CHAPTER 2

# IMPORTING DATA INTO AVS

## Introduction

The process of scientific visualization is, in very simple terms, one of transforming data into pictures. AVS contains a comprehensive set of modules for transforming data and for generating and manipulating 2D images and 3D geometric scenes. These modules and viewers are discussed in detail in the remainder of this *User's Guide* and in the *AVS Module Reference Manual*. This chapter presents various techniques for importing data into AVS.

Unfortunately, scientific data is computed or acquired in a plethora of different formats. Perhaps at some point in the future a standard format will be adopted for representing and storing scientific data, but today, no such standard exists. In order to use AVS effectively with *your data*, you will have to spend some time translating the data into a format that AVS can understand.

This chapter is designed to help you achieve this goal. The first section, "AVS Data Types," discusses the basic types of data used by AVS, and will guide you in selecting a data type which closely matches your own data. The next section, "Data Import Strategies," presents a series of choices available to you for importing your data into AVS. The last five sections, "Field Data," "Geometric Data," "Unstructured Cell Data," "Colormap Data," and "Molecule Data Type" present a description of each major AVS data type, AVS modules for reading these data types, and external file formats for storing AVS-compatible data sets.

Once you have imported your data into AVS, the power of interactive scientific visualization will be at your fingertips—allowing you to experience the same benefits of insight and discovery that thousands of AVS users have experienced.

## AVS Data Types

This section briefly describes each of the basic AVS data types and gives a few examples of how each data type is used. To help guide you in selecting which data type is appropriate for your application, a quick reference table is presented at the end of this section indicating the appropriate data type for various scientific and engineering applications.

AVS has two main types of data: **primitive data** and **aggregate data**.

## Primitive Data

**Primitive data** includes simple scalar types (byte, integer, single-precision floating-point, and double-precision floating point) and text strings. Primitive data types are the basic building blocks of aggregate data types, and are also used by AVS to represent parameters.

### byte

A single byte, or 8 bits. On all machines, a byte is used to represent an unsigned integer in the range 0..255.

Bytes are typically used by aggregate data types to represent pixel or voxel intensities. If the intensities are discrete and within the ranges specified above, byte data significantly reduces the amount of memory required to store a scientific data set when compared to integers or floating-point values.

### integer

A single machine word. On 32-bit machine architectures, integers are four bytes and can be used to represent signed integer values in the range -2**31 to 2**31 (-2147483648..2147483647). On 64-bit architectures, these ranges are much larger.

Integers are used by aggregate data when discrete values are appropriate, but greater than the range afforded by bytes (0..255). For example, integers can be used to represent voxel intensities. Integers can also be used as parameters. Examples include data replication (zoom=2), downsizing (downsize=4), or an 2D orthogonal slice plane setting within a 3D volume of data (k=21).

### single-precision floating point

A single machine word used to represent a floating-point quantity in *single* format. On 32-bit machine architectures, single-precision floating point values are four bytes.

Single-precision floating-point data values are used in aggregate data to represent continuously varying phenomena, e.g. temperature or velocity. Single-precision floating-point parameters are used for continuously varying values. Examples include the diffuse component of a gradient shader, or the x value of a bilinear interpolator.

### double-precision floating point

Two machine words used to represent a floating-point quantity in *double* format. On 32-bit machine architectures, double-precision floating point values are two words, or eight bytes.

Double-precision floating-point data is used in cases where the accuracy of single-precision is not great enough to contain the significant components of the input data values.

### text strings

An array of bytes, each of which represents a single character.

Text string parameters are typically used to represent file names. Aggregate data uses text strings for labels on elements of data; for example, in a multi-dimensional fluid flow data set, a text string might be used as a label for each of the vector components: density, x-momentum, y-momentum, etc.

## Aggregate Data

**Aggregate data** is used by AVS to represent the major data types for scientific and engineering data visualization, including field data, geometric data, unstructured cell data, and colormap data. AVS has other aggregate data types, including pixel maps, and user-defined data. These two data types are generally not required by a novice or intermediate-level AVS user to import data into AVS, and are therefore not described in this chapter. For a full description of AVS data types, consult the *AVS Developer's Guide*.

### Field Data

Field data is an *n*-dimensional array with an *m*-dimensional vector of values at each array location (where *n* and *m* are any integers). The physical location of each data element (if it exists) is either implied or specified explicitly with 3D coordinates.

Field data is typically used to represent two-dimensional images (e.g. satellite images or x-rays) and three-dimensional volumes (e.g. MRI scans, CAT scans, numerical simulation data, fluid flow data). It can also be used to represent one-dimensional data (e.g. x-y plots, or a series of points in space).

### Geometric Data

Geometric data consists of three-dimensional objects that are constructed out of one or more of the following primitives: polyhedra, polygons, meshes, spheres, and polytriangles.

Geometric data is used to represent any three-dimensional shape, including real-world objects (e.g. rotor blades in jet turbines, a brake assembly in an automobile, or a film advance mechanism in a camera), objects represented as geometries (e.g. ball and stick models of molecules), and objects created by visualization techniques (e.g. an arbitrary slice plane through a volume, or an isosurface).

### Unstructured Cell Data

Unstructured cell data (UCD) consists of a geometric model built of individual cells. The cells, including 1D, 2D, and 3D shapes, are defined by nodes, or vertices. Data, either scalar or vector, can be assigned to the entire model, each individual cell, and each vertex of each cell.

Unstructured cell data is used in finite element problems, including structural analysis and certain areas of computational fluid dynamics.

### Molecule Data Type

The molecule data type (MDT) consists of ten defined CHEM objects that can be combined hierarchically in linked lists to represent the structure of molecules. At the root of the hierarchy is the CHEMmolecule. A CHEMmolecule is a data structure containing the molecule's name, and a unit in angstroms of bohrs. Chained off of the CHEMmolecule is an arbitrarily long list of CHEMatoms (data for component atoms), CHEMchemunits (data for defining molecule substructures), CHEMquantums (data for classical quantum chemistry), and user-defined data.

The molecule data type can be used to model chemical data for problems in classical, substructure, and quantum chemistry.

The molecule data type and the *libchem* library that manipulates it is described fully in its own document, the *AVS Chemistry Developer's Guide.* It is discussed only briefly in this *User's Guide.*

### Colormap Data

A colormap is an arbitrary-sized one-dimensional array of 4D vectors. Vector components include normalized floating-point values for hue, saturation, brightness, and opacity.

Colormaps are used to specify colors of data values that are displayed on the screen. For example, an image composed of 8-bit pixels can be pseudo-colored by a colormap that ranges continuously in hue from blue to red. A pixel intensity of 0 is colored blue when it is displayed and a pixel intensity of 255 is colored red. All intermediate pixel intensities are colored with a hue that is in the color spectrum between blue and red.

## AVS Data Type Reference Table

The following table presents a few common scientific applications and the AVS data types frequently used to represent various types of data. If you do not find your application in this table, try to locate one that has similar characteristics.

**Table 2-1. AVS Data Type/Application Cross Reference Table**

| Application | AVS Data | Examples |
|---|---|---|
| CFD | *field* | Simulation grids |
| | | Finite difference data |
| | *ucd* | Finite element model and data |
| | *geometry* | Structures |
| MCAE | *ucd* | Finite element model and data |
| | *geometry* | Structures |
| | *field* | Finite difference data |
| Molecular Chemistry | *molecule data type* | Ball and Stick models |
| | *geometry* | Microscopic and x-ray imaging |
| | *field* | Scatter data |
| Quantum Chemistry | *molecule data type* | Molecular orbitals |
| | *field* | Molecular interactions |
| | | Subatommic particles |
| Seismic | *field* | 3D volumes - sampled data |
| | | 3D volumes - simulated data |
| | *geometry* | Structures |
| Strategic Imaging | *field* | 2D images |
| | | Digital terrain maps |
| | | Multispectral images |
| | *geometry* | Structures |
| Medical Imaging | *field* | 2D images - CAT/MRI/PET scans |
| | | x-rays |
| | | 3D volumes - CAT/MRI/PET scans |
| | *geometry* | Anatomical structures |
| Meteorology | *field* | 2D images |
| | | 3D volumes - sampled data |
| | | 3D volumes - simulated data |

## Data Import Strategies

There are basically two choices a user has when importing data into AVS: *(1)* use an existing AVS module, or *(2)* write a customized new module.

You must first determine if an existing AVS module can satisfy your data input requirements. To do so, you must understand each of the AVS data types, the existing modules for reading AVS data, and the format of external AVS-compatible data. The following sections of this chapter will allow you to make this assessment.

As will be shown, many "data importing" modules are provided with AVS. These modules may be already-compiled utilities, or they may exist as source code examples that you can compile.

It is possible that you will be unable to find a module suited to your data format. However, it is likely that one exists elsewhere. Contact your sales office to see if a "user-contributed" module exists for your data format—many users of AVS could have data similar to your own and might already have created a module to import it.

If an existing AVS module is not available for importing your data, you will have to write a new module, tailored to match your specific data format. One of the primary goals of AVS is to minimize the amount of programming necessary if, and when, a new module must be developed.

Once it has been determined that programming is necessary, several options are available:

1. Write a shell-level "translator" that converts your data format into the appropriate AVS data format.
2. Modify your application to produce data in AVS format.
3. Write an AVS module that converts your data format into the appropriate AVS data format.
4. Modify your application as an AVS module that can be incorporated into AVS networks.

If programming is required, the most effective use of AVS will generally involve writing a module. Once written, the module can become part of many different visualization networks, and can easily be modified and enhanced to satisfy additional requirements. AVS has example module source programs in Fortran and C for each of its internal data formats, located in the */usr/avs/examples* directory. You can use these samples as a guide to writing your own modules.

There are many additional benefits to incorporating a data-producing application into an AVS module. Once an application has been transformed into an AVS module, the user can gain interactive control over various parameters (e.g. starting conditions, boundary conditions, constraints). In effect, the process of visualization can be tightly integrated with the simulation or data acquisition process.

The remainder of this chapter is devoted to helping you determine if there is an existing solution to the problem of importing your data into AVS. The emphasis here is on the "no-programming required" approach. However, pointers are given to existing sample source for data importing modules. It is strongly suggested that the user consult the *AVS Developer's Guide* before programming AVS modules.

## *Field Data*

A **field** is a generalization of the familiar array structure. Whereas each element of an array has a single data value (e.g. byte or integer), each element of an AVS field can have a list of data values. Thus, a field can be described as an *n*-dimensional array with an *m*-dimensional vector of values at each array location (where *n* and *m* are any integers). Moreover, the field can include coordinate data, so that each field element is mapped to a real-world location.

The AVS field datatype has the following components:

**ndim**
The number of computational dimensions in the field. For an image, **ndim** = 2. For a volume, **ndim** = 3.

**dim1**
**dim2**
**dim3**
**...**
The dimension size of each axis (the array bound for each dimension of the computational array). The number of **dim***x* entries is based on the value of **ndim**.

**nspace**
The dimensionality of the physical space that corresponds to the computational space (number of physical coordinates per field element).

**veclen**
The number of data values for each field element. All the data values must be of the same primitive type (e.g. **integer**), so that the collection of values is conceptually a **veclen**-dimensional vector. If **veclen**=1, the single data value is, effectively, a scalar. Thus, the term **scalar field** is often used to describe such a field.

**data_type**
The primitive data type of all the data values: must be one of **byte, integer, single,** or **double**.

**field_type**
The field type, one of **uniform, rectilinear,** or **irregular**.

**min_ext** and **max_ext**
The minimum and maximum coordinate value that any member data point occupies in space, for each axis in the data.

**min_data** and **max_data**
The minimum and maximum data value in the field.

**labels**
A title for each of the individual elements in a vector of values.

**units**
A string that describes the unit of measurement for each vector element.

**field_data**
The data values for the field.

**coordinate_data**
The computational-to-physical space mapping coordinate data values.

Other references to the AVS field format can be found in the **read field** page of the *AVS Module Reference Manual*, and in the "AVS Data Types" chapter of the *AVS Developer's Guide*.

AVS field data is classified into three basic categories, according to the specification of computational-to-physical space mapping. As mentioned above, the three types of fields are **uniform**, **rectilinear**, and **irregular**.

## Uniform Fields

A uniform field has no computational-to-physical space mapping *between* data elements. The field implicitly takes its mapping from the organization of the computational array of field elements. However, AVS allows an entire uniform field to be mapped into physical space by specifying physical coordinates for the **bounds** of the field dataset. Example

Consider the following 2D integer-valued array (using a FORTRAN-style notation):

DATA( *I*, *J* )   *I*=1,2   *J*=1,5

DATA(1,1) = 12
DATA(2,1) = 17

DATA(1,2) = 4
DATA(2,2) = 0

DATA(1,3) = 10
DATA(2,3) = -5

DATA(1,4) = 16
DATA(2,4) = 16

DATA(1,5) = 16
DATA(2,5) = 8

This array describes a 2D **computational** space, with *I* and *J* dimensions. The size of the *I* dimension is 2; the size of the *J* dimension is 5. The data is of type *integer*.

Since there is only one data value for each field element, this is said to be a **scalar field**. The following notation might be used to indicate the values of a **vector field**:

DATA(2,3) = (2.51, 1.09, 5.73)

The field is still 2-dimensional, but the data value is said to be a *3-vector*. Such a data value might be used to represent a velocity vector, or to represent a temperature-pressure-humidity measurement at each location in space.

In the absence of any additional information, there is a natural mapping between the computational space and a 2D **physical** space, the X-Y coordinate plane (see Figure 2-1.) .

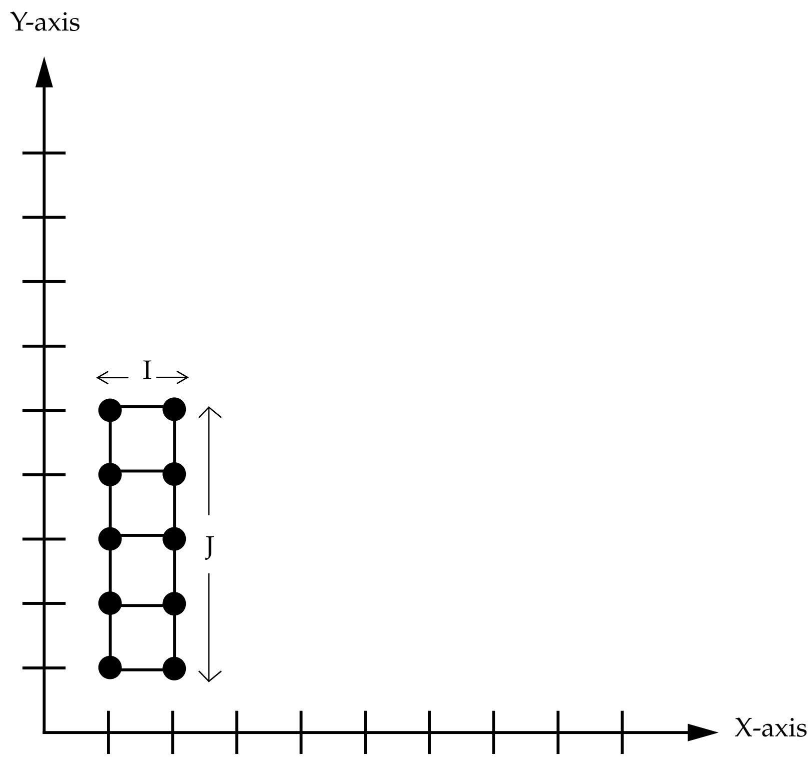


**Figure 2-1 Uniform 2D Field**

The physical space is a uniformly-spaced lattice. Accordingly, a field with no coordinate data for each of the data values is said to have the field type **uniform**.

## *Rectilinear Fields*

In a rectilinear field data set, each array index in each dimension of the computational space is mapped to a physical coordinate. This produces a physical space whose axes are orthogonal, but the spacing among the elements is not necessarily equal. Example

An explicit mapping between the computational and physical spaces can be established by specifying coordinate data for all of the points along *each axis*:

X-coordinates: 0, 3, 6, 9, 12
Y-coordinates: 20*log(1), 20*log(2), 20*log(3), 20*log(4), 20*log(5)

From the preceding example, array element DATA(1,3) would be mapped to physical location (0, 20*log(3)) according to this scheme. This mapping from computational space to the X-Y plane can be pictured as in Figure 2-2.

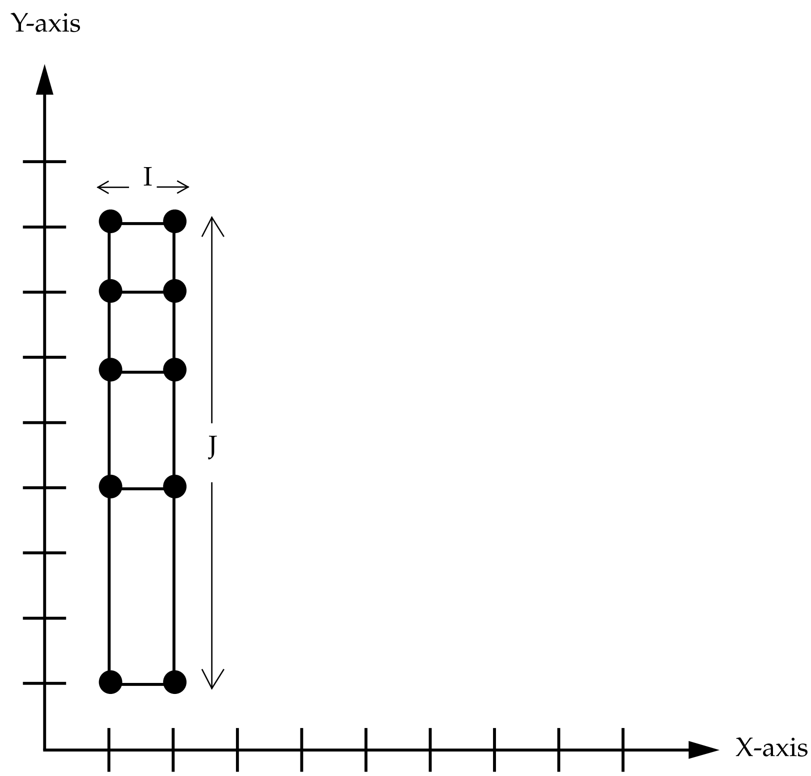


**Figure 2-2 Rectilinear 2D Field**

Note that in a **rectilinear** field, lines connecting the lattice points are always mutually orthogonal—all the angles are right angles.

## Irregular Fields

For irregular fields, there is no restriction on the correspondence between computational space and physical space. Each element in the computational space is assigned its own physical coordinates. Example

The first example is extended once again, this time to include a coordinate pair for each of the data elements:

DATA(1,1) -> (1.0, 1.0)
DATA(2,1) -> (7.0, 1.0)

DATA(1,2) -> (3.0, 3.0)
DATA(2,2) -> (6.0, 2.5)

DATA(1,3) -> (4.0, 4.5)
DATA(2,3) -> (5.5, 4.5)

DATA(1,4) -> (3.5, 6.0)
DATA(2,4) -> (4.5, 5.5)

DATA(1,5) -> (3.5, 7.5)
DATA(2,5) -> (6.0, 8.0)

This mapping from computational space to the X-Y plane can be pictured as in Figure 2-3.

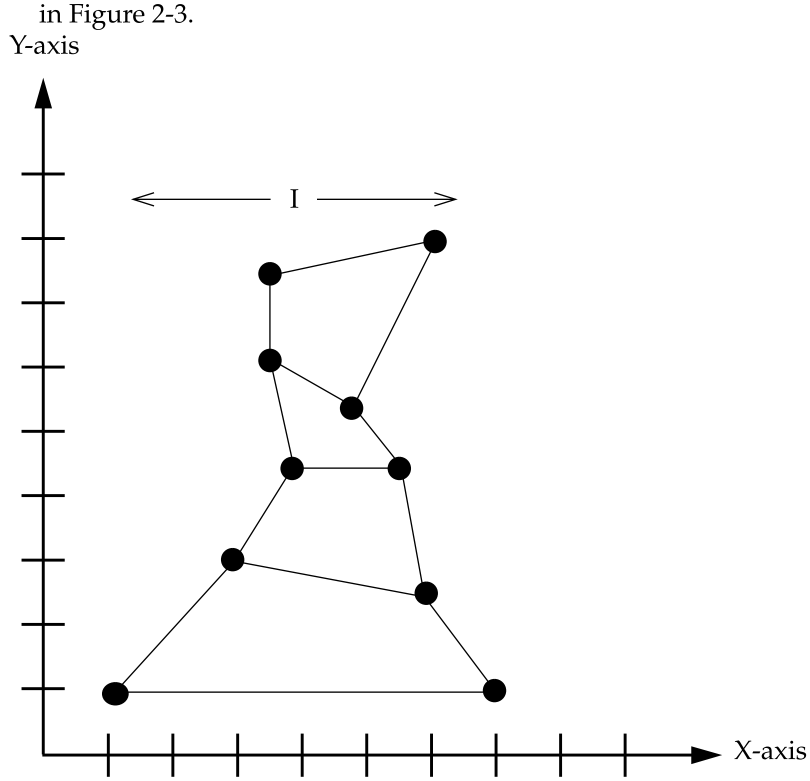


**Figure 2-3 Irregular or Curvilinear 2D Field**

Note that there is nothing in this scheme that restricts the physical space to having the same number of dimensions as the computational space. For example, the field element DATA(2,3) could be mapped to the physical point (4.5, 5.5, -8.1) in 3D space. This kind of mapping can be used to "wrap" a plane (computational space) around a sphere (physical space), or to warp a flat plane into a 3D manifold.

For additional examples, see the "AVS Data Types" chapter in the *AVS Developer's Guide*.

## AVS Data Interchange Application: ADIA

The **AVS Data Interchange Application** (**ADIA**) is an interactive tool for importing a wide variety of binary and ASCII data formats into an AVS field. ADIA is similar in functionality to the **read field** module described below, with these additional features:

- It can read 16-bit "halfword" data. Many medical imaging applications produce 12-bit data in two 8-byte halfwords. (The data will be represented in the AVS field as 32-bit integers.)
- It supports variables and expressions, making it possible to define skips and spaces necessary to read the input data as a function, rather than as an absolute number.
- It is an interactive facility contained within AVS—one does not need to use a text editor to externally edit an ASCII header.
- It can read data format information, such as the dimensions of a dataset, from the input dataset and use these values as part of its variables and expressions.
- It creates ASCII data forms that define how to read a particular input dataset format. These data forms can be reused and exchanged.

ADIA is not documented in this manual. Instead, it is fully described in the *AVS Applications Guide* manual.

## AVS Module: read field

The **read field** module has two input modes, "native field input" and "data-parsing input". In its first input mode, it reads an AVS **field** data structure from a disk file into a network. The format of an AVS **field file** is discussed below in the section "AVS Field File Format".

In its second input mode, it converts data stored in ASCII, FORTRAN unformatted, or pure binary data files into AVS field format. **read field** can thus be used to import *many* datasets into AVS.

### Native Field Input

**read field** can read files in the native AVS field file format into an AVS network. An AVS field file (suffix *.fld*) has the following components:

- An ASCII header that describes the field

- Two separator characters that divide the ASCII header from the data and coordinate information
- A binary area containing the data and coordinate information

The **write field** module creates files in this format.

### ASCII Header

The ASCII header contains a series of text lines, each of which is either a comment or a *TOKEN=VALUE* pair. For example, the following header defines a field of type "field 2D 4-vector byte", which is the AVS image format:

```
# AVS field file
#
ndim=2          # number of computational dimensions
dim1=512
dim2=480
nspace=2        # number of physical dimensions
veclen=4
data=byte
field=uniform
```

The first two lines are comments, indicated by the **#** character. Note that the first line of the header *must* begin as follows:

```
# AVS
```

In this example, comments also occur at the end of the 3rd and 6th lines. Any characters following (and including) **#** in a header line are ignored. Comments are not required.

### Separator Characters

The ASCII header must be followed by two formfeed characters (i.e. **Ctrl-L**, octal 14, decimal 12, hex 0C), in order to separate it from the binary area. This scheme allows you use the **more**(1) shell command to examine the header. When **more** stops at the formfeeds, press **q** to quit. This avoids the problem of the binary data garbling the screen.

### Binary Area

The binary area contains both **data values** and **coordinate mapping** data. This section describes the components of the binary are by looking at how the size of this area is calculated.

The size of the **data values** section of the binary area is calculated by taking the product of:

- All the dimensions of the field: *(dim1 * dim2 * ... dimx)*,
- The number of data values per field element: *(veclen)*, and
- The byte size of the primitive data type: *(data_size)*.

Thus, the size (in bytes) of the data values section is:

( *dim1 * dim2 * ... dimx * veclen * data_size* )

In the block of data values:

- All the data values for a field element are stored together.
- The first array index varies most quickly (*FORTRAN-style*).

The size of the **coordinate mapping** data section of the binary area depends on the field type: uniform, rectilinear, or irregular.

- For **uniform** fields, coordinate mapping data contains two values for each physical dimension (n-space) of the data, one associated with the minimum extent of the data, and the other associated with the maximum extent.

  The minimum and maximum extent values in the coordinate binary area are copies of the **min_ext** and **max_ext** values in the field data structure, *except* when the field has been cropped, downsized, or interpolated. Then the field data structure contains the original field's **min_ext** and **max_ext** values, while the coordinate section of the binary area contains the minimum and maximum extent of the subsetted data. Mapper modules can use this additional extent information to properly locate their geometric representation of the subsetted data in world coordinate space. The extents in the coordinate binary area are stored in this order: minimum x, maximum x, minimum y, maximum y, minimum z...etc.

  The size of the coordinate mapping data section for a uniform field is simply:

  ( 2 * nspace ) * sizeof(float)

  **NOTE:** *sizeof(float)* refers to the size, in bytes, of a single-precision floating-point number. On 32-bit architectures, this will be 4.

- For **rectilinear** fields, there is one coordinate for each array index in each dimension of computational space. Thus, the size of the coordinates area is:

  ( *dim1 + dim2 ... + dimx ) * sizeof(float)*

  All of the X-coordinates are stored together, at the beginning of the coordinates area. Following these are all the Y-coordinates, and so on.

- For **irregular** fields, each field element is mapped to a point in *nspace*-dimensional physical space. Thus, the size of the coordinates area is:

  ( *dim1 * dim2 ... * dimx ) * nspace * sizeof(float)*

  As with rectilinear fields, all of the X-coordinates are stored together, at the beginning of the coordinates area. Following these are all the Y-coordinates, and so on.

### Example 1

The following ASCII header describes a volume (3D uniform field) with a single byte of data for each field element. This format might be used to represent CAT scan data, where all of the 2D slices are equally spaced:

```
# AVS field file
ndim=3           # number of dimensions in the field
dim1=64          # dimension of axis 1
dim2=64          # dimension of axis 2
dim3=64          # dimension of axis 3
nspace=3         # number of physical coordinates per point
veclen=1         # number of components at each point
data=byte        # data type (byte, integer, float, double)
field=uniform    # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

(64 * 64 * 64) * 1 * 1 = 262,144 bytes

The coordinates area occupies (2 * 4) * 3 bytes. The total binary area occupies 262,168 bytes.

### *Example 2*

The following ASCII header describes a volume (3D uniform field) whose data for each field element is a 3D vector of single-precision values. This format might be used to represent a volume of data from a sampling device where the samples are not equally spaced.

```
# AVS field file
ndim=3           # number of dimensions in the field
dim1=27          # dimension of axis 1
dim2=25          # dimension of axis 2
dim3=32          # dimension of axis 3
nspace=3         # number of physical coordinates per point
veclen=3         # number of components at each point
data=float       # data type (byte, integer, float, double)
field=uniform    # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

(27 * 25 * 32) * 4 * 3 = 259,200 bytes

The coordinates area occupies (2 * 4) * 3 bytes. The total binary area occupies 259,224 bytes.

### *Example 3*

The following ASCII header describes an irregular volume (3D irregular field) with one single-precision value for each field element. The binary area includes an (X,Y,Z) coordinate triple for each field element, indicating the corresponding point in physical space. This format might be used to represent velocity data from a simulation of fluid flow around a curved object (e.g. an airplane wing). Note that the data type is specified as **xdr_float**. This means that the data file is written in Sun's external data representation (XDR) format. This allows for transporation of field files between machines of dissimilation data storage architectures ("big-endian" vs "little-endian"). When using **read field**, if **Auto** is selected on the control panel, **read field** will look at the data specification. If it is simply **float**, (or **integer**, or **double**), it will assume the file is written in the system's native format. If it is **xdr_integer**, **xdr_float**, or **xdr_double** it will translate the XDR format into its native format. If **Portable** is selected rather than **Auto**, it will always assume the file is written in XDR format.

```
# AVS field file
ndim=3            # number of dimensions in the field
dim1=40           # dimension of axis 1
dim2=32           # dimension of axis 2
dim3=32           # dimension of axis 3
nspace=3          # number of physical coordinates per point
veclen=1          # number of components at each point
data=xdr_float    # data type (byte, integer, float, double)
field=irregular   # field type (uniform, rectilinear, irregular)
```

In the binary area, the data area occupies this amount of space:

(40 * 32 * 32) * 4 * 1 = 163,840 bytes

The coordinates area occupies this amount of space:

(40 * 32 * 32) * 4 * 3 = 491,520 bytes

### Data-Parsing Input

In its second input mode, **read field** can convert a certain class of data stored in ASCII, FORTRAN unformatted, or pure binary data files into AVS field format. To import data into AVS, you must create an ASCII description file that defines the structure of the AVS field to make. The first part of this description file is identical in format and meaning to the ASCII header file described above.

The second part of this file contains commands that specify which files contain the data or coordinate information, its data type (ASCII, FORTRAN unformatted, or binary), and simple parsing instructions. **read field** can read a file that is parseable by this general scheme:

**skip** *n* lines or bytes
move over an **offset** of *m* columns on this line (ASCII only)
read the value
do until # *of values needed*
{
  take *p* **stride**(s) to the next value
  read the value
}

Note that **read field** follows parsing instructions you supply; it does not "figure out" the format of a data file.

The ASCII description file, data, and coordinate information for rectilinear and irregular data can all be read from different files. If the resulting AVS field contains a vector of data values at each point, each vector element can also be read from a separate file.

The ASCII description file must have a *.fld* file suffix or the **read field** file browser will not display the file.

**read field** data parsing capability is meant to be used only once, in order to convert data to AVS field format. The parsing activity makes **read field** run more slowly than when it reads a file that is already in AVS field format. Once you have read your data using **read field**'s data-parsing mode, you should use the **write field** module to store it permanently on disk in AVS field file format.

### ASCII Description File

As the example below shows, the ASCII description file contains a series of text lines that define the AVS field to construct. Each line is either:

- A comment
- A required line in the form *token*=*value*
- An optional line in the form *token*=*value*
- A **variable** or **coord** parsing specification

The following ASCII description file imports three-dimensional curvilinear data with a vector of values at each point into an AVS field of type "field 3D 3-vector irregular float". This type of data often occurs in computational fluid dynamics applications. The data and coordinate information are in separate files, both of which were written as straight binary data. Both files happen to have a serial organization. In the data file, all of vector element 1's values appear, then all of vector element 2's, then all of vector element 3's values. In the X, Y, Z coordinate file, all the X coordinate values appear, then all the Y's, then all the Z's.

Each line's meaning is explained in detail below.

```
# AVS field file     the string "# AVS" must be the first five
#                    characters in the file.
#
#                    When a '#' character appears in a line,
#                    the rest of the line is a comment
#
ndim=3                  # REQUIRED - the number of dimensions
dim1=40                 # REQUIRED - dimension of axis 1
dim2=32                 # REQUIRED - dimension of axis 2
dim3=32                 # REQUIRED - dimension of axis 3
nspace=3                # REQUIRED - coordinates per point
veclen=3                # REQUIRED - components at each point
data=float              # REQUIRED - data type
field=irregular         # REQUIRED - field type
min_ext=-1.0 -1.0 -1.0  # OPTIONAL - coordinate space extent
max_ext=1.0 1.0 1.0     # OPTIONAL - coordinate space extent
label=x-velocity        # OPTIONAL - label for variable 1
label=y-velocity        # OPTIONAL - label for variable 2
label=z-velocity        # OPTIONAL - label for variable 3
unit=miles-per-second   # OPTIONAL - unit label for variable 1
unit=miles-per-second   # OPTIONAL - unit label for variable 2
```

```
unit=miles-per-second   # OPTIONAL - unit label for variable 3
min_val=-2.1 -0.3 -3.7  # OPTIONAL - minimum data values
max_val=5.79 3.54 1.50  # OPTIONAL - maximum data values
#
# For each coordinate X, Y, and Z: data reading instructions
#
coord 1 file=/jetdata/wing.bin filetype=binary skip=12
coord 2 file=/jetdata/wing.bin filetype=binary skip=163852
coord 3 file=/jetdata/wing.bin filetype=binary skip=327692
#
# For each value in the vector: data reading instructions
#
variable 1 file=/jetdata/wdata.bin filetype=binary skip=28
variable 2 file=/jetdata/wdata.bin filetype=binary skip=163868
variable 3 file=/jetdata/wdata.bin filetype=binary skip=327708
```

Any characters following (and including) **#** in a header line are ignored.

**NOTE:** The first five characters in the ASCII description file *must* be "# AVS" or **read field** will not recognize the file as valid.

The example above shows all of the required *TOKEN=VALUE* token names: an ASCII description file that is missing one or more of these lines causes **read field** to generate an error. Required *TOKEN=VALUE* pairs are stored in the AVS field that **read field** produces as output.

Optional *TOKEN=VALUE* pairs are stored in the output AVS field as well, if they are provided. **min_ext** and **max_ext** are stored in the output AVS field even if they are not specified, as **read field** calculates them if they are not provided.

The **variable** and **coord** lines are not stored in the output AVS field. They are only instructions to **read field**.

With the exception of filenames, ASCII description file specifications are *not* case-sensitive. You can surround the **=** character with any amount of white space (including none at all). For example, "dim2 = 32", "DIM 2 =32", and "Dim2=32" are all equivalent.

Below is a complete description of all of the tokens recognized by the **read field** parser:

**ndim =** *value* (required)
The number of computational dimensions in the field. For an image, **ndim** = 2. For a volume, **ndim** = 3.

**dim1 =** *value* (required)
**dim2 =** *value* (required, depending on total number of dimensions)
**dim3 =** *value* (required, depending on total number of dimensions)
**...**
The dimension size of each axis (the array bound for each dimension of the computational array). The number of **dim***x* entries must match the value of **ndim**. For instance, if you specify a 3D computational space field

(**ndim**=3), you must specify the length of the X dimension (**dim1**), the length of the Y dimension (**dim2**), and the length of the Z dimension (**dim3**). Note that counting is 1-based, not 0-based. If you have scatter data (**ndim=1**) in 3D coordinate space (**nspace=3**), there will be only **dim1=***value.*

**nspace =** *value* (required)

The dimensionality of the physical space that corresponds to the computational space (number of physical coordinates per field element).

In many cases, the values of **nspace** and **ndim** are the same — the physical and computational spaces have the same dimensionality. But you might embed a 2D computational field in 3D physical space to define a manifold; or you might embed a 1D computational field in 3D physical space to define an arbitrary set of points (a "scatter").

**veclen =** *value* (required)

The number of data values for each field element. All the data values must be of the same primitive type (e.g. **integer**), so that the collection of values is conceptually a **veclen**-dimensional vector. If **veclen**=1, the single data value is, effectively, a scalar. Thus, the term *scalar field* is often used to describe such a field.

**data = byte** (one of the four options is required)
**data = integer**
**data = float**
**data = double**

The primitive data type of all the data values. **xdr_integer, sdr_float,** and **xdr_double** may also be specified. If **Auto** is selected on input, **read_-field** will examine the **data=** field. If the specification is unqualified, it assumes the file is written in the native format for the platform ("big-endian" vs "little-endian"). If it is **xdr_*value***, it will read it as Sun's external data format (XDR) and translate it into the native format. If **Portable** is selected, it assumes XDR format no matter what is placed here.

**field = uniform** (one of the three options is required)
**field = rectilinear**
**field = irregular**

The field type. A **uniform** field has no computational-to-physical space mapping. The field implicitly takes its mapping from the organization of the computational array of field elements.

For a **rectilinear** field, each array index in each dimension of the computational space is mapped to a physical coordinate. This produces a physical space whose axes are orthogonal, but the spacing among elements is not necessarily equal.

For an **irregular** field, there is no restriction on the correspondence between computational space and physical space. Each element in the computational space is assigned its own physical coordinates.

**min_ext** = *x-value [y-value] [z-value]...* (optional)
**max_ext** = *x-value [y-value] [z-value]...* (optional)

The minimum and maximum coordinate value that any member data point occupies in space, for each axis in the data. If you do not supply this value, **read field** calculates it and stores it in the output AVS field data structure. This value can be used by modules downstream to, for example, size the **volume bounds** drawn around the data in the Geometry Viewer or put minimum and maximum values on coordinate parameter manipulator dials (**probe**). Values can be separated by blanks and/or commas.

If you do not know the extents, don't guess—let **read field** calculate them. Most downstream modules use whatever values are supplied, without checking their validity. If the wrong numbers are specified, incorrect results will be computed.

**label** = *string1 [string2] [string3]...* (optional)

Allows you to title the individual elements in a vector of values. These labels are stored in the output AVS field data structure. Subsequent modules that work on the individual vector elements (for example, **extract scalar**) will label their parameter widgets with the strings provided here instead of the default "Channel 0, Channel 1...", etc. You can either use one **label** line as shown here, or separate label lines as shown in the example above. In either case, the labels are applied to the elements of the vector in the order encountered. You can also label single scalar values, though downstream modules may ignore such a label. Any alphanumeric string is acceptable. Strings can be separated by blanks and/or commas.

**unit** = *string1 [string2] [string3]...* (optional)

Allows you to specify a string that describes the unit of measurement for each vector element. You can either use one *unit* line as shown here, or separate unit lines as shown in the example above. In either case, the unit specifications are applied to the elements of the vector in the order encountered. You can also specify the unit for a single scalar value, though downstream modules may ignore it. Any alphanumeric string is acceptable. Strings can be separated by blanks and/or commas.

**min_val** = *value [value] [value]...* (optional)
**max_val** = *value [value] [value]...* (optional)

For each data element in a scalar or vector field, allows you to specify the minimum and maximum data values. These values are stored in the output AVS field data structure. This is used by subsequent modules that need to normalize the data. Values can be separated by blanks and/or commas.

**read field** does not calculate these values if you do not supply them (unlike **min_ext** and **max_ext**). If you do not know these values, don't guess—just leave these optional lines out. In this case, the **write field** module can, at your instruction, compute these values when it creates an AVS field file. Most downstream modules use whatever values are supplied, without checking their validity. If the wrong numbers are specified, incorrect results will be computed.

**variable** *n* **file=***filespec* **filetype=***type* **skip=***n* **offset=***m* **stride=***p*
**coord** *n* **file=***filespec* **filetype=***type* **skip=***n* **offset=***m* **stride=***p*

Both of these specifications must be on a *single* line. There is no support for continuation characters.

**variable** specifies where to find *data* information, its type, and how to read it.

**coord** specifies where to find *coordinate* information, its type, and how to read it. It is used when the data is **rectilinear** or **irregular**.

The individual parameters are interpreted as follows:

- *n* — An integer value that specifies which element of a data vector or which coordinate (1 for x, 2 for y, 3 for z, etc.) the subsequent read instructions apply to. **n** does not default to 1 and must be specified.
- **file** = *filespec* — The name of the file containing the data or coordinates. The *filespec* can be an absolute full pathname to a file, or it can be a *filespec* relative to the directory that contains the field ASCII header. For example, an absolute pathname might be */home/myuserid/experiment/data*. In a relative pathname specification, if the ASCII file of field parsing instructions exists in the file */home/myuserid/experiment/readit.fld* and the data and coordinates file are in the subdirectory */home/myuserid/experiment/data*, you can name these files as *xdata/xyzs* and *data/values*. The advantage of this second approach is that you can move the directories containing your data around without having to change the contents of the ASCII parsing instruction file.
- **filetype = ascii — ascii** means that the data or coordinate information is in an ASCII file. In ASCII files, float data can be specified in either real (0.1) or scientific notation (1.00000e-01) format interchangeably.
- **filetype = unformatted** — the file is written in FORTRAN unformatted format. (FORTRAN unformatted data is binary data with additional words written at the beginning and end of each data block stating the number of bytes or words in the data block.) In general, **read field** can read unformatted files where all variables of one type (for example, all the X coordinates) were output as one "record" in a single write statement. This is usually the case.
- **filetype = binary** — the file is written in straight binary format, such as that produced by Unix output routines, *write* and *fwrite*.

  Note the warning on binary compatibility among different hardware platforms on the **read field** man page.

  In each case, **read field** will use the data type specified in the earlier **data={byte,float,integer,double}** statement when it interprets the file.
- **skip** = *n* — For **ascii** files, **skip** specifies the number of *lines* to skip over before starting to read the data. Lines are demarked by newline characters.

  For **binary** or **unformatted** files, **skip** specifies the number of *bytes* to skip over before starting to read the data.

  There are two motivations for **skip**. First, data files often include header information irrelevant to the AVS field data type. Second, if the file contains, for example, all X data values, then all Y data values,

**skip** provides a way to space across the irrelevant data to the correct starting point.

**skip** can only be used once at the start of the file. There is no way to **skip**, read, **stride**, then **skip** again.

You must simply know what value to use for **skip** based on your knowledge of the software that produced the original data file, the number of data elements, and the type (byte, float, double, integer, etc.)

**skip** defaults to 0.

- **offset** = *m* — **offset** is only relevant to ASCII files; it is ignored for binary and unformatted files. **offset** specifies the number of columns to space over before starting to read the first datum. (The **stride** specification determines how subsequent data are read.) Hence, to read the fourth column of numbers in an ASCII file, use **offset=3**.

  In ASCII files, columns must be separated by one or more blank characters. Commas, semicolons, TAB characters, etc., are *not* recognized as delimiters. If necessary, edit ASCII files to meet this restriction.

  **offset** defaults to 0 (the first column, no columns spaced over).

- **stride** = *p* — **stride** assumes you are "standing on" the data value just read. **stride** specifies how many "strides" must be taken to get to the next data value. In ASCII files, **stride** means stride forward *p* delimited items. In binary and unformatted files, **stride** means stride forward *p* * *the size of the data type* (byte, float, double, integer). In a file where the data or coordinate values are sequential, one after the other, the **stride** would be 1. Note that this presumes homogeneous data in binary and unformatted files — double-precision values could not be intermixed with single precision values.

  **stride** defaults to 1.

  The stride value will be repeatedly used until the number of data items indicated by the product of the dimensions (e.g. dim1 * dim2 * dim3) have been read.

### *Example 1*

Here are some **skip**, **offset**, and **stride** examples for ASCII data. "A's" are vector component 1; "B's" are vector component 2.

ASCII file organization 1:

```
X      Y      Z      A      B
1      1      1      A1     B1
2      2      2      A2     B2
3      3      3      A3     B3
4      4      4      A4     B4
5      5      5      A5     B5
```

To read A: skip=1, offset=3, stride=5 To read B: skip=1, offset=4, stride=5

ASCII file organization 2:

```
A1     A2     A3     A4     A5
A6     A7     A8     A9     A10
A11    A12    A13    A14    A15
B1     B2     B3     B4     B5
B6     B7     B8     B9     B10
B11    B12    B13    B14    B15
```

To read A: skip=0, offset=0, stride=1 To read B: skip=3, offset=0, stride=1

ASCII file organization 3:

```
A1      B1      A2      B2      A3      B3
A4      B4      A5      B5      A6      B6
A7      B7      A8      B8      A9      B9
A10     B10     A11     B11     A12     B12
```

To read A: skip=0, offset=0, stride=2 To read B: skip=0, offset=1, stride=2

ASCII file organization 4:

```
TEMP1=A1 TEMP2=A2 TEMP3=A3 TEMP4=A4
TEMP5=A5 TEMP6=A6 TEMP7=A7 TEMP8=A8
PRESS=B1 PRESS=B2 PRESS=B3 PRESS=B4
PRESS=B5 PRESS=B6 PRESS=B7 PRESS=B8
```

**read field** cannot read this file until the data labels and equal signs are edited out.

### *Example 2*

You have some 3-dimensional, curvilinear data that projects the amount and location of wood that will be eaten after five years by a colony of termites that has entered a 14th century Scandanavian grain silo structure at a particular spot in its base. The data is in one ASCII file, *decay.dat*, as a long sequential, numbered list of 1250 consumed-wood values that looks like this:

```
1,1002.707;
2,1443.971;
3,1307.069;
4,1240.354;
5,1778.715;
    ...
```

The coordinates that correspond to the data values are in a separate ASCII file, *loc.dat*, that looks like this:

```
LOC,1,0,0.2500000,0.0000000e+00,1.105255,0.0000000e+00;
LOC,2,0,0.2500000,0.0000000e+00,1.000000,0.0000000e+00;
LOC,3,0,0.5000000,0.0000000e+00,1.552552,0.0000000e+00;
LOC,4,0,0.5000000,0.0000000e+00,1.442042,0.0000000e+00;
LOC,5,0,0.5000000,0.0000000e+00,1.331531,0.0000000e+00;
    ...
```

In the data file, the second column represents the data. In the coordinate file, the fourth through sixth columns are the x, y, and z coordinates, respectively.

First, to read this data, you must use a text editor to globally edit out the commas and semi-colons, changing them to spaces. The files now look like:

```
1 1002.707
2 1443.971
  ...

LOC 1 0 0.2500000 0.0000000e+00 1.105255 0.0000000e+00
LOC 2 0 0.2500000 0.0000000e+00 1.000000 0.0000000e+00
  ...
```

The following ASCII description file, *decay.fld*, would import the data into AVS field format:

```
# AVS Field File
#
# Termite Decay after Five Years
#
ndim=3            # number of dimensions in the field
dim1=25           # dimension of axis 1
dim2 =10          # dimension of axis 2
dim3 =5           # dimension of axis 3
nspace=3          # number of physical coordinates
veclen=1          # number of elements at each point
data=float        # data type (byte, integer, float, double)
field=irregular   # field type (uniform, rectilinear, irregular)
coord 1 file=/Termite/loc.dat filetype=ascii offset=3 stride=7
coord 2 file=/Termite/loc.dat filetype=ascii offset=4 stride=7
coord 3 file=/Termite/loc.dat filetype=ascii offset=5 stride=7
variable 1 file=/Temite/decay.dat filetype=ascii offset=1 stride=2
```

### *Example 3*

The following ASCII description file specifies how to convert the volume data in the file */usr/avs/data/volume/hydrogen.dat* into an AVS field. *hydrogen.dat* is a series of binary byte values that represent the probability of finding an electron at various locations around a hydrogen nucleus. The first three bytes in the file give the X, Y, and Z dimensions of the data—however, this information is not part of the actual data and must be skipped over. You could examine these three bytes and determine what to use for the dimensions in the ASCII description file. Thereafter, it is just a matter of reading successive bytes. **offset** is not used because this is not an ASCII file. **stride** is allowed to default to 1.

```
# AVS field file
ndim=3             # number of dimensions in the field
dim1=64            # dimension of axis 1
dim2=64            # dimension of axis 2
dim3=64            # dimension of axis 3
nspace=3           # number of physical coordinates per point
veclen=1           # number of components at each point
data=byte          # data type (byte, integer, float, double)
field=uniform      # field type (uniform, rectilinear, irregular)
variable 1 file=/usr/avs/data/volume/hydrogen.dat filetype=binary skip=3
```

### *Example 4*

This ASCII description file specifies how to use **read field** to convert the image data in */usr/avs/data/image/mandrill.x* into an AVS field. The first two words in *mandrill.x* are 32-bit integers that specify the horizontal and vertical dimensions of the image. This information must be skipped over — you must supply it in the ASCII description file. Thereafter, *mandrill.x* is a succession of 32-bit straight binary words, one word per pixel. However, in AVS, each of these words is considered to be a vector of 4 bytes. The first byte is the "alpha" (or "transparency") value for the pixel, and the second through fourth bytes are the red, green, and blue values for each pixel. Thus, this whole file is treated as a series of binary bytes.

```
# AVS field file
#
ndim = 2                        # number of dimensions in the field
nspace=2                        # number of physical coordinates
dim1=500                        # dimension of axis 1
dim2=480                        # dimension of axis 2
veclen=4                        # number of components at each point
data=byte                       # data type (byte, integer, float, double)
field=uniform                   # field type (uniform, rectilinear, irregular)
label = alpha, red, green, blue     # labels the vector elements
variable 1 file=/usr/avs/data/image/mandrill.x filetype=binary skip=8 stride=4
variable 2 file=/usr/avs/data/image/mandrill.x filetype=binary skip=9 stride=4
variable 3 file=/usr/avs/data/image/mandrill.x filetype=binary skip=10 stride=4
variable 4 file=/usr/avs/data/image/mandrill.x filetype=binary skip=11 stride=4
```

### *Example 5*

This ASCII description file reads a FORTRAN unformatted ARC 3D dataset. The file is 34x34x34, made up of floating point numbers. It is irregular; therefore there is both computational and coordinate data in two separate files. The vector length is six. The data file is written as a 24 byte header that must be skipped over followed by all vector 1 values, all vector 2 values, etc. The coordinate file is written as a 12 byte header (a full word for each of the X, Y, and Z dimensions) followed by all X cooordinates, all Y coordinates, then all Z coordinates. The person is using a relative file specification—the filenames will be interpreted relative to the directory of this ASCII description file.

```
# AVS field file
# to read an Arc 3D file that's 34x34x34z
ndim = 3
dim1 = 34
dim2 = 34
dim3 = 34
nspace = 3
veclen = 6
data = float
field = irregular
#
coord 1 file=for003.dat filetype=unformatted skip=20 stride=1
coord 2 file=for003.dat filetype=unformatted skip=157236 stride=1
coord 3 file=for003.dat filetype=unformatted skip=314452 stride=1
#
```

```
variable 1 file=for004.dat filetype=unformatted skip=32 stride=1
variable 2 file=for004.dat filetype=unformatted skip=157248 stride=1
variable 3 file=for004.dat filetype=unformatted skip=314464 stride=1
variable 4 file=for004.dat filetype=unformatted skip=471680 stride=1
variable 5 file=for004.dat filetype=unformatted skip=628896 stride=1
variable 6 file=for004.dat filetype=unformatted skip=786112 stride=1
```

Given that the coordinate file header is 12 bytes, why is the **skip** value 20? It is 20 because **read field** must be directed to skip over the one word FORTRAN unformatted record header, and the one word FORTRAN unformatted record trailer (12+4+4=20). The same 20 bytes must be added to the **skip** value for **coord**s 2 and 3. Similarly, the data file's 24 byte header must have 8 bytes added to it for a total of 32. **read field** correctly deals with the remaining "invisible" FORTRAN unformatted record header and trailer words in the remainder of the file, provided that all values pertaining to a dimensions (X, Y, or Z) and/or all values pertaining to a vector (e.g., all x-momentums) were written as one record. It will also work if the records were written as repeating groups (e.g., X, Y, Z; X, Y, Z; X, Y, Z; etc.) or (vec1, vec2, vec3, vec4, vec5, vec6: repeat). It will not work if the output was generated as "first half of X's; second half of X's", since the intermediate FORTRAN formatting words will throw off its **stride**s.

### Hints on Using read field

If you did not write the application that produces your data, and you instead use an application acquired in binary form from an outside source, then you may not know what the format of the application's binary or unformatted Fortran data files actually *is*. This can be difficult to determine just by looking at the raw data files. However, most established packages do document their internal file format and you can usually contact the vendor's support organization for this information.

It can be hard to tell when you've read a dataset correctly. To check on this, use the **print field** module. The **print field** module will display the contents of an AVS field in ASCII format. You can look at the results in **print field**'s output display window. If this display window is too narrow, **print field** also writes its output to a temporary file that you can view with any text editor.

You can also use the **volume bounds** module to outline the resultant field's extents. If there has been a mistake reading the coordinate information for rectilinear and irregular datasets, it will usually show up very graphically with **volume bounds**.

Here is one useful "debugging" network for importing uniform or irregular volume data.

```
                         read field
                             |       |
 generate colormap           |       |--------------+
          |                  |       |              |
          |    +-------------|       |              |
          |    |             |  print field   write field
        color range          |
               |             |
               |             |--------------------+
               |             |                    |
          any-mapper-module          volume bounds
                             |                    |
                             |--------------------+
                             |
                             |
          geometry viewer
```

You can replace *any-mapper-module* with any of the modules that produce a colored geometry picture of the data (such as the **orthogonal slice/field to mesh** module pair). If you are trying to produce a vector field, it might be easier to try reading just one vector element at first until all the other variables are correct. You would then include an **extract scalar** module between the **read field** and mapper module in this network.

Once you have read the data correctly, you should write it permanently to disk in binary native AVS field format with the **write field** module. **read field** reads native format AVS fields much more quickly than it is able to parse foreign-format files.

You can also be creative with **read field**. For example, many imaging applications produce a series of separate images that actually represent slices through a volume (such as density slices through a cranium, for example). The scientist may wish to deal with the images as a 3D volume of data. You could write a very simple C or Fortran program that generates the uniform X and Y coordinates of the images as ASCII data, as well as a series of rectilinear Z coordinates that position the slices accurately in space. You can then use **read field** to read the original data file, and the new coordinate file to produce a rectilinear 3D volume representation of the data.

### *read field Limitations*

**read field** does have limitations:

- Its parsing commands primarily deal with files that are structured with header information that can be skipped over, followed by data in a regular, repeating format. **read field**, for example, could not be used to read an image file that contained run-length encoded data.
- In many established packages (including PLOT3D) the data that is stored in the input data file is not the final data that is actually used by the packages when they perform their computations. For instance, values such as temperature or friction may be *calculated* from the raw data using well-known constant values and formulae. **read field** cannot, in principle, a priori know what these automatic calculations should be. It may be nec-

essary to write modules that further process the original field data into its final, usable form.

## AVS Module: read plot3d

The **read plot3d** module in the Unsupported module library reads computational fluid dynamics data files in the National Aeronautics and Space Administration's PLOT3D format and converts them into AVS field format. (To access the Unsupported module library in the Network Editor, use the **Module Tools** submenu and the **Select Module Library** button.) There are two types of PLOT3D files, the XYZ grid files that specify the irregular coordinate information, and the Q solution files that contain a vector of values for each point in the grid.

XYZ and Q file pairs can contain a single set of grid/data mappings, or multiple grid/data mappings. The XYZ file can also contain an IBLANK value for each point, although these data values will not be stored in the output field. The data within the files can be in either binary, or Fortran formatted or unformatted format. XYZ grid file and Q solution file formats must match in all respects.

**read plot3d** requires that you know the format (dimensionality, whole/plane, number of grids, binary/formatted/unformatted, and whether IBLANK values are present) of the PLOT3D files that you are trying to read. It does not check to verify that the values it is given map reasonably to the data.

Q solution files contain three to five floating point values for each point in the grid: X momentum (1D), Y momentum (1D and 2D), Z momentum (1D, 2D, and 3D), density, and stagnation. The four header values (FSMACH, ALPHA, RE and TIME) are ignored.

**read plot3d** does impose some practical limits to the size of the data: No one dimension can be larger than 1,000,000; the output data can have no more than 1,000,000,000 points in any one grid; and the maximum number of data grids is 50.

**read plot3d** outputs an AVS field (field irregular float 1D, 2D, or 3D of 3-, 4-, or 5-vector). The AVS field output will match the dimensionality of the original PLOT3D dataset. At each point in the grid will be three to five floating point values: density, X momentum (and Y momentum, and Z momentum, if appropriate), and stagnation, in that order. **Note:** The output AVS field represents only the *first* grid of multi-grid parameter files. There is no way to pack multiple grids into a single AVS field.

An additional module, **cfd values**, is often used in conjunction with **read plot3d**. **cfd values** inputs the field data from **read plot3d** (including density, x-momentum, y-momentum, z-momentum, and stagnation), and allows the user to compute the following values:

- energy

- pressure
- enthalpy
- mach number
- temperature
- total pressure
- total temperature

## AVS Module: read image

Although there is a distinct AVS image file format for storing images on disk, once AVS reads an image into a network, it treats image data as a 2D 4-vector byte AVS *field*. Therefore, the best way to approach importing image data is to view the problem as converting your image data into an AVS field.

### read image Data File Format

The **read image** module and the Image Viewer subsystem (also implemented as the **image viewer** module) can read a file that contains an image—a 2D array of pixel values. In AVS, such files should have names that end with a *.x* suffix.

The file must begin with a two-word header, which specifies the dimensions of the image:

first word: number of pixels in horizontal direction (32-bit integer)
second word: number of pixels in vertical direction (32-bit integer)

There is no explicit limit on the size of an image.

The remainder of the file is a sequence of 4-byte (32-bit) words, one for each pixel of the image. The pixels are arranged rowwise; there is no padding at the end of a row.

The four bytes of a pixel are interpreted as four component values in the range 0..255. Three of the bytes are the red, green, and blue color components. The fourth byte is an auxiliary field, which is used by some AVS modules to represent an opacity/transparency value:

```
+-----------+-----------+-----------+-----------+
| auxiliary |    red    |   green   |   blue    |
+-----------+-----------+-----------+-----------+
```

The AVS *image* data file format is shown below. Image data files should have names that end with a *.x* suffix.

```
number_of_pixels_in_x_dimension
number_of_pixels_in_y_dimension
pixel_1
pixel_2
pixel_3
```

```
  .
  .
  .
pixel_n
```

In the example above, the total number of pixels is:

(*number_of_pixels_in_x_dimension * number_of_pixels_in_y_dimension*)

**Note:** The AVS image data type is primarily designed to hold data that was originally produced as RGB (red, green, blue) data, whether as full 24-bit true color or some more limited format. Many other kinds of scientific data are *also* referred to as "image data." For example, medical imaging devices produce 2D or 3D arrays of numbers that represent some quantity, such as density. This is usually called "image data." However, such data is best represented as an *AVS field*, not as an AVS *.x* image. A field can always be converted into an image that the AVS Image Viewer can manipulate.

## AVS Module: read volume

Although there is a distinct AVS volume file format for storing 3D uniform volumes on disk (where each element is a byte holding a value from 0-255); once AVS reads an volume into a network, it treats the volume data as a 3D uniform scalar byte AVS *field*. Therefore, the best way to approach importing this narrow class of volume data is to view the problem as converting your volume data into an AVS field. You are referred to the "Field Data" discussion earlier in this chapter.

### read volume Data File Format

Measurement data often takes the form of a 3-dimensional array, which corresponds to a uniform lattice in 3D space. Each array value indicates one measurement (temperature, pressure, etc.) at the corresponding lattice point. Such data can be represented as a *uniform 3D field*, as described in the preceding section. For convenience, AVS also provides a simpler *volume data* format to accommodate this type of data.

The AVS volume data format requires that each value in the data array be a byte. (For other data types (e.g. single-precision), you must use the more general *field* construct.) Volume data files should have names that end with a *.dat* suffix.

**Note:** The **volume data** and **field** file formats are not compatible.

A volume data file begins with a three-byte header, which specifies the size of the array in the first (X), second (Y), and third (Z) dimensions. Since each dimension's size must be expressed as a 1-byte number, the largest array supported by this file format is 255x255x255.

The remaining contents of the file are the values of a 3D array of bytes, in column-major order ("FORTRAN-style"). For example, the values in a 50x20x10 array would be stored as follows (using FORTRAN notation):

DATA(1,1,1)
DATA(2,1,1)
DATA(3,1,1)
...
DATA(50,1,1)
DATA(1,2,1)
DATA(2,2,1)
DATA(3,2,1)
...
DATA(1,20,1)
DATA(2,20,1)
DATA(3,20,1)
...
DATA(1,20,2)
DATA(2,20,2)
DATA(3,20,2)
...
DATA(1,20,10)
DATA(2,20,10)
DATA(3,20,10)
...
DATA(50,20,10)

The volume data file format is shown below:

```
number_of_voxels_in_x_dimension
number_of_voxels_in_y_dimension
number_of_voxels_in_z_dimension
voxel_1
voxel_2
voxel_3
  .
  .
  .
voxel_n
```

In the example above, the total number of voxels is:

(*number_of_voxels_in_x_dimension* *
*number_of_voxels_in_y_dimension* *
*number_of_voxels_in_z_dimension*)

## Programming Examples

If none of the above alternatives for importing data into AVS field format is satisfactory, you will need to write a module that will read the data into AVS field format. AVS contains a library of routines for building fields, and storing and accessing field elements. These routines are described in "Appendix A" of the *AVS Developer's Guide*.

There are several source code examples of modules that read data into AVS field format. All are found in the directory */usr/avs/examples*.

**read_image.c**
**read_image_f.f**
These modules, one in C and one in Fortran, read data in AVS image file format into a 2D 4-vector byte field.

**read_vol.c**
**read_vol_f.f**
These modules, one in C and one in Fortran, read data in AVS volume file format into a 3D scalar byte field.

**read_scans.c**
This example module reads a set of hypothetically-formatted images stored in separate files, each representing a planar scan, into a uniform 3D integer field (a 3D volume of integer data). It assumes a certain location for the input files, and a pattern to their file names; fixed-size images (64 x 64), a fixed-size header in each image file, an integer of data for each pixel, and that the image data is stored in "row major order", that is that the second index of the data varies most rapidly (1,1 1,2 1,3, etc.). *read_scans.c* can be used as a template for a variety of image file formats that need to be converted to 2D or 3D AVS fields.

**read_plot3d.c**
The file *read_plot3d.c* is a C program that reads NASA PLOT3D 3D whole format datasets into AVS field format.

**Note:** The **read plot3d** example is *not* the same as the **read plot3d** module in the AVS Unsupported library.

This is probably the most sophisticated example source code module since its module description section includes the definition of several different kinds of interface widgets, it must deal with allocating memory to store the field based upon the changing size of the input datasets, it reads column-major order data, it reads vector data instead of simple scalars, and it uses formulae to produce output values not found in the input data.

## Geometry Data

AVS geometry data contains a description of three-dimensional geometries and scenes, including object type, coordinate data, surface attributes, rendering modes, material properties, and transformations.

AVS objects are one of the following types:

**Polyhedron**
A list of vertices with an indirect list of pointers into these vertices for each polygon.

**Polygon**
A list of vertices for each polygon.

**Mesh**
A 2D array of values, either scalars (for a height field) or vertices.

**Sphere**
A list of center points and radii.

**Polytriangle**
A single list of vertices representing polylines, disjoint lines, or a triangle mesh, where the connectivity is implied by the particular data type.

AVS has routines that allow a module to change several properties of an object, including:

- The geometric data defining the object
- Surface or line color
- Render mode (Gouraud, Phong, wireframe, etc.)
- Parent (the name of the parent object)
- Object material properties
- Object, camera, and light transformation
- Object visibility, deletion
- Object color, light source color and camera background color
- Camera background color
- Light source on/off, type
- Texture mapping
- Transformation mode (controls how objects are transformed)
- Selection mode (controls how objects are picked)
- Center of rotation and scaling
- Viewable region of data
- Viewing projection

AVS represents graphical objects in an external binary data format called *geom*. *geom* format files are created using the *libgeom* library documented in the "Geometry Library" appendix to the *AVS Developer's Guide*.

## *AVS Module: geometry viewer*

The **geometry viewer** module provides access within an AVS network to the complete Geometry Viewer subsystem. Many different modules can supply input geometries. That is, many *geometry*-format outputs can be connected to

**geometry viewer**'s geometry input port. All the objects will be combined into a single scene. Each module providing input to **geometry viewer** can define attributes and geometries for any number of objects. Each of these modules can also define a hierarchical relationship among its objects.

You can also invoke **geometry viewer** with no inputs, so that the "scene" is initially empty. Objects can be added to a scene either by upstream modules or by the **Read Object** selection on the **geometry viewer** control panel. Geometries and descriptions sent by upstream modules can be saved to files using the **Save Object** and **Save Scene** selections. In this way, you can save visualization results and retrieve them later with **Read Scene** or **Read Object**.

## AVS Module: read geom

The **read geom** module reads a file containing an AVS *geometry* and outputs the geometry to one or more modules connected to its output port. The resulting object will be named after the file from which it was read.

This module reads *geom* files only. See the section below on "AVS Geometry Filters" for a description of how to create AVS *geom* files.

## AVS Module: pdb to geom

The **pdb to geom** module converts molecular data in Brookhaven Protein Data Bank format into AVS *geom* format. It is in the Data Input column of the Network Editor's module palette. It expects an input file with a *.pdb* suffix. There are two sample input datasets, */usr/avs/filter/example/bwdna.pdb* and */usr/avs/filter/example/crambin.pdb*. (This latter can also be found in the */usr/avs/data/pdb* directory.)

For instance, the table below shows part of a file written in the Brookhaven Protein Data Bank format. This file defines the structure of a particular protein molecule called "crambin". There is an AVS data input module to read files in this format. Users can supply their own data input modules for other data formats.

```
ATOM      1  HN1 THR     1       17.017  14.972   4.068
ATOM      2  HN2 THR     1       16.297  13.912   2.883
ATOM      3  N   THR     1       16.982  14.095   3.587
ATOM      4  HN3 THR     1       17.707  14.470   3.008
ATOM      5  CA  THR     1       16.949  12.808   4.348
ATOM      6  C   THR     1       15.686  12.779   5.142
ATOM      7  O   THR     1       15.236  13.827   5.603
   ...
```

## AVS Geometry Filters

There are a number of geometry filters supplied with AVS that will convert data into the *geom* format. These filters fall into two classes:

- Filters that read several file formats commonly found in the technical community. These geometry filters can be used to import existing datasets into AVS *geom* format.
- Filters that read ASCII files in a format unique to AVS. Each of these geometry filters inputs ASCII data to create one of the fundamental geometry structures manipulated by the *libgeom* programming library, such as polygons, polyhedrons, meshes, and spheres. These AVS-specific geometry filters can be used in two ways:
  - You might be able to construct an ASCII file version of your geometry data that one of these geometry filters could read to create a *geom* object. However, the geometry filters are probably not robust enough to be used on a production basis for a real application. The *mesh.c* filter has the most general utility.
  - The geometry filters serve as example programs that show how to construct the primitive *geom* structures using *libgeom* calls. Two of the geometry filters are written in Fortran in addition to C.

The sources to geometry filters are all located in the directory */usr/avs/filter*. The executable versions of geometry filters are all located in the directory */usr/avs/bin*. Sample datasets in the formats these geometry filters expect are located in */usr/avs/filter/example*.

The following common file formats have geometry filters:

**Table 2-2. AVS-Readable Geometry File Formats: General Use**

| File Format | File Suffix | Source Filename | Executable Filename |
|---|---|---|---|
| Mathematica ThreeScript | *.ts* | -- | ts_to_geom |
| Movie BYU | *.byu* | byu.c | byu_to_geom |
| Protein Data Bank | *.pdb* | pdb.c | pdb_to_geom |
| UNC | *.ppoly* | ppoly.c | ppoly_to_geom |
| Wavefront | *.wfront* | wfront_geom.c | wfront_to_geom |

There is a sample Movie BYU format dataset in */usr/avs/filter/example/cube.byu*, and a sample Mathematica ThreeScript dataset in */usr/avs/filter/example/cone.ts*.

The following AVS-specific geometry filters are provided:

**Table 2-3. AVS-Readable Geometry File Formats: AVS Specific**

| Object Type | Source Filename | Executable Filename |
|---|---|---|
| polyhedron | polyh.c | polyh_to_geom |
| disjoint polygon | polygon.c, polygon.f | polyg_to_geom |

**Table 2-3. AVS-Readable Geometry File Formats: AVS Specific**

| Object Type | Source Filename | Executable Filename |
|---|---|---|
| mesh | mesh.c, mesh.f | mesh_to_geom |
| sphere | sphere.c | sphere_to_geom |

The source to each AVS-specific geometry filter includes a description of the ASCII file format it expects to read. There are sample polygon, polyhedron, and mesh ASCII data files in */usr/avs/filter/example*.

The remainder of this section describes how to use these geometry filters.

## Automatic Data Filtering

AVS geometry filters are normally executed at the shell level. The geometry filter facility is unique in this regard—if, during a **Read Object** function within the AVS Geometry Viewer, AVS determines (using the filename extension) that the file you select is in a known data format, it invokes the appropriate geometry filter *automatically* to create a corresponding *geom* file on disk. It then reads in the object from the *.geom* file.

When you execute the AVS function **Read Object**, you select a filename in the File Browser window. If the file has a *.geom*, *.scr*, or *.obj* extension, AVS reads it in directly.

If the file has another extension, AVS determines whether the file contains data in a "known format," as follows:

- It looks in its */usr/avs/bin* directory for utility programs named *..._to_geom*. Each such geometry filter determines a particular filename extension. For instance, the geometry filter utility *wfront_to_geom* determines the extension *.wfront*.
- It compares the extension of the **Read Object** filename with its list of geometry filter extensions.
- If there is a match, AVS automatically invokes the appropriate geometry filter utility, creating a *geom*-format file of the same name, with a *.geom* extension. This file is created in the directory that the original file is located in.
- The **Read Object** function is completed by reading in the newly created *.geom* file.

## Shell-Level Usage of Geometry Filter Utilities

Each of the geometry filters can also be invoked from the shell, in the usual fashion. Each one reads a single file from *stdin* and writes a single *geom*-format file to *stdout*. Typically, you should redirect *stdout* to a file whose extension is *.geom*, so that it is directly readable by the AVS application. Command-Line Options

Most of the geometry filters don't accept any command line options. The exceptions are as follows:

The **ts_to_geom** filter utility accepts the following options:

**-bbox** *xmin xmax ymin ymax zmin zmax*
Define a bounding box to scale the object down non-uniformly.

**-bratios** *x y z*
Alter the aspect ratio of the bounding box. *x*=1, *y*=1, *z*=1 is a uniform aspect ratio. *x*=1, *y*=1, *z*=0.5 forces the Z dimension to be half the size of the X and Y dimensions.

**-boxed**
Put a wireframe box around the object.

**-noscale**
Do not attempt to scale the object at all.

The **pdb_to_geom** filter accepts the following option:

**-balls**
Use the sphere representation instead of the default ball-and-stick representation.

### Postprocessor Filters

AVS supplies an additional set of geometry filters, which you can use to postprocess the output of the *geom*-format converters. For instance, if a file in Movie BYU format defines an object with normals that point inward, you can make a *.geom* file with normals that point outward as follows:

```
byu_to_geom < myobject.byu | geom_flip > myobject.geom
```

The **geom_flip** postprocessor reads and writes a *geom*-format file, flipping the normals of the object defined therein.

The following table lists the postprocessor geometry filters supplied. Except for **send_to** and **animate_to**, each filter reads a file from *stdin* and writes a file to *stdout*.

**Table 2-4. Postprocessor Filters**

| Filter Name | Description |
|---|---|
| **geom_flip** | Flip normals of object. |
| **geom_pickable [-non]** | Make all objects pickable (non-pickable), so that their attributes can be set (cannot be set) individually. |
| **geom_scale** | Scales the object uniformly, so that it lies within the unit cube, (-1,-1,-1) to (1,1,1). Also, converts the objects's normals to unit length (normalizes them). |
| **geom_to_normls [-scale ]** *length* | Produces a disjoint-line object that represents the normals of the input object. The **-scale** option specifies the length of the normals. The default length is 1. |
| **geom_to_text** | Produces an ASCII version of the input *.geom* file. |
| **text_to_geom** | Produces a *.geom* version of the input ASCII file. The combination of **geom_to_text** and **text_to_geom** can be used to transfer geometry files between machines with different byte ordering or different word sizes. |
| **geom_split [***name***]** | Creates a series of *.geom* files, each of which contains one of the objects defined in the input file. This is used for input files that contain more than one *geom* object. Each output file is named *name.n***.geom**. If you don't specify the optional *name*, the files are named *split0.geom*, *split1.geom*, etc. |
| **send_to** *filename* | Causes a currently-executing AVS program to read the specified file, which should be a *.geom* file. |
| **animate_to -file** *name geom-file1 geom-file2* | Creates a script that defines a cycle object, stores the script as *name.***obj**, and casues a currently-executing AVS program to read the script file. The cycle includes the specified *geom-file* sequence. |

## Programming Examples

If none of the geometry filters listed above is satisfactory for your application, then you can either write a custom geometry filter, or write an AVS geometry-producing module. As mentioned earlier, the latter approach will be the most effective solution.

In either case, you must use the *libgeom* calls documented in the *AVS Developer's Guide* "Geometry Library" appendix to create an AVS-compatible geometry. The source code versions of all of the geometry filters described above are useful templates for generating geometry filters. There are also several geometry-producing modules in the */usr/avs/examples* directory.

## Unstructured Cell Data

Unstructured cell data (UCD) is commonly used in structural analysis and computational fluid dynamics. A UCD data structure consists of an irregular coordinate **structure** (or "model") made up of **cells**. Cells may be points, lines, quadrilaterals, triangles, tetrahedrons, pyramids, prisms, or hexahedrons. Each cell has a corresponding number of **nodes**. Data can be associated with the entire structure, with each cell, and with each node. The data is structured as a set of **components**. Each component can be either a scalar or a vector.

The UCD data type and the library of routines for creating and manipulating UCD are described in detail in the *AVS Developer's Guide* "Unstructured Cell Data" appendix.

## AVS Module: read ucd

**read ucd** reads an AVS-compatible UCD structure from a file. The file must have a *.inp* suffix or it will not be displayed in **read ucd**'s file browser.. The file may be ASCII or binary.

Binary UCD files have a different format than ASCII UCD files. Specifically, if a file is binary then it is assumed that it is in the format output by the module **write ucd**.

ASCII UCD files have a simple format described below. For a more detailed description of both ASCII and binary file formats, see the "Unstructured Cell Data" appendix of the *AVS Developer's Guide*.

### ASCII UCD File Format

The input file cannot contain blank lines or lines with leading blanks. The numbers down the left correspond to the above descriptions and are not part of the ASCII file. Comments, if present, must precede all data in the file—comments within the data will cause read errors. The general order of the data is:

1. Numbers defining the overall structure, including the number of nodes, the number of cells, and the length of the vector of data associated with the nodes, cells, and the model.
2. For each node, its node-id and the coordinates of that node in space. Node-ids must be integers, but any number including non-

sequential numbers can be used. Mid-edge nodes are treated like any other node.

3. For each cell: its cell-id, material, type (hexahedral, pyramid, etc.), and the list of node-ids that correspond to each of the cell's verticies. (The UCD appendix shows the order in which cell verticies are numbered.)
4. For the data vector associated with nodes, how many components that vector is divided into (e.g., a vector of 5 floating point numbers may be treated as 3 components: a scalar, a vector of 3, and another scalar, which would be specified as 3 1 3 1).
5. For each node data component, a component label/unit label pair, separated by a comma.
6. For each node, the vector of data values associated with it.
7. That is the end of the node definitions. Cell-based data descriptions, if present, then follow in the same order and format as items 4, 5, and 6.
8. The single model-based data descriptions, if present, comes last.

```
    # <comment 1>
         .
         .
         .
    # <comment n>
1.  <num_nodes>  <num_cells>  <num_ndata>  <num_cdata>  <num_mdata>
2.  <node_id 1>  <x>  <y>  <z>
    <node_id 2>  <x>  <y>  <z>
         .
         .
         .
    <node_id num_nodes>  <x>  <y>  <z>
3.  <cell_id 1>  <mat_id>  <cell_type>  <cell_vert 1> ... <cell_vert n>
    <cell_id 2>  <mat_id>  <cell_type>  <cell_vert 1> ... <cell_vert n>
        .
        .
        .
    <cell_id num_cells>  <mat_id>  <cell_type>  <cell_vert 1> ...<cell_vert n>
4.  <num_comp for node data>  <size comp 1>  <size comp 2>...<size comp n>
5.  <node_comp_label 1> ,  <units_label 1>
    <node_comp_label 2> ,  <units_label 2>
        .
        .
        .
    <node_comp_label num_comp> ,  <units_label num_comp>
6.  <node_id 1>  <node_data 1> ... <node_data num_ndata>
    <node_id 2>  <node_data 1> ... <node_data num_ndata>
        .
        .
        .
    <node_id num_nodes>  <node_data 1> ... <node_data num_ndata>
7.  <num_comp for cell's data>  <size comp 1>  <size comp 2>...<size comp n>
    <cell-component-label 1> ,  <units-label 1>
```

```
    <cell-component-label 2> ,  <units-label 2>
        .
        .
        .
    <cell-component-label n> ,  <units-label n>
    <cell-id 1> <cell-data 1> ... <cell-data num_cdata>
    <cell-id 2> <cell-data 1> ... <cell-data num_cdata>
        .
        .
        .
    <cell-id num_cells> <cell-data 1> <cell-data num_cdata>
8.  <num_comp for model's data>  <size comp 1>  <size comp 2>...<size comp n>
    <model-component-label 1> ,  <units-label 1>
    <model-component-label 2> ,  <units-label 2>
      .
      .
      .
    <model-component-label n> ,  <units-label n>
    <model-id> <model-data 1> <model-data num_mdata>
```

The UCD structure and library will support either integer or character node-, cell-, and model-ids, (referred to in the library documentation as **names**). However, the **read ucd** module only accepts integer node-ids, cell-ids, and model-ids. This is shown in the example below. The ids do not have to be consecutively numbered.

Also note that, at present, most of the UCD modules do not make use of cell and model-based data, thus the input data examples all show "0" for **<num-cdata>** and **<num-mdata>**. User-written modules can use the UCD library to manipulate cell- and model-based data.

### *Example ASCII UCD File*

The following is an example of a simple UCD file. This UCD structure has 8 nodes in 1 hexahedral cell. Associated with each node is a single scalar data value, making up one component that this person labels "stress," and specifies a "lb/in**2" unit label. There is no cell- or model-based data. See the "Unstructured Cell Data" appendix in the *Developer's Guide* for more examples.

```
#
# Simple AVS UCD File
#
8 1 1 0 0                     <--8 nodes, 1 cell, 1 component of node data
1  0.000   0.000   1.000      <--Node coordinates
2  1.000   0.000   1.000
3  1.000   1.000   1.000
4  0.000   1.000   1.000
5  0.000   0.000   0.000
6  1.000   0.000   0.000
7  1.000   1.000   0.000
8  0.000   1.000   0.000
1 1 hex 1 2 3 4 5 6 7 8   <--cell id, material id, cell type, cell vertices
1 1                        <--num data components, size of each component
stress, lb/in**2           <--Component name, units name
1  4999.9999               <--Data value for each node component
```

```
2  18749.9999
3  37500.0000
4  56250.0000
5  74999.9999
6  93750.0001
7  107500.0003
8  5000.0001
```

## Programming Examples

The AVS release includes several sample programs, one in Fortran and threee in C, that deal with AVS UCD-format files. These sample programs can be used as models for your own modules.. The UCD library these samples call is documented in the "Unstructured Cell Data Library" appendix to the *AVS Developer's Guide*.

### read_ucd.c

The file */usr/avs/examples/read_ucd.c* is a C program source. It reads either a binary or ASCII format UCD dataset (file suffix *.inp*), creating the UCD structure format used by the various UCD visualization modules. **Note:** ignore the binary section of this example. It reads AVS 3 format UCD binary files; a format that has changed with the AVS 4 release. The ASCII section is still valid.

### gen_ucd.f

The file */usr/avs/examples/gen_ucd.f* is the Fortran source to a module that first generates its own scalar or vector hexahedral data, then writes it out as a UCD structure.

### ucd_thresh.c

The file */usr/avs/examples/ucd_thresh.c* is the C source to the **ucd threshold** module. It is a filter that both reads an existing UCD structure and creates a new UCD structure as output. There are some extra variables declared in the header that the program does not actually use.

### ucd_extract.c

The file */usr/avs/examples/ucd_extract.c* is the C source to the **ucd extract** module. It is a filter that reads an existing UCD structure, extracts various components from the data, and outputs a new UCD structure. As with *ucd_thresh.c*, there are some extra variables declared in the header that the module does not use.

## Colormap Data

An AVS colormap is a data structure which implements a transfer function that assigns a color to each value between an upper and a lower bound. A colormap consists of:

- Four arrays of floating-point values, one each for hue, saturation, brightness, and opacity. Each value is normalized between 0.0 and 1.0 inclusive.
- An integer indicating the number of colors — this is the length of each of the four arrays.
- Floating-point lower and upper bounds that determine the resolution of the colormap. The lower bound is a data value that maps to the first element of each array. The upper bound is a data value that maps to the last element in each array. All intermediate data values are mapped to elements within the arrays.

The hue, saturation and brightness (HSB) color space can be thought of as an inverted cone:

- The **hue** axis runs circularly around the cone. Example hue values and corresponding hues are given below:

  *0.00 = red*
  *0.16 = yellow*
  *0.33 = green*
  *0.50 = cyan*
  *0.66 = blue*
  *0.83 = magenta*
- The **saturation** axis runs from the center of the cone (white) to its perimeter (fully saturated color). Example saturation values are:

  *0.00 = white*
  *0.50 = partially saturated* **hue**
  *1.00 = fully saturated* **hue**
- The **brightness** axis runs from the tip of the cone (black) to the base (white). Example brightness values are:

  *0.00 = black*
  *0.50 = partially darkened* **hue**
  *1.00 = full intensity* **hue**

The **opacity** value determines the amount of transparency the color has:

*0.00 = completely transparent*
*0.50 = partially transparent*
*1.00 = completely opaque*

## AVS Module: generate colormap

The **generate colormap** module produces an AVS *colormap* data structure, for use by modules that transform input data into color values. This module also allows the user to both read and write AVS-compatible colormaps. These *colormaps* are stored on disk as ASCII files, in the following format:

```
number_of_entries
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
hue  saturation  brightness  opacity
 ...
low_value high_value
```

All values in this file are in floating-point format, with the exception of **number_of_entries**, which is an integer. The hue, saturation, brightness, and opacity values are normalized to the range 0.0 - 1.0. For examples of files containing AVS colormaps, look at several of the files in the directory */usr/avs/data/colormap*.

## Molecule Data Type

The molecule data type (MDT) consists of ten defined CHEM objects that can be combined hierarchically in linked lists to represent the structure of molecules. At the root of the hierarchy is the CHEMmolecule. A CHEMmolecule is a data structure containing the molecule's name, and a unit in angstroms or bohrs. Chained off of the CHEMmolecule is an arbitrarily long list of CHEMatoms, CHEMchemunits, CHEMquantums, and user-defined data.

A CHEMatom is a data structure containing the atom's name, a color to represent it, its x, y, z position in space within the molecule, and its radius. The connectivity of the CHEMatom to other atoms, and the nature of the chemical bond (none, single, double, triple, hydrogen, disulfide, etc.) is specified by a link to a CHEMcandb data structure. CHEMatoms also contain a provision for user-defined data. By default, this area specifies the atom's atomic number, weight, hybridization type and charge.

CHEMchemunits are used to represent chemical substructures in the CHEMmolecule. For example, if the CHEMmolecule were DNA, CHEMchemunit could specify the animo acids of which it is composed.

The CHEMquantum data structure stores quantum information about the parent molecule such as charge, number of electrons, alpha and beta spin, a basis set for gaussian functions, molecular orbit, numbers of shells, etc.

The molecule data type and the *libchem* library that manipulates it is described in its own document, the *AVS Chemistry Developer's Guide.*

Within the AVS release:

***/usr/avs/examples/chemistry***
Shows source code examples to chemistry modules.

***/usr/avs/data/chemistry***
Shows sample chemistry data files.

***/usr/avs/chem_lib***

Contains sample AVS modules that illustrate use of the chemistry data type. They are not intended as "production" modules. These modules are not in the Network Editor's list of default module libraries. To use the chemistry modules, use the Network Editor's **Read Module Library** function on the **Module Tools** menu to load the file */usr/avs/chem_lib/chemistry*

***/usr/avs/networks/chemistry***

Shows the sample chemistry modules at work in networks. These may be read into the Network Editor using the **Read Network** function on the **Network Tools** menu.

*libchem*, the molecule data type, and the contents of the example directories above are also referred to collectively as the "Chemistry Developer's Kit" (CDK).

## AVS Module: Read structure file

Unlike the other AVS data types, no "standard" file format for the molecule data type has yet been defined. Data content and representations tend to differ more widely within the chemistry field than many other disciplines.

Nonetheless, the files */usr/avs/examples/chemistry/CHEMcon_r.c* and *CHEMcon_rf.f* are C and FORTRAN examples that read files containing AVS molecule data type information and produce MDTs as outputs. Example files in this format are found in */usr/avd/data/chemistry*.