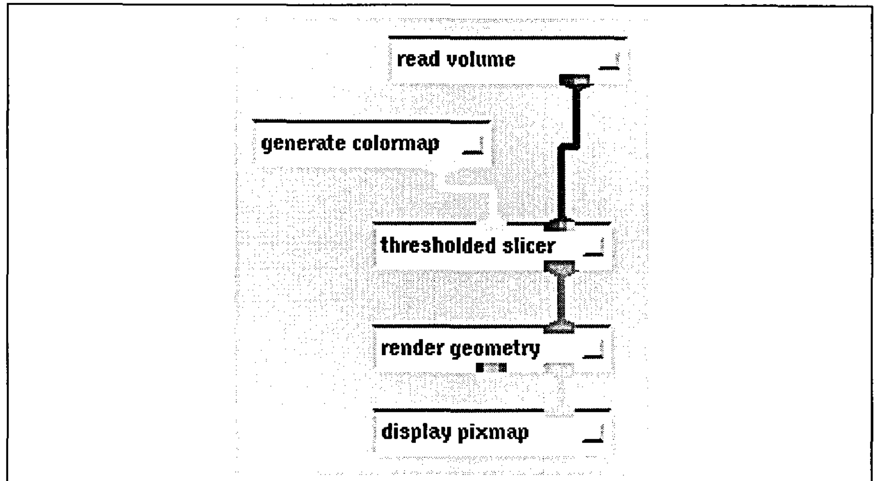
Geometry (geometry)

The output is a geometric representation of a volume slice.

Example

The network in Figure 135 shows the usage of the **thresholded slicer** module for byte data in the range 0-255.

Figure 135
thresholded slicer module in an example network



Related modules

Modules that could provide input to **thresholded slicer** are read volume, color range, and generate colormap.

Modules that could process **thresholded slicer**'s output is render geometry.

See also

The example script THRESHOLDED SLICER demonstrates the **thresholded slicer** module.

tracer

Perform ray-traced volumetric rendering on 3D field

Summary

Name	tracer
Type	mapper
Inputs	field 3D 4-vector byte field 2D scalar float
Outputs	field 2D 4-vector byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
alpha scale	float	1.0	0.0	1.0
perspective	float	0.0	0.0	1.0
Width	integer	64		
Height	integer	64		
interpolate	toggle	off		

Description

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, consequently **tracer**'s input must be a 3D field of color values. In other words, the data at each point of the volume must be a 4-vector of bytes in the alpha-red-green-blue format.

Use the **colorizer** or **gradient shade** module to generate such color fields. You can use **tracer** with nonbyte data, because **colorizer** and **gradient shade** take in a field containing any data (byte, integer, float, double), and output a 3D field of color values.

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 to edit the opacities associated with the data.

The method used by **tracer** is considerably faster than that used by **vbuffer** and avoids the image anomalies that **vbuffer** displays when volumes are rotated.

Inputs

Data Field (required; field 3D 4-vector byte)

The input data must be a 3D block of voxels. That is, the data at each point of the 3D volume field must be a 4-vector of bytes in the alpha-red-green-blue format. Voxel data is produced by passing 3D field data through **colorizer**.

Upstream Transform (optional; field 2D scalar float)

This port can receive a 4 by 4 transformation matrix describing rotations and translations to apply to the volume data. This matrix can come from an appropriate downstream module, **euler transformation**, or **display tracker**. 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. This allows you to directly manipulate the volume by moving the mouse in **display tracker**'s window.

Parameters

alpha scale

A floating-point value between 0.0 and 1.0 that is multiplied by the alpha byte of every voxel in the volume. This determines how transparent 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.

perspective

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 (that is, non-perspective) view of the volume. As the perspective value increases, the point from which rays emanate 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

Value that 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 **Width** can bring clipped regions of the window back into view.

Height

Value that 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** on, changing **Height** can bring clipped regions of the window back into view.

interpolate

Allows you to choose between two ray-tracing algorithms:

Voxel Approximation (default)

The 3D field is broken into cells, or voxels. Each voxel has a single opacity, color and set of shading parameters. These values are taken from the vertex at the voxel's upper left corner and are assumed to be uniform throughout the voxel.

The length of a ray's path through a voxel is computed. 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 **Trilinear Interpolation**. Use it to get a quick look at the data.

Trilinear Interpolation

It is not assumed in this algorithm 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, but takes much longer to complete.

Outputs

Data Field (field 2D 4-vector byte)

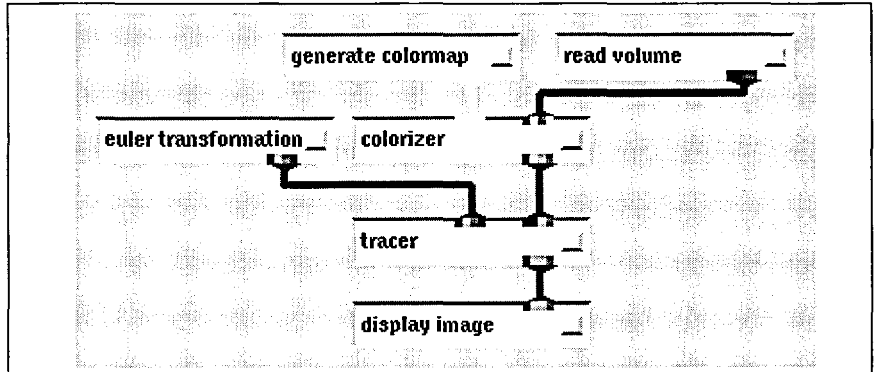
The output field is an image.

Examples

1.

The network in Figure 136 reads in volume data, assigns color values to each element of the data, and passes this field to **tracer**. The module **euler transformation** allows you to rotate the volume to produce views from any angle.

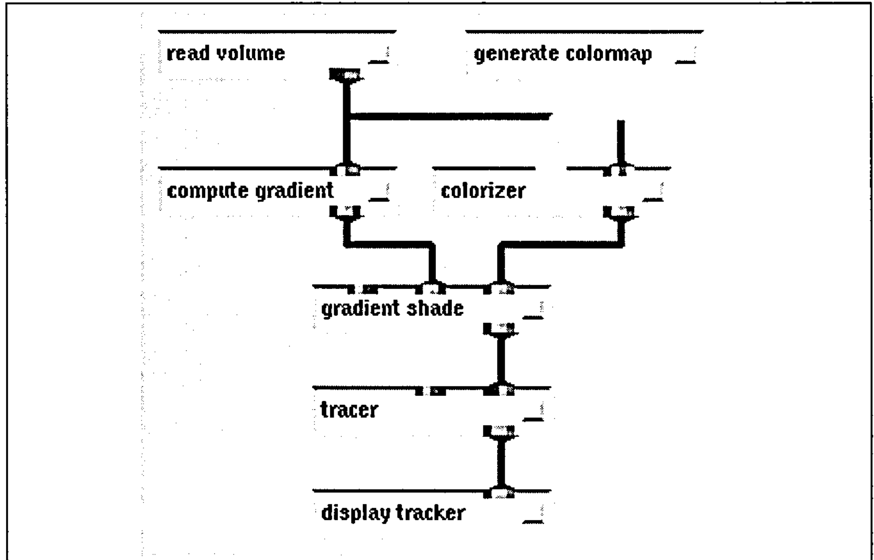
Figure 136
tracer module in an example network



2.

Another technique is to apply a light source to the data. In order to do this, the gradient of the data (that approximates the surface normal) must be computed. The **gradient 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** or **euler transformation.** Without connecting **display tracker** (or **euler transformation)** to **gradient shade,** the light source would appear attached to any object transformations. A network for doing this gradient shading is in Figure 137.

Figure 137
tracer module in an example network



Related modules

Modules that could provide the **Data Field** input are read field, colorizer, and gradient shade.

Modules that could provide the **Transformation Matrix** input are euler transformation and display tracker.

Modules that can process **tracer**'s output are display tracker, display image, image viewer, and hq display image.

See also

The example script TRACER demonstrates the **tracer** module.

transpose

Exchange dimensions in a 2D or 3D data set

Summary

Name	transpose
Type	filter
Inputs	field 2D/3D *n*-vector *any-data any-coordinates*
Outputs	field 2D/3D *n*-vector *any-data any-coordinates*

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

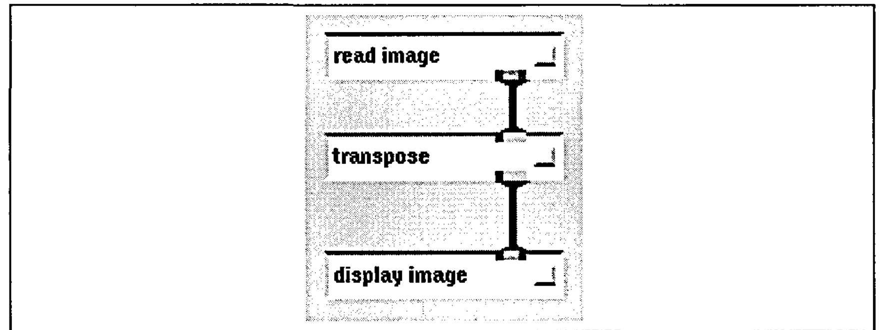
Data Field (field 2D/3D *n*-vector *any-data any-coordinates*)

The output field has the same dimensionality and type as the input field.

Example

The network in Figure 138 reads in an image and then swaps the XY dimensions.

Figure 138
transpose module in an example network



Related modules

mirror

tristate

Send a tristate value to one or more module(s) tristate parameter port(s)

Summary

Name	tristate
Type	data input
Inputs	none
Outputs	tristate

Parameters

Name	*Type*	*Default*	*Min*	*Max*
tristate	integer	0	0	2

Description

The **tristate** module sends a single tristate value to one or more tristate parameter ports on one or more receiving modules. This makes it possible for you 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 a receiving module, you must make the receiving module's parameter port visible.

Parameters

tristate

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).

Outputs

Integer (tristate)

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

Related modules

integer and oneshot

tube

Convert lines to cylindrical tubes

Summary

Name	tube				
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 a geometry, replacing a set of disjoint lines with tubes constructed out of eight polygons.

Inputs

Geometry (required; geometry)

A geometry containing lines created by another 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 a geometry, representing each input line segment as a set of polygons.

Example

Figure 139 shows **tube** in a network.

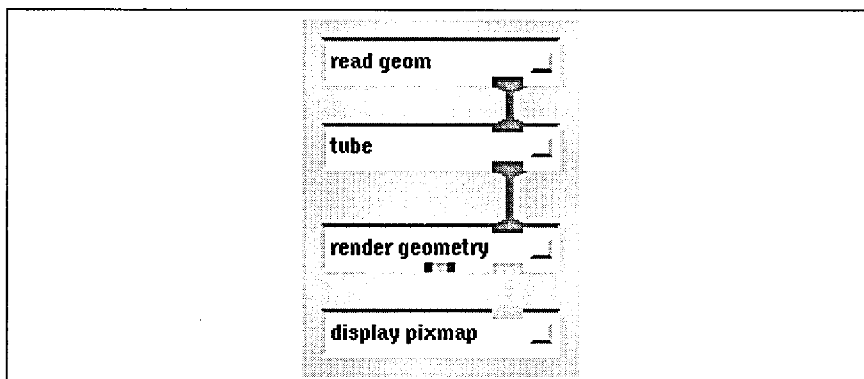


Figure 139
tube module in an example network

Related modules

read geom, offset, shrink, flip normal, wireframe, and render geometry

Limitations

Cylinders are not capped, and adjacent line segments are not joined.

For thick cylinders, there may be surface intersections at the joins.

Spheres and other geometric shapes are not passed through.

All color information from the lines is lost, and the tubes are gray.

See also

The example script TUBE demonstrates the **tube** module.

ucd anno

Show data values of cells or nodes of a UCD structure

Summary

Name ucd anno

Type mapper

Inputs ucd structure
upstream geometry

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
cell data	choice	<data 1>		
label id	boolean	on		
label value	boolean	off		
cell nodes	boolean	off		
title	boolean	off		
Text Size	integer	2	1	5

Description

ucd anno makes it possible to see the values of specific cells and nodes of a UCD structure 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 **render geometry** module. The **ucd anno** module 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. Use the **node data** and **cell data** radio buttons to select which data component you want **ucd anno** to display.

ucd anno takes two inputs: a UCD structure and an upstream geometry that it receives when it is in a network with **render geometry.** When you click the left mouse button on the image of the UCD structure, the **render geometry** 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 **display pixmap**'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 bring the label above the structure's surface.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format.

upstream geometry (optional; invisible, autoconnect)

When the **ucd anno** module coexists with the **render geometry** module in a network, **render geometry** 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 invisible. This makes it possible to see the values of specific cells and nodes simply by clicking on them.

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 has received data, the default "<data *x*>" is displayed. If there is no node data in the structure, "<no data>" is displayed on the button.

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 has received data, the default "<data *x*>" is displayed. If there is no cell data in the structure, "<no data>" is displayed on the button.

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 nodes of the cell that has been clicked on. Thus, for a hexahedron, **ucd anno** would display the node data at each of the cell's eight 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.

Outputs

Geometry (geometry)

ucd anno's outputs consist of the selected UCD structure values output as a geometry.

Example

The network in Figure 140 reads in a UCD structure and annotates it. The selected values are displayed by **render geometry** along with the UCD structure itself.

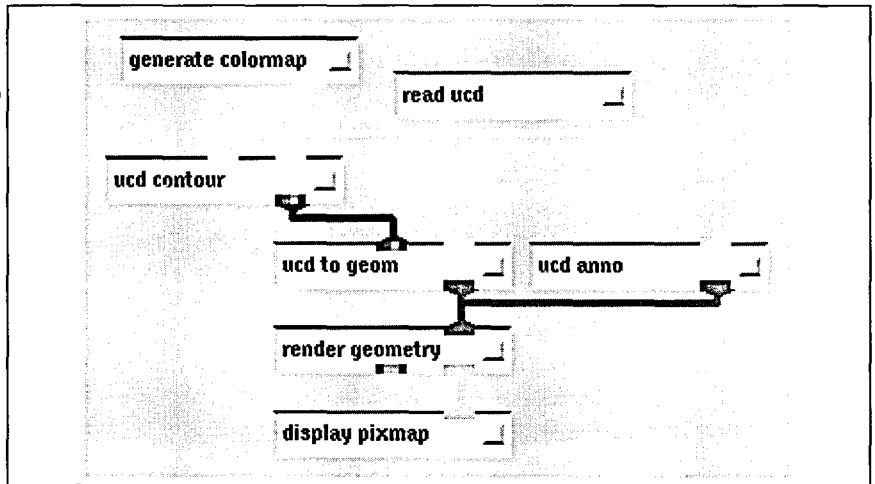


Figure 140
ucd anno module in an example network

Related modules

Modules that could provide the **UCD Structure** input are field to ucd, ucd crop, ucd threshold, ucd extract, and ucd hex to tet.

Module that can process **ucd anno**'s output is render geometry.

Limitations

ucd anno does not yet support the display of cell-based data.

See also

The example script UCD ANNO demonstrates the **ucd anno** module.

ucd cell to node

Convert UCD cell data into node data

Summary

Name	ucd cell to node
Type	filter
Inputs	ucd structure
Outputs	ucd structure
Parameters	none

Description

The **ucd cell to node** module accepts a UCD structure with cell data as input and computes node data based on the cell data values. The output is a UCD structure containing only node data. The algorithm used averages the cell values from each cell containing a given node and places that value in the data slot for that node. Because no other ConvexAVS UCD module can access cell data, the **ucd cell to node** module is necessary to visualize UCD structures.

Inputs

UCD Structure (required; ucd structure)

The input structure is a UCD structure with only cell data.

Outputs

UCD Structure (ucd structure)

The output structure is a UCD structure with only node data.

Example

The network in Figure 141 reads in a 3D vector UCD structure and computes the magnitude of the vectors.

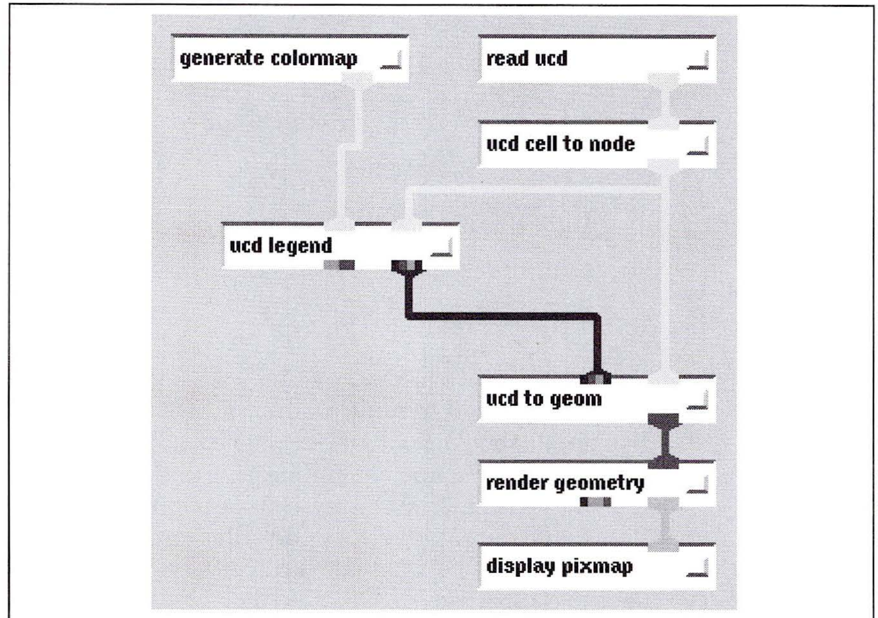


Figure 141
ucd cell to node module in an example network

Limitations

This module will ignore any node data already present in a UCD structure.

ucd contour

Generate list of color values associated with unstructured cell data

Summary

Name	ucd contour		
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.

A UCD structure has a number of nodes. Each of these nodes may have an arbitrary number of data components associated with it. 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 following formula:

$$color_index = \frac{node_value - min_node_value}{max_node_value - min_node_value} * max_colormap_value$$

The *color_index* is an index into the input colormap and is used to compute the 3-vector real value for a given color.

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. You may change the input colormap using **generate colormap's** colormap widget. The output by **ucd contour** does not include the alpha or opacity information contained in a colormap.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Colormap (required; colormap)

A colormap. **ucd contour** maps node values in the input structure to colors in the colormap.

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 has received data, the default "<data *x*>" 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 1D 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 output is not a ConvexAVS colormap.

Example

The network in Figure 142 reads in a UCD structure, colors each node based on the node's value, and displays the result.

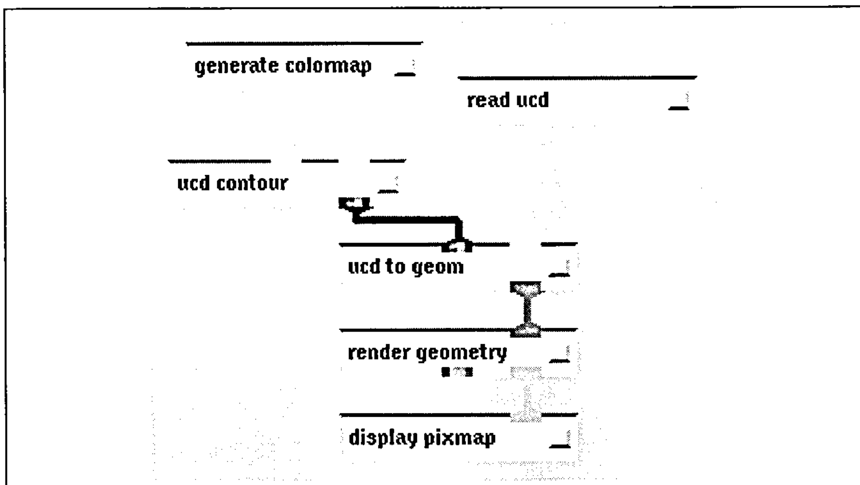


Figure 142
ucd contour module in an example network

Related modules

Modules that could provide the **UCD Structure** input are read ucd, ucd crop, and ucd threshold.

Modules that could provide the **Colormap** input are generate colormap and color range.

Modules that can process **ucd contour**'s output are ucd iso, ucd probe, ucd rslice, ucd to geom, and ucd slice 2d.

See also

The example scripts UCD ISO, UCD PROBE, and UCD EXTRACT demonstrate the **ucd contour** module.

ucd crop

Subset UCD structure data using slice plane or box

Summary

Name	ucd crop		
Type	filter		
Inputs	ucd structure upstream transform		
Outputs	ucd structure geometry		
Parameters	*Name*	*Type*	*Default*
	choice	radio	plane
	side	radio	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. Before cropping a UCD structure, the subsetting tool must be moved from its default location. To initiate the actual cropping operation, you must select **Do Crop**.

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. In other words, any cells in the input structure that lie on the highlighted 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, any cells that are outside the bounds of the wireframe box are cropped from the output structure.

The **ucd crop** module is similar to the **ucd threshold** module. **ucd crop**, however, eliminates nodes from a UCD structure based on their X-, Y-, and 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. The **ucd to geom** module is used to convert the structure output by **ucd crop** into a geometry so it can be visualized using **render geometry**.

Because **ucd crop** outputs the slice plane and box subsetting tools as geometry objects, they can be sent directly to **render geometry** and manipulated using the mouse like other geometry objects. Enter the Geometry Viewer and select the crop tool object as the current object. When **ucd crop** is linked in a network to **render geometry**, manipulating the subsetting tools with the mouse causes **render geometry** 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

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Upstream Transform (required; invisible, autoconnect)

When the **ucd crop** module coexists with the **render geometry** module in a network, **render geometry** 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 invisible. This gives direct mouse-manipulation control over **ucd crop**'s subsetting tools.

Parameters

choice

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.

side

inside — A radio button that selects the inside view.

outside — A radio button that selects the outside view.

Do Crop

A boolean switch that initiates the cropping function.

Outputs

UCD Structure (ucd structure)

The output structure is the cropped UCD structure.

Geometry (geometry)

The geometry object that **ucd crop** outputs represents the subsetting tool currently selected.

Example

The network in Figure 143 reads in a UCD structure and crops it. The **ucd crop** module outputs a geometry that gets passed directly to **render geometry**. It also outputs the cropped UCD structure from which a geometry is formed. **read ucd** also sends the UCD structure to a second **ucd to geom** module. If you switch to the **geometry viewer** and specify a wireframe representation mode for the geometry output by this module, the structure output by the second **ucd to geom** functions as a wireframe bounding volume around the structure.

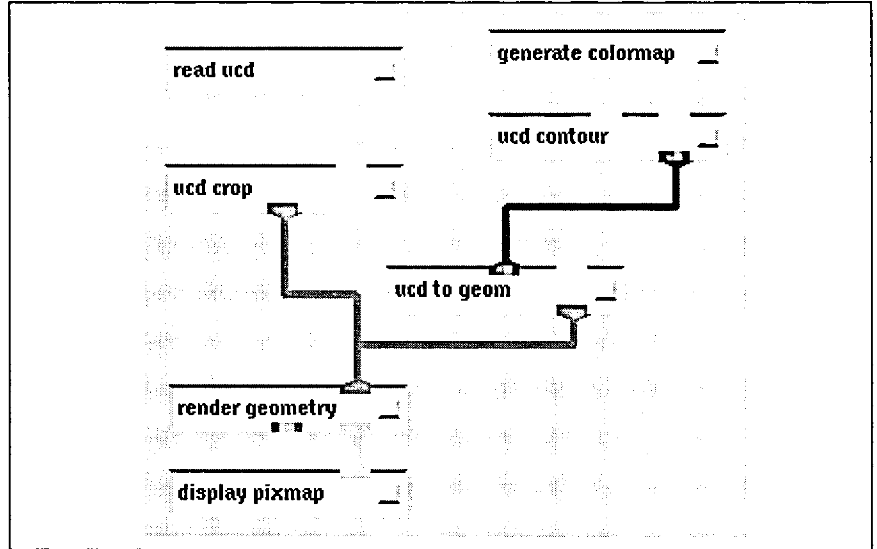


Figure 143
ucd crop module in an example network

Related modules

Modules that could provide the **UCD Structure** input are read ucd, field to ucd, and ucd extract.

Other modules that subset UCD structures are ucd threshold and ucd rslice.

Module that can process the cropped UCD structure output is ucd to geom.

Module that can process the subsetting tool output is render geometry.

See also

The example script UCD CROP demonstrates the **ucd crop** module.

ucd extract

Extract single node component from a UCD structure

Summary

Name	ucd extract		
Type	filter		
Inputs	ucd structure		
Outputs	ucd 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 could consist of three components: 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.

Inputs

UCD Structure (required; ucd structure)

The input structure is in 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 has received data, the default "<data *x*>" is displayed. If there is no node data in the structure, "<no data>" is displayed on the button.

Outputs

UCD Structure (ucd structure)

The output structure is the same as the input structure except that the node data is reduced to one component.

Example

The network in Figure 144 reads in a UCD structure, extracts one component of the node data, and colors each node based on the value of that component.

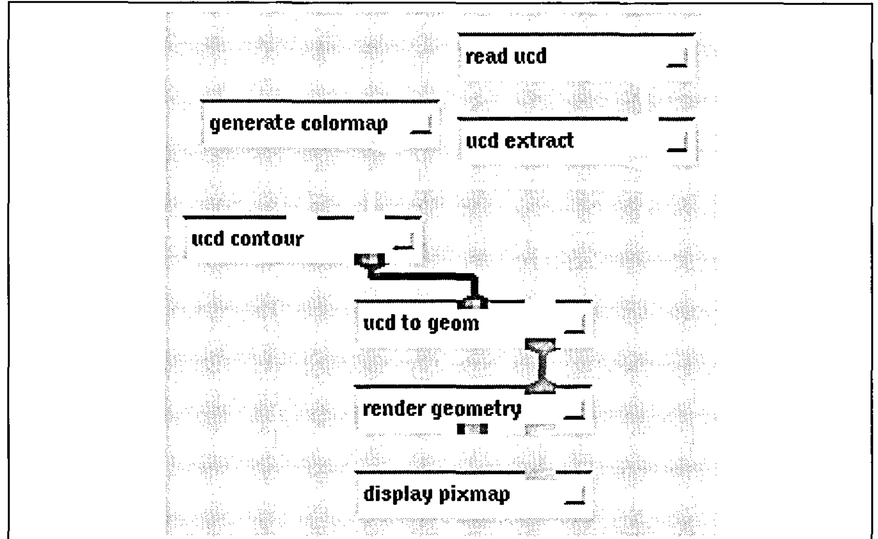


Figure 144
ucd extract module in an example network

Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

Modules that can process **ucd extract**'s output are ucd to geom, ucd crop, ucd threshold, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice 2d, ucd legend, ucd probe, ucd streamline, and write ucd.

See also

The example script UCD EXTRACT demonstrates the **ucd extract** module.

ucd hex to tet

Convert a UCD structure from hexahedral cells to tetrahedral cells

Summary

Name	ucd hex to tet		
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 into 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 five tetrahedra or into 24 tetrahedra. When data cannot be properly decomposed into five 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

UCD Structure (required; ucd structure)

The input is a UCD structure that has cells that are hexahedral.

Parameters

24 Tet

When **24 Tet** is selected, hexahedral cells are decomposed into 24 tetrahedra instead of the default, which is five.

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 has received data, the default "<data *x*>" 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

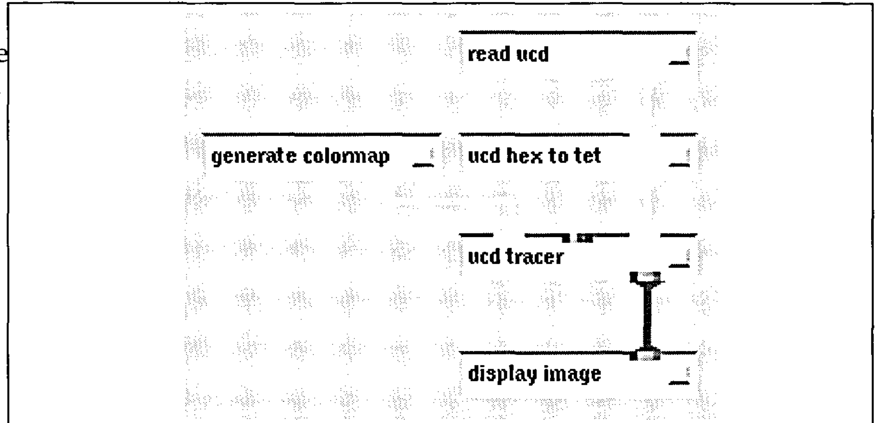
UCD Structure (ucd structure)

The output is a UCD structure that has cells that are tetrahedral.

Example

The network in Figure 145 reads in a UCD structure, which is converted from hexahedral cells to tetrahedral 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.

Figure 145
ucd hex to tet module in an example network



Related modules

Module that could provide the **UCD Structure** input is read ucd.

Module that can process **ucd hex to tet**'s output is ucd tracer.

ucd hog

Show UCD structure node vector values as line segments in 3D space

Summary

Name ucd hog

Type mapper

Inputs ucd structure
colormap
upstream transform

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
Scale	float	0.2	none	none
Arrows	boolean	off		
N Segment	integer	16	2	64
choice	choice	point		
Normalize Vectors	boolean	off		

Description

ucd hog takes in a UCD structure whose node values include a 3-vector float component. This module interprets the three values of the vector as the X-, Y-, and 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. To move the sample probe, select it by clicking on it with the left mouse button. Alternately, you can enter the Geometry Viewer and make it 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.

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 **node data** choices 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 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).

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.

Because 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.

Inputs

UCD Structure (required; ucd structure)

> The input data must be a UCD structure. The structure must include a node data component that is a 3-vector of floats to be interpreted as vectors in 3-space.

Colormap (optional; colormap)

> A colormap that is used by **ucd hog** to associate colors with vector values. This is a regular colormap and not the **color field** output by **ucd contour** and **ucd field legend**.

Upstream Transform (required; invisible, autoconnect)

> When the **ucd hog** module coexists with the **render geometry** module in a network, **render geometry** feeds information on how the sampling probe has been moved to this input port on the **ucd hog** module. The two modules connect automatically through a data pathway that is 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 the label attached to each node data component. Before the module receives data, the default "<data *x*>" is displayed. If there is no node data in the structure, "<no data>" is displayed on the button.

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 Segments

An integer value that 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**.

choice

Specifies the type of sample taken from the vector field: point, line, circle, plane, space, or nodes (where arrows will be drawn at all nodes).

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

Geometry (geometry)

The output geometry is a collection of line segments representing the 3-vector component of nodes near the sample locations.

Example

The network in Figure 146 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. The module **ucd to geom** provides a frame within which to view the hedgehog of vectors. To do this, switch into the Geometry Viewer and change the rendering mode for the geometry output by **ucd to geom** to wireframe. 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.

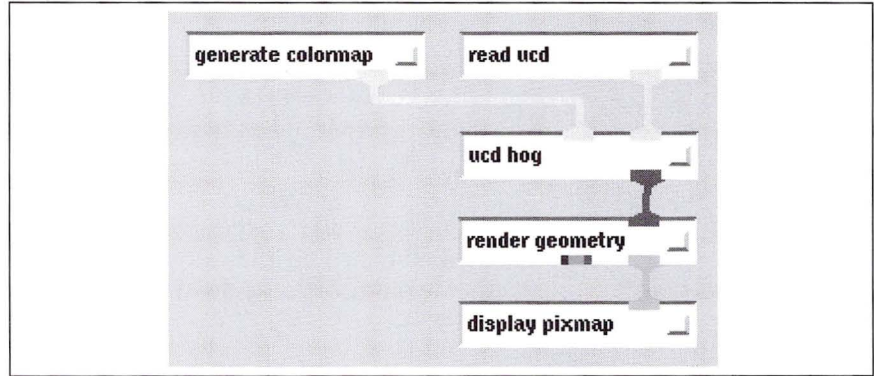


Figure 146
ucd hog module in an example network

Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

Module that can process **ucd hog**'s output is render geometry.

See also

The example script UCD HOG demonstrates the **ucd hog** module.

ucd iso

Generate an isosurface for a UCD structure with scalar node data

Summary

Name ucd iso

Type mapper

Inputs ucd structure
colormap
value

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
Level	float	1.0	1.0	9.0
Map Scalar	boolean	off		
map data	choice	<data 1>		

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** choices to select which component of the node data to use when computing the isosurface.

The **Level** value can be set in two ways: use the floating-point parameter dial or input from the **ucd legend** module.

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.

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; ucd structure)

The input data must be a UCD structure. The structure must include a scalar node data component.

Colormap (optional; colormap)

A colormap that is used by **ucd iso** to associate colors with the output isosurface. This is a regular colormap and not the **color field** output by **ucd contour** and **ucd field legend**.

Information (optional; value)

ucd iso can receive input from the module **ucd legend** through its leftmost input port. This tells **ucd iso** what the floating-point 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 *n*>" is displayed. If there is no node data in the structure, "<no data>" is displayed on the buttons.

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 **Level** value.

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.

map data

Radio buttons to use when the input field has more than one scalar component of its node data and you need to choose which component's values to use in determining the isosurface's color.

Outputs

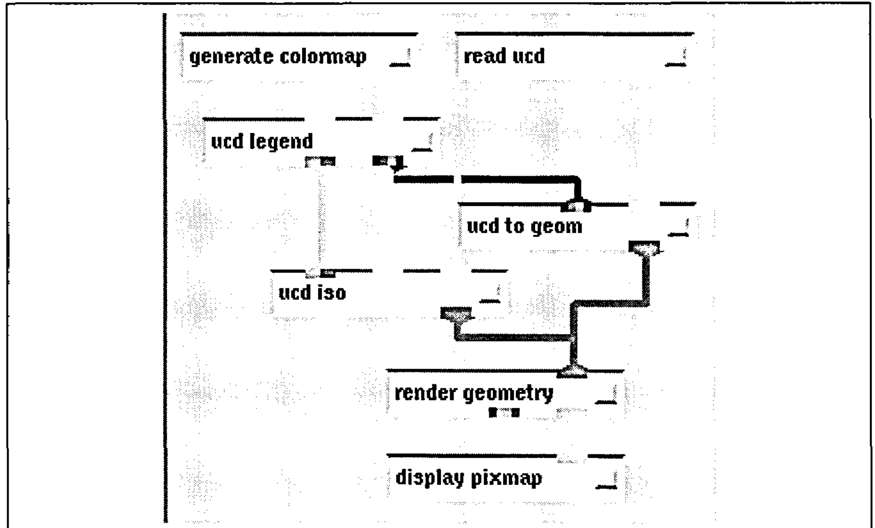
Isosurface (geometry)

A surface that represents the isosurface.

Example

The network in Figure 147 reads in a UCD structure and generates an isosurface for some node value. The **generate colormap** module provides a colormap to color the isosurface, and the module **ucd legend** provides a widget for picking values for the **Level** parameter. The module **ucd to geom** provides a frame within which to view the isosurface. To do this, switch into the Geometry Viewer and change the rendering mode for the geometry object produced by **ucd to geom** to wireframe.

Figure 147
ucd iso module in an example network



Related modules

Modules that can provide the **UCD Structure** input are read ucd and field to ucd.

Module that can provide the **Colormap** and **Information** inputs is ucd legend.

Module that can process **ucd iso**'s output is render geometry.

See also

The sample script UCD ISO demonstrates the **ucd iso** module.

ucd legend

Creates a color legend relating UCD structure data to a color scale

Summary

Name	ucd legend
Type	mapper
Inputs	ucd structure colormap
Outputs	field 1D 3-vector real range

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 colors UCD structures. To do this, it takes in a 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 structures 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 threshold**. Using a dial, you specify a single value or a range of values. 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.

A UCD structure can have 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** choices, you can select a scalar data component for **ucd legend** to use in its color legend.

ucd legend prints the scale representing the range of values associated with the selected node component 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 the title of the color legend.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Colormap (required; colormap)

A 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 has received data, the default "<data *n*>" 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.

value

If the **range** parameter is 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.

lo value
hi value

If the **range** parameter is not 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

Value (range)

ucd legend outputs either a single floating-point value or two values representing the minimum and maximum of a range.

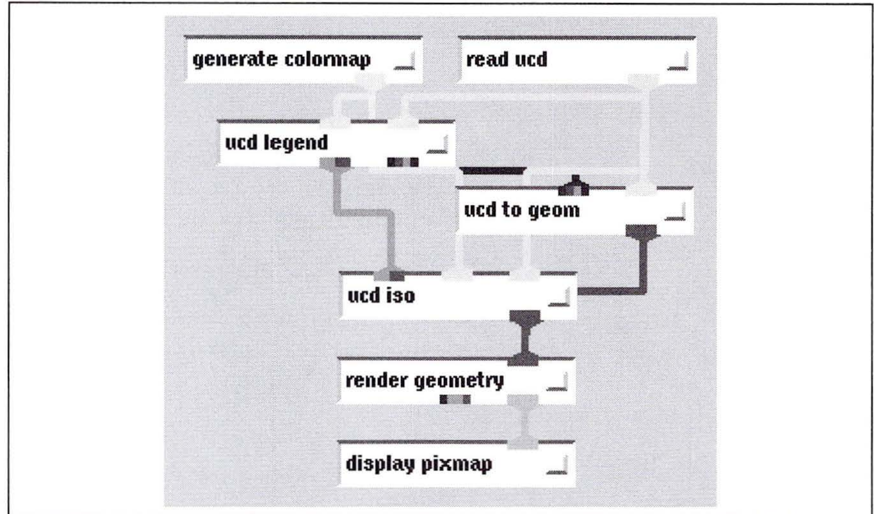
Color Field (field 1D 3-vector real)

The color field is a 1D 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 output is not the same as a ConvexAVS colormap.

Example

The network in Figure 148 reads in a UCD structure. **ucd legend** generates a color field that associates a color with each node in the structure. The color field is then used to color an isosurface of the structure.

Figure 148
ucd legend module in an example network



Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

Module that could provide the **Colormap** input is generate colormap.

Modules that can process **ucd legend**'s output are ucd iso and ucd threshold.

Module that can produce color fields is ucd contour.

See also

The example script UCD THRESHOLD demonstrates the **ucd legend** module.

ucd offset

Deform a UCD structure based on vector values at each node

Summary

Name	ucd offset		
Type	filter		
Inputs	ucd structure		
Outputs	ucd structure		
Parameters	*Name*	*Type*	*Default*
	node data	choice	<data 1>
	offset factor	float	1.0

Description

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. If the nodes of a UCD structure have more than one 3-vector component, then use the **node data** choices 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**.

For example, if a UCD structure has a node component that is a 3-vector of values representing a displacement field in the X-, Y-, Z-directions, **ucd offset** translates the X-, Y-, and Z-location of each node proportional to the displacement field values at that node.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Parameters

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 *n*>" is displayed. If there are no vector components of the node data, **ucd offset** prints an error message. If there are several vector data components, these buttons let you select which component to use in calculating the offset of the UCD structure.

offset factor

A floating-point value that is used to scale the magnitude of the deformation.

Outputs

UCD Structure (ucd structure)

The output structure is the deformed UCD structure.

Example

The network in Figure 149 reads in a UCD structure, deforms it, then displays the result.

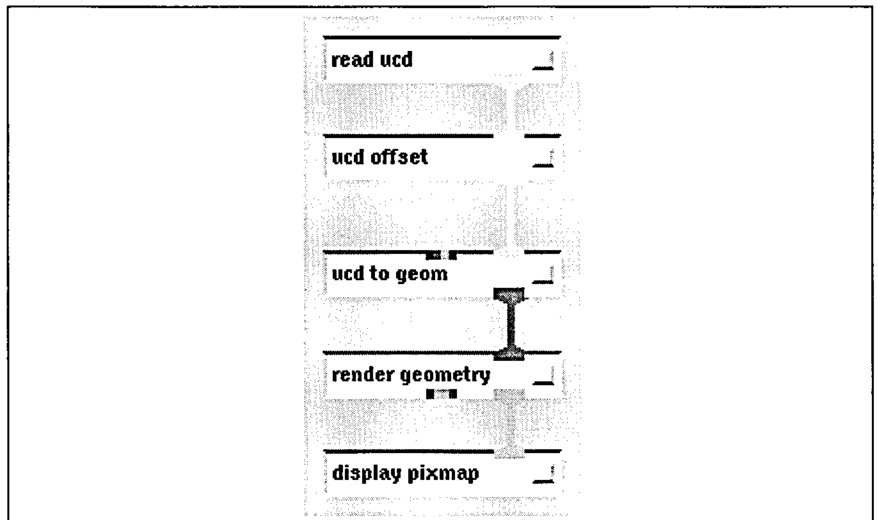


Figure 149
ucd offset module in an example network

Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

Modules that can process **ucd offset**'s output are ucd to geom, ucd crop, ucd threshold, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice 2d, ucd legend, ucd probe, ucd streamline, and write ucd.

See also

The example script UCD OFFSET demonstrates the **ucd offset** module.

ucd print

Create an ASCII readable version of a UCD structure

Summary

Name	ucd print
Type	data output
Inputs	ucd structure
Outputs	none

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Output File	typein	on		
Node Browser	(dummy)	off		
Component	dial	0	0	5
Start	dial	0	0	5
End	dial	5	0	5
Display Mode	radio			

Description

The **ucd print** module creates a human-readable version of the contents of a UCD structure. The information is displayed on the control panel. **ucd print** is useful whenever you need to inspect the actual contents of a UCD structure.

By default, **ucd print** displays header information. The control panel contains a radio-button selector that allows you to display different pieces of the UCD structure to the browser window.

Parameters

Output File

A typein that determines the temporary file used by **ucd print** to cache the browser info. You can change this file if storage in your /tmp directory is a problem for any reason.

Node Browser

This parameter is used to implement the browser screen itself. You do not directly manipulate this parameter.

Component

This parameter selects the data component to display.

Start

Selects the starting cell/node for which to display the data. The node/cell selected will be the first one placed in the browser window.

End

Selects the last cell/node for display. The node/cell selected will be the last one available to the browser window.

Node Display

Selects the display mode. The choices are:

Header — The Header selection consists of the following information: UCD structure 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. 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 the ucd_defs.h include file. The X-,Y-, and Z-extents are the extents of the mesh portion of the UCD structure (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 structure. 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 structure 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 structure.

Node data — The Node data selection displays the data associated with the UCD structure's 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 structure only contains cell data, this information may not be available.

Node pos.	The Node positions selection displays the node positions in XYZ-coordinates. This data is always present in a UCD structure.
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 structure.
cell info	The cell info selection allows you to display the material type, individual cell names, cell types, and `mid_edge` flag for each cell in the UCD structure being examined.
cell_data	The cell_data selection does the same thing for cell data as the node data selection does for node data.

Example

The network in Figure 150 converts some data into a UCD structure, displays the contents of the new UCD structure, and gives you the option of writing the new UCD structure permanently to a file.

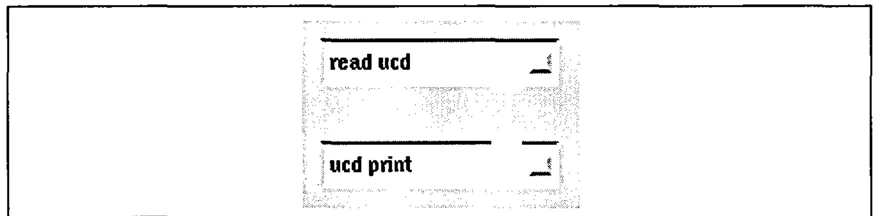


Figure 150
ucd print module in an example network

Related modules

print field and read ucd

Limitations

ucd print writes to the /tmp directory by default. This can cause problems if: (1) there is no /tmp mounted on your system, (2) the /tmp directory does not have much room in it or has inaccessible protections, or (3) the module is being run remotely.

ucd probe

Interactively show numeric data values in a geometry rendered UCD structure

Summary

Name ucd probe

Type mapper

Inputs ucd
field 1D 3-vector real
upstream geometry
field irregular

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
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	2	1	7

Description

The **ucd probe** module displays the numeric data values associated with a specific cell in a UCD structure. It works for structures that have been rendered as geometries. The **ucd probe** module lets you see the values in a UCD structure by simply pointing and clicking on the cell you are interested in if the **Pick Geometry** parameter is set.

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.

You cannot rotate or translate the probe using the middle or left mouse button. This is the case even if probe is selected as the current object in the Geometry Viewer.

To move the probe object through space and have it display the values of UCD cells, press the left mouse button. The probe object will snap to the UCD cell that is below the mouse cursor. This is a point-and-click technique of data sampling. Alternately, if you hold the left mouse-button down as you move the mouse, the probe follows the cursor around the display window, continuously reporting its position and the values of cells it passes over.

The Geometry Viewer tells the **ucd probe** module what UCD cell the mouse cursor was over when the button was pressed. **ucd probe** then reports the data values of the nodes that make up the vertices of the selected cell.

When reporting values for nodes with several data components, **ucd probe** lists the values of all the node components.

ucd probe outputs a geometry object representing the cell that has been selected. 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 the rendering mode for this geometry to lines, you can produce a wireframe representation of the structure.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Color Field (optional; field 1D 3-vector real)

The color field is a 1D 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 that represents the selected UCD cell. Two modules output the color field data type: **ucd contour** and **ucd legend**. The color field is not the same as a colormap.

Upstream Geometry (optional; invisible, autoconnect)

Used by the **ucd probe**'s point-and-click technique, this invisible port is what the **render geometry** 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.

Upstream Transform (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. **ucd probe** takes these locations and displays the data values associated with them.

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 has received data, the default "<data *n*>" 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 a dissecting needle.

Pick Geometry

When selected, lets you see the values in a UCD structure by simply pointing and clicking on the cell you are interested in.

Label Options

label nodes When **label nodes** is selected, **ucd probe** displays the data for all the nodes of the cell that has been clicked on.

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.

label cell 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 *n*>" is displayed. If there is no cell based data in the structure, "<no data>" is displayed on the button.

Text Size

An integer that controls the font size of the output strings.

Outputs

Geometry (geometry)

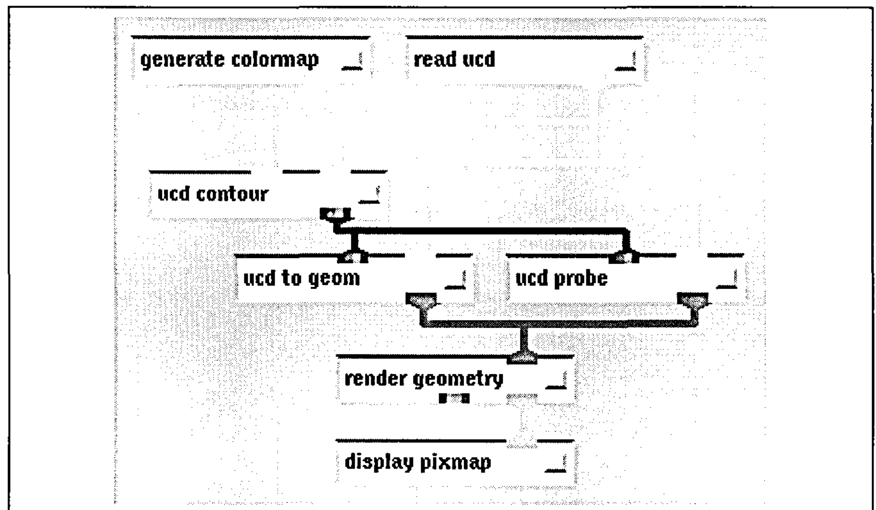
The output geometry has three parts:

- 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 network in Figure 151 reads in a UCD structure, which is passed to **ucd to geom** and **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 lines, you can produce a rendering of the selected cell within a wireframe model of the structure.

Figure 151
ucd probe module in an example network



Related modules

Modules that could provide the **Data Field** input are read ucd and field to ucd.

Modules that could provide the **Color Field** input are ucd contour and ucd legend.

Module that could provide the **Sample Field** input is samplers.

Module that can process **probe** output is render geometry.

See also

The example script UCD PROBE demonstrates the **ucd probe** module.

ucd rslice

Slice away portions of a UCD structure

Summary

Name	ucd rslice
Type	mapper
Inputs	ucd structure 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. To initiate the slicing operation, you must select the **Do Slice** parameter.

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 that 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 **Do Slice** parameter 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

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Color Field (optional; field 1D 3-vector real)

This input field is a 1D 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 is not the same as a 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 that rotates the slice plane around the UCD structure's X-axis.

y-rot

A floating point value that rotates the slice plane around the UCD structure's Y-axis.

Distance

A floating-point value between -2.0 and 2.0 that 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.

Outputs

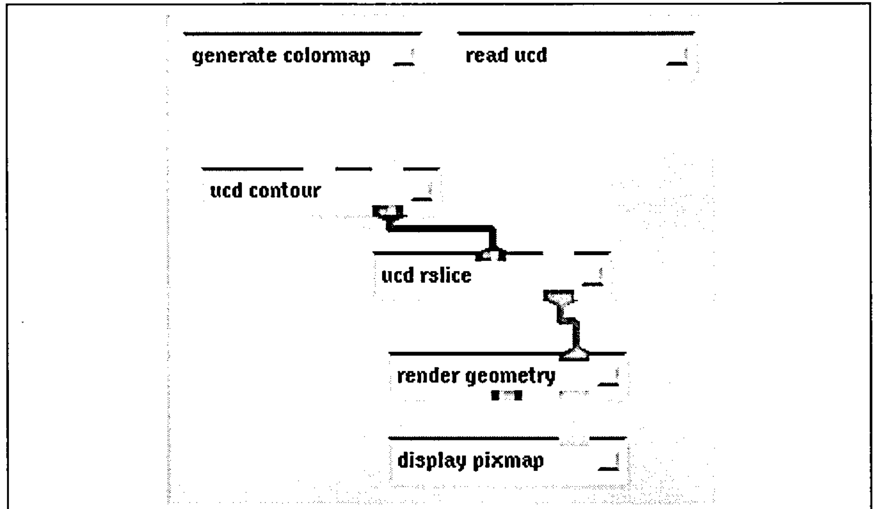
Geometry (geometry)

ucd rslice outputs a geometry that includes the slice plane and the portion of the UCD structure remaining after the slice operation is performed.

Example

The network in Figure 152 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 **render geometry.**

Figure 152
ucd rslice module in an example network



Related modules

Modules that could provide the **UCD Structure** input are read ucd, field to ucd, and ucd extract.

Other modules that subset UCD structures are ucd threshold and ucd crop.

Module that can process **ucd rslice**'s output is render geometry.

See also

The example script UCD RSLICE demonstrates the **ucd rslice** module.

ucd slice 2d

Extract 2D slice from a UCD structure

Summary

Name	ucd slice 2d				
Type	mapper				
Inputs	ucd structure colormap				
Outputs	geometry geometry				
Parameters	*Name*	*Type*	*Default*	*Min*	*Max*
	node data	choice	<data 1>		
	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	on		

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**.

Each time the slice plane is reoriented, the **Do Slice** parameter is turned off. This lets you adjust the slice plane until it is where you want and only then perform the slicing operation. 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. **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.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Colormap (optional; colormap)

This input field is a colormap. There is one color for each node in the input UCD structure.

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 has received data, the default "<data *n*>" is displayed. If there is no node data in the structure, "<no data>" is displayed on the button.

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 extracts the slice.

x-rot

A floating-point value that rotates the slice plane around the UCD structure's X-axis.

y-rot

A floating-point value that rotates the slice plane around the UCD structure's Y-axis.

Distance

A floating-point value between -1.0 and 1.0 that 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.

Transform Slice

When this is 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 **render geometry** module. It must be turned on when **ucd slice 2d** is sending its slice plane output to one **render geometry** module and its 2D slice output to another **render geometry** module.

Outputs

Geometry (geometry)

The geometry object that **ucd slice 2d** outputs from the left output port represents the 2D slice of the UCD structure.

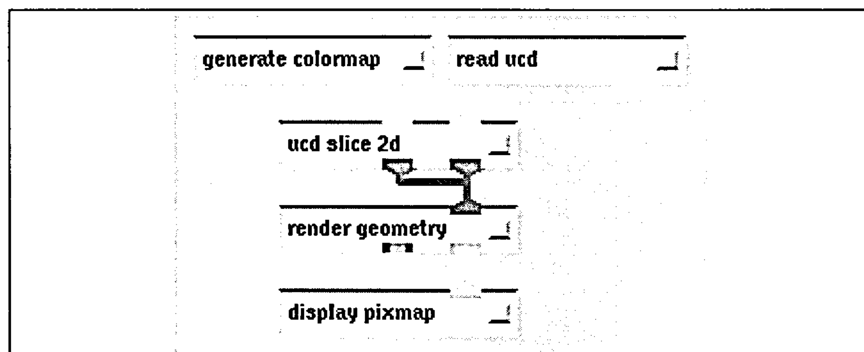
Geometry (geometry)

The geometry object that **ucd slice 2d** outputs from the right output port represents the slice plane.

Example

In Figure 153, **ucd slice 2d** sends both the slice plane output and the 2D slice output to a single **render geometry** module.

Figure 153
ucd slice 2d module in an example network



Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

Modules that could provide the **Colormap** input are ucd contour and ucd legend.

Module that can process **ucd slice 2d**'s output is render geometry.

See also

The example script UCD SLICE 2D demonstrates the **ucd slice 2d** module.

ucd streamline

Generate stream lines for a UCD structure with vector node data

Summary

Name ucd streamline

Type mapper

Inputs ucd structure
colormap
upstream transform

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
node data	choice	<data 1>		
N Segment	integer	16	2	64
choice	choice	point		
N Steps	integer	2	2	10
Integration	choice	1st Order		
Backward	boolean	off		
Color Streams	boolean	off		
Interaction Mode	choice	Immidiate		
Start Streams	boolean	off		

Description

The **ucd streamline** module generates colored streamlines based on the vector node data in a UCD structure. It places a sample of points at a starting location in the UCD structure. The number of points is parameter-controlled. The orientation of the points is controlled by the sample probe, which can be moved around in space like any other geometry object.

To initiate the calculation of **stream lines**, you must select the **Start Streams** parameter. To move the probe, select it by clicking on it, or by entering the Geometry Viewer and making it the current object. After each run through, the probe will be hidden from view. Moving it or changing a parameter will make it visible again.

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. If the nodes of a structure have more than one 3-vector component, use the **node data** choices to select which component to use in calculating the stream lines.

The vectors at each node can be viewed as exerting force on the 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 vectors at the nodes of the UCD structure.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format. It must have at least one node component that is a 3D vector, representing the components of a velocity vector.

Colormap (required; colormap)

A colormap that is used by **ucd streamline** to associate colors with vector values. This is a regular colormap and not the color field output by **ucd contour** and **ucd legend**.

Upstream Transform (optional; invisible, autoconnect)

When the **ucd streamline** module coexists with the **render geometry** module in a network, **render geometry** 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 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 has received data, the default "<data *n*>" 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.

choice

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 increments for which stream lines are computed within each cell of the UCD structure. As the number of segments increases, so does the accuracy of the stream lines.

Integration

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
- **3rd Order** uses a 3rd order Runge-Kutta method

Backward

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

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.

Interaction Mode

Immidiate See immediate updates as you alter parameters or move the probe. This is the default.

Wait Wait until parameters are set or the probe is moved. The **Start Streams** parameter needs to be selected before seeing any results.

Button Up Wait until the probe is moved with the cursor and you release the mouse button before streams start. This is an abbreviated version of the **Wait/Start Streams** combination.

Start Streams

A parameter that initiates the calculation of stream lines. This button (along with **Wait**) allows you to manipulate the sample probe until you are satisfied with its position and only then begin computing stream lines.

Outputs

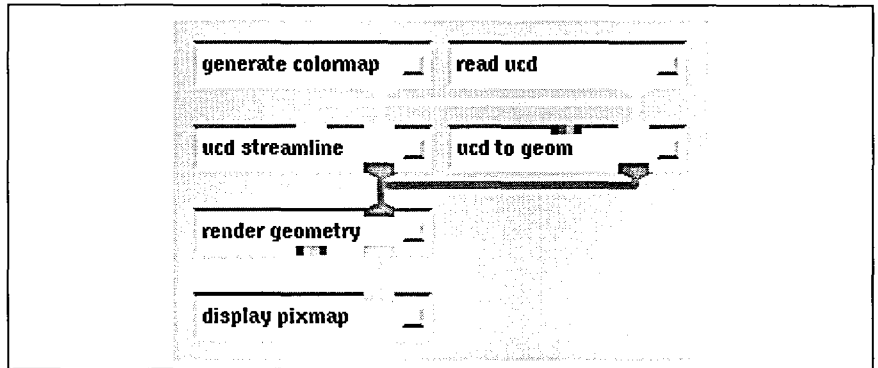
Stream Lines (geometry)

A set of colored disjoint lines.

Example

The network in Figure 154 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. The module **ucd to geom** provides a frame within which to view the streamlines. To do this, switch to the Geometry Viewer and change the rendering mode for the geometry object produced by **ucd to geom** to wireframe.

Figure 154
ucd streamline module in an example network



Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

Module that could provide the **Colormap** input is generate colormap.

Module that can process **ucd streamline**'s output is render geometry.

See also

The example script UCD STREAMLINE demonstrates the **ucd streamline** module.

ucd threshold

Restrict values in a UCD structure

Summary

Name	ucd threshold		
Type	filter		
Inputs	ucd structure values		
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 that **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 cutoff 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

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Value (required; values)

ucd threshold must receive input from **ucd 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

UCD Structure (ucd structure)

The output structure is the threshold filtered UCD structure.

Example

The network in Figure 155 reads in a UCD structure. The structure is passed to the **ucd 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. **ucd legend** also outputs a color field that gets passed to **ucd to geom** so that the data is colored.

Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

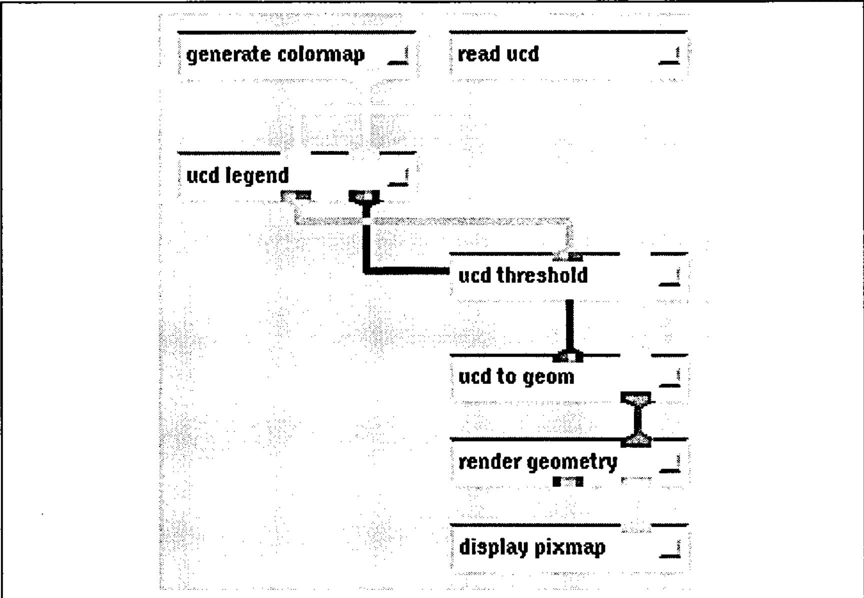
Module that provides the **Value** input is ucd legend.

Modules that can process **ucd threshold**'s output are ucd to geom, ucd crop, ucd anno, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice 2d, ucd probe, ucd streamline, and write ucd.

See also

The example script UCD THRESHOLD demonstrates the **ucd threshold** module.

Figure 155
ucd threshold module in an example network



ucd to geom

Convert a UCD structure into a geometry

Summary

Name ucd to geom

Type mapper

Inputs ucd structure
field 1D 3-vector real

Outputs geometry

Parameters

Name	*Type*	*Default*	*Min*	*Max*
Shrink	boolean	off		
Shrink Factor	integer	10	0	100
Geometry Display Mode	choice	External Faces		

Description

ucd to geom converts a UCD structure into a geometry that can be rendered using the **render geometry** 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 or thousands of nodes and cells, many of which are likely to be interior and hidden. You can use the **Geometry Display Mode** parameter 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.

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.

ucd to geom can receive an optional color field through its left input port. The color field is an array of color values—one color for each node in the input UCD structure. This results in the structure being rendered as a colored geometry object. The color field can be generated by the modules **ucd contour** or **ucd legend**.

Inputs

UCD Structure (required; ucd structure)

The input structure is in UCD format.

Color Field (optional; field 1D 3-vector real)

The color field is a 1D 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**. This not the same as a ConvexAVS colormap.

Parameters

Shrink

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.

Geometry Display Mode

External Faces	Show exterior faces.
External Node Faces	Show exterior node faces.
All Faces	Show all faces (this allows mid-edge nodes to be displayed).

Outputs

Geometry (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

The network in Figure 156 shows **ucd to geom** in a sample network.

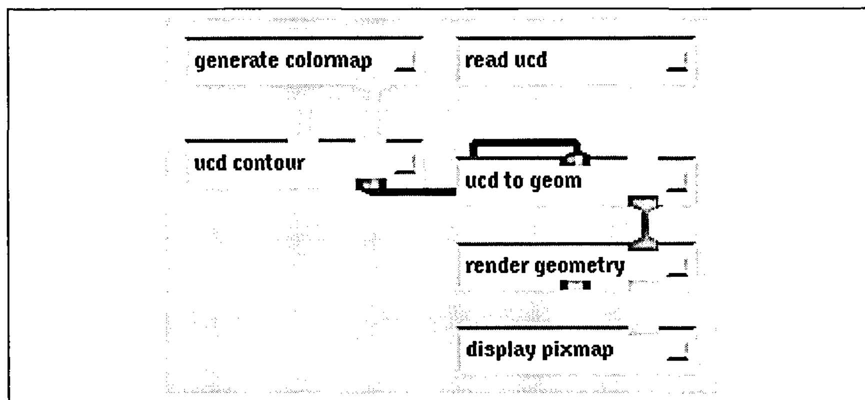


Figure 156
ucd to geom module in an example network

Related modules

Modules that could provide the **UCD Structure** input are read ucd, field to ucd, and ucd extract.

Modules that could provide the **Color Field** input are ucd contour and ucd legend.

Module that can process **ucd to geom**'s output is render geometry.

See also

The example scripts READ UCD, UCD THRESHOLD, and UCD CROP demonstrate the **ucd to geom** module.

ucd tracer

Perform ray-traced volumetric rendering on a UCD structure

Summary

Name ucd tracer

Type mapper

Inputs ucd structure
field 2D scalar float
colormap

Outputs field 2D 4-vector byte

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. By default, **ucd tracer**'s **exterior** parameter is on, and only an object's surface is ray-traced.

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, it travels no further, and the pixel associated with that ray is colored red. If the tetrahedron is nearly transparent, it confers only a fraction of its color to the pixel while 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; ucd structure)

The input structure is in UCD format. The structure's cells must be tetrahedrons.

Tracker Information (optional; field 2D scalar float)

The middle input port on **ucd tracer** can receive a 4 by 4 transformation matrix describing rotations and translations to apply to the UCD structure. The matrix can come from the module **euler transformation** or **display tracker**. This allows you to rotate the structure in 3-space.

Colormap (required; colormap)

A colormap that is used by **ucd tracer** to associate colors with UCD node values. This is a ConvexAVS colormap and not the color field output by **ucd contour** and **ucd legend**.

Parameters

size

Value that 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

A floating-point value that is multiplied by the alpha value of every node in the structure. This determines how transparent 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

If **exterior** is selected, then only the surface of the UCD structure is ray-traced. This is the default.

Outputs

Image (field 2D 4-vector byte)

The output field is an image.

Example

The network in Figure 157 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.

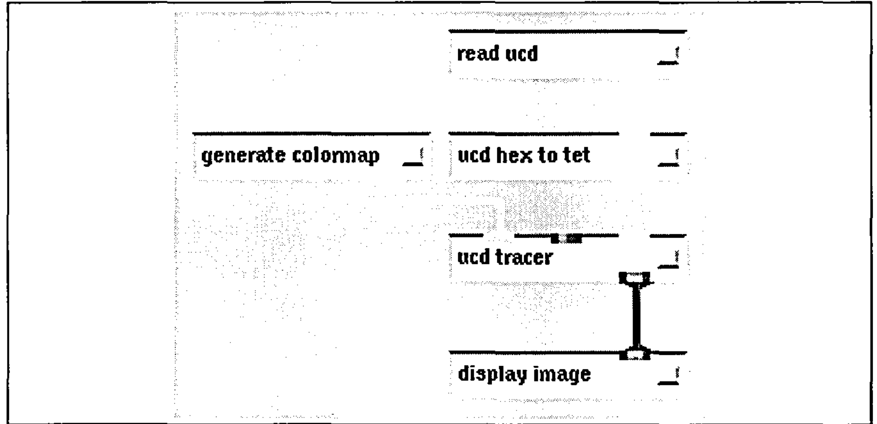


Figure 157
ucd tracer module in an example network

Related modules

Modules that could provide the **UCD Structure** input are read ucd and ucd hex to tet.

Modules that can process **ucd tracer**'s output are display tracker, display image, and image viewer.

See also

The example script UCD TRACER demonstrates the **ucd tracer** module.

ucd vecmag

Compute the magnitude of a vector UCD structure

Summary

Name	ucd vecmag
Type	filter
Inputs	ucd structure
Outputs	ucd structure
Parameters	none

Description

The **ucd vecmag** module accepts a vector UCD structure as input and computes the magnitude of each vector data value. The output is a scalar UCD structure consisting of the magnitudes.

The magnitude equation is:

$$Magnitude(x,y,z) = \sqrt{(V_x(x,y,z) \cdot V_x(x,y,z)) + (V_y(x,y,z) \cdot V_y(x,y,z)) + (V_z(x,y,z) \cdot V_z(x,y,z))}$$

Inputs

UCD Structure (required; ucd structure)

The input UCD structure must represent a volume of elements, with a 3D vector of floating-point data values for each node.

Outputs

UCD Structure (ucd structure)

The output UCD structure has a single floating-point value for each input node.

Example

The network in Figure 158 reads in a 3D vector UCD structure and computes the magnitude of the vectors.

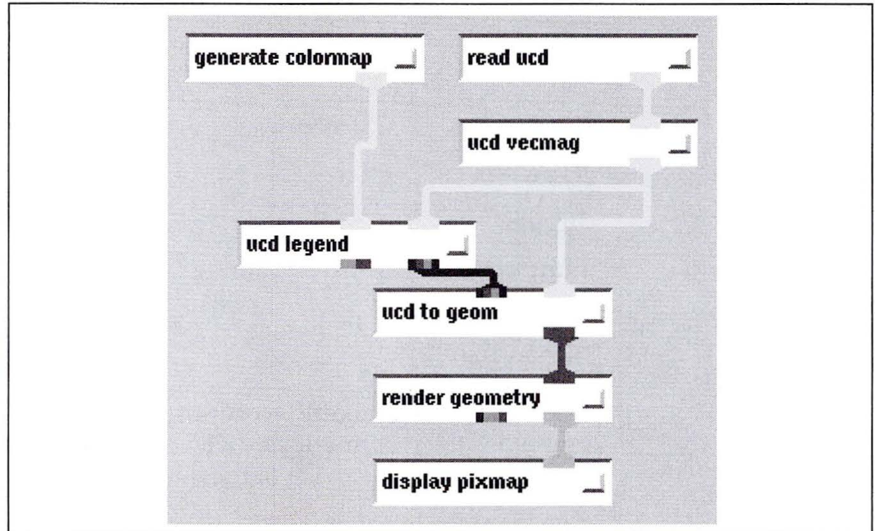


Figure 158
ucd vecmag module in an example network

Related modules

vector mag

Limitations

This module works only with 3D UCD structure node data.

vbuffer

Perform volumetric rendering on volume data

Summary

Name	vbuffer
Type	data output
Inputs	field 3D uniform scalar colormap
Outputs	pixmap field 2D uniform 4-vector byte

Parameters

Name	*Type*	*Default*	*Min*	*Max*
X Rotate	real	0.0	-180	180
Y Rotate	real	0.0	-180	180
Z Rotate	real	0.0	-180	180
ZTranslate	real	-2.5	-10	10
Scale X	real	0.5	0.0	5.0
Scale Y	real	0.5	0.0	5.0
Scale Z	real	0.5	0.0	5.0
Image Size	int	200	10	1024
Background	real	0.0	0.0	1.0
Cell-Based	logical	off	off	on
Re-use Scripts	logical	on	off	on
Script File	string	/tmp/vbuf.inp		
Scalar File	string	/tmp/vbuf.dat		
Image File	string	/tmp/vbuf.image		

Description

The **vbuffer** module creates a volumetric rendering of a 3D uniform scalar field, using RGB color and opacity transfer functions. The technique employed uses two levels of interpolation:

- A cell-averaged or voxel representation produces lower quality and lower apparent resolution images at moderate execution speeds.
- A trilinear or cell-based interpolation results in very high quality images with a slower execution time.

The **vbuffer** module is actually a wrapper for `vbuf`, which performs the image computation:

- **vbuffer** places the parameters and transfer functions in a file for `vbuf` to read.
- **vbuffer** executes `vbuf` and waits for it to exit.

- `vbuf` reads the data from the file, computes an image, writes it to a file, and exits.
- **vbuffer** reads the image that was written to a file and places it on the output port.

You can choose between two rendering algorithms:

Trilinear interpolation: This algorithm uses an inverse-mapping scheme to generate images. The 3D field data is decomposed into 8-node cells, with one field value at each vertex of the cell. Each cell is processed by accumulating color and opacity along an integration volume that maps into one or more pixels.

At several locations through this volume, both the value of the scalar field and its gradient are interpolated. The sampled scalar field value is used to map into transfer functions, which determine the color and opacity at the point. The color for the pixel is determined by a product of the diffuse illumination (due to the dot product of the light source and gradient vectors), the opacity, and the sampled color as accumulated along the volume. After all the pixels that the cell projects into are processed, the algorithm moves on to the next cell. Partial pixel contributions are saved into an in-memory frame buffer.

Voxel Approximation: The 3D field is decomposed into cells, as described above. But no interpolation is performed within the cell. Each cell has a single opacity, color, texture color, surface gradient, and set of shading parameters. This method is much faster than the Trilinear Interpolation method. Use it to get a quick (albeit ragged) look at the data. It is most useful for selecting the opacity and color transfer functions. This is the default.

This module is in the unsupported library.

Inputs

Data Field (required; field 3D uniform scalar)

The input field must be 3-dimensional. The data for each field element must be a scalar.

Colormap (required; colormap)

Any colormap.

Parameters

X Rotate

The X-rotation of the data field in degrees.

Y Rotate

The Y-rotation of the data field in degrees.

Z Rotate

The Z-rotation of the data field in degrees.

Z Translate

The Z-translation of the field. The coordinate system is right handed, so only that data which has a negative Z-coordinate will be visible.

Scale X

The X-scale factor of the data field. By default the highest resolution axis of the data is scaled between -1.0 and 1.0, and other axes are scaled accordingly by this.

Scale Y

The Y-scale factor of the data field.

Scale Z

The Z-scale factor of the data field.

Image Size

The resolution in pixels of the computed image. The image is always square. The algorithm complexity scales with the number of pixels and the number of cells that have a non-zero opacity.

Background

The background brightness. Zero corresponds to a black background, 1.0 to a white background.

Cell-Based

A toggle switch between the Voxel Approximation algorithm and the Trilinear Interpolation algorithm.

Re-use Scripts

vbuffer generates a script to run `vbuf`. This toggle switch allows you either to reuse these scripts, data files and images or to build new scripts and files with each new invocation.

Script File

The name of the `vbuf` script file.

Scalar File

The name of the `vbuf` data input file.

Image File

The name of the `vbuf` output image file.

Outputs

Pixmap (pixmap)

A pixmap containing the 2D image rendered by **vbuffer**.

Data Field (field 2D uniform 4-vector byte)

The output is a 2D image rendered by **vbuffer**.

Limitations

Only uniform fields can be rendered by **vbuffer**.

Some image anomalies can occur along cell boundaries. This is view-orientation dependent.

The execution time can be dramatically reduced by limiting the number of cells that contain a nonzero amount of opacity.

Related modules

Module that could provide the **Data Field** input is read volume.

Modules that could be used in place of **vbuffer** are isosurface and dot surface.

Module that can process **vbuffer** output is display pixmap.

vector PLOT3D

Calculate derived PLOT3D vector functions

Summary

Name	vector PLOT3D
Type	filter
Inputs	field 3D 3-space irregular 5-vector real field 3D scalar uniform integer field 1D scalar uniform float
Outputs	field 3D 3-space irregular 3-vector real
Parameters	*Name* *Type* func name choice

Description

This module allows you to visualize some of the vector fields that can be derived from the basic PLOT3D data. The source code is designed so that each of the functions is implemented separately from the ConvexAVS flow. Each of the functions and their names (for the select button) are stored in a table. In this way, you can extend the functionality of this module beyond a list of functions implemented in the standard PLOT3D viewer by adding new functions and names. The source for this module is in the /usr/avs/examples/convex/plot3d/modules directory.

Inputs

Data Field (required; field 3D 3-space irregular 5-vector real)

This input data is the 5-vector field output by **read PLOT3D**. It is the most important field because it contains the mesh and solution data.

Blanking Data Field (optional; field 3D scalar uniform integer)

This field contains the blanking records. If a PLOT3D data set contains blanking records and this is not connected, unpredictable results may occur.

Global Parameters (required; field 1D scalar uniform float)

This field contains the four global parameters (free stream Mach number, angle-of-attack (alpha), Reynolds number, and the time). Not all derived vector functions make use of this data, but it must be connected so that those derived vector functions that do require the information can be calculated.

Outputs

Vector Field (field 3D 3-space irregular 3-vector real)

A vector field representing the function selected.

Parameters

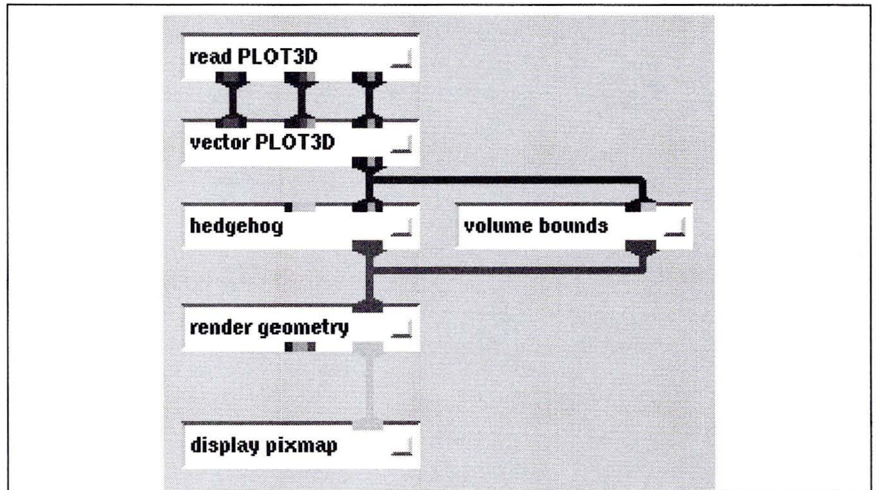
func name

A choice parameter that selects the desired vector function of the PLOT3D values to calculate. Implemented functions are: velocity and Perturbation Velocity.

Example

The example in Figure 159 reads in a PLOT3D data set and does a hedgehog on one of the derived vector fields.

Figure 159
vector PLOT3D module in an example network



Related modules

density PLOT3D, momentum PLOT3D, stagnation PLOT3D, read PLOT3D, and scalar PLOT3D

Related programs

export_PLOT3D and import_PLOT3D

vector curl

Compute the curl of a vector field

Summary

Name	vector curl
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D 3-vector float uniform
Parameters	none

Description

The **vector curl** module accepts a vector field as input and computes the curl of that field as output. This is related to the divergence as follows:

$$\mathrm{curl}\boldsymbol{F} = \nabla \times \boldsymbol{F}$$

where F is the vector input field.

The equations used to compute the curl are:

$$newV_x(x, y, z) = (V_z(x, y+dy, z) - V_z(x, y-dy, z)) - (V_y(x, y, z+dz) - V_y(x, y, z-dz))$$

$$newV_y(x, y, z) = (V_x(x, y, z+dz) - V_x(x, y, z-dz)) - (V_z(x+dx, y, z) - V_z(x-dx, y, z))$$

$$newV_z(x, y, z) = (V_y(x+dx, y, z) - V_y(x-dx, y, z)) - (V_x(x, y+dy, z) - V_x(x, y-dy, z))$$

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 3-vector float uniform)

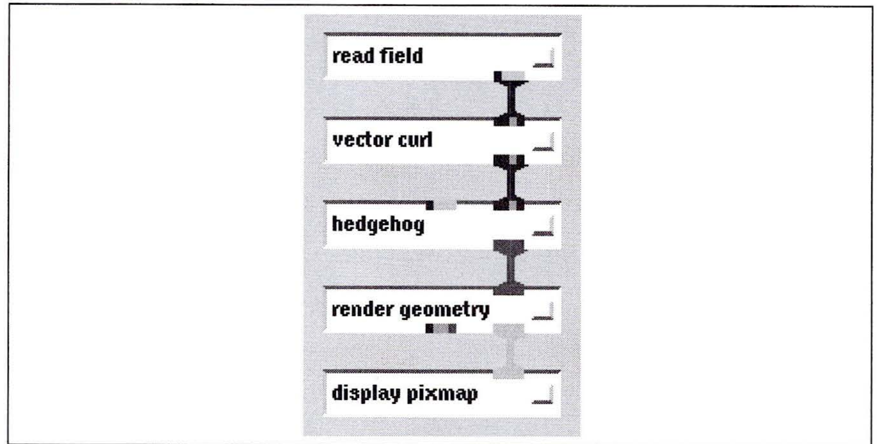
The output field is in the same format as the input field.

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

Example

The network in Figure 160 shows **vector curl.**

Figure 160
vector curl module in an example network



Related modules

gradient shade and 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

Compute the divergence of a vector field

Summary

Name	vector div
Type	filter
Inputs	field 3D 3-vector float uniform
Outputs	field 3D scalar float uniform
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:

$$\mathrm{div}\boldsymbol{F} = \nabla \bullet \boldsymbol{F}$$

where *F* is the vector input field.

The equations used to compute the divergence are:

$$\begin{aligned} Divergence(x, y, z) = {} & (V_x(x + dx, y, z) - V_x(x - dx, y, z)) + \\ & (V_y(x, y + dy, z) - V_y(x, y - dy, z)) + \\ & (V_z(x, y, z + dz) - V_z(x, y, z - dz)) \end{aligned}$$

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)

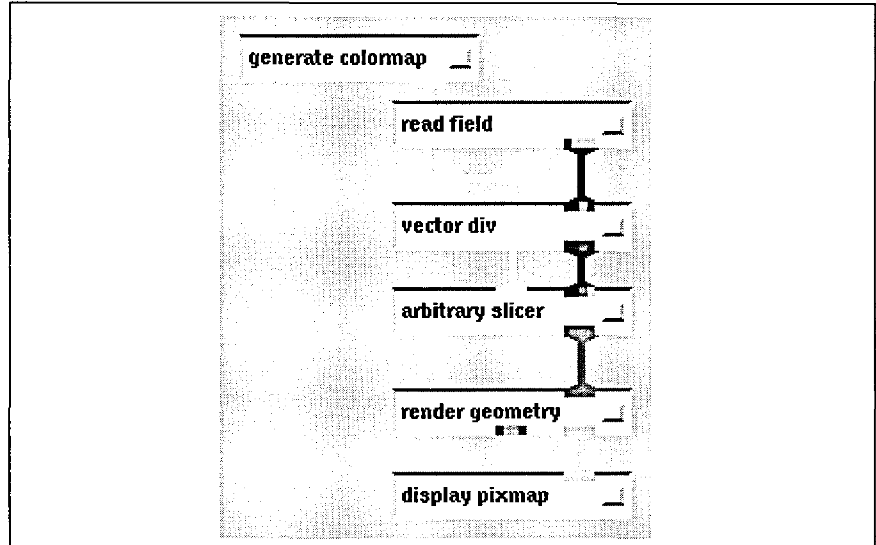
The output field has a single floating-point value for each input field element.

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

Example

The network in Figure 161 reads in a 3D vector field and computes its divergence.

Figure 161
vector div module in an example network



Related modules

vector curl, vector div, vector norm, vector mag, hedgehog, and stream lines

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.

vector grad

Compute the vector gradient of a 3D scalar field

Summary

Name	vector grad
Type	filter
Inputs	field 3D scalar float uniform
Outputs	field 3D 3-vector float uniform
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 shown in Figure 162.

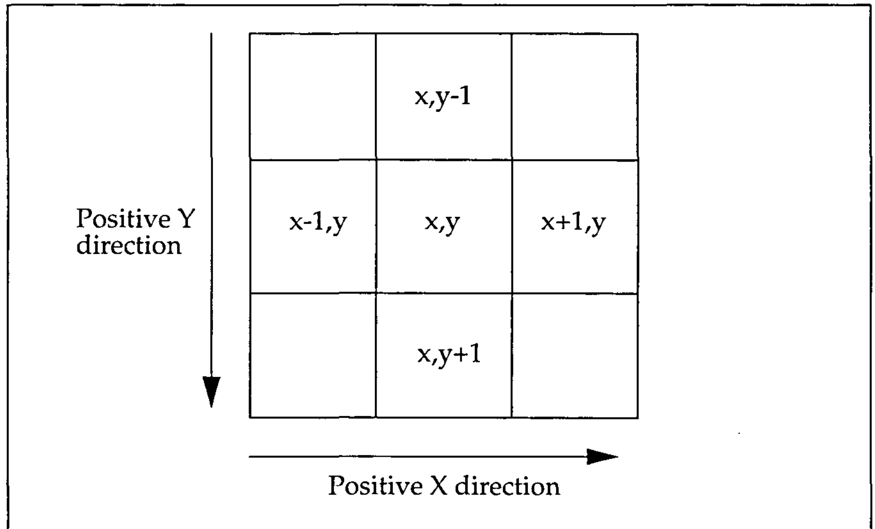


Figure 162
Computing the gradient

$$\Delta x(x, y, z) = data(x+1, y, z) - data(x-1, y, z)$$

$$\Delta y(x, y, z) = data(x, y+1, z) - data(x, y-1, z)$$

$$\Delta z(x, y, z) = data(x, y, z+1) - data(x, y, z-1)$$

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**. **vector grad** followed by **vector norm** produces the same results as **compute gradient**.

Inputs

Data Field (field 3D scalar float uniform)

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 uniform)

The output field has a 3D vector of floating-point data values for each element.

Example

The network in Figure 163 reads a 3D scalar field, computes its gradient, and then uses the **hedgehog** module to display the resulting vector field.

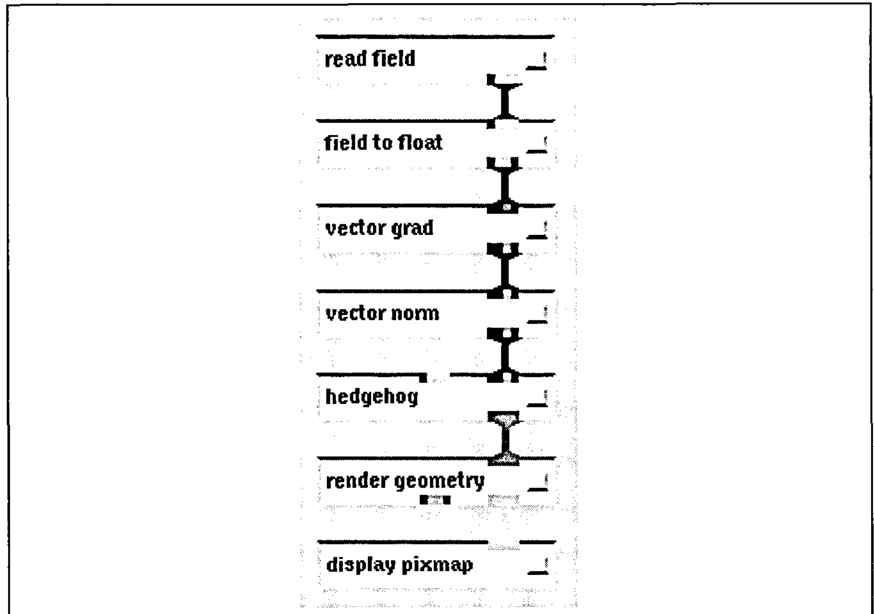


Figure 163
vector grad module in an example network

Related modules

vector curl, vector div, vector norm, vector mag, hedgehog, particle advector, and stream lines

Limitations

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

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

Compute the magnitude of a vector field

Summary

Name	vector mag
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:

$$Magnitude(x, y, z) = \sqrt{(V_x(x, y, z) \cdot V_x(x, y, z)) + (V_y(x, y, z) \cdot V_y(x, y, z)) + (V_z(x, y, z) \cdot V_z(x, y, z))}$$

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)

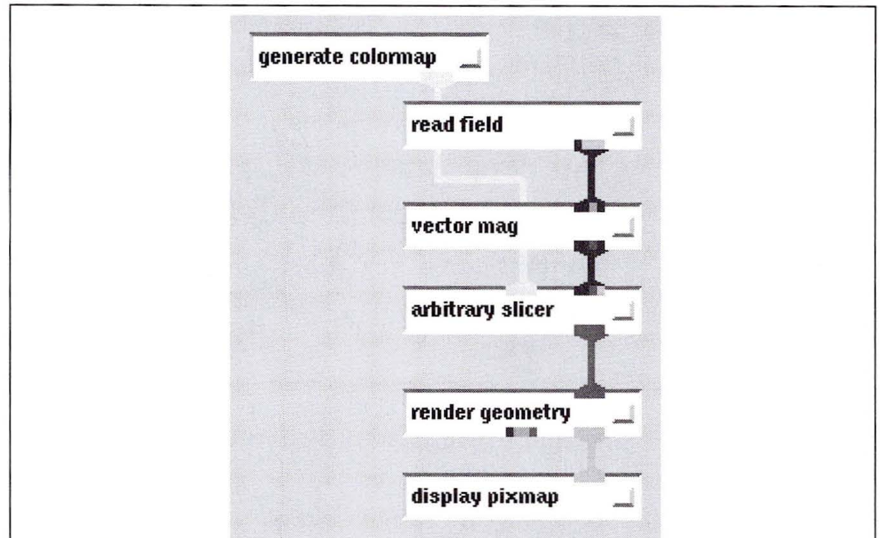
The output field has a single floating-point value for each input field element.

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

Example

The network in Figure 164 reads in a 3D vector field and computes the magnitude of the vectors.

Figure 164
vector mag module in an example network



Related modules

vector curl, vector div, vector norm, vector mag, hedgehog, particle advector, gradient shade, and stream lines

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.

vector norm

Normalize a vector field

Summary

Name	vector norm
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 is:

$$Magnitude(x, y, z) = \sqrt{(V_x(x, y, z) \cdot V_x(x, y, z)) + (V_y(x, y, z) \cdot V_y(x, y, z)) + (V_z(x, y, z) \cdot V_z(x, y, z))}$$

$$newV_x = \frac{V_x}{Magnitude}$$

$$newV_y = \frac{V_y}{Magnitude}$$

$$newV_z = \frac{V_z}{Magnitude}$$

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 network in Figure 165 reads a 3D scalar field, computes its gradient, then uses the **hedgehog** module to display the resulting vector field.

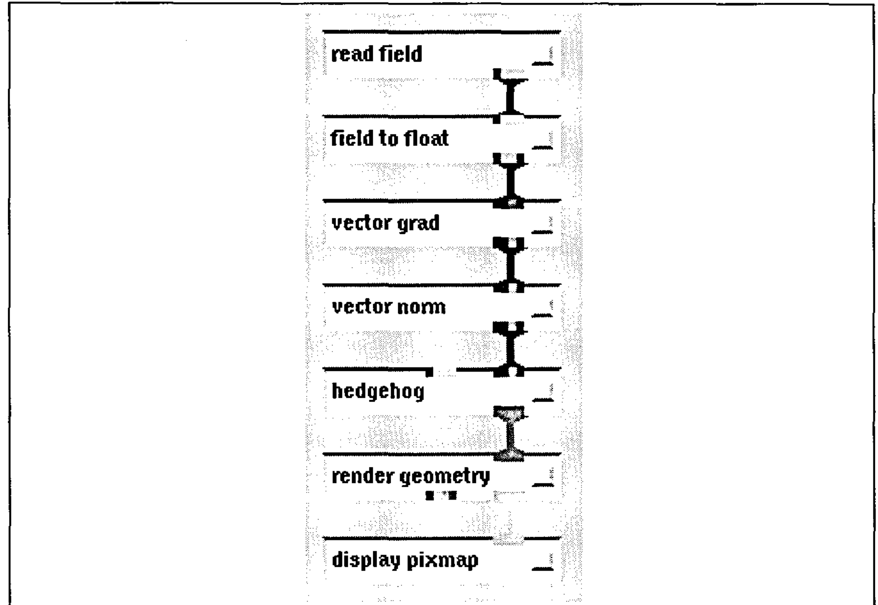


Figure 165
vector norm module in an example network

Related modules

vector curl, vector div, vector norm, vector mag, hedgehog, particle advector, gradient shade, and stream lines

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.

volume bounds

Generate bounding box of 3D 3-vector field

Summary

Name	volume bounds	
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

Description

The **volume bounds** module generates lines that indicate the bounding box of a 3D data set. It is frequently used in conjunction with other geometry-based volume-visualization modules (for example, **bubbleviz, isosurface, hedgehog, arbitrary slicer**) because it provides some volumetric context for the data.

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

Bounding Box (geometry)

The output geometry consists of the lines that form the bounding box.

Parameters

Hull

Draws lines for the perimeter of the data set.

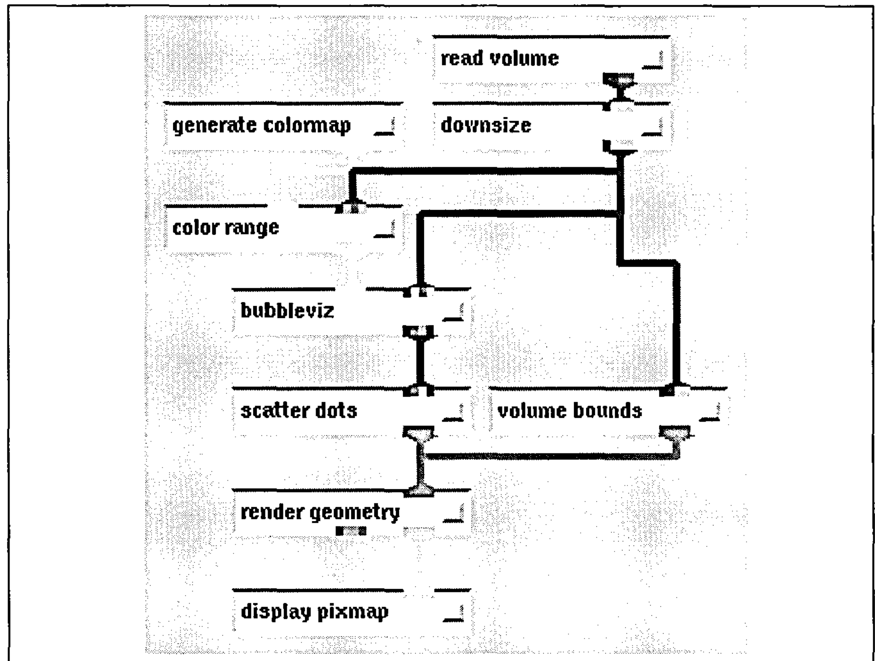
Min I
Max I
Min J
Max J
Min K
Max K

These toggle switches provide further help in 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.

Example

The network in Figure 166 shows a usage of **volume bounds**.

Figure 166
volume bounds module in an example network



Related modules

read volume and volume manager

See also

The example scripts HEDGEHOG and PROBE demonstrate the **volume bounds** module.

volume manager

Share volumes among subnetworks

Summary

Name	volume manager		
Type	data input		
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 "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 **read volume** for a description of the volume format.

This module is in the unsupported library.

Parameters

VOLUMGR Select

A choice that determines how newly-read volumes will be placed into 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 a volume file to read.

Volume Choices

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

Outputs

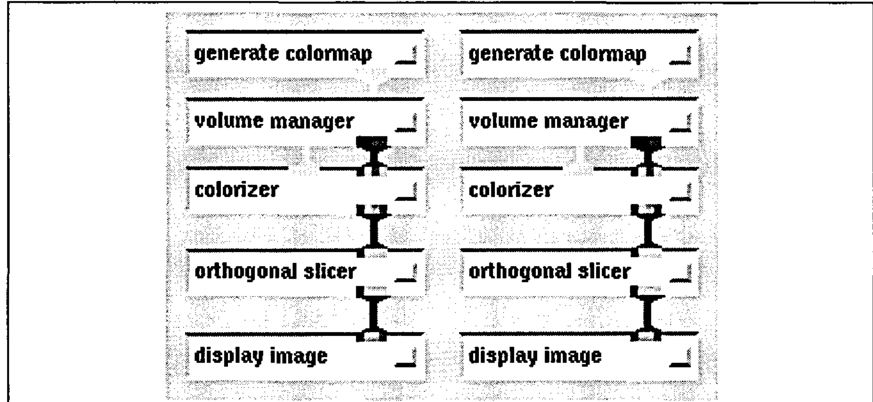
Data Field (field 3D scalar byte)

The output is the byte data cast as the scalar data in a 3D field.

Example

The networks in Figure 167 show how **volume manager** could be used to display two volumes.

Figure 167
volume manager module in example networks



Related modules

Modules that can input colormaps are generate colormap and read colormap.

Modules that can filter data are 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, and gradient shade.

Modules that can map data are dot surface, arbitrary slicer, bubbleviz, orthogonal slicer, field to mesh, isosurface, and volume bounds.

Modules that can render data are display image and render geometry.

Limitations

The cached volumes are not freed until all **volume manager** modules are destroyed. Because volume data can be large, caching multiple volume data sets can use a lot of memory.

wireframe

Convert object from surface to wireframe representation

Summary

Name	wireframe
Type	filter
Inputs	geometry
Outputs	geometry
Parameters	none

Description

The **wireframe** module transforms a 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 geometry created with the geometry library or produced by another module.

Outputs

Geometry (geometry)

A geometry that represents the same object as the input data.

Examples

1.

The network in Figure 168 shows the use of **wireframe** to generate a wireframe version of a polygonal object.

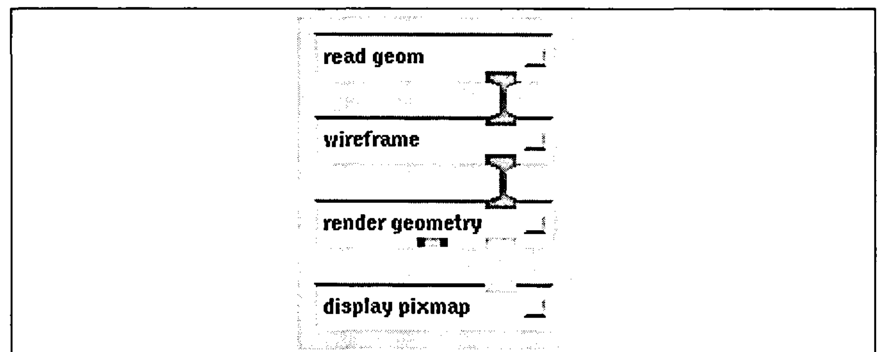
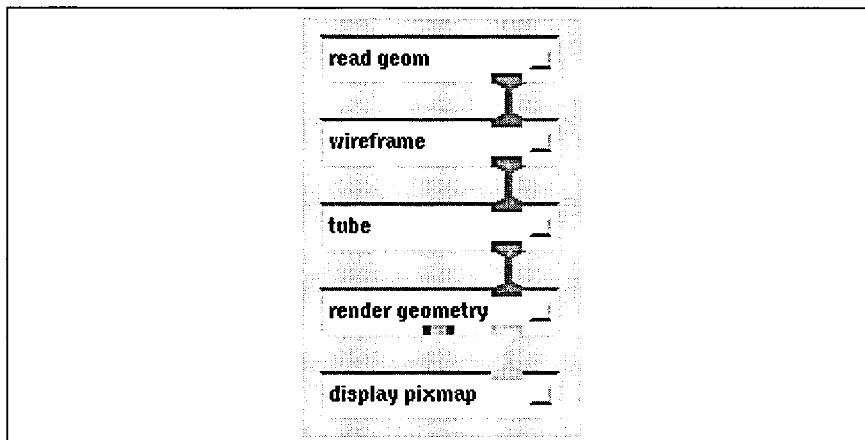


Figure 168
wireframe module in an example network

2.

The network in Figure 169 uses **wireframe** and **tube** to have a geometry involving spheres drawn with cylinders instead of lines.

Figure 169
wireframe module in an example network



Related modules

read geom, offset, shrink, flip normal, tube, and 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

Write a field description to disk

Summary

Name	write field	
Type	data output	
Inputs	field *any-dimension n*-vector *any-data any-coordinates*	
Outputs	none	
Parameters	*Name*	*Type*
	Write Field Browser	browser

Description

The **write field** module writes a 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 on the **read field** module reference.

Inputs

Data Field (field *any-dimension n*-vector *any-data any-coordinates*)

The input can be any field.

Parameters

Write Field Browser

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 or cancel the operation.

After the field file is written, the file name 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 file name.

Examples

1.

Figure 170 shows an example of a file produced by **write field**. The field has three dimensions: 64 by 64 by 64. There is a single byte at each point, and the field is uniform.

Figure 170
Example write field file

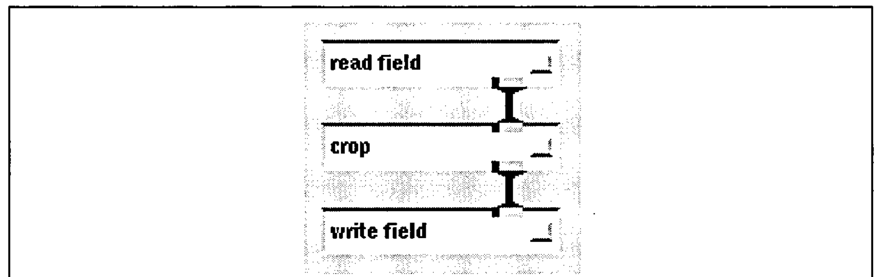
```
ndim=3 # number of dimensions in the field
dim1=64 # dimension of axis 1
dim2=64 # dimension of axis 2
dim3=64 # dimension of axis 3
nspace=3 # number of physical coordinates per point
veclen=1 # number of components at each point
data=byte # data type (byte, integer, float, double)
field=uniform # field type (uniform, rectilinear, irregular)

262,144 bytes of data
```

2.

The network in Figure 171 reads in a field, crops it, and then writes the resultant field to a file.

Figure 171
write field module in an example network



Related modules

Data input modules that produce field output are image manager, read field, read image, read volume, and volume manager.

Filter modules that produce field output are clamp, geom to scatter, colorizer, gradient shade, combine scalars, histogram stretch, compute gradient, interpolate, contrast, mirror, crop, threshold, dot surface, transpose, downsize, vector curl, extract scalar, vector div, field to byte, vector grad, field to double, vector mag, field to float, vector norm, and field to int.

Mapper modules that produce field output are bubbleviz, orthogonal slicer, and pixmap to image.

Limitations

write field produces an error message if it cannot open the file or if there is not enough space to write the complete file.

See also

The example script WRITE FIELD demonstrates the **write field** module.

write frame seq

Write image sequences to a file

Summary

Name write frame seq

Type data output

Inputs field 2D 4-vector byte uniform

Outputs none

Parameters

Name	*Type*	*Default*	*Values*
status	text		
current frame	integer	0	
erase to end	oneshot		
erase all	oneshot		
delete frame	oneshot		
new frame	radio	append	append insert replace off
Write Sequence Browser	browser		
compression technique	radio	none	none color cell run length

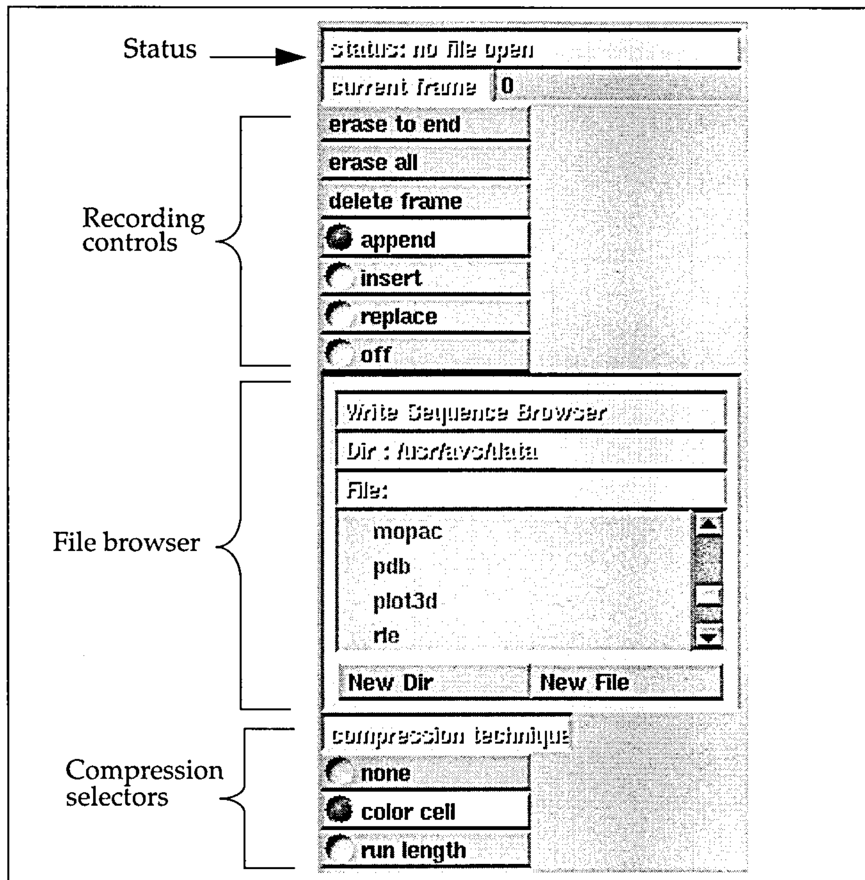
Description

The **write frame seq** module writes an image sequence to a file. It can optionally compress the images and has several controls for editing sequences.

This file can be read through the **read frame seq** module.

Figure 172 shows the **write frame seq** control panel.

Figure 172
write frame seq
control panel



Inputs

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

The input image must be a uniform 2D field with a 4-vector of byte data values at each location in the field. The image can be any size. Any image that can be sent to the **display image** module can be used here.

Parameters

status

This text widget displays the current sequence file and the number of frames in the file.

current frame

An integer typein that may be used to specify the current frame. The **erase to end** and **delete frame** buttons and the **insert** and **replace** modes use the current frame.

erase to end

A oneshot button that, when pressed, erases all frames from the current frame through the end of the sequence.

erase all

A oneshot button that erases all frames in the file. When the **erase all** button is pressed, a confirmation dialog box appears.

delete frame

A oneshot button that, when pressed, deletes the current frame from the file. This does not create a gap in the frame sequence. Instead, frames following the deleted position shift down.

new frames

A set of radio buttons that may be used to select what happens to new frames. There are four choices:

append	Appends each new frame to the end of the sequence (default).
insert	Inserts each new frame before the current frame, then increments the current frame number, so the next frame will follow the new frame in the sequence.
replace	Replaces the current frame with the new frame, then increments the current frame number.
off	Turns off recording. New frames are ignored.

Write Sequence Browser

A file browser used to select the sequence file.

compression technique

A radio button set used to choose the compression technique. The default compression technique is **none**. The frames are stored uncompressed.

color cell The frames are stored using color cell compression. Color cell compression is a lossy compression technique that requires 2 bits per pixel, regardless of image complexity.

run length The frames are stored using an enhanced run length encoding. Run length encoding is a loss-less compression technique. A much better compression rate is possible if the frames have zeros in their alpha channel. Frames typically compress to 8-10 bits per pixel if the alpha channel is zero, depending on image complexity. The **replace alpha** module may be used upstream of **write frame seq** to clear the alpha channel.

Example

The networks in Figure 173 use the **write frame seq** module to generate an image sequence files.

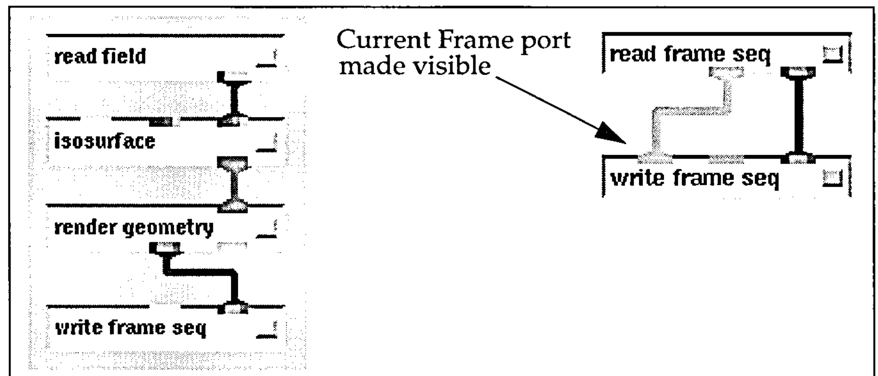


Figure 173
write frame seq module in example networks

You can revise the image sequence by writing portions of an image file to a new file using the write controls to selectively stop and start the recording.

Note

Although images are compressed, a long sequence can easily add up to many megabytes of storage.

If you enable compression, the images require about 1/15th the space (using color cell), but it takes somewhat longer to read the file.

Related modules

AVS Animator, read frame seq, output VideoCreator, output ImageNode

See also

Refer to *Animating AVS Data Visualizations* for detailed information about using this module.

write hdf field

Write a field as a scientific data set to an HDF file

Summary

Name	write hdf field	
Type	data output	
Inputs	field *any-dimension* scalar float uniform string	
Outputs	none	
Parameters	*Name* Write HDF Field	*Type* browser

Description

The **write hdf field** module writes a field as a Scientific Data Set (SDS) to an HDF file. The Hierarchical Data Format (HDF) from the National Center for Supercomputing Applications (NCSA) facilitates the transfer of scientific data and images between computers.

Multiple data sets may reside in an single HDF file, and this module allows you to add data sets to a new file or to an already existing file. Whenever you select a file from the **Write HDF Field** browser, the module writes an additional data set to that file.

This module was developed using NCSA HDF Version 3.1.

Inputs

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

The input is a field.

Label (optional; string)

This input port allows a string to be written to an HDF file as a data set label.

Parameters

Write HDF Field

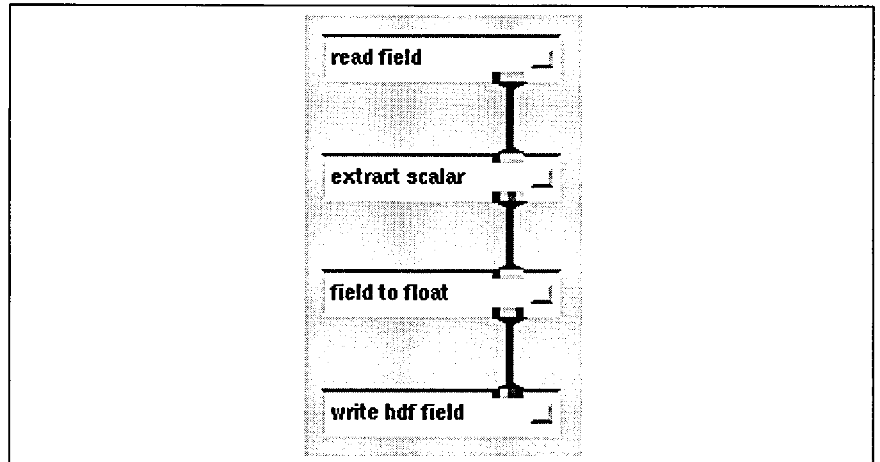
A file browser that allows you to specify the name of the HDF file to be written.

After the HDF file is written, the file name is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering any additions to the file.

Example

The network in Figure 174 shows **write hdf field.**

Figure 174
write hdf field module in an example network



Related modules

The **read hdf field** module will take the HDF file produced by **write hdf field** and read it as a scientific data set into a field.

Considerations

write hdf field performs a significant amount of error checking. If an error is detected while writing the HDF file, an error dialog box appears on the screen along with the type of error.

Limitations

It is possible to corrupt an HDF file if it is opened for writing by more than one module at a time.

See also

read field, write field, and print field

Acknowledgments

Portions developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.

write hdf image

Store image data to an HDF file

Summary

Name	write hdf image	
Type	data output	
Inputs	field 2D 4-vector byte uniform	
Outputs	none	
Parameters	*Name*	*Type*
	Write HDF Image	browser

Description

The **write hdf image** module writes an image data structure as a 24-bit raster image in an HDF file. The Hierarchical Data Format (HDF) from the National Center for Supercomputing Applications (NCSA) facilitates the transfer of scientific data and images between computers.

Multiple images may reside in a single HDF file, and this module allows you to add images to a new file or to an already existing file. Whenever you select a file from the **Write HDF Image** browser, the module writes an additional image to that file.

This module was developed using NCSA HDF Version 3.1.

Inputs

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

The input can be any image.

Parameters

Write HDF Image

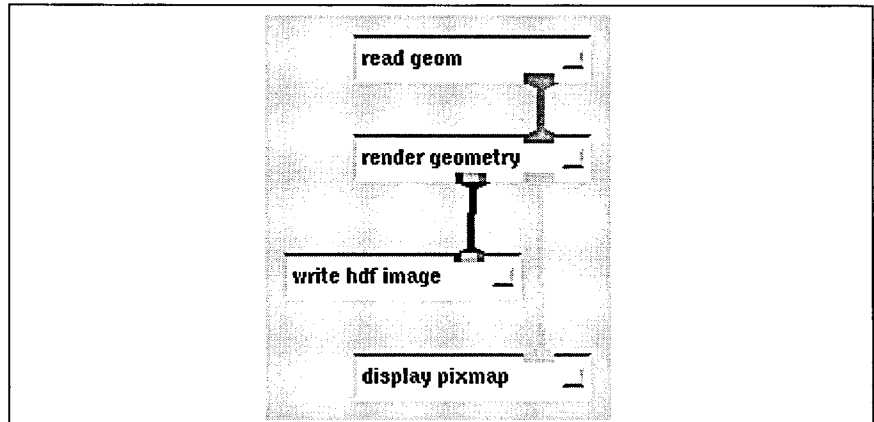
A file browser that allows you to specify the name of the HDF file to be written.

After the HDF file is written, the file name is reset to NULL. This prevents subsequent changes upstream in the network from automatically triggering any additions to the file.

Example

The network in Figure 175 shows **write hdf image.**

Figure 175
write hdf image module in an example network



Related modules

The **read hdf image** module will take the HDF file produced by **write hdf image** and read it as a 24-bit raster image into a 2D 4-vector byte field.

Limitations

It is possible to corrupt an HDF file if it is opened for writing by more than one module at a time.

See also

read image and write image

Acknowledgments

Portions developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.

write image

Store image data in a file

Summary

Name	write image	
Type	data output	
Inputs	field 2D 4-vector byte uniform	
Outputs	none	
Parameters	*Name*	*Type*
	Write Image Browser	browser

Description

The **write image** module writes an image data structure to a file. This structure takes the form of "field 2D 4-vector byte." The image format for this module and **read image** follows:

```
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)
```

Inputs

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

The input can be any image.

Parameters

Write Image Browser

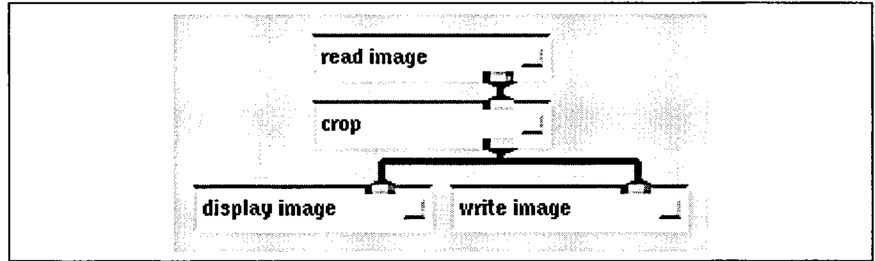
A file browser that allows you to specify the name of the image file to be created. The file suffix .x is appended to the name automatically. If the file already exists, **write image** issues a warning message and has you confirm or cancel the operation.

After the image file is written, the file name 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 file name.

Example

The network in Figure 176 shows **write image**.

Figure 176
write image module in an example network



Related modules

Modules that can provide additional image processing are contrast, threshold, histogram stretch, clamp, and interpolate.

Modules that can decompose or compose images from separate bands are extract scalar and combine scalars.

Module that can show images is display image.

Modules that can take output and write it as an image are render geometry and pixmap to image.

See also

read image and image viewer

The example script WRITE IMAGE demonstrates the **write image** module.

write ucd

Write unstructured cell data to disk

Summary

Name	write ucd	
Type	data output	
Inputs	ucd structure	
Outputs	none	
Parameters	*Name*	*Type*
	file name	browser
	File Format	radio button

Description

The **write ucd** module writes a UCD structure to disk.

write ucd outputs either an ASCII or binary file. This file format is read by the module **read ucd**. The format of these files, however, is not the same.

Inputs

UCD Structure (required; ucd structure)

The input can be any UCD structure.

Parameters

file name

A file browser that allows you to specify the name of the UCD file to be created.

After the UCD file is written, the file name 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 file name.

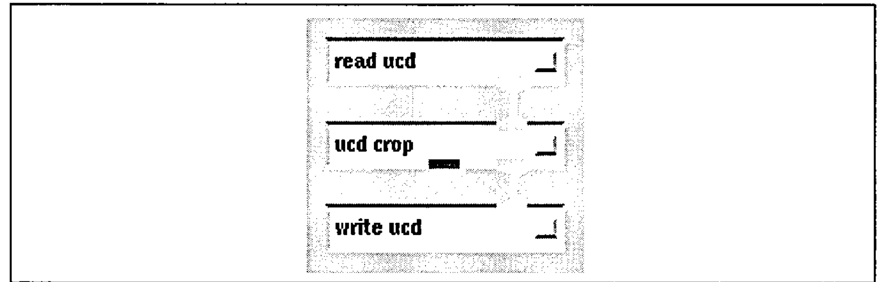
File Format

Buttons that allow you to choose which format you want the file written in. Choices are **Binary** (default) and **ASCII**.

Example

The network in Figure 177 reads in a UCD structure, crops it, and writes the resulting structure to disk.

Figure 177
write ucd module in an example network



Related modules

Modules that could provide the **UCD Structure** input are read ucd and field to ucd.

Limitations

write ucd produces an error message if it cannot open the file or if there is not enough space to write the complete file.

See also

The example script WRITE UCD demonstrates the **write ucd** module.

write volume

Write volume data to a file

Summary

Name	write volume	
Type	data output	
Inputs	field 3D scalar byte uniform	
Outputs	none	
Parameters	*Name*	*Type*
	Write Volume Browser	browser

Description

The **write volume** module writes volume data to a file. The input volume is in the format "field 3D scalar byte." The data format for this module and **read volume** is:

```
(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
```

Each time the file is written, the file name is reset to NULL. This prevents successive changes upstream in the network to automatically trigger a volume data file to be written. A new file name must be entered each time the file is to be written out.

This module is used to preprocess 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

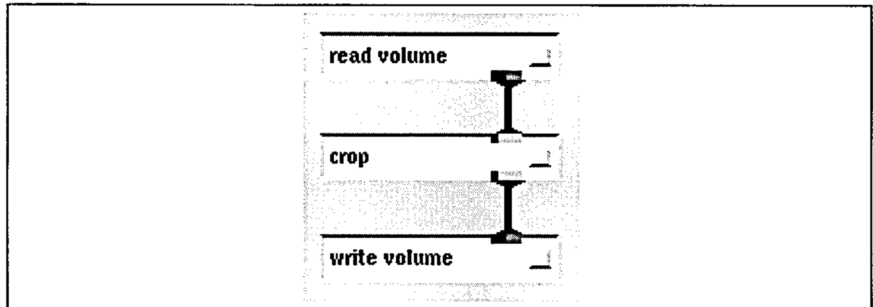
Write Volume Browser

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 or cancel the operation.

Example

The network in Figure 178 shows **write volume**.

Figure 178
write volume module in an example network



Related modules

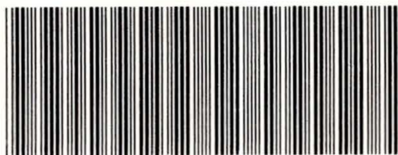
read volume, clamp, contrast, crop, downsize, histogram stretch, interpolate, mirror, threshold, and transpose

Limitations

The format of volume databases on disk is severely limiting. The dimensions are restricted to a maximum of 256 in X, Y, and Z. The data also must be in the range 0–255.

See also

The example script WRITE VOLUME demonstrates the **write volume** module.



Order Number
DSW-305

Document Number
710-013030-003