The
Connection Machine
System

# CM/AVS User's Guide

Version 1.0
February 1993

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, February 1993

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1264
(617) 234-1000

# Contents

# About This Manual

## Objectives of This Manual

This manual describes CM/AVS. Working in conjunction with the Application Visualization System (AVS), CM/AVS provides a graphic programming environment for building distributed visualization applications. This manual tells how to build applications that include computation on a CM-5 system, and how to create your own CM/AVS modules.

## Intended Audience

This manual is intended for

- Users who are familiar with the AVS product and who want to visualize data on a CM-5 system. You must also be familiar with using a CM-5.
- Application developers who want to write modules that are compatible with CM/AVS. You should be an experienced C or Fortran programmer, knowledgeable about AVS, and familiar with C* or CM Fortran and using a CM-5.

## Revision Information

This is a new manual.

## Organization of This Manual

This manual contains the following chapters:

**Chapter 1 Introduction**
An overview of CM/AVS concepts and a list of installed components.

**Chapter 2** **Using CM/AVS Modules**
How to set up the environment to run CM/AVS modules locally on a CM-5 partition manager or remotely from a workstation. How to to build a simple application "network" to use a CM/AVS module remotely.

**Chapter 3** **Writing CM/AVS Modules**
A brief discussion of the properties that differentiate serial and parallel fields. How to allocate and access parallel fields. Example module. How to compile, debug, and link modules.

**Appendix A** **CM/AVS Routines**
Descriptions of the CM/AVS routines.

**Appendix B** **CM/AVS Modules**
Descriptions of the CM/AVS modules.

**Appendix C** **Unsupported Programs and Modules**
Descriptions of programs and modules that are included in the CM/AVS package without guarantee or support.

## Related Documents

The following document contains information concerning the hardware and software requirements and installation of CM/AVS:

- *CM/AVS Release Notes* for Version 1.0.

You should have the complete AVS document set. The following manuals are required:

- *AVS User's Guide*
  An introduction to AVS. To use CM/AVS effectively, you must be familiar with the concepts introduced in this manual.

- *AVS Developer's Guide*
  How to write AVS modules.

- *AVS Module Reference*
  Detailed descriptions of all the AVS modules.

- *AVS Tutorial Guide*
  A tutorial introduction to using AVS.
- *AVS Applications Guide*
  Information on using the Module Generator.

## Notation Conventions

The table below displays the notation conventions observed in this manual.

| Convention | Meaning |
|---|---|
| **`bold typewriter`** | UNIX and CM System Software commands, command options, and filenames, when they appear embedded in text. Also, programming language elements, such as keywords, operators, and function names, when they appear embedded in text. |
| **`% bold typewriter`**<br>`regular typewriter` | In interactive examples, user input is shown in **`bold typewriter`** and system output is shown in `regular typewriter` font. |
| `typewriter` | Code examples and code fragments. |
| *italics* | Argument names and placeholders in function and command formats. |

# Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

If your site has an applications engineer or a local site coordinator, please contact that person directly for support. Otherwise, please contact Thinking Machines' home office customer support staff:

| | |
|---|---|
| **Internet Electronic Mail:** | `customer-support@think.com` |
| **uucp Electronic Mail:** | `ames!think!customer-support` |
| **U.S. Mail:** | Thinking Machines Corporation<br>Customer Support<br>245 First Street<br>Cambridge, Massachusetts 02142-1264 |
| **Telephone:** | (617) 234-4000 |

Chapter 1

# Introduction

CM/AVS adapts and extends the Application Visualization System (AVS) to the realm of the CM-5. AVS provides a graphic programming environment in which a user builds a distributed visualization application. An application may involve diverse operations such as filtering, graphing, volume rendering, polygon rendering, image processing, and animation. CM/AVS enables an application to operate on data that is distributed on CM-5 processing nodes and to interoperate with data from other sources. CM/AVS also facilitates the incorporation of CM-5 code into a CM/AVS application.

CM/AVS is not run separately from AVS. A user runs AVS normally, using CM/AVS modules and functions to handle data on the CM-5.

## 1.1 Overview of CM/AVS

The building blocks of an AVS application program are small, packaged units of code, called *modules*. Most modules process typed data input(s) into typed data output(s). Each module performs a given function. The function may be as simple as adding two arrays, or as complicated as extracting isosurfaces of a volume. When a CM/AVS module is used, the function is performed on a CM-5.

Modules are connected to form larger applications, called *networks*. In a network, information is passed between the modules as various data types. Only the *field* data type, which represents an array of data, is relevant to CM/AVS. CM/AVS supports a *parallel field* that accommodates the distribution of data across the CM-5 processing nodes. CM/AVS includes routines to allocate the parallel arrays, and to access the data and coordinates as CM Fortran arrays or C* parallel variables.

When CM/AVS modules that operate on parallel data are connected with AVS modules that operate on serial data, CM/AVS routines convert the data between parallel and serial fields as required. The conversion is transparent to the user and to the module writer.

The AVS Network Editor visual interface makes it easy to build application networks graphically. Alternatively, the Network Editor may be driven by the AVS Command Language Interpreter.

## 1.2 The CM/AVS Package

The CM/AVS software package includes:

- A set of modules that handle data on a CM-5. The modules are described in Appendix B.
- A set of routines that provide general operations for parallel fields. The routines are used by the CM/AVS modules and users may incorporate them in their own C* or CM Fortran modules. The concepts that differentiate serial and parallel fields are discussed in Chapter 3. The routines are described in Appendix A.
- On-line code examples, help files, and release notes.

CM/AVS is installed on the CM-5 compile server as follows:

| | |
|---|---|
| CM/AVS libraries | **`/usr/lib`** |
| CM/AVS include files | **`/usr/include`** |
| Combined module binary, list-dir file, and library description file | **`/usr/lib/cmavs_library`** |
| Examples | **`/usr/examples/cmavs`** |
| Module help files | **`/usr/doc/cmavs/modules`** |

| | |
|---|---|
| Release notes | **`/usr/doc/cmavs-1.0.releasenotes`** |
| Source (by license only) | **`/usr/src/cmavs`** |

The directory **`/usr/examples/cmavs/unsupported`** contains items that are supplied without guarantee or support. The **`README`** file in this directory contains information about its contents. Appendix C contains additional information about unsupported programs and modules.

# Chapter 2

# Using CM/AVS Modules

This chapter tells how to execute the CM/AVS modules from a workstation and from a CM-5 partition manager. The following topics are discussed:

- The CM/AVS Modules
- Using CM/AVS Modules in a Network
- Preparing to Run Remote CM/AVS Modules
- Running a Remote CM/AVS Module — Tutorial
- An Important Note about Performance
- Running CM/AVS Modules Locally
- Cleaning Up

For a more thorough discussion of remote module execution, please refer to the *AVS User's Guide.*

## 2.1 The CM/AVS Modules

CM/AVS provides the following modules. Most are CM-5 versions of AVS modules. A detailed description of each module appears in Appendix B.

```
antialias cm
clamp cm
color range cm
colorizer cm
combine scalars cm
```

```
compare field cm
compute gradient cm
contrast cm
downsize cm
extract scalar cm
fft cm
field math cm
field to byte cm
field to double cm
field to float cm
field to int cm
luminance cm
orthogonal slicer cm
threshold cm
```

The following modules are unsupported. Detailed descriptions of these modules appear in Appendix C.

```
field to polygons
field to spheres
```

## 2.2 Using CM/AVS Modules in a Network

AVS supports distributed computation over a heterogeneous network of computers. While you run the AVS kernel on a local graphics workstation, you can execute modules locally or on other workstations or systems. Using CM/AVS, you can also execute CM/AVS modules on a CM-5.

You use CM/AVS modules in exactly the same manner as AVS modules. When you build a network, you may interconnect AVS modules and CM/AVS modules.

If you run AVS on a CM-5 partition manager, you can run CM/AVS modules locally. (We do not recommend this as a good use of partition manager resources.)

## 2.3 Preparing to Run Remote CM/AVS Modules

It is likely that you will run AVS on a local workstation and run the CM/AVS modules on the CM-5. The remote use of any modules, including CM/AVS, requires some preparation.

AVS uses a "hosts file" to find remote modules. The file identifies remote hosts and the directories on those hosts that contain modules. The hosts file format is described under Remote Module Execution in the Advanced Network Editor chapter in the *AVS User's Guide*.

You may choose to rely on the system administrator to maintain the file **/usr/avs/runtime/hosts**. Alternatively, you may choose to create and maintain a private hosts file. In either case, your **.avsrc** initialization file must point to a legitimate hosts file.

To create and use a **.avsrc** file, follow these steps:

1. Create a **.avsrc** file. AVS looks for this file first in the current directory, and then in your home directory. We recommend putting it in your home directory.

   A minimal **.avsrc** file might look like this:

   ```
       # Point to a file containing remote hosts
       Hosts /home/yourname/.avs-hosts
   ```

   where the specified hosts file is **/home/yourname/.avs-hosts.**

2. Check the hosts file:

   - It must reside at the pathname specified in the **.avsrc** file. The pathname must be valid on the system where AVS is invoked.

   - It must contain one line of information for every remote host/directory combination where you want AVS to look for modules.

   Each information line in a hosts file contains four fields, in this order:

   (1) A logical name that identifies a particular combination of a remote host and a module directory. This logical name will appear in the Remote Host Browser. (Just the host name may be used for this purpose, unless more than one module directory on the host is of interest. In that case, each of the directories requires its own line in the hosts file.)

(2) Both the name of the remote shell program (the path to `rsh`) and the actual host name of the remote machine. (For CM/AVS modules, the host name should be a CM-5 partition manager.) You may add options to `rsh`. The entire field is enclosed in double quotes.

(3) The directory on the remote host that should be searched for modules.

(4) The default data directory on the remote host.

To make the remote CM/AVS modules available, the hosts file must contain a line that specifies a CM/AVS modules directory on a partition manager. The line might look like this:

```
pep.think.com "/usr/ucb/rsh pep.think.com -n"
                  /usr/lib/cmavs_library
                  /usr/avs/data
```

**`pep.think.com`** is the logical name of the CM-5 partition manager (**`pep.think.com`**) and the module directory **`/usr/lib/cmavs_ library`**. (Field 1)

**`/usr/ucb/rsh`** is the command to run a command shell on the remote machine, whose real name is **`pep.think.com`**; **`-n`** is an **`rsh`** option that prevents input conflicts with the caller. (Field 2)

**`/usr/lib/cmavs_library`** is the directory that contains CM/AVS modules. (Field 3)

**`/usr/avs/data`** is the data directory. (Field 4)

## 2.4 Running a Remote CM/AVS Module — Tutorial

In this section, we build a simple network to turn an RGB image into a greyscale image. We use a CM/AVS module, **`luminance cm`**, in the network. The module is the CM-5 version of the AVS **`luminance`** module.

First, make the preparations described in Section 2.3. Then, follow these steps:

1. Start AVS and bring up the AVS network editor. From the menu in the upper left of the network editor menu, select **Module Tools**, as shown in Figure 1.

2. *Optional:* If you want the CM/AVS module icons to appear under a `CM Modules` Library header instead of being incorporated in the AVS module lists, do the following:

   Select **Edit Module Library**, then select **Create Empty Library** on the resulting pop-up window. Enter `CM Modules` in the pop-up prompt for a name, then select **OK**. Close the **Edit Module Library** pop-up window. `CM Modules` appears as the selected AVS Module Library header in the AVS Module Palette.

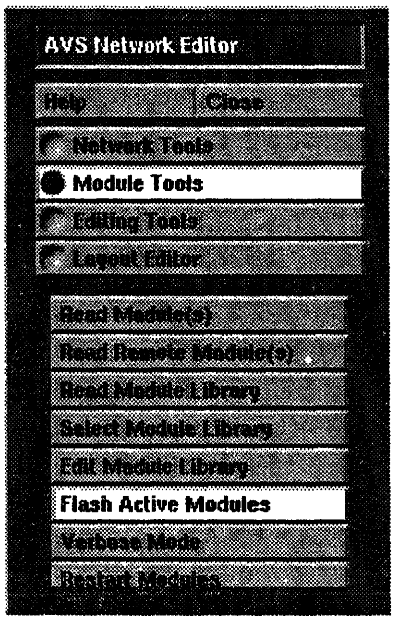

**Figure 1. Module Tools menu.**

3. Select **Read Remote Module(s)** from the **Module Tools** menu. This brings up a Remote Host Browser containing a list of available hosts. Select a CM-5 partition manager (one that is named in `.avs-hosts`) from this list. In this example, we use `pep`, as shown in Figure 2.

   The selection causes the display of the contents of the current directory on the host. If the binary file `cmavs-modules` is not in the contents list, change to the directory that contains it (`/usr/lib/cmavs_library`). Select `cmavs-modules` from the list and close the module selection window.

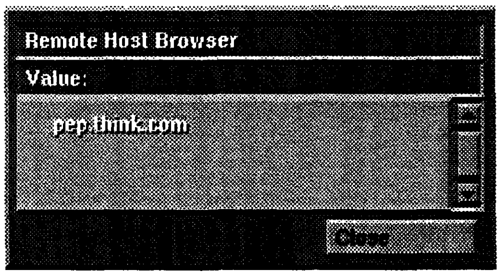

**Figure 2. Remote Host Browser.**

When the read of the binary file is complete, an icon representing each of its modules appears under the appropriate Library header. Each icon has a button on thc right side. On a remote module, this button is colored pink.

4. Drag one instance each of the following modules into the network editor workspace. Place them as shown in Figure 3. The appearance of colored badges designating input and output ports indicates that the AVS modules are active and ready to accept connections. *Wait for the ports to appear* on one module before you drag in the next one. (Section 2.5 explains the benefit of waiting.)

   `read image` (**Data Input** list)

   `luminance cm` (**Filter** list)

   `colorizer cm` (**Filter** list)

   `display image` (**Data Output** list)

   When `display image` becomes active, the image display window comes up. Reposition it, if you like. (It will expand to about four times its initial size to accomodate the result.)

5. When all four modules are active, connect the output of `read image` to the input of `luminance cm`: position the cursor over the `read image` output port, press the middle mouse button, move the cursor to the `luminance cm` input port, and release. Similarly, connect the output of `luminance cm` to the input to `colorizer cm`, and the output of `colorizer cm` to the input to `display image`. See Figure 3.

6. Select **`read image`** on the Network Control Panel and read the image **`/usr/avs/data/image/mandrill.x`**. (This image file is included with AVS.) The modules in the network change color when they are active, so you can watch as the image data progresses through the network. During processing, it is actually transferred to the CM-5 for luminance and colorizer calculations, then back to the local workstation to be displayed. See Figure 3 and Figure 4.

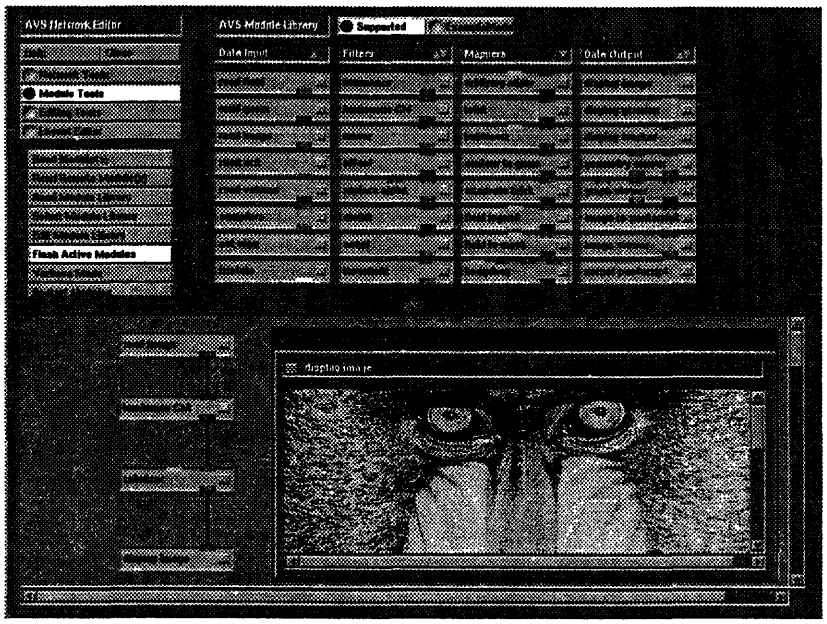

**Figure 3. The complete network and display.**

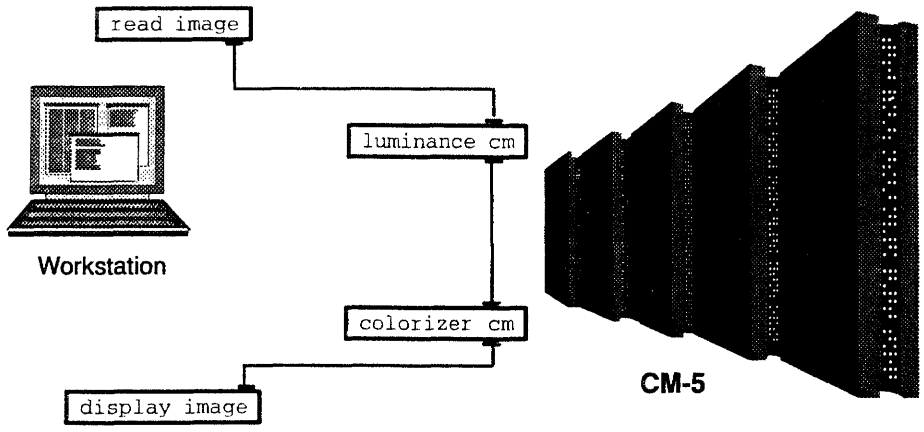

**Figure 4. Example network: processing locations.**

## 2.5 An Important Note about Performance

To realize the best performance for your application, you must take a bit of care when you add modules to a network. After you drag a module into the workspace, *wait for its ports to appear* before you drag in another module.

This can have an effect on performance because multiple CM/AVS or AVS modules may be linked into a single binary, enabling multiple modules in a network to run in a single process. Field transfers between modules in a single process consist of a simple pointer copy. By contrast, field transfers between processes must use sockets. If you add new modules to your network too quickly, AVS may not have time to ascertain whether or not it can fulfill the module requests with a single process.

## 2.6 Running CM/AVS Modules Locally

Running AVS on a CM-5 partition manager does not make the best use of the partition manager resources. However, it can be done, provided you follow these steps to make the CM/AVS modules accessible to the AVS Network Editor:

1. Start AVS and bring up the AVS Network Editor. From the menu in the upper left of the Network Editor menu, select **Module Tools**, as shown in Figure 5.

2. *Optional:* If you want the CM/AVS module icons to appear under a `CM Modules` Library header instead of being incorporated in the AVS module lists, do the following:

   Select **Edit Module Library**, then select **Create Empty Library** on the resulting pop-up window. Enter `CM Modules` in the pop-up prompt for a name, then select **OK**. Close the **Edit Module Library** pop-up window. `CM Modules` appears as the selected AVS Module Library header in the AVS Module Palette.

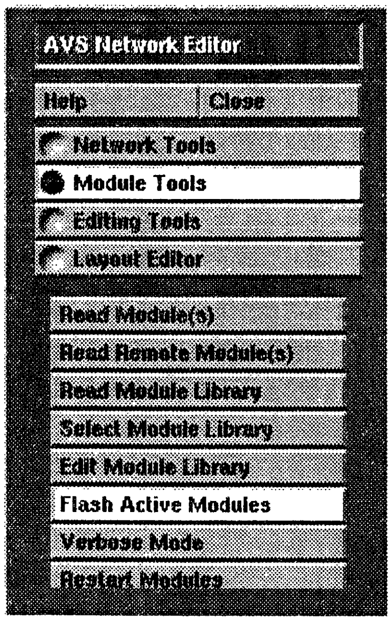

**Figure 5. Module Tools menu**

3. Select **Read Module(s)** from the Network Editor list. A display shows the contents of the current directory. If the binary file **`cmavs-modules`**

is not in the contents list, change to the directory that contains it (**/usr/lib/cmavs_library**). Select **cmavs-modules** from the list, then close the module selection window. (From this point, you may pick up with Step 4 in the tutorial, Section 2.4, if you like.)

## 2.7 Cleaning Up

If AVS terminates abnormally (if the kernel crashes or if there are network problems, for example), CM/AVS modules may be left running on the CM-5. Therefore, it is a good idea to use **cmps** and check the partition manager process status after AVS terminates. Be sure that no stray modules continue to run and waste system resources.

# Chapter 3

# Writing CM/AVS Modules

You can develop new code or adapt existing code to be compatible with the CM/AVS environment. You may combine your modules in a network with modules from other sources.

This chapter talks about the aspects of code that are unique to handling parallel arrays, including the topics listed below:

- The Parallel Field Type
- Using AVS Field Routines on Parallel Fields
- Allocating Parallel Fields
- Accessing Field Data and Coordinates

The chapter includes a sample module, and concludes with instructions for compiling and debugging your modules.

## 3.1 The Field Type

The data that is passed between AVS modules is categorized by type. Only the field data type is pertinent to CM/AVS.

An AVS field is an *n*-dimensional array of byte, short integer, integer, floating-point, or double-precision floating-point numbers. AVS fields contain some descriptive information, such as the number of dimensions and the type of coordinate mapping, but the bulk of a field is in its data and coordinate arrays.

A field is defined in a *computation space* where the axes are orthogonal and each data point is unit distance away from its neighbors along any axis.

This computation space is mapped into a *coordinate space* in one of three ways;

| | |
|---|---|
| *uniform* | The coordinate space is determined by minimum and maximum values along each axis and is mapped directly onto the Cartesian grid defined by these extents. |
| *rectilinear* | The neighbors along any axis may be different distances apart; for each axis there is a separate array that gives the mapping from computation to coordinate space. |
| *irregular* | Each data point also has an explicit coordinate stored with it; this can be used to represent curvilinear volumes. The connectivity is still topologically rectilinear. |

Fields are described in further detail in the *AVS Developer's Guide.*

## 3.2 The Parallel Field Type

The defining feature of a *parallel field* is the distribution of its data across the CM-5 processing nodes. The field's coordinate array may also be stored on the processing nodes. If the mapping from computation space to coordinate space is rectilinear or irregular, then the coordinate array is automatically placed on the processing nodes. If the mapping is uniform, then the few floating point-numbers that describe the mapping are left on the partition manager; the coordinate array for a uniform field is never put on the processing nodes.

In this discussion, we use the term *parallel field* for a field whose data is distributed over the CM-5 processing nodes. The term *serial field* refers to a field whose data lives in the memory of some scalar machine: either your local workstation or a CM-5 partition manager. When this distinction does not matter, we simply use the term *field.* CM/AVS provides routines for allocating and accessing *parallel* fields.

### 3.2.1 Declaring a Parallel Field

Parallel fields are declared in the same way as standard AVS fields. In C*, a parallel field is declared as a structure or as a pointer to a structure, as appropriate:

```
AVSfield *field;
```

In CM Fortran, a parallel field is declared as an opaque integer, which should be operated on only with AVS or CM/AVS routines:

```
integer field
```

### 3.2.2 Passing a Parallel Field

Parallel fields must be passed as single arguments to CM Fortran functions. This means that any CM Fortran module must include in its module description function a call to **AVSset_module_flags** with the **SINGLE_ARG_DATA** flag set:

```
      call AVSset_module_flags(SINGLE_ARG_DATA)
```

For coroutine modules that output parallel fields, the **COROUT_UNPACK_ARGS** flag must also be set. In CM Fortran, module flags may be combined by using the **IOR** intrinsic:

```
      call AVSset_module_flags(IOR(SINGLE_ARG_DATA,
     $   COROUT_UNPACK_ARGS ))
```

## 3.3 Using AVS Field Routines on Parallel Fields

Most of the standard AVS field access routines work correctly on parallel fields; the exceptions are the ones that that touch the field data or coordinates. CM/AVS provides equivalents for these, as listed in Table 2.

**Table 2. Standard AVS field routine equivalents for parallel fields.**

| Standard AVS Field Routine | CMAVS Replacement |
|---|---|
| `AVSfield_alloc` | Use `CMAVSfield_alloc` or `CMAVSdata_alloc`. |
| `AVSfield_free` | Use `AVSdata_free`. |
| `AVSfield_data_offset` `AVSfield_data_ptr` | Convert the field into a CMF array or C* pvar with `CMAVSfield_data_get`. |
| `AVSfield_points_offset` `AVSfield_points_ptr` | Use these routines only for UNIFORM CM/AVS fields. For others, use `CMAVSfield_points_get`. |
| `AVSfield_reset_minmax` | Use `CMAVSfield_reset_minmax`. |
| `AVSfield_copy_points` | Use only for UNIFORM CM/AVS fields. For others, use `CMAVSfield_copy_points`. |
| `AVSbuild_field` `AVSbuild_2d_field` `AVSbuild_3d_field` | *Obsolete after AVS 2.0.* |

## 3.4 Allocating Parallel Fields

There are two ways that a parallel field can come into being: it may be explicitly allocated, or an input port may be declared as parallel, causing the received field to reside on the processing nodes.

### 3.4.1 Parallel Input Ports

To direct a module to distribute data on the CM-5 processing nodes, use the `PARALLEL` flag with `AVScreate_input_port`. (This flag may be `OR`'ed with others, such as `REQUIRED`.) If the `PARALLEL` flag is not set, the data is placed on the partition manager.

For example, a CM Fortran module that reads an image and processes it on the processing nodes might contain this input port definition:

```
        inport = AVScreate_input_port('input field',
     $             'field 2D 4-vector byte',
     $             IOR(REQUIRED, PARALLEL))
```

The AVS Network Editor displays ports for parallel fields and serial fields the same way, and allows connections between the two.

**NOTE**

A connection between an AVS module and a CM/AVS module can work only if the modules *are not linked in the same binary.* If this condition is not met, the results may appear to be correct at first; however, errors may appear later.

## 3.4.2 Explicit Allocation

To allocate a CM/AVS field, one may call **CMAVSdata_alloc** or **CMAVSfield_alloc**.

Given a dimension array and a string describing the desired field, **CMAVSdata_alloc** returns a parallel field.

```
output = CMAVSdata_alloc
             ("field 2D scalar byte",dims)
```

The string describing the field is the same as that used by **AVSdata_alloc**.

**NOTE**

**CMAVSdata_alloc** may be used only to allocate fields. If the string describes any other type, an error is raised.

**CMAVSfield_alloc** takes an AVS field as a template and allocates a corresponding parallel field. The new field may take its dimensions from the template or from an explicit dimensions array. This CM Fortran code fragment allocates an output field with the same properties as the input field:

```
iresult = AVSfield_make_template(input, template)
output = CMAVSfield_alloc(template,0)
```

The resulting output field is a duplicate of the input field, and it is guaranteed to be on the processing nodes, even if the input field was not.

## 3.5 Accessing Field Data and Coordinates

CM/AVS provides some special routines to gain access to a parallel field's data and coordinates.

### 3.5.1 Access Routines

There are two CM/AVS routines that give access to the data and coordinates in a CM/AVS field: **CMAVSfield_data_get** and **CMAVSfield_points_get**:

```
void:void *
    CMAVSfield_data_get(AVSfield *field, shape S);

float:void *
    CMAVSfield_points_get(AVSfield *field shape S);
```

The C* interface to these routines takes both a field and a shape, and returns a pointer to a parallel variable. The parallel pointer refers to the coordinates in the first argument (**AVSfield**). You must pre-allocate the shape using **CMAVSfield_alloc_points_shape**. You may use the same shape to construct pointers to any fields that have the same rank and dimensions.

In CM Fortran, the situation is slightly more complicated because both these routines construct and return arrays of arbitrary rank. There is no way to express such a generic array constructor in CM Fortran itself, so the routines instead return an opaque object that can be passed to a routine expecting a CM Fortran array.

This approach is similar to that used by the CM Fortran utility function **CMF_ALLOCATE_ARRAY**. CM/AVS uses a small array on the partition manager to hold an array descriptor. This one-dimensional array of integers must have length **CMF_SIZEOF_DESCRIPTOR**, which is defined in **/usr/include/cm/CMF_defs.h**.

```
      include '/usr/include/cm/CMF_defs.h'
      include '/usr/include/cm/cmavs.inc'
      integer field

      integer desc(CMF_SIZEOF_DESCRIPTOR)
  CMF$LAYOUT desc(:serial)


      call CMAVSfield_data_get(field, desc)

      call CMAVSfield_points_get(field, desc)
```

The **CMF$LAYOUT** directive is not actually needed, but its use in the documentation and example code emphasizes that **desc** must not be a parallel array. The descriptor array returned by **CMAVSfield_data_get** or **CMAVS-field_points_get** may be passed to any routine expecting a CM Fortran array. Section 3.5.5 shows how to declare the layout of these arrays.

Note that both the C* and CM Fortran access routines are really returning pointers to a memory location on the processing nodes. Be careful not to refer to one of these pointers after freeing a field; it will no longer refer to valid data.

### 3.5.2 Primitive Data Types

When you declare a C* parallel variable or a CM Fortran array, the primitive data type must correspond with the AVS type, as shown in Table 3.

**Table 3. Primitive data types.**

| AVS Type | C* Type | CM Fortran Type |
|---|---|---|
| `AVS_TYPE_BYTE` | unsigned char | integer |
| `AVS_TYPE_SHORT` | short | integer |
| `AVS_TYPE_INTEGER` | integer | integer |
| `AVS_TYPE_REAL` | float | real |
| `AVS_TYPE_DOUBLE` | double | double-precision |

CM/AVS byte and short fields are promoted to integers for CM Fortran, since CM Fortran does not support parallel arrays of bytes or shorts, and it is simpler to manipulate integer fields. During this promotion, shorts are sign-extended to form integers, and bytes are not sign extended: shorts are in the range -32768 <= x <= 32767 and bytes are in the range 0 <= x <= 255.

**NOTE**

Even though the field data is promoted to integers, the *min_data* and *max_data* values are still kept as bytes and shorts. With Version 5.0, AVS provides **`AVSfield_get_minmax_as_int`** and **`AVSfield_set_minmax_as_int`**, which automatically coerce shorts and bytes to ints.

When you convert the coordinates in a CM/AVS field to a C* parallel variable or CM Fortran array, the result is always stored as single-precision floating-point numbers.

### 3.5.3 Data Array Layout

An AVS field is essentially an *n*-dimensional Cartesian grid, where each point in the grid may contain a single value or a vector of values. The length of this vector is given by the *veclen* member of the `field` structure.

In C*, the field data is stored in an *n*-dimensional shape. In this shape, we allocate a 1-dimensional per-processor array of length *veclen* using the appropriate primitive data type.

In CM Fortran, the field data is stored in an array with *n*+1 dimensions. The first axis has `:SERIAL` ordering (elements along this axis reside in the same physical processor) and length *veclen*. We call this serial axis the "vector axis." To make it easier to write modules that are independent of vector-length, this vector axis is present even for scalar fields; in this case it is of length one. The remaining axes have `:NEWS` ordering.

### 3.5.4 Coordinate Array Layout

There is only one valid type for coordinate arrays: single-precision floating point. This applies to coordinates on the processing nodes or on the partition manager.

Uniform fields *always* have their coordinates stored on the partition manager.

Irregular fields are placed in a floating-point array with *ndim*+1 axes, where *ndim* is the dimensionality of the data array. The first *ndim* are `:NEWS` axes whose length is given by the corresponding entry in the dimensions array. The remaining axis is `:SERIAL` and of length *nspace*, where *nspace* is the dimensionality of the space in which the data points exist.

Rectilinear fields are placed in a 1-dimensional floating-point array with a single `:NEWS` axis. The length of the array is the sum of the lengths of all axes in the field.

### 3.5.5 Declaring the Arrays

Assume that you have a 2-dimensional uniform field, with a 4-vector of bytes at every point. In CM Fortran, the field data would be loaded into an array declared

```
     integer array(4, x, y)
  CMF$LAYOUT(:serial, :news, :news)
```

where *x* and *y* are the lengths of the field's axes.

If you have a 3-dimensional scalar field, the field data would be loaded into an array declared

```
     integer array(1, x, y, z)
  CMF$LAYOUT(:serial, :news, :news, :news)
```

## 3.6 Luminance Module Example

As a simple example, consider a module that takes the luminance of an image. In AVS, an image is represented by a 2-dimensional field with a 4-vector of bytes at every point. The coordinate mapping is usually uniform.

The luminance of an image is a weighted sum of the color components at each pixel. The first byte in each 4-vector is the alpha component; this component is typically used to store opacity, and it is not used to compute the luminance. The remaining bytes are the red, green, and blue components; we combine these, using weights appropriate for the NTSC luminance. This choice of weights makes our simple routine compatible with the AVS **luminance** module.

### 3.6.1 Luminance Module Code

A copy of the module **luminance.fcm** is included with other examples in the directory

```
/usr/examples/cmavs
```

Below is the CM Fortran code for the module. Note that the bytes of the image are automatically promoted to integers by CM/AVS; this makes it easier to deal with byte fields in CM Fortran. Note also that we have a separate routine, **luminance_compute**, which extracts parallel arrays from the CM/AVS fields and passes them to the function that actually computes the luminance.

```
C**********************************************************************
C A luminance module
C**********************************************************************

C----------------------------------------------------------------------
C Describe the module to AVS
C----------------------------------------------------------------------
      subroutine AVSinit_modules
      implicit none
      include 'avs/avs.inc'
      include 'cm/cmavs.inc'
      integer iport, oport
      external luminance_compute

C Set the module name and type
      call AVSset_module_name('luminance CM', 'filter')

C Make sure we pass in the args as integers
      call AVSset_module_flags(IOR(SINGLE_ARG_DATA,
     $      IOR(COOPERATIVE,REENTRANT)))

C Create an input port for the required field input
      iport = AVScreate_input_port('input field',
     $     'field 2D 4-vector byte',
     $     IOR(REQUIRED, PARALLEL))

C Create an output port for the result
      oport = AVScreate_output_port('output field',
     $     'field 2D scalar byte')

      call AVSset_compute_proc(luminance_compute)

      return
      end

C----------------------------------------------------------------------
C Unpack the structure members we need, create CMF arrays that point
C to the field data, and call the routine that does the real work
C----------------------------------------------------------------------
      integer function luminance_compute(in, out)
      implicit none
      include 'avs/avs.inc'
      include '/usr/include/cm/CMF_defs.h'
      include 'cm/cmavs.inc'

      integer in, out

      integer indesc(CMF_SIZEOF_DESCRIPTOR),
     $     outdesc(CMF_SIZEOF_DESCRIPTOR)
      integer iresult, dims(2)
```

```
C Now get pointers to the arrays containing the AVS field data
      call CMAVSfield_data_get(in,indesc)
      iresult= AVSfield_get_dimensions(in, dims)

C If there is already output data, deallocate it.
      if (out .ne. 0) then
         call AVSdata_free("field",out)
      endif
      out = CMAVSdata_alloc("field 2D scalar byte",dims)

C Get a pointer to the output data
      call CMAVSfield_data_get(out,outdesc)

C Copy the points from input to output
      iresult= CMAVSfield_copy_points(in,out)

C Call the real function that does the work
      call luminance_internal(indesc, outdesc, dims(1), dims(2))

C Return 1 to indicate success
      luminance_compute = 1
      return
      end

C---------------------------------------------------------------------
C The real workhorse
C---------------------------------------------------------------------
      subroutine luminance_internal(in, out, x, y)

      integer x, y
      integer in(4,x,y), out(1,x,y)
CMF$LAYOUT in(:serial,:news, :news), out(:serial,:news, :news)

C
C Set up the weights for NTSC luminance
C
      double precision red_weight, green_weight, blue_weight

      parameter (red_weight   = .299,
     $     green_weight = .587,
     $     blue_weight  = .114)


      out(1,:,:) = in(2,:,:) * red_weight +
     $     in(3,:,:) * green_weight +
     $     in(4,:,:) * blue_weight

      return
      end
```

## 3.7 The CM/AVS Header Files

The routines that you write must include the standard AVS header files. In addition, they must include the CM/AVS files that define all the appropriate symbols and return types for the CM/AVS routines.

The header file for CM Fortran routines is

```
/usr/include/cm/cmavs.inc
```

The include file for C* routines is

```
<cm/cmavs.h>
```

## 3.8 The CM/AVS Libraries

The CM/AVS subroutine and coroutine libraries are listed below.

For a sparc processor:

| | |
|---|---|
| CMF subroutine | `libcmavsflow_f_cm5_sparc_sp.a` |
| CMF coroutine | `libcmavssim_f_cm5_sparc_sp.a` |
| C* subroutine | `libcmavsflow_c_cm5_sparc_sp.a` |
| C* coroutine | `libcmavssim_c_cm5_sparc_sp.a` |

For a vector unit processor:

| | |
|---|---|
| CMF subroutine | `libcmavsflow_f_cm5_vu_sp.a` |
| CMF coroutine | `libcmavssim_f_cm5_vu_sp.a` |
| C* subroutine | `libcmavsflow_c_cm5_vu_sp.a` |
| C* coroutine | `libcmavssim_c_cm5_vu_sp.a` |

These libraries act in conjunction with the standard AVS libraries:

| | |
|---|---|
| FORTRAN subroutine | `libflow_f.a` |
| FORTRAN coroutine | `libsim_f.a` |
| C subroutine | `libflow_c.a` |
| C coroutine | `libsim_c_.a` |

When you link a CM/AVS module, specify the CM/AVS library first, then the corresponding AVS library. To build a CM Fortran subroutine module, for example, link against `libcmavsflow_f_cm5_sparc_sp.a` first, then against `libflow_f.a`.

## 3.9 Compiling a Module

To compile a CM Fortran subroutine module, start a shell on a CM-5 compile server and invoke the CM Fortran compiler:

For a sparc processor:

```
cmf -cm5 -sparc -o module_name module_name.fcm \
-lcmavsflow_f_cm5_sparc_sp -L/usr/avs/lib -lflow_f \
-lgeom -lutil -lm
```

For a vector unit processor:

```
cmf -cm5 -vu -o module_name module_name.fcm \
-lcmavsflow_f_cm5_vu_sp -L/usr/avs/lib -lflow_f \
-lgeom -lutil -lm
```

To compile a C* subroutine module, start a shell on a CM-5 compile server and invoke the C* compiler:

For a sparc processor:

```
cs -cm5 -sparc -o module_name module_name.cs \
-lcmavsflow_c_cm5_sparc_sp -L/usr/avs/lib -lflow_c \
-lgeom -lutil -lm
```

For a vector unit processor:

```
cs -cm5 -vu -o module_name module_name.cs \
-lcmavsflow_c_cm5_vu_sp -L/usr/avs/lib -lflow_c \
-lgeom -lutil -lm
```

To compile *coroutine* modules, replace `flow` with `sim` in the `lcmavsflow...` and `lflow_c` library names above.

## 3.10 Debugging a Module

AVS reports run-time errors in a dialog box. However, the run-time error messages are not as detailed as those issued by a debugger, and some problems may appear to be downstream from the actual error.

To obtain detailed debugging messages, follow these steps:

1. Compile all the files for your module with the `-g` switch.
2. Select **Read Remote Modules** (or **Read Modules**, if you are on the partition manager) to add your module to the palette.
3. Start a shell on the machine that will run this module (for CM/AVS modules, this is the partition manager).
4. In this shell, change to the directory containing your module and invoke **avs_dbx**:

   ```
   avs_dbx -debug prism your_module
   ```

   The `-debug` switch lets you specify your preferred debugger. You may substitute "`prism -C`" (including the quotes) for `prism` .
5. Drag your module into a network. It will not fire immediately. Instead, you will see the following message in the window where you invoked **avs_dbx**:

   ```
   your_module instance waiting, fire when ready...
   ```
6. Set the desired breakpoints.
7. Launch the module by telling the debugger to run it.

## 3.11 Getting Help

Man pages for all the CM/AVS modules are viewable through AVS after you have "read" them following the instructions in Sections 2.3 and 2.4. To view them:

1. In the shell where you will invoke the AVS kernel, set the environment variable AVS_HELP_PATH as follows:

C shell:

```
setenv AVS_HELP_PATH /usr/doc/cmavs/modules
```

Bourne or Korn shell:

```
AVS_HELP_PATH=/usr/doc/cmavs/modules
export AVS_HELP_PATH
```

2. Open the AVS network editor.
3. In the module library list, find the CM/AVS module whose man page you want to view. Using the right mouse button, select the button on the right of the module icon.
4. On the resulting pop-up menu, select **Show Module Documentation.** The man page will appear in the AVS viewer.

## 3.12 Multiple-Module Binaries

CM/AVS modules may be linked together into a single binary in exactly the same way as AVS modules. With the exception noted below, this is desirable, because it enables multiple modules in a network to run in a single process.

**NOTE**

A connection between an AVS module and a CM/AVS module can work only if the modules *are not* linked in the same binary.

Field transfer between modules in a single process can be considerably faster than field transfer between modules in separate processes. The former involves a simple pointer copy, while the latter uses sockets to transfer all the data.

Appendix A

# CM/AVS Routines

This appendix contains descriptions of the supported user-visible routines in the CM/AVS libraries, in alphabetical order.

A module may use the CM/AVS routines to

- allocate parallel arrays
- gain access to the data and coordinates as CM Fortran arrays or C* parallel variables
- query whether or not a field is parallel

CM/AVS provides the following routines:

```
CMAVScorout_init
CMAVSdata_alloc
CMAVSfield_alloc
CMAVSfield_alloc_data_shape
CMAVSfield_alloc_points_shape
CMAVSfield_copy_points
CMAVSfield_data_get
CMAVSfield_points_get
CMAVSfield_reset_minmax
CMAVSis_field_on_CM
```

These routines should be used in conjunction with the standard AVS routines. Most of the standard AVS routines also work on parallel fields. The exceptions are listed in Table 1 in Chapter 3.

## A.1 CMAVScorout_init

Initializes a CM/AVS coroutine module.

C* Binding

```
#include <cm/cmavs.h>
void
    CMAVScorout_init(int argc, char *argv[],
         int (*desc)());
```

CMF Binding

```
include '/usr/include/cm/cmavs.inc'
SUBROUTINE CMAVScorout_init(desc)
external desc
```

This subroutine should be used instead of **AVScorout_init** to initialize a CM/AVS coroutine module. It must precede any other AVS or CM/AVS routines.

The subroutine sets up some internal data structures, then calls the user-supplied module description function **desc**.

For the C* interface, *argc* and *argv* are the same arguments that are passed to **main**.

## A.2 CMAVSdata_alloc

Allocates a parallel field based on a descriptive string.

C* Binding

```
#include <cm/cmavs.h>
void *
    CMAVSdata_alloc(char *string, int *dims);
```

CMF Binding

```
include '/usr/include/cm/cmavs.inc'
character*n string
integer dims()
integer function CMAVSdata_alloc(string,dims)
```

This routine allocates a parallel field based on a descriptive string. The behavior is similar to **`AVSdata_alloc`**, except that it allocates the data and points on the CM-5 processing nodes. The **`string`** argument is a descriptive string in the same form that is used for **`AVSdata_alloc`**.

**NOTE**

The **`string`** argument must describe a field; trying to allocate any other AVS object such as **`ucd`** or **`geom`** on the processing nodes will result in a fatal run-time error.

The **`dims`** argument is an array of integers that tells us how much space to allocate for this field on the CM-5. In C*, this routine returns a pointer to an AVSfield structure. In CM Fortran, it returns an opaque integer that can be used anywhere a parallel field is needed.

## A.3 CMAVSfield_alloc

Allocates a field structure for a parallel field using the given template field.

C* Binding

```
#include <cm/cmavs.h>
AVSfield *
    CMAVSfield_alloc(CMAVSfield *template, int *dims)
```

CM Fortran Binding

```
include /usr/include/cm/cmavs.inc'
integer template
integer dims()
integer function CMAVSfield_alloc(template, dims)
```

This routine allocates a field structure for a parallel field using the given template field. The template may be either a parallel or serial AVS field. The newly allocated field will always be a parallel field.

The *dims* argument is an array of integers that tells how much space to allocate for this field on the processing nodes. If you use zero in CM Fortran, or NULL in C*, instead of a dimensions array, the dimensions are taken from the template field.

In C*, this routine returns a pointer to an AVSfield structure. In CM Fortran, it returns an opaque integer that can be used anywhere a CM/AVS field is needed.

## A.4 CMAVSfield_alloc_data_shape

Allocates a C* shape that can contain the data from a field.

C* Binding

```
#include <cm/cmavs.h>
shape
    CMAVSfield_alloc_data_shape(AVSfield *field)
```

CM Fortran Binding

Not applicable.

This routine allocates a C* shape that can be used to refer to the data in any parallel field having the same rank and dimensions as the intput field. Note that the field's *veclen* does not matter; vectors become C* per-processor arrays that do not affect the choice of shape.

Each time you call **CMAVSfield_alloc_data_shape**, a new shape is allocated, even if you use the same field as input.

To deallocate the shape that this routine allocates, you must use the C* routine **deallocate_shape**. The field data is not affected when you free the shape that points to it.

## A.5 CMAVSfield_alloc_points_shape

Allocate a C* shape that can contain the points from a field.

C* Binding

```
#include <cm/cmavs.h>
shape
    CMAVSfield_alloc_points_shape(AVSfield *field)
```

CM Fortran Binding

Not applicable.

This routine allocates a C* shape that can be used to refer to the coordinates in any parallel field having the same rank and dimensions as the input field.

Do not call this routine on a uniform field; the coordinates for a uniform field can never reside on the processing nodes.

For a rectilinear field, this routine returns a one-dimensional shape with a number of positions equal to the sum of the field dimensions. For an irregular field, it returns a shape of rank *ndim*, where the number of positions in each axis is given by the dimensions array.

Each time you call **CMAVSfield_alloc_points_shape**, a new shape is allocated, even if you use the same field as input.

To deallocate the shape that this routine allocates, you must use the C* routine **deallocate_shape**. The field coordinates are not affected when you free the shape that points to them.

## A.6 CMAVSfield_copy_points

Copies points array from infield to outfield.

C* Binding

```
#include <cm/cmavs.h>
int CMAVSfield_copy_points(AVSfield *infield,
    AVSfield *outfield)
```

**CMF Binding**

```
include '/usr/include/cm/cmavs.inc'
integer infield, outfield
integer function
        CMAVSfield_copy_points(infield,outfield)
```

This routine copies a points array from an infield to an outfield. It works only if the points arrays are both on the partition manager or both on the processing nodes. The routine returns 1 on success, 0 on failure.

## A.7 CMAVSfield_data_get

Gets access to the data portion of a parallel field by returning a pointer to a C* parallel variable or filling in a CM Fortran array descriptor.

**C* Binding**

```
#include <cm/cmavs.h>
void:void *
    CMAVSfield_data_get(AVSfield *field, shape S)
```

**CM Fortran Binding**

```
    include '/usr/include/cm/CMF_defs.h'
    include '/usr/include/cm/cmavs.inc'
    integer field
    integer basevec(CMF_SIZEOF_DESCRIPTOR)
CMF$LAYOUT basevec(:serial)
    subroutine CMAVSfield_data_get(field, basevec)
```

This routine returns a pointer to a C* parallel variable or fills in a CM Fortran array descriptor, thereby giving access to the data portion of a parallel field. (The C* parallel variable is allocated in the specified shape.) Once the descriptor is loaded with a CM Fortran array descriptor, it may be passed to any CM Fortran routine that is expecting a parallel array of the appropriate rank.

Note the following:

- The C* interface takes both a field and a shape; it returns a pointer to a parallel variable. The parallel pointer refers to the data in the first argument (`AVSfield`). You must pre-allocate the shape using

**CMAVSfield_allocate_points_shape**. You may use the same shape to construct pointers to any fields that have the same rank and dimensions.

- If you dispose of the field (as with **AVSdata_free**) you should no longer refer to any arrays created from that field.
- A pointer returned by **CMAVS_field_data_get** can be invalidated if you make another call to **CMAVSfield_data_get** on the same field.
- If you use **CMAVSfield_data_get** on a field whose data resides on the partition manager, a fatal error occurs.

## A.8 CMAVSfield_points_get

Returns a pointer to the parallel coordinate data from a CM/AVS field.

C* Binding

```
#include <cm/cmavs.h>
float:void *
    CMAVSfield_points_get(AVSfield *field, shape S)
```

CMF Binding

```
    include '/usr/include/cm/CMF_defs.h'
    include '/usr/include/cm/cmavs.inc'
    integer field
    integer basevec(CMF_SIZEOF_DESCRIPTOR)
CMF$LAYOUT basevec(:serial)
    subroutine CMAVSfield_points_get
         (field, basevec)
```

This routine returns a pointer to the parallel coordinate data from a CM/AVS field. It works only when the coordinate array resides on the processing nodes, and it does not work on uniform fields.

Note the following:

- The C* interface takes both a field and a shape; it returns a pointer to a parallel variable. The parallel pointer refers to the data in the first argument (**AVSfield**). You must pre-allocate the shape using **CMAVSfield_allocate_points_shape**. You may use the same shape to construct pointers to any fields that have the same rank and dimensions.

- A pointer returned by **CMAVS_field_points_get** can be invalidated if you make another call to **CMAVSfield_points_get** on the same field.
- If you use **CMAVSfield_points_get** on a field whose data resides on the partition manager, a fatal error occurs.

## A.9 CMAVSfield_reset_minmax

Recomputes the minimim and maximum values for the field's computational data and stores those values with the field.

C* Binding

```
#include <cm/cmavs.h>
void
    CMAVSfield_reset_minmax(AVSfield *field)
```

CM Fortran Binding

```
    include '/usr/include/cm/cmavs.inc'
    integer field
    SUBROUTINE CMAVSfield_reset_minmax(field)
```

This routine recomputes the **min** and **max** values for the field's computational data and stores those values with the field. The routine works for both parallel and serial fields.

## A.10 CMAVSis_field_on_CM

Accepts a pointer to an AVS field, and returns true if the field is a parallel field.

### C* Binding

```
#include <cm/cmavs.h>
bool
    CMAVSis_field_on_CM(AVSfield *field);
```

### CM Fortran Binding

```
      include '/usr/include/cm/cmavs.inc'
      integer field
      logical function CMAVSis_field_on_CM(field)
```

# Appendix B

# CM/AVS Modules

This appendix contains man pages for the following CM/AVS modules, in alphabetical order:

| | |
|---|---|
| **antialias cm** | **fft cm** |
| **clamp cm** | **field math cm** |
| **color range cm** | **field to byte cm** |
| **colorizer cm** | **field to double cm** |
| **combine scalars cm** | **field to float cm** |
| **compare field cm** | **field to int cm** |
| **compute gradient cm** | **luminance cm** |
| **contrast cm** | **orthogonal slicer cm** |
| **downsize cm** | **threshold cm** |
| **extract scalar cm** | |

With the exception of **fft cm**, all the CM/AVS modules are AVS modules that have been adapted for the CM-5, and they may be interchanged with their AVS counterparts. For example, **downsize cm** and **downsize** are interchangeable.

See Appendix C and **/usr/examples/cmavs/unsupported/README** for information about unsupported modules.

## NAME

**antialias cm** - antialias an image

## SUMMARY

| | |
|---|---|
| **Name** | antialias cm |
| **Type** | filter |
| **Inputs** | field 2D uniform 4-vector byte *(image)* |
| **Outputs** | field 2D uniform 4-vector byte *(image)* |
| **Parameters** | none |

## DESCRIPTION

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

It should be noted that the CM implementation uses a different algorithm than the serial version. This will probably be corrected in a later release.

## INPUTS

| | |
|---|---|
| **Image** | (required; field 2D uniform 4-vector byte)<br>The image to be antialiased. |

## OUTPUTS

| | |
|---|---|
| **Image** | (field 2D uniform 4-vector byte)<br>The output antialiased image. This image is half the size of the input image in each dimension. |

## EXAMPLE 1

The following network reads an image, antialiases it on the CM-5, and displays it through the **image viewer**.

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

## RELATED MODULES

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

**colorizer cm**
**composite**
**convolve**
**field math cm**
**localops**
**read image**
**replace alpha**

Modules that can process **antialias cm** output:

**extract scalar cm**
**image viewer**
**display image**

## NAME

**clamp cm** - restrict values in data field to user-specified range

## SUMMARY

| | | | | | |
|---|---|---|---|---|---|
| **Name** | clamp cm | | | | |
| **Type** | filter | | | | |
| **Inputs** | field *any-dimension n-vector any-data any-coordinates* | | | | |
| **Outputs** | field of same type as input | | | | |
| **Parameters** | **Name** | **Type** | **Default** | **Min** | **Max** |
| | clamp_min | float | 0.0 | none | none |
| | clamp_max | float | 255.0 | none | none |

## DESCRIPTION

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

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

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

**clamp_min** <= *value* <= **clamp_max**

If appropriate, **clamp cm** 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 cm** works with uniform, rectilinear and irregular fields, whether they are vector or scalar.

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

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

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

## INPUTS

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

## PARAMETERS

| | |
|---|---|
| **clamp_min** | A floating-point number that specifies the minimum output value. |
| **clamp_max** | A floating-point number that specifies the maximum output value. |

## OUTPUTS

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

## EXAMPLE

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

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

## RELATED MODULES

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

**read volume**
*any other filter module*

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

**threshold cm**

Modules that can process **clamp cm** output:

**colorizer cm**
*any other filter module*

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

**statistics**
**print field**
**generate histogram**

## SEE ALSO

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

## NAME

**color range cm** - scale AVS colormap to the range of data in a parallel field

## SUMMARY

| | |
|---|---|
| **Name** | color range cm |
| **Type** | data |
| **Inputs** | field (*any-dimension* scalar *any-data any-coordinates*)<br>colormap |
| **Outputs** | colormap |
| **Parameters** | *none* |

## DESCRIPTION

**color range cm** adjusts the minimum and maximum values of a colormap to those of a parallel field, thus normalizing the colormap to the range of the data in the field. To do this, **color range cm** examines a parallel 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 cm** also stores the minimum and maximum data values into its output AVS colormap data structure.

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

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

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

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

**Color Map** (required; colormap)
This is the original AVS colormap. Any minimum or maximum values that may have been set in the input colormap are ignored.

## OUTPUTS

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

## EXAMPLE

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

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

## RELATED MODULES

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

**read field**
**extract scalar cm** (for fields with vectors)

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

**generate colormap**

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

**arbitrary slicer**
**bubbleviz**
**colorizer cm**
**field legend**
**field to mesh**
**isosurface**
**probe**

## SEE ALSO

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

## NAME

**colorizer cm** - convert field of data values to color values

## SUMMARY

| | |
|---|---|
| **Name** | colorizer cm |
| **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 cm** 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*:

```
                                         COLORMAP

                           Aux       Red        Green      Blue
                                     Value      Value      Value
                       |_________|_________|_________|_________|
  Input             1  |_________|_________|_________|_________|
  Value             2  |_________|_________|_________|_________|
________            3  |_________|_________|_________|_________|
e.g. 147__
          |       146  |_________|_________|_________|_________|
          +--> 147  |_________|_________|_________|_________| Output
                  148  |_________|_________|_________|_________|   Value
                       |         |         |         |         |
```

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

## INPUTS

**Data Field**　(required; field *any-dimension* scalar *any-coordinates*) The principal input data for the **colorizer cm** module is a field, which can be of any dimensionality. The data at each point of the field may be of any data type.

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

## OUTPUTS

**Field of Colors** (field *any-dimension* 4-vector byte *any-coordinates*) Each input value is transformed into a color value, which is structured as four bytes, as illustrated above. The red, green, and blue bytes specify a true-color pixel value. The *auxiliary* byte is typically used to specify an opacity value (lo-value = completely transparent; hi-value = completely opaque).

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

The **min_val** and **max_val** attributes of the output field are invalidated. The dimensions of the 4-vector output data are assigned the labels "Alpha", "Red", "Green", and "Blue".

## EXAMPLE

The following network reads in an AVS image, which is a 2D field of 4-vector bytes. **extract scalar cm** 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 cm**:

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

## RELATED MODULES

Modules that could provide the **Data Field** input: **read volume field to byte** Modules that could provide the **Color Map** input: **generate colormap** Modules that could be used in place of **colorizer cm: arbitrary slicer** Modules that can process **colorizer cm** output: **alpha blend gradient shade display image tracer**

## SEE ALSO

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

## NAME

**combine scalars cm** - combine scalar fields into a vector field

## SUMMARY

| | |
|---|---|
| **Name** | combine scalars cm |
| **Type** | filter |
| **Inputs** | field *any-dimension* scalar *any-data any-coordinates* (channel 0 — optional) |
| | field *any-dimension* scalar *any-data any-coordinates* (channel 1 — optional) |
| | field *any-dimension* scalar *any-data any-coordinates* (channel 2 — optional) |
| | field *any-dimension* scalar *any-data any-coordinates* (channel 3 — optional) |
| **Outputs** | field *same-dimension* 1D–4D *same-data* |

**Parameters**

| Name | Type | Default | Min | Max |
|---|---|---|---|---|
| Vector | Len | Dial | 4 | 14 |

## DESCRIPTION

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

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

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

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

## INPUTS

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

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

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

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

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

## PARAMETERS

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

## OUTPUTS

**Field** (field *same-dimension* 1D–4D *same-data*) The scalar input streams are assembled into a single output stream consisting of vectors, whose dimension is specified by **Vector Length.** The coordinate type (e.g. uniform, rectilinear, or irregular) of the output field is the same as the leftmost, non-empty input field. The field's **min_val, max_val, veclen, label,** and **unit** are updated.

## EXAMPLE 1

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

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

## EXAMPLE 2

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

```
                            READ IMAGE
                                 |
                                 |
             ____________________|____________________
             |                   |                   |
   EXTRACT SCALAR [CM]  EXTRACT SCALAR [CM]  EXTRACT SCALAR [CM]
         [red]               [green]               [blue]
             |_______________    |______     __________|
                            |          |     |
                            |    /---- | ----'
                            |    |     |
                       COMBINE SCALARS CM
                                 |
                          DISPLAY IMAGE
```

## RELATED MODULES

**extract scalar cm**

## SEE ALSO

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

## NAME

**compare field cm** - compare two fields, display and write data difference

## SUMMARY

| | |
|---|---|
| **Name** | compare field cm |
| **Type** | data output |
| **Inputs** | field *any-dimension n-vector any-data any-coordinates*<br>field *same-dimension same-vector same-data same-coordinates* |
| **Outputs** | none |

**Parameters**

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

## DESCRIPTION

The **compare field cm** module compares *any* two identically-structured AVS fields. It will print out differences between the headers if they are different. If the headers are the same, it will proceed to do a comparison of the data contents of the two fields. If the fields are not identical in their data components, **compare field cm** 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 cm** displays its output can be resized, like any other widget, using the AVS Layout Editor. For a detailed description of how to do this, see the section titled "Layout Editor," in the chapter "Advanced Network Editor" of the *AVS User's Guide*.

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

## INPUT

**Input Field 1**　(required; field *any-dimension n-vector any-data any-coordinates*) The input field can be 1, 2, 3, or 4 dimensional; it can be vector or scalar, can contain byte, int, float or double data, and can have uniform, rectilinear, or irregular coordinates.

**Input Field 2** (required; field *any-dimension n-vector any-data any-coordinates*) The second input field must match the first in the number of dimensions (Ndim), the size of each dimension (Dims), the number of coordinate dimensions (Nspace), the vector length (Veclen), the data type (byte, float, double, etc.), and the type of coordinate system (uniform, rectilinear, curvilinear), if a comparison of the two fields' data is to be done.

## PARAMETERS

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

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

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

## EXAMPLE 1

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

```
              READ IMAGE
                   |
                   |
     --------------|
     |             |
THRESHOLD CM       |
     |             |
     --------|     |
             |     |
          COMPARE FIELD CM
```

## RELATED MODULES

**print field**

## LIMITATIONS

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

## SEE ALSO

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

## NAME

**compute gradient cm** - compute gradient vectors for 2D or 3D data set

## SUMMARY

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

**Parameters**

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

## DESCRIPTION

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

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

```
          -------------------------
          |       |       |       |
   Y-1    |       | X,Y-1 |       |
          |       |       |       |
          -------------------------
          |       |       |       |
    Y     | X-1,Y | X, Y  | X+1,Y |
          |       |       |       |
          -------------------------
          |       |       |       |
   Y+1    |       | X,Y+1 |       |
          |       |       |       |
          -------------------------
             X-1      X      X+1

   Delta_x[X][Y]  = data[X-1][Y]  - data[X+1][Y]
   Delta_y[X][Y]  = data[X][Y-1]  - data[X][Y+1]

   Delta_z[X][Y]  = 2D Height Dial                       for 2D data
   Delta_z[X][Y][Z]  = data[X][Y][Z-1]  - data[X][Y][Z+1]   for 3D data
```

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

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

## INPUTS

**Data Field** (required; field 2D/3D scalar byte *any-coordinates*) The input field may be either 2D or 3D. The data at each point of the field must be a single byte. The byte values will be interpreted as integers in the range 0..255.

## PARAMETERS

**2D Height** (appears for 2D data only) Supplies the Z-coordinate of the gradient. It can be used 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.

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

## OUTPUTS

**Data Field** (field *same-dimension* 3-vector real *same-coordinates*) The output field has the same dimensionality as the input field. For each element, the output data is a 3D vector of reals, representing the 3D gradient.

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

**EXAMPLE 1**

The following network shades a 2D image:

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

**EXAMPLE 2**

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

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

**EXAMPLE 3**

The following network shades a 3D image:

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

## RELATED MODULES

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

## LIMITATIONS

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

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

## SEE ALSO

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

## NAME

**contrast cm** - 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 of same type as input |

**Parameters**

| Name | Type | Default | Min | Max |
|---|---|---|---|---|
| cont_in_min | float | 0.0 | *none* | *none* |
| cont_in_max | float | 255.0 | *none* | *none* |
| cont_out_min | float | 0.0 | *none* | *none* |
| cont_out_max | float | 255.0 | *none* | *none* |

## DESCRIPTION

The **contrast cm** module transforms all the values in a field. Two different types of transformation take place:

- o **Linear transform:** All values that fall within the "input range" specified by the **cont_in_min** and **cont_in_max** parameters are transformed linearly to the "output range" **cont_out_min .. cont_out_max.**

$$new\_value = \frac{(cont\_out\_max - cont\_out\_min) * (value - cont\_in\_min)}{(cont\_in\_max - cont\_in\_min)}$$

  (More precisely, this is an *affine transformation.*) In essence, this transformation "stretches" or "compresses" one specified range of data to fit another specified range.

- o All values that fall outside the specified input range are "clamped" to the limit values of the output range.

The **contrast cm** module typically is used to remove low-level noise from images and volumes, or to increase the contrast in faded images and volumes.

## INPUTS

**Data Field** (required; field *any-dimension n-vector any-data any-coordinates*) The input data may be an AVS field of any dimensionality.

## PARAMETERS

| | |
|---|---|
| **cont_in_min** | Specifies the bottom of the range of input values that will be transformed linearly. |
| **cont_in_max** | Specifies the top of the range of input values that will be transformed linearly. |
| **cont_out_min** | Specifies the bottom of the range of output values. All values ≤ **cont_in_min** will be transformed to this value. |
| **cont_out_max** | Specifies the top of the range of output values. All values ≥ **cont_in_max** will be transformed to this value. |

## OUTPUTS

| | |
|---|---|
| **Data Field** | The output field has the same dimensionality and type as the input field. |

If the input field has byte values, appropriate new **min_val** and **max_val** values are written to the output field.

## EXAMPLE 1

The following diagram shows how field values are transformed given these parameters:

**cont_in_min** = 100
**cont_in_max** = 500
**cont_out_min** = 3000
**cont_out_max** = 6000

```
Outputs
        |      |                            |
 6000   ------------------------------------xxxxxxxxxxxxxxxxxx
        |      |                       x    |
        |      |                 x          |
        |      |           x                |
        |      |     x                      |
 3000   xxxxxxxx----------------------------|-----------------
        |      |                            |
        |      |                            |
        |      |                            |
        |      |                            |
        +------|----------------------------|-----------------
              100                          500
                         Inputs
```

You can use **contrast cm** to make a negative out of an image by "flipping" the output values (e.g. **cont_out_min** = 255; **cont_out_max** = 0).

## EXAMPLE 2

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

```
                         READ IMAGE
                              |
      |---------------------|---------------------|
      |                     |                     |
   EXTRACT               EXTRACT               EXTRACT
  SCALAR CM             SCALAR CM             SCALAR CM (red)
      |                     |                     |
      |                     |                 CONTRAST CM
      |                     |                     |
      |---------------------|---------------------|
                            |
                    COMBINE SCALARS CM
                            |
                            |
                      DISPLAY IMAGE
```

## RELATED MODULES

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

**read volume**

## SEE ALSO

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

## NAME

**downsize cm** - reduce size of data set by sampling

## SUMMARY

| | |
|---|---|
| **Name** | downsize cm |
| **Type** | filter |
| **Inputs** | field 2D/3D *n-vector any-data any-coordinates* |
| **Outputs** | field of same type as input |

**Parameters**

| Name | Type | Default | Min | Max |
|---|---|---|---|---|
| downsize | integer | 8 | 1 | 16 |

## DESCRIPTION

The **downsize cm** module changes the size of the input data set by subsampling the data. It extracts every *n*th element of the field along each dimension, where *n* is the value of the **downsize factor** parameter. This technique preserves the aspect ratio of the input data.

This module is useful for operating on a reduced amount of data, in order to adjust other processing parameters interactively, or save memory. After the parameter values have been set, you can remove the **downsize cm** module, so that the full data set is used for final processing.

Alternatively, retain the **downsize cm** module in the network, so that you can interactively choose between image quality (**downsize factor** = 1 for highest-resolution data) and execution speed (**downsize factor** > 1 for lower-resolution data).

## INPUTS

**Data Field**　(required; field 2D/3D *n-vector any-data any-coordinates*)
The input data may be any AVS field.

## PARAMETERS

**downsize**　Determines how data elements from the field are sampled. Increasing this parameter causes more elements to be skipped over, thus *decreasing* the size of the output.

## OUTPUTS

**Data Field**

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

The **min_val** and **max_val** attributes of the output field are invalidated. Note that the extent is unmodified; this module changes the resolution of the data within the physcial space delimited by the extents. It does not alter the physical extents of the data.

## EXAMPLE

The following diagram shows how a **downsize factor** of 4 reduces a 2D field. Each element of the field is represented by a hyphen or an o. Only the o's are included in the output field.

```
o - - - o - - - o - - - o
- - - - - - - - - - - - -
- - - - - - - - - - - - -
- - - - - - - - - - - - -
o - - - o - - - o - - - o
- - - - - - - - - - - - -
- - - - - - - - - - - - -
- - - - - - - - - - - - -
o - - - o - - - o - - - o
- - - - - - - - - - - - -
- - - - - - - - - - - - -
- - - - - - - - - - - - -
o - - - o - - - o - - - o
```

## LIMITATIONS

**downsize cm** works for 2D, and 3D data sets only.

## RELATED MODULES

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

**read volume**
**read field**
**filter modules**

## SEE ALSO

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

## NAME

**extract scalar cm** - extract a scalar field from a vector field

## SUMMARY

| | | | |
|---|---|---|---|
| **Name** | extract scalar cm | | |
| **Type** | filter | | |
| **Inputs** | field *any-dimension n-vector any-data any-coordinates* (*n* = 1..25) | | |
| **Outputs** | field *same-dimension* scalar *same-data same-coordinates* | | |
| **Parameters** | **Name** | **Type** | **Default** |
| | Channel *n* | radio buttons | Channel 0 |

## DESCRIPTION

The **extract scalar cm** 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 cm** module, which composes vector fields from individual scalar fields.

## INPUTS

**Data Field** (required; field *any-dimension n*-vector *any data any-coordinates*) The input data may be any field whose data values are vectors with 25 or fewer dimensions. Even scalar fields may be used, since their data values are considered to be 1D vectors.

## PARAMETERS

**Channel** *n* Selects the dimension of the input data values to be output. A set of radio buttons appears, showing the labels that are attached to the dimensions of the *n*-vector data.

## OUTPUTS

**field** (*same-dimension* scalar *same-data same-coordinates*)
The output field has the same dimensionality as the input field. The data for each element is reduced from a vector to a scalar. The *veclen*, *min_val*, *max_val*, *label*, and *unit* values in the field are updated.

## EXAMPLE 1

This examples displays a slice of the Y-component of the gradient field of a volume:

```
                      READ VOLUME
                           |
                    COMPUTE GRADIENT
                           |
                 EXTRACT SCALAR CM [1]    (0=X, 1=Y, 2=Z)
                           |
                  ORTHOGONAL SLICER CM
                           |
GENERATE COLORMAP   FIELD TO BYTE CM
        |_______________   |
                       |   |
                     COLORIZER CM
                           |
                     DISPLAY IMAGE
```

For additional examples, see the **combine scalars cm** manual page.

## RELATED MODULES

**combine scalars cm**

## SEE ALSO

The AVS example scripts CONTOUR GEOMETRY, CONTRAST, as well as others demonstrate the AVS **extract scalar** module. The **extract scalar cm** module may be substituted in many of these examples.

## NAME

**fft cm** - do a Fast Fourier Transform on a field

## SUMMARY

| | |
|---|---|
| **Name** | fft cm |
| **Type** | filter |
| **Inputs** | field float (each axis must be length power of two) |
| **Outputs** | field |
| **Parameters** | op |

## DESCRIPTION

The **fft cm** module takes a floating point {1|2|3}D 2-vector and, depending on the "op" parameter, does either a forward or inverse Fast Fourier Transform on it. The module uses the simple FFT routine in the Thinking Machines CMSSL library.

## INPUTS

| | |
|---|---|
| **field** | (required; field 1D, 2D, or 3D 2-vector float)<br>The field to be transformed. |

## OUTPUTS

| | |
|---|---|
| **field** | (required; field 1D, 2D, or 3D 2-vector float)<br>The transformed field. |

## EXAMPLE

The following network reads an image, crops it to dimensions which are a power of two, pulls out a 2-vector, converts it to floating point, does a forward FFT, then an inverse FFT, coverts the data back to bytes, extracts the first channel, and recombines it into a greyscale image, and displays it. The result should be the greyscale equivalent of the first extracted channel of the image.

```
  READ IMAGE
      |
     CROP
      |
EXTRACT VECTOR
      |
FIELD TO FLOAT CM
      |
FFT CM (forward)
      |
FFT CM (inverse)
      |
FIELD TO BYTE CM
      |
EXTRACT SCALAR CM
      |
COLORIZER CM
      |
IMAGE VIEWER
```

## RELATED MODULES

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

**read field**

Modules that can process **fft cm** output:

**write field**
**image viewer**
**display image**

## NAME

**field math cm** - perform math operations between fields

## SUMMARY

| | |
|---|---|
| **Name** | field math cm |
| **Type** | filter |
| **Inputs** | field *any-dimension n-vector any-data any-coordinates*<br>field *same-dimension same-vector any-data same-coordinates* (OPTIONAL) |
| **Outputs** | field *same-dimension same-vector any-data same-coordinates* |

**Parameters**

| **Name** | **Type** | **Default** | **Min** | **Max** |
|---|---|---|---|---|
| choice | choice | + | | |
| Normalize | boolean | off | | |
| Constant | float typein | 0.0 | unbounded | unbounded |

## DESCRIPTION

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

The unary operations are Not, Square, and Sqrt. The binary operations are +, -, *, /, And, Or, Xor, Left-Shift, Right-Shift, and RMS (Root Mean Square). Unary operations are performed against the right port field only. The field that is connected to the left port is ignored. If only one field is provided as an operand for a binary operations, the field must be attached to the right port and the binary operations are performed on the right port field and the **Constant** input parameter.

When two fields are connected to the module, the **Constant** parameter is not displayed and the fields are evaluated against each other.

The input fields must be of the same dimensionality, size, and vector length. When the fields contain different data types, the output field will have the more elaborate data type.

When the fields have different coordinate types, the output field will have the same coordinate type as the right input port field.

During computation, byte data is converted to integer, while short, integer, and float data are converted to double. The result is then converted back to the appropriate output data type. If **Normalize** is off, the data is "clamped" to the range:

```
0...255]                        byte
[-32767...32767]                short
[-2147483647...2147483647]      integer
```

If **Normalize** is on, the result is normalized to between:

| | |
|---|---|
| `[0...255]` | `byte` |
| `[0...32767]` | `short` |
| `[0...2147483647]` | `integer` |
| `[0...1]` | `float, double` |

## INPUTS

**Data Field** (required; field *any-dimension n-vector any-data any-coordinates*)
The rightmost input field is used as the input to unary operations, or the first operand for binary operations.

**Data Field** (optional; field *same-dimension same-vector any-data same-coordinates*)
The left field is the second operand in binary operations. It must have the same dimension, size, and vector length as the first input field.

## PARAMETERS

**+**

**-**

*****

**/**

**And** (bitwise)

**Or** (bitwise)

**Xor** (bitwise)

**Not** (bitwise)

**Left-Shift** (bitwise)

**Right-Shift** (bitwise)

**Square**

**Sqrt**

**RMS** (Root Mean Square)
A choice of operations. For binary operations, if the left port field (field2) is not provided, the **Constant** parameter is used as the second operand (i.e. field2 is replaced by

**Constant).**

```
+               field1  +  field2
-               field1  -  field2
*               field1  *  field2
/               field1  /  field2 (result is 0 if field2 is 0)
And             field1 AND field2 |
Or              field1 OR  field2 |
Xor             field1 XOR field2 | not applicable for
Not             NOT field1        | floats and doubles
Left-Shift      field1 << field2  |
Right-Shift     field1 >> field2  |
Square          field1 * field1
Sqrt            sqrt (field1)
RMS             sqrt (field1**2 + field2**2)
```

**Normalize** Selecting **Normalize** causes the results of the operation to be normalized to between 0 and 1 for floats and doubles, 0 and 255 for bytes, 0 and 32767 for shorts, and 0 and 2147483647 for integers. **Normalize** is off by default.

**Constant** A floating point typein to specify the constant value to use as the second operand in binary operations. If two fields are connected to the module, **Constant** is ignored and disappears from the control panel. The default is 0.0. There is no upper or lower limit.

## OUTPUTS

**Data Field** (field *same-dimension same-vector any-data same-coordinates*)
The output field has the same form as the input fields.
If the input fields were of different data types, the output field will have the more elaborate data type. If the input fields had different coordinate types, the output field will have the same coordinate type as the right input port field.

## EXAMPLE 1

The following network inverts (flips the look-up table) an image using the Not function, with Normalize on. The same effect can be achieved by multiplying the image by -1.

```
 READ IMAGE
     |
FIELD MATH CM
     |
DISPLAY IMAGE
```

## EXAMPLE 2

This network does a logical AND on a volume against the constant 128 (0x80), which produces a volume with only 0s and 255s based on whether the source voxel was greater or less than 128.

```
        READ VOLUME
             |
       FIELD MATH CM
             |
  ORTHOGONAL SLICER CM
             |
        COLORIZER CM
             |
       DISPLAY IMAGE
```

## RELATED MODULES

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

Any module that outputs a field

Modules that can process **field math cm** output:

Any module that inputs a field

## SEE ALSO

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

## NAME

**field to byte cm** - transform any field to a byte-valued field

## SUMMARY

| | |
|---|---|
| **Name** | field to byte cm |
| **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 cm** 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 (i.e., **compute gradient cm**).

By default, the input data is normalized to the range 0..255 If the toggle parameter **byte_normalize** is turned off, the data is "clamped" to that range instead. (See below for details.)

## INPUTS

**Data Field** (required; field *any-dimension n-vector any-data any-coordinates*) The input data may be any AVS field.

## PARAMETERS

**byte_normalize** This is a toggle parameter:

**If on:** The data is transformed linearly into the range 0..255:

```
                 (value - min) * 255
new_value =    --------------------
                     max  -  min
```

**If off:** The data is "clamped" so that no value falls outside the range 0..255:

| | |
|---|---|
| If *value* < 0 | *new_value* = 0 |
| If 0 <= *value* <= 255 | *new_value* = *value* |
| If *value* > 255 | *new_value* = 255 |

## OUTPUTS

**Data Field** (field *same-dimension same-vector* byte *same-coordinates*)
The output field has the same dimensionality as the input field, but each scalar value is forced to be a byte.

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

## RELATED MODULES

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

**read volume**

Modules that could be used in place of **field to byte cm**:

**field to int cm**
**field to float cm**
**field to double cm**

Modules that can process **field to byte cm** output:

**read volume**

## SEE ALSO

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

## NAME

**field to double cm** - transform any field to a field of double-precision floating point values

## SUMMARY

| | |
|---|---|
| **Name** | field to double cm |
| **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 | on | on, off |

## DESCRIPTION

The **field to double cm** module takes a field of data (*byte*, *real*, *double*, or *integer*) and converts it to an *double* field. This may be useful for computing fields at greater data resolutions.

By default, the input data is simply cast (re-typed) to be double-precision floating point. If the toggle parameter **double_normalize** is turned on, the data is also normalized to the range 0..1. (See below for details.)

## INPUTS

**Data Field** (required; field *any-dimension n-vector any-data any-coordinates*) The input data may be any AVS field.

## PARAMETERS

**double_normalize** This is a toggle parameter:

**If on:** The data is transformed linearly into the range 0..1:

```
                      (value - min)
new_value =      -------------------
                       max  -  min
```

**If off:** The data is converted to double-precision floating point format.

## OUTPUTS

**Data Field** (field field *same-dimension same-vector* double *same-coordinates*
The output field has the same dimensionality as the input field, but each scalar value is forced to be a double-precision number.

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

## RELATED MODULES

**read volume**
**field to byte cm**
**field to int cm**
**field to float cm**

## SEE ALSO

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

## NAME

**field to float cm** - transform any field to a field of single-precision floating point values

## SUMMARY

| | |
|---|---|
| **Name** | field to float cm |
| **Type** | filter |
| **Inputs** | field field *any-dimension n-vector any-data any-coordinates* |
| **Outputs** | field *same-dimension same-vector* float *same-coordinates* |

**Parameters**

| **Name** | **Type** | **Default** | **Choices** |
|---|---|---|---|
| float normalize | toggle | off | on, off |

## DESCRIPTION

The **field to float cm** module takes a field of data (*byte*, *short*, *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 (re-typed) to be single-precision floating point. If the toggle parameter **float normalize** is turned on, the data is also normalized to the range 0..1. (See below for details.)

## INPUTS

**Data Field** (required; *any-dimension n-vector any-data any-coordinates*)
The input data may be any AVS field.

## PARAMETERS

**float normalize** This is a toggle parameter:

**If on:** the data is transformed linearly into the range 0..1:

```
                     (value - min)
new_value =     -------------------
                      max  -  min
```

**If off:** the data is converted to single-precision floating point format.

## OUTPUTS

| | |
|---|---|
| **Data Field** | (field *same-dimension same-vector* float *same-coordinates* The output field has the same dimensionality as the input field, but each scalar value is forced to be a single-precision number. Appropriate new values of the **min_val** and **max_val** attributes are written to the output field. |

## RELATED MODULES

**read volume**
**particle advector**
**samplers**
**field to byte cm**
**field to short**
**field to int cm**
**field to double cm**

## SEE ALSO

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

## LIMITATIONS

Overflow or underflow may occur when converting a double field to a float field with **float normalize** turned off.

## NAME

**field to int cm** - transform any field to an integer-valued field

## SUMMARY

| | | | | |
|---|---|---|---|---|
| **Name** | field to int cm | | | |
| **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 | on | on, off |

## DESCRIPTION

The **field to int cm** module takes a field of data (*byte*, *short*, *real*, *double*, or *int*) and converts it to an *int* field. This may be useful for performing integer math with greater precision (-231-1 to 231-1, -2147483647...2147483647) than that offered by byte fields (0..255).

By default, the input data is "clamped" to the range -231-1...231-1. If the toggle parameter **int_normalize** is turned on, the data is normalized to 0...231-1 instead. (See below for details.)

## INPUTS

**Data Field** (required; field *any-dimension n-vector any-data any-coordinates*) The input data may be any AVS field.

## PARAMETERS

**int normalize** This is a toggle parameter:

**If on:** the data is transformed linearly into the range 0..231-1:

```
                    (value - min) * 2147483647
new_value =    -------------------------
                         max  -  min
```

| | |
|---|---|
| **If off:** | the data is "clamped" so that no value falls outside the range -2147483647...2147483647. Values greater than 2147483647 are set to 2147483647. Values less than -2147483647 are set to -2147483647. |

## OUTPUTS

**Data Field** (field *same-dimension same-vector* integer *same-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.

## RELATED MODULES

**field to byte cm**
**field to short**
**field to float cm**
**field to double cm**

## SEE ALSO

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

## NAME

**luminance cm** - compute the luminance of an image

## SUMMARY

| | |
|---|---|
| **Name** | luminance cm |
| **Type** | filter |
| **Inputs** | field 2D uniform 4-vector byte *(image)* |
| **Outputs** | field 2D uniform scalar byte |
| **Parameters** | none |

## DESCRIPTION

The **luminance cm** module computes the luminance (brightness) of an image, then outputs a 2-dimensional field of the same dimensions, but with a *scalar* byte value for each pixel in the original image instead of the full four-byte alpha, red, green, blue vector.

The luminance (I) is calculated as follows:

$$I = (0.299 * \text{red}) + (0.587 * \text{green}) + (0.114 * \text{blue})$$

This luminance byte value can be used to produce a black and white version of the original image (with **colorizer cm**), or substituted back into the alpha byte of the original image (with **replace alpha**) to produce transparency effects.

## INPUTS

| | |
|---|---|
| **Image** | (required; field 2D uniform 4-vector byte)<br>The image whose luminance to calculate. |

## OUTPUTS

| | |
|---|---|
| **Data Field** | (field 2D uniform scalar byte)<br>The output field has the same dimension as the input image, but with a scalar byte value representing the image luminance at each original pixel instead of color value. |

## EXAMPLE 1

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

```
                          READ IMAGE
                              |
                        LUMINANCE CM
                              |
                        COLORIZER CM
                              |
                        IMAGE VIEWER
```

## EXAMPLE 2

This network takes a geometry, displays it on the screen, then converts the screen pixmap to an image, computes its luminance, uses that to create an alpha mask, renders a shaded background and composites the rendered image over the shaded background. The **contrast** modules controls should be set to : minimum and maximum input contrast, both 1; minimum output contrast 0, and maximum output contrast, 255. If the original geometry were */usr/avs/data/geometry/jet.geom* and the **background** module were set to produce a sky-like pattern, this would produce a jet over a sky field.

```
                               READ GEOM
                                   |
                            GEOMETRY VIEWER
                                   |----------------------|
                                   |                      |
     ______________________________|                DISPLAY IMAGE
     |                |            |
 BACKGROUND     LUMINANCE CM       |
     |                |            |
     |          CONTRAST CM        |
     |                |------|     |
     |                  REPLACE ALPHA
     |                             |
     |---------------------|       |
                            COMPOSITE
                                |
                           IMAGE VIEWER
```

## RELATED MODULES

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

Any module that produces an image as output

Modules that can process **luminance cm** output:

**colorizer cm**
**contrast cm**
Any modules that can process a 2D scalar field

Other related modules:
**background**
**composite**
**replace alpha**
**extract scalar cm**

## SEE ALSO

The AVS example script LUMINANCE demonstrates the AVS **luminance** module.

## NAME

**orthogonal slicer cm** - slice through 3D or 2D field with plane perpendicular to coordinate axis

## SUMMARY

| | |
|---|---|
| **Name** | orthogonal slicer cm |
| **Type** | mapper |
| **Inputs** | field 3D or 2D *n-vector any-data any-coordinates* |
| **Outputs** | field 2D or 1D *n-vector same-data same-coordinates* |

| **Parameters** | **Name** | **Type** | **Default** | **Min** | **Max** | **Choices** |
|---|---|---|---|---|---|---|
| | slice plane | int | 0 | 0 | 255 | on, off |
| | axis | choice | K | | | I,J,K |

## DESCRIPTION

The **orthogonal slicer cm** 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, however, the **orthogonal slicer cm** module can take slices of vector-valued fields, also. It passes through whatever data type is presented to it; e.g. if the input is a "field 3D 3-vector float", the output is a "field 2D 3-vector float".

## INPUTS

**Data Field** (field 2D/3D *n-vector any-data any-coordinates*)
The input may be any 3D or 2D *field.*

## PARAMETERS

**slice plane** Determines the value of the array index to be held constant. This value is reset to zero each time a new data field is input.

**axis** Selects the dimension (I, J, or K) in which the array index is to be held constant.

## OUTPUTS

**Data Field** (field 1D/2D *n-vector any-data any-coordinates*)
The output field is 2D instead of 3D (or 1D instead of 2D), and has the same type of data as the input field.

Appropriate new values for **min_ext** and **max_ext** are written to the output field.

## EXAMPLE 1

The following network takes a slice from a scalar volume and displays it:

```
      READ VOLUME
           |
   ORTHOGONAL SLICER CM
           |            GENERATE COLORMAP (optional)
           |   |-----------------|
           |   |
      COLORIZER CM
           |
     DISPLAY IMAGE
```

The **colorizer cm** module is necessary because the output of **orthogonal slicer cm** is a "field 2D scalar byte", which must be cast into an AVS *image* field for display.

**EXAMPLE 2**

For reasonably small volumes, a better way to construct this network is:

```
   READ VOLUME
       |           GENERATE COLORMAP (optional)
       |   |----------------|
       |   |
   COLORIZER CM
       |
ORTHOGONAL SLICER CM
       |
  DISPLAY IMAGE
```

This network has the effect of colorizing the entire volume once, which make the slicing operation more efficient. It does this at the expense of allocating more memory up front.

**EXAMPLE 3**

**Irregular Data: orthogonal slicer cm** supports the passing of "points" data for *rectilinear* and *irregular* data. This is an important module for visualizing curved data sets. For example:

```
                               READ FIELD (irregular data)
                         ____________|____________
                         |                       |
               ORTHOGONAL SLICER CM               |
GENERATE COLORMAP        |                       |
        |_________       |                VOLUME BOUNDS
                 |       |                       |
               FIELD TO MESH                     |
                     |___________       _________|
                                |       |
                             GEOMETRY VIEWER
```

(This is the reason for labeling the axis control with "I, J, and K": frequently, the data is *not* aligned to the X, Y, and Z axes. **orthogonal slicer cm** takes slices through the logical data set, not the physical one.)

## EXAMPLE 4

The following network shows how to use **orthogonal slicer cm** to plot the values of one scan-line of an image:

```
          READ IMAGE
               |
               |
        EXTRACT SCALAR CM
               |
               |
  ORTHOGONAL SLICER CM (set to middle of image)
               |
               |
         GRAPH VIEWER
```

## RELATED MODULES

**field to mesh**
**colorizer cm**

## SEE ALSO

The AVS example scripts ANIMATED INTEGER, COLOR RANGE, and VECTOR CURL demonstrate the AVS **orthogonal slicer** module.

## NAME

**threshold cm** - restrict values in data field

## SUMMARY

| | | | | | |
|---|---|---|---|---|---|
| **Name** | threshold cm | | | | |
| **Type** | filter | | | | |
| **Inputs** | field *any-dimension n-vector any-data any_coordinates* | | | | |
| **Outputs** | field of same type as input | | | | |
| **Parameters** | **Name** | **Type** | **Default** | **Min** | **Max** |
| | thresh_min | float | 0.0 | none | none |
| | thresh_max | float | 255.0 | none | none |

## DESCRIPTION

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

- o Any value less than the value of the **threshold_min** parameter is set to 0.
- o Any value greater than the value of the **threshold_max** parameter is set to 0.
- o Values within the **threshold_min**-to-**threshold_max** range are not changed.

After being **threshold**'ed, a data set's values are all either zero, or in this range:

**thresh_min** <= *value* <= **thresh_max**

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

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

## INPUTS

**Data Field** (required; field *any-dimension n-vector any-data any_coordinates*)
The input data may be any AVS field.

## PARAMETERS

| | |
|---|---|
| **thresh_min** | The minimum threshold value. |
| **thresh_max** | The maximum threshold value. |

## OUTPUTS

**Field Data**　The output field has the same dimensionality as the input field.

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

## RELATED MODULES

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

**read volume**
*any other filter module*

Modules that could be used in place of **threshold cm**:

**clamp cm**

Modules that can process **threshold cm** output:

**colorizer cm**
*any other filter module*

## SEE ALSO

The AVS example scripts CONTOUR GEOMETRY, and THRESHOLDED SLICER demonstrate the AVS **threshold** module.

Appendix C

# Unsupported Programs and Modules

This appendix contains information about programs and modules that are included with CM/AVS but are not guaranteed or supported.

These items reside in the directory **`/usr/examples/cmavs/unsupported`**.

## C.1 Programs

The following programs are documented in this appendix:

**`avstoppm`**

**`ppmtoavs`**

## C.2 Modules

The following modules are documented in this appendix:

**`field to polygons`**

**`field to spheres`**

## NAME

**avstoppm** - convert AVS images (.x format) to PPM format

## SYNOPSIS

**avstoppm** [ *infile* ] [ *outfile* ]

## DESCRIPTION

**avstoppm** converts images from AVS format (filenames end in .x) to PPM format. If no filenames are given, **avstoppm** reads from *stdin* and writes to *stdout*. If one filename is given, it is the input filename, and output goes to *stdout*. If two filenames are given, the first is the input file and the second is the output file (which is first truncated if it exists).

NOTE: The PPM format is part of the **pbmplus** package. For more information on the pbmplus package, send mail to jef@well.sf.ca.us (Jeff Poskanzer); the **pbmplus** package can be retrieved via FTP from **archive.cis.ohio-state.edu:/pub/pbmplus/pbmplus.tar.Z,** among many other places.

**avstoppm** resides in **/usr/examples/cmavs/unsupported.**

## OPTIONS

There are no options to **avstoppm.**

## SEE ALSO

**libppm**(3), **ppmto...**(1) (converters for ppm to many image formats)

## DIAGNOSTICS

The diagnostics are intended to be self-explanatory.

## BUGS

None known.

## NAME

**ppmtoavs** - convert PPM format images to AVS images (.x format)

## SYNOPSIS

**ppmtoavs** [ *infile* ] [ *outfile* ]

## DESCRIPTION

**ppmtoavs** converts images from PPM format to AVS image format. The **alpha** bytes of the AVS image are all zeros (but they can be set to any value within AVS by using the *replace alpha* module).

NOTE: The PPM format is part of the **pbmplus** package. For more information on the pbmplus package, send mail to jef@well.sf.ca.us (Jeff Poskanzer); the **pbmplus** package can be retrieved via FTP from **archive.cis.ohio-state.edu:/pub/pbmplus/pbmplus.tar.Z,** among many other places.

If no filenames are given, **ppmtoavs** reads from *stdin* and writes to *stdout*. If one filename is given, it is the input filename, and output goes to *stdout*. If two filenames are given, the first is the input file and the second is the output file (which is first truncated if it exists).

**ppmtoavs** resides in **/usr/examples/cmavs/unsupported**.

## OPTIONS

There are no options to **ppmtoavs**.

## SEE ALSO

**libppm**(3), **ppmto...**(1) (converters for ppm to many image formats)

## DIAGNOSTICS

The diagnostics are intended to be self-explanatory.

## BUGS

None known.

## NAME

**field to polygons** (unsupported) - translates a coordinate field into a set of polygons

## SUMMARY

| | |
|---|---|
| **Name** | field to polygons |
| **Type** | mapper |
| **Inputs** | polygon list 2D 3-space irregular float<br>colormap **Outputs** polygon geom (*geom*) |

| **Parameters** | **Name** | **Type** | **Default** | **Choices** |
|---|---|---|---|---|
| | Use Color | toggle | off | on, off |

**field_to_polygons** resides in **/usr/examples/cmavs/unsupported**.

## DESCRIPTION

**field to polygons cm** translates a field containing vertex coordinates into a geometry describing polygons. The polygon list's points array contains the vertex coordinates and the data array (optionally) contains the color information.

The **Use Color** parameter determines how the polygon will be colored. (See below for details.)

## INPUTS

**polygon list**

(required; field 2D 3-space [1 or 3]-vector irregular float)
The first dimension of the input field must be equal to the number of sides of the polygons. All polygons must have the same number of sides. The second dimension is the number of polygons. The points array describes the coordinates of the polygons. The data array optionally describes the polygons color. The vector length restriction has effect only if the **Use Color** parameter is on.

**colormap**

(colormap)
This colormap is used to color the polygons when the **Use Color** toggle is on and a 1D data field is suppied. The default colormap is a linear ramp from black to white with a low value of 0.0 and a high value of 255.0.

## OUTPUTS

| | |
|---|---|
| **polygon geom** | (geom) The output geometry containing the polygon objects. |

## PARAMETERS

**Use Color** This is a toggle parameter:

| | |
|---|---|
| **If on:** | The polygon vertices will be colored using the field's data and, optionally, the given colormap. There are two techniques used to color the data depending on the vector length of the polygon list. If the vector length is one, the value of each element is used as an index into the colormap. If the vector length is three, they are interpreted as the red, green and blue color values. |
| **If off:** | (default) The polygons will be drawn without explicit color information. Generally, this results in the polygons being drawn in white. In this case, the polygon list's data array is ignored. |

## EXAMPLE

```
   READ FIELD
        |
FIELD TO POLYGONS
        |
 GEOMETRY VIEWER
```

## RELATED MODULES

**field to spheres (unsupported)**

## NAME

**field to spheres** (unsupported) - translates a coordinate field into a set of spheres

## SUMMARY

| | | | | |
|---|---|---|---|---|
| **Name** | field to spheres | | | |
| **Type** | mapper | | | |
| **Inputs** | sphere list - field 3-space irregular float<br>colormap - colormap | | | |
| **Outputs** | sphere geom - geom | | | |
| **Parameters** | **Name** | **Type** | **Default** | **Choices** |
| | size | dial | 0.0 | |
| | Use Color | toggle | off | on, off |

**field_to_spheres** resides in **/usr/examples/cmavs/unsupported.**

## DESCRIPTION

**field to spheres** translates an irregular field describing a set of three space coordinates into a geometry containing sphere objects. The sphere list's points array describes the spheres' positions. Optionally, the field's data array describes the spheres' colors.

If the **size** parameter is equal to 0.0, the spheres will be drawn as single pixels. If the size is greater than 0.0, the spheres will be drawn as uniformly sized spheres with radii equal to the size.

The **Use Color** parameter determines how the spheres will be colored. (See below for details.)

## INPUTS

| | |
|---|---|
| **sphere list** | (required; field 3-space [1 or 3]-vector irregular float)<br>This field must be a list of irregular points in 3-space. The points array describes the coordinates of the spheres. The data array optionally describes the spheres color. The vector length restriction has effect only if the **Use Color** parameter is on. |
| **colormap** | (colormap)<br>This colormap is used to color the spheres when the **Use Color** toggle is on and a 1D data field is suppied. The default colormap is a linear ramp from black to white with a low value of 0.0 and a high value of 255.0. |

## PARAMETERS

**size**

(dial)
The size of the spheres. If equal to zero, the spheres are rendered as single pixels. Otherwise, the value is used as the radius in world coordinates.

**Use Color**

This is a toggle parameter.

**If on:** The spheres will be colored using the field's data and, optionally, the given colormap. There are two techniques used to color the data depending on the vector length of the sphere list. If the vector length is one, the value of each element is used as an interpolated index into the colormap. If the vector length is three, the three values are interpreted as the red, green and blue color values. They should lie in the range from 0.0 to 1.0.

**If off:** The spheres will be drawn without explicit color information (no color information is encoded with the sphere data). Generally, this results in the spheres being drawn in white. In this case, the sphere list's data array is ignored.

## OUTPUTS

**sphere geom**

(geom)
The output geometry containing the sphere objects.

## EXAMPLE

```
    SAMPLERS
        |
 FIELD TO SPHERES
        |
 GEOMETRY VIEWER
```

## RELATED MODULES

### scatter dots

**scatter dots** is very similar to **field to spheres**, differing in the following ways:

- The radius of each sphere may be specified independently in scatter dots.
- **scatter dots** requires more information to be specified per sphere.
- **scatter dots** is slower.
- **scatter dots** requires a 1D coordinate field, whereas **field to spheres** is independent of the field's dimension.

# Index