read field

NAME

read field – read AVS field from a disk file, or import data files into AVS field format

SUMMARY

Name	read field		
Availability	Imaging, UCD, Volume, FiniteDiff module libraries		
Type	data		
Inputs	none		
Outputs	field *same-dimension same-vector same-data same-coordinates*		
Parameters	*Name*	*Type*	*Default*
	Read File	browser	
	Auto/ Portable(XDR)	choice	Auto

DESCRIPTION

The **read field** module has two input modes:

- In its first input mode, it reads an AVS *field* data structure from a disk file into a network. The format of an AVS *field file* is discussed below in the "Native Field Input" section.
- In its second input mode, it converts data stored in ASCII, Fortran unformatted, or pure binary data files into AVS field format. **read field** can thus be used to import *some* datasets into the AVS system. (The **file descriptor** module also performs this function, but with more flexibility.)

The two input modes—"native field input" and "data-parsing input"—are described separately in the sections below.

PARAMETERS

Read File A file browser window to specify the name of the file to be read.

Auto/Portable(XDR)

A pair of radio buttons that control how **read field** will interpret binary AVS field input files.

Auto

If **Auto** is selected, then **read field** will examine the ASCII header's "data=" line. If the file is described as just "data=integer", or "data=float", then **read field** assumes that the field file's binary data format is compatible with the system on which the **read field** module is executing. If the file is described as "data=xdr_float", "data=xdr_integer", or "data=xdr_double", then **read field** assumes that the binary area of the field file is written in machine-independent XDR (external data representation) format and will translate the binary portion of the field file into the binary format of the system on which the **read field** module is executing.

Portable(XDR)

If this is selected, then **read field** assumes that the binary portion of the field file is written in machine-independent XDR format (no matter what the ASCII header says) and will translate the binary portion of the field file into the binary format of the system on which the **read field** module is executing.

See the "Binary Compatibility on Different Hardware Platforms" section below for more information on this feature.

NATIVE FIELD INPUT

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

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

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

ASCII Header

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

```
# AVS field file
# creation date: Fri Aug 23 11:23:27 1991
#
ndim=2                          # number of dimensions in the field
dim1=500                        # dimension of axis 1
dim2=480                        # dimension of axis 2
nspace=2                        # number of physical coordinates per point
veclen=4                        # number of components at each point
data=byte                       # portable data format
field=uniform                   # field type (uniform, rectilinear, irregular)
min_ext=0.000000 0.000000                       # coordinate space extent
max_ext=499.000000 479.000000                   # coordinate space extent
label= alpha red green blue
min_val=0 0 0 0                 # minimum data values for each data component
max_val=0 255 255 255           # maximum data values for each data component
```

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

```
# AVS
```

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

Separator Characters

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

Binary Area

The size (in bytes) of the binary area depends on the field type:

- For **uniform** fields, the binary area contains data values followed by the coordinate values.

  Coordinate information is limited to minimum and maximum extent fullword values for each physical dimension (n-space) of the data. The minimum and maximum extent values in the coordinate binary area are copies of the **min_ext** and **max_ext** values in the field data structure, *except* when the field has been cropped, downsized, or interpolated. Then the field data structure contains the

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

Thus, the size of the binary area is the product of the following numbers:

value of **dim1**	(product of sizes of computational dimensions
value of **dim2**	yields total number of field elements)
...	
value of **dim***x*	
value of **veclen**	*(number of data values per field element)*
size of **data**	*(byte size of primitive data type)*

Plus:

8 ∗ value of **nspace**	*(2 coordinates per dimension, 4 bytes per coordinate)*

In the stream of data values:

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

- For **rectilinear** fields, the binary area contains both data values and coordinates for each scalar data value or vector of data values. The data values occupy the same amount of space as for a **uniform** field. Each coordinate is a single-precision floating-point number (4 bytes), and there is one coordinate for each array index in each dimension of computational space. Thus, the size of the coordinates area is:

  $(dim1 + dim2 ... + dimx) * 4$

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

- For **irregular** fields, the data area contains both data values and coordinates. The data values occupy the same amount of space as for a **uniform** field. Each coordinate is a single-precision floating-point number (4 bytes), and each field element is mapped to a point in *nspace*-dimensional physical space. Thus, the size of the coordinates area is:

  $(dim1 * dim2 ... * dimx) * nspace * 4$

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

Binary Compatibility on Different Hardware Platforms

Memory addressing on 32-bit systems is usually divided into two major hardware classes:

"Big-endian"

32-bit words are divided into 4 8-bit bytes, where the high-order byte is byte 0. Systems with this organization include Sun, Hewlett-Packard, and IBM workstations.

"Little-endian"

32-bit words are divided into 4 8-bit bytes, where the low-order byte is byte 0. Systems with this organization include Digital Equipment Corporation workstations.

Binary byte data are compatible between the two kinds of systems. Binary integer, floating point, and double-precision floating point data are *not* compatible between the two kinds of systems. For example, an integer AVS field file written on a Sun workstation would not normally be readable on a DEC workstation.

To make AVS field data interchangeable among platforms, the **write field** module has a **Native/Portable(XDR)** switch. Selecting **Portable(XDR)** will write the binary area of the field in Sun's external data representation (XDR). The field header will show "data=xdr_integer|xdr_float|xdr_double". If **Native** is selected, the field header will contain a comment at the end of the "data=" line stating what platform the field file was created on. **read field** uses its **Auto/Portable(XDR)** switches to either examine the ASCII header for the "data=xdr_" flag, or to force reading the data file as XDR format no matter what the ASCII header says. (Note: XDR format is simply 32-bit "big-endian" integers and IEEE standard format floating point.)

EXAMPLE 1

The following ASCII header describes a volume (3D uniform field) with a single byte of data for each field element. This format might be used to represent CAT scan data.

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

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

$$(64 * 64 * 64) * 1 * 1 = 262{,}144 \text{ bytes}$$

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

EXAMPLE 2

The following ASCII header describes a volume (3D uniform field) whose data for each field element is a 3D vector of single-precision values. This format might be used to represent the wind velocity at each point in space. This field file is written in XDR format.

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

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

$$(27 * 25 * 32) * 4 * 3 = 259{,}200 \text{ bytes}$$

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

EXAMPLE 3

The following ASCII header describes an irregular volume (3D irregular field) with one single-precision value for each field element. The binary area includes an (X,Y,Z)

coordinate triple for each field element, indicating the corresponding point in physical space. This format might be used to represent fluid flow data.

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

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

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

The coordinates area occupies this amount of space:

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

DATA-PARSING INPUT MODE

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

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

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

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

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

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

Suggestion: While experimenting with **read field**'s ASCII description file, connect its output port to the **print field** module's input port and use **print field**. This allows you to examine the results online, to see whether the data is being interpreted correctly.

read field chronicles its progress in a status display below the file browser widget as it works through the input files to assemble the AVS field.

ASCII Description File

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

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

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

Each line's meaning is explained in detail below.

```
# AVS field file     the string "# AVS" must be the first
#                    five characters in the file
#                    when a '#' character appears in a line,
#                    the rest of the line is a comment
#
ndim=3                      # REQUIRED--the number of dimensions in the field
dim1=40                     # REQUIRED--dimension of axis 1
dim2=32                     # REQUIRED--dimension of axis 2
dim3=32                     # REQUIRED--dimension of axis 3
nspace=3                    # REQUIRED--number of coordinates per point
veclen=3                    # REQUIRED--number of components at each point
data=float                  # REQUIRED--data type (byte,integer,float,double)
field=irregular             # REQUIRED--field type (uniform, rectilinear,irregular)
min_ext=-1.0 -1.0 -1.0      # OPTIONAL--coordinate space extent
max_ext=1.0 1.0 1.0         # OPTIONAL--coordinate space extent
label=x-velocity            # OPTIONAL--component label for variable 1
label=y-velocity            # OPTIONAL--component label for variable 2
label=z-velocity            # OPTIONAL--component label for variable 3
unit=miles-per-second       # OPTIONAL--describes unit of measure for variable 1
unit=miles-per-second       # OPTIONAL--describes unit of measure for variable 2
unit=miles-per-second       # OPTIONAL--describes unit of measure for variable 3
min_val=-2.18 -0.32 -3.73   # OPTIONAL--minimum data values per component
max_val=5.79 3.54 1.50      # OPTIONAL--maximum data values per component
#
# For each coordinate X, Y, and Z, where to find it and how to read it
#
coord 1 file=/usr/userid/data/wing.bin filetype=binary skip=12
coord 2 file=/usr/userid/data/wing.bin filetype=binary skip=163852
coord 3 file=/usr/userid/data/wing.bin filetype=binary skip=327692
#
# For each value in the vector, where to find it and how to read it
#
```

```
variable 1 file=/usr/userid/data/wdata.bin filetype=binary skip=28
variable 2 file=/usr/userid/data/wdata.bin filetype=binary skip=163868
variable 3 file=/usr/userid/data/wdata.bin filetype=binary skip=327708
```

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

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

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

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

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

With the exception of filenames, ASCII description file specifications are *not* case-sensitive.

- You can surround the = character with any amount of white space (including none at all). For example, "dim2 = 32", "DIM 2 =32", and "Dim2=32" are all equivalent.
- Value strings *do not* have to be padded out to 11 characters.

ndim = *value* (required)

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

dim1 = *value* (required)
dim2 = *value* (required, depending on total number of dimensions)
dim3 = *value* (required, depending on total number of dimensions)
...

The dimension size of each axis (the array bound for each dimension of the computational array). The number of **dim***x* entries must match the value of **ndim**. For instance, if you specify a 3D field (**ndim**=3), you must specify the length of the X dimension (**dim1**), the length of the Y dimension (**dim2**), and the length of the Z dimension (**dim3**).

Note that counting is 1-based, not 0-based.

nspace = *value* (required)

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

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

veclen = *value* (required)

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

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

The primitive data type of all the data values. It is possible to specify "data=xdr_integer | xdr_float | xdr_double" in data parsing input mode as well as native field input mode. However, it will only work correctly in the case where the original binary file is in 32-bit big-endian format. The reverse case will not work.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The individual parameters are interpreted as follows:

n — An integer value that specifies which element of a data vector or which coordinate (1 for x, 2 for y, 3 for z, etc.) the subsequent read instructions apply to. **n** does not default to 1 and must be specified.

file = *filespec*

The name of the file containing the data or coordinates. The *filespec* can be an absolute full pathname to a file, or it can be a *filespec* relative to the directory that contains the field ASCII header. For example, an absolute pathname might be */home/myuserid/experiment/data1*. **Note:** the *$AVS_PATH* environment variable is not recognized nor interpreted correctly. You must use a full absolute pathname.

In a relative pathname specification, if the ASCII file of field parsing instructions exists in the file */home/myuserid/experiment/readit.fld* and the data and coordinate files are in the subdirectory */home/myuserid/experiment/data*, you can name these files as *data/xyzs* and *data/values*. The advantage of this second approach is that you can move the directories containing your data around without having to change the contents of

the ASCII parsing instruction file.

filetype = **ascii**
filetype = **unformatted**
filetype = **binary**

ascii means that the data or coordinate information is in an ASCII file. In ASCII files, float data can be specified in either real (0.1) or scientific notation (1.00000e-01) format interchangeably.

unformatted means that the data or coordinate information is in a file that was written as Fortran unformatted data. (Fortran unformatted data is binary data with additional words written at the beginning and end of each data block stating the number of bytes or words in the data block.). When you are figuring out the **skip** and **stride** values below, you must count the additional words surrounding any header information that must be **skip**ped over; but ignore the size words when reading the actual data. See the example below.

binary means that the file is written in straight binary format. such as that produced by Unix output routines, write and fwrite.

Note the warning on binary compatibility among different hardware platforms earlier on this man page.

In each case, **read field** will use the data type specified in the earlier **data={byte,float,integer,double}** statement when it interprets the file.

skip = *n*

For **ascii** files, **skip** specifies the number of *lines* to skip over before starting to read the data. Lines are demarked by newline characters.

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

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

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

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

skip defaults to 0.

offset = *m*

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

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

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

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

stride defaults to 1.

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

Here are some **skip**, **offset**, and **stride** examples for ASCII data. "A's" are vector component 1; "B's" are vector component 2. There are more examples at the end of this manual page.

ASCII file organization 1:

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

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

ASCII file organization 2:

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

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

ASCII file organization 3:

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

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

ASCII file organization 4:

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

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

EXAMPLE 4

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

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

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

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

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

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

```
1 1002.707
2 1443.971
  ...

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

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

```
# AVS Field File
#
# Termite Decay after Five Years
#
  ndim=3           # number of dimensions in the field
  dim1=25          # dimension of axis 1
  dim2 =10         # dimension of axis 2
  dim3 =5          # dimension of axis 3
  nspace=3         # number of physical coordinates
  veclen=1         # number of elements at each point
```

```
  data=float       # data type (byte, integer, float, double)
  field=irregular  # field type (uniform, rectilinear, irregular)
coord 1 file = where.coord filetype=ascii offset = 3 stride = 7
coord 2 file = where.coord filetype=ascii offset = 4 stride = 7
coord 3 file = where.coord filetype=ascii offset = 5 stride = 7
variable 1 file = decay.dat filetype=ascii offset =1 stride = 2
```

In this example, the ASCII description file *decay.fld* is in the same directory as the *where.coord* and *decay.dat* files. If it were in a different directory, you could either give a pathname relative to *decay.fld*'s position, (e.g., *../data/where.coord* or *data/decay.dat*, etc.), or an absolute pathname to the files.

EXAMPLE 5

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

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

EXAMPLE 6

This ASCII description file specifies how to use **read field** to convert the image data in *$AVS_PATH/data/image/mandrill.x* into an AVS field. The first two words in *mandrill.x* are 32-bit integers that specify the horizontal and vertical dimensions of the image. This information must be skipped over — you must supply it in the ASCII description file. Thereafter, *mandrill.x* is a succssion of 32-bit straight binary words, one word per pixel. However, in AVS, each of these words is considered to be a vector of 4 bytes. The first byte is the "alpha" (or "transparency") value for the pixel, and the second through fourth bytes are the red, green, and blue values for each pixel. Thus, this whole file is treated as a series of binary bytes. Note that, because the *$AVS_PATH* construct is not recognized, the example uses a full absolute pathname of */usr/avs/...* to find the file.

```
# AVS field file
#
ndim = 2                        # number of dimensions in the field
nspace=2                        # number of physical coordinates
dim1=500                        # dimension of axis 1
dim2=480                        # dimension of axis 2
veclen=4                        # number of components at each point
data=byte                       # data type (byte, integer, float, double)
```

```
field=uniform                   # field type (uniform, rectilinear, irregular)
label = alpha, red, green, blue     # labels the vector elements
variable 1 file=/usr/avs/data/image/mandrill.x filetype=binary skip=8 stride=4
variable 2 file=/usr/avs/data/image/mandrill.x filetype=binary skip=9 stride=4
variable 3 file=/usr/avs/data/image/mandrill.x filetype=binary skip=10 stride=4
variable 4 file=/usr/avs/data/image/mandrill.x filetype=binary skip=11 stride=4
```

EXAMPLE 7

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

```
# AVS field file
# to read an Arc 3D FORTRAN unformatted file that's 34x34x34
ndim = 3
dim1 = 34
dim2 = 34
dim3 = 34
nspace = 3
veclen = 6
data = float
field = irregular
#
coord 1 file=for003.dat filetype=unformatted skip=20 stride=1
coord 2 file=for003.dat filetype=unformatted skip=157236 stride=1
coord 3 file=for003.dat filetype=unformatted skip=314452 stride=1
#
variable 1 file=for004.dat filetype=unformatted skip=32 stride=1
variable 2 file=for004.dat filetype=unformatted skip=157248 stride=1
variable 3 file=for004.dat filetype=unformatted skip=314464 stride=1
variable 4 file=for004.dat filetype=unformatted skip=471680 stride=1
variable 5 file=for004.dat filetype=unformatted skip=628896 stride=1
variable 6 file=for004.dat filetype=unformatted skip=786112 stride=1
```

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

RELATED MODULES

The **file descriptor** module can also be used to import data into AVS. It has some additional capabilities such as the ability to read 16-bit halfword data, to read some

parsing information (such as the dimensions of the data) directly from the data file itself, and to use variables and expressions for skips, offsets, and strides. The **data dictionary** modules can use the data forms that **file descriptor** constructs to repeatedly read external format data.

The **write field** module will take the AVS field produced by **read field** and write it to disk as a permanent AVS field file. The **read field** module can then read the data much more quickly whenever you need to use it.

The **print field** module displays the ASCII header and contents of an AVS field interactively on the screen. Connect it to **read field**'s output port while experimenting with ASCII description files to verify that the data is being read correctly.

ERROR CHECKING

read field performs a significant amount of error checking. If an error is detected while reading the field, an error dialog box appears on the screen, indicating the line in which the error occurred (if it was in the ASCII header), along with the type of error.

SEE ALSO

The example scripts PRINT FIELD, CONTRAST, FIELD MATH, as well as others demonstrate the **read field** module.

NAME

read ucd – read UCD structure from a disk file

SUMMARY

Name	read ucd	
Availability	UCD module library	
Type	data	
Inputs	none	
Outputs	ucd structure	
Parameters	*Name*	*Type*
	Read UCD	browser

DESCRIPTION

read ucd reads a UCD structure from a file, which must have a *.inp* suffix. The file may be ASCII or binary. The cell connectivity list is calculated automatically.

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

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

PARAMETERS

Read UCD A file browser window to specify the name of the UCD file to be read. Files must have a *.inp* suffix or they will not appear in the browser.

OUTPUTS

UCD structure

The output structure is in AVS unstructured cell data format.

ASCII FILE FORMAT

If a UCD file is in ASCII, it has the following format. For a more complete description of UCD file formats, as well as a discussion of UCD data in general, see the "Unstructured Cell Data" appendix of the *AVS Developer's Guide*.

Comments, if present, must precede all data in the file—comments within the data will cause read errors. The general order of the data is:

1. Numbers defining the overall structure, including the number of nodes, the number of cells, and the length of the vector of data associated with the nodes, cells, and the model.
2. For each node, its node-id and the coordinates of that node in space. Node-ids must be integers, but any number including non-sequential numbers can be used. Mid-edge nodes are treated like any other node.
3. For each cell: its cell-id, material, type (hex, prism, pyr, tet, quad, tri, line, pt), and the list of node-ids that correspond to each of the cell's verticies. (The UCD appendix shows the order in which cell verticies are numbered.)
4. For the data vector associated with nodes, how many components that vector is divided into (e.g., a vector of 5 floating point numbers may be treated as 3 components: a scalar, a vector of 3, and another scalar, which would be specified as 3 1 3 1).
5. For each node data component, a component label/unit label pair, separated by a comma.

6. For each node, the vector of data values associated with it.
7. That is the end of the node definitions. Cell-based data descriptions, if present, then follow in the same order and format as items 4, 5, and 6.
8. The single model-based data descriptions, if present, comes last.

The input file cannot contain blank lines or lines with leading blanks. The numbers down the left correspond to the above descriptions and are not part of the ASCII file.

```
   # <comment 1>
     .
     .
     .
   # <comment n>
1. <num_nodes> <num_cells> <num_ndata> <num_cdata> <num_mdata>
2. <node_id 1> <x> <y> <z>
   <node_id 2> <x> <y> <z>
     .
     .
     .
   <node_id num_nodes> <x> <y> <z>
3. <cell_id 1> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
   <cell_id 2> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
     .
     .
     .
   <cell_id num_cells> <mat_id> <cell_type> <cell_vert 1> ... <cell_vert n>
```

Note: valid strings for <cell-type> are: pt, line, tri, quad, tet, pyr, prism, and hex.

```
4. <num_comp for node data> <size comp 1> <size comp 2>...<size comp n>
5. <node_comp_label 1> , <units_label 1>
   <node_comp_label 2> , <units_label 2>
     .
     .
     .
   <node_comp_label num_comp> , <units_label num_comp>
6. <node_id 1> <node_data 1> ... <node_data num_ndata>
   <node_id 2> <node_data 1> ... <node_data num_ndata>
     .
     .
     .
   <node_id num_nodes> <node_data 1> ... <node_data num_ndata>
7. <num_comp for cell's data> <size comp 1> <size comp 2>...<size comp n>
   <cell-component-label 1> , <units-label 1>
   <cell-component-label 2> , <units-label 2>
     .
     .
     .
   <cell-component-label n> , <units-label n>
   <cell-id 1> <cell-data 1> ... <cell-data num_cdata>
   <cell-id 2> <cell-data 1> ... <cell-data num_cdata>
     .
```

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

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

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

SAMPLE UCD FILE

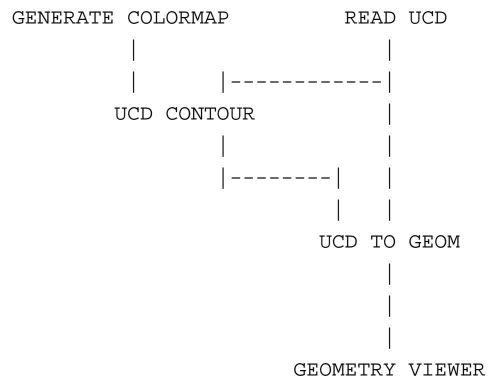
The following is an example of a simple UCD file. This UCD structure has 8 nodes in 1 hexahedral cell. Associated with each node is a single scalar data value, making up one component that this person labels "stress," and specifies a "lb/in**2" unit label. There is no cell- or model-based data. See the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

```
8 1 1 0 0                    1.  8 nodes, 1 cell, 1 component of node data
1  0.000   0.000   1.000     2.  for each node, its id and node coordinates
2  1.000   0.000   1.000
3  1.000   1.000   1.000
4  0.000   1.000   1.000
5  0.000   0.000   0.000
6  1.000   0.000   0.000
7  1.000   1.000   0.000
8  0.000   1.000   0.000
1  1 hex 1 2 3 4 5 6 7 8     3.  cell id, material id, cell type, cell vertices
1 1                          4.  num data components, size of each component
stress, lb/in**2             5.  component label, units label
1  4999.9999                 6.  data vector for each node
2  18749.9999
3  37500.0000
4  56250.0000
5  74999.9999
6  93750.0001
7  107500.0003
8  5000.0001
```

EXAMPLE

The following network reads in a UCD ASCII file (*.inp* suffix), and displays it:

read ucd



RELATED MODULES

Modules that can process **read ucd**'s output:

ucd to geom, ucd crop, ucd threshold, ucd extract, ucd hex to tet, ucd anno, ucd contour, ucd hog, ucd iso, ucd offset, ucd rslice, ucd slice2d, ucd legend, ucd probe, ucd streamline, write ucd, ucd tracer.

SEE ALSO

The example script READ UCD demonstrates the **read ucd** module.