4 File Formats

This chapter describes all the file formats currently supported by SciAn, giving details on the implementation and any special controls they may use.

Because file formats appear more quickly than manual revisions, SciAn may have additional file formats which are not covered here. To find out about any additional file formats, look on the `ftp` server (`ftp.scri.fsu.edu`) in the `pub/SciAn/technotes` directory.

File readers

As described in the user's manual, file readers are SciAn objects whose purpose it is to read datasets from files. A file reader typically reads one type of file which can be identified by its file name extension, the portion of the file name which is after the last period in the name.

All file readers have control panels. To see a control panel, choose Show File Readers from the File menu. A window will come up showing icons for all the file readers. Select the reader of interest and press the Show Controls button.

All file readers have a text box in their control panel to change the extension which indicates this particular file type. To change it, enter the new extension in the text box and press the Enter key.

Depending on the file type, file readers can read several datasets from one file, or different timesteps of a dataset from several files. Some file readers are inherently able to read time-dependent data. For others, it is usually possible to read time-dependent data using a naming convention. Simply keep the timesteps as separate datasets, and name them each with a string of the form *name*@*time*, where *name* is the name of the time-dependent dataset, and *time* is the time of this particular time step. The time can be a real number, in which case it is interpreted as a real number of seconds since midnight, or it can be a 24-hour clock time (e.g. 12:03). A check box in the file reader's control panel selects whether this feature is enabled or not.

All control panels also have a Save All Settings button. Pressing this button saves all the settings of the file reader so that they are read the next time SciAn is started. Settings are saved in a directory called `.scianSettings` in your home directory. If the directory does not exist, it will be created automatically. Each setting file is a fragment of a script which gives a snapshot of the important variables in the file reader.

Additional controls for particular file readers are described in the following sections.

G90 (Gaussian 90) format

This information will be supplied Really, Really Soon Now.

HDF (Hierarchical Data Format)

The Hierarchical Data Format (HDF) is a file format for storing self-descriptive datasets which was developed at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign. SciAn uses the Scientific Data Set (SDS) capability of HDF to store multidimensional data over rectilinear Cartesian grids. Extensions are included for time-dependent data, vector data, and missing data points. [We're working on VSET for non-structured grids, too. -EMP]

HDF files are written and read using a library, which must be compiled and linked into SciAn as well as into any program which writes HDF files. There are several versions available for a variety of machines.

Obtaining HDF

This information is provided for your information only. We didn't write HDF and aren't responsible for maintaining it. However, we think it's great and are grateful to NCSA for making such a useful tool available to the research community.

The HDF library and documentation are available directly from NCSA via anonymous `ftp` from `ftp.ncsa.uiuc.edu`.

Once it has been downloaded, it must be compiled. Refer to the NCSA documentation for information on compiling it. When compiled, you will have a library called `libdf.a`, which must be linked into SciAn and into any programs which are to write HDF files. Copy this library to the `/usr/local/bin` directory.

To recompile SciAn with the HDF library do `make INSTALL`, which will ask you whether you want to install the HDF library. Then recompile SciAn by doing `make`.

Note *The SciAn makefile looks for libraries in* `/lib`, `/usr/lib`, *and* `/usr/local/lib`. *If for some reason you cannot put the HDF library in one of these directories or you cannot call it* `libdf.a`, *you should be able to hack SciAn to be able to find them anyway. Look in* `lfiles.*.make`.

Writing HDF files

Detailed information on writing HDF SDS files is provided in the documentation from NCSA. As SciAn does not use all the facilities of HDF and also contains extensions, this section briefly explains the process of writing an HDF file, describing in particular extensions and restrictions on the way SciAn handles HDF files. All routines that begin with `DFSD` mentioned here are HDF routines, and you can get more information about them by looking in the HDF documentation.

The HDF SDS format supported by SciAn can express scalar and vector fields defined over rectilinear Cartesian grids with one or more dimensions. It can deal with missing data points and time dependency.

Before you write an HDF file, you must have this information:

- The name you want the dataset to have
- The number of topological dimensions of the dataset
- For each topological dimension, the name of the axis
- For each topological dimension, the number of steps in the dimension
- For each topological dimension, the position of each step in the dimension
- If the dataset is time-dependent, the time values for every time step. This can be expressed in time step numbers, seconds, or clock time (e.g. 12:37).
- The minimum and maximum values in the data. This is required if there are missing data points. It is optional otherwise, but it's still a good idea to include. The minimum and maximum should bracket all interesting portions of the field comfortably.

The data label as given in the `DFSDsetdatastrs` call is used to give the name of the dataset. If no `DFSDsetdatastrs` call was issued, the file name is used instead.

The name of each axis is specified in the `DFSDsetdimstrs` call. The axis names will only be read if the Read Axis Names check box is in the HDF control panel is on. See "HDF controls" below for more information. The name of the first axis should be `X`, the second `Y`, and the third `Z`. If you wish to use other axis names, you will have to create a file named `.scianAxes` in your home directory. This file must contain a series of text lines, each of the form

newAxis = *oldAxis*

where *newAxis* is the axis name to define and *oldAxis* is the axis to which it corresponds. Upper and lower case differences are ignored when matching to an axis name in a file. Here is an example `.scianAxes` file which defines `Latitude` and `Longitude` as axes corresponding to `X` and `Y` in a flat Mercator projection:

```
#Axes file containing supplemental axes to SciAn

Longitude = X
Latitude = Y
```

The dimension scale as specified in `DFSDsetdimscale` is used to determine the spacing along each of the axes of the grid. The spacing is always assumed to be in a Cartesian coordinate system. If no `DFSDsetdimscale` call was issued, a uniform scale is guessed from the dimensions.

Extensions to HDF

The *name@time* syntax can be used in the dataset name to specify a time slice. The dataset name is given in the `DFSDsetdatastrs` function call. Instead of the name of the dataset, use a string of the form *name@time*. For example, a dataset named `Q` at 317 seconds may be called `Q@317`. SciAn will assemble all the time steps into a single time-dependent dataset when the timesteps are read in.

To encode missing data in the file, use `DFSDsetrange` to set a reasonable maximum and minimum for the data. Choose a single data value far outside the range to be the missing data value. A group of radio buttons in the file reader control panel specifies what to do with data points that fall outside the range. Data outside the range can be used as it is, clipped to the maximum and minimum, or treated as missing data within SciAn. For more information, see “HDF controls” below.

Warning `DFSDsetrange` *replaces an earlier HDF command,* `DFSDsetmaxmin`*. Either will work, depending on which version of HDF you have installed, but beware! The C usage is different.* `DFSDsetmaxmin` *took the values as arguments, but* `DFSDsetrange` *takes the addresses of the variables.*

If the first dimension of the dataset is 2 or 3, the dataset can be read as vector instead of scalar data. The 2 or 3 elements over the first dimension are used to form the X and Y or X, Y, and Z components of the vector. For example, say you had a 2-D vector field that was defined over a grid 20 by 30 units wide. Define an array to hold the vector like this:

```
float vector[20][30][2];
```

in C or

```
REAL VECTOR(20, 30, 2)
```

in FORTRAN. Take as an example the vector at location (14, 22) in the array. Then in C, `vector[14][22][0]` holds the X component of the vector, and `vector[14][22][1]` holds the Y component of the vector. In FORTRAN, `VECTOR(14, 22, 1)` holds the X component and `VECTOR(14, 22, 2)` holds the Y component. When writing the dataset to HDF, assume it is a dataset with three dimensions: 14, 22, and 2. SciAn will decode the information into a vector dataset when it is read in.

This feature works whether the vector dimension is the first or last dimension. In this example, it works with `float vector[2][20][30];` as well.

This feature can be turned on or off using a check box in the HDF file reader control panel, as described in the next section.

HDF controls

Bring up the File Readers window by choosing Show File Readers from the File menu. You will see a control panel like Figure 4-1.

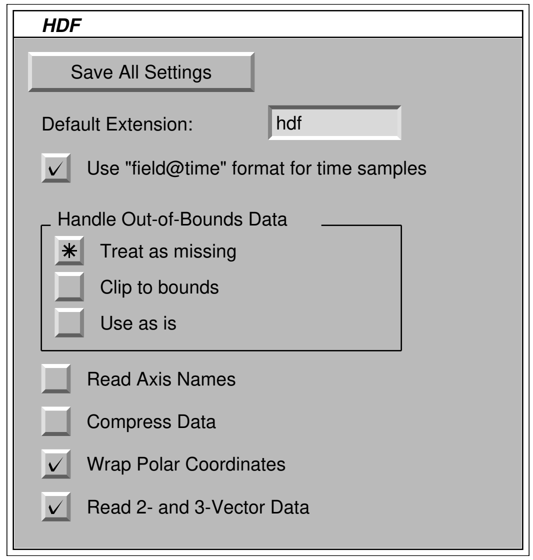

Figure 4-1. *The HDF Control Panel*

The top three items are common to all file readers. At the top is a Save All Settings button. This button allows you to save all the settings for this control panel in a file in the `.scianSettings` directory in your home directory. The Default Extension text box gives the file name extension (the portion of the file name after the dot) that SciAn uses to recognize a file as an HDF file. Change it and press the Enter key to specify a new extension. Do not include the dot when typing the extension. Below the text box is a check box which allows the *name*@*time* format to be used to specify time samples. Remember that this name is the dataset name as given by the `DFSDsetdatastrs` function, not the name of the file.

Below these are controls specific to the HDF file reader. The Handle Out-of-Bounds Data radio buttons control how data points outside the range given by `DFSDsetrange` are treated. If Treat as missing is on, any data points outside the range are treated as missing data. If Clip to bounds is on, data points are automatically clipped so that they lie within the bounds. If Use as is is on, data points are used as is. All data points which are outside the bounds will be treated as underflow and overflow data within SciAn.

Below the radio button group is a Read Axis Names check box. If this check box is off, the first axis will always be considered X, the second Y, and the third Z. If it is on, the axis names will be read from the file. The axis names must have been set in the HDF file using the `DFSDsetdimstrs` function. It is important that this check box be set only if the axes have been set in the file. Otherwise, a bug in the HDF code can cause a crash.

Next is a `Compress Data` check box. Normally, datasets use four bytes per data value. When this box is on, datasets will use only one byte per value. The corresponding values will come from a lookup table generated from the minimum and maximum values. This significantly reduces the amount of memory used to store the data, with the disadvantage of reducing the precision somewhat. The reduction of precision can be noticed in visualizations of very smooth datasets. However, for most real-world datasets, it does not matter.

Next is a Wrap Polar Coordinates check box. Normally, polar coordinates are simply stretched out into a plane using the Mercator projection. When the values go through some transitions, say from 359 to 0, there can be discontinuities in the dataset resulting in problems with visualization objects. If this check box is on, polar coordinates are wrapped to change monotonically.

Wrapping of polar coordinates only occurs when the fourth parameter in the `DFSDsetdatastrs` call is `"spherical"` or `"polar"`. It only occurs when the units argument as set in the `DFSDsetdimstrs` call is `"degrees"`, `"radians"`, or `"gradians"`.

At the bottom is a Read 2- and 3-Vector Data check box, which controls whether 2- and 3-vector datasets are read using the dimension convention listed above.

Example code

Following is an example of writing an HDF file.

```
/*exampleHDF.c
  Example program to write an HDF file

  Compile

  cc exampleHDF.c -lm -ldf -o exampleHDF

  Eric Pepke
*/

#include <stdio.h>
#include <math.h>

#define XSIZE 50
#define YSIZE 50
#define ZSIZE 50

/*Define this if you have a newer version of HDF*/
#define NEWHDF

main()
{
    int i, j, k;
    int dimSizes[3];
    float array[XSIZE][YSIZE][ZSIZE];       /*Data array*/
    float xScale[XSIZE];                    /*Scales*/
    float yScale[YSIZE];
    float zScale[ZSIZE];
    float min, max;

    /*Fill array*/
    for (i = 0; i < XSIZE; ++i)
    {
       for (j = 0; j < YSIZE; ++j)
       {
           for (k = 0; k < ZSIZE; ++k)
           {
              array[i][j][k] =  sin(((double) i) / XSIZE * 6.0)
                              + sin(((double) j) / YSIZE * 8.0)
                              + sin(((double) k) / ZSIZE * 10.0);
           }
```

```
        }
    }
    min = -3.0;
    max = 3.0;

    /*Set up the dimension sizes*/
    dimSizes[0] = XSIZE;
    dimSizes[1] = YSIZE;
    dimSizes[2] = ZSIZE;
    DFSDsetdims(3, dimSizes);

    /*Set up the scales along each dimension.  Here they're regularly
      spaced, but they could be irregularly spaced.*/
    for (i = 0; i < XSIZE; ++i) xScale[i] = (float) i;
    for (j = 0; j < YSIZE; ++j) yScale[j] = (float) j;
    for (k = 0; k < ZSIZE; ++k) zScale[k] = (float) k;
    DFSDsetdimscale(1, XSIZE, xScale);
    DFSDsetdimscale(2, YSIZE, yScale);
    DFSDsetdimscale(3, ZSIZE, zScale);

    /*Set up the names of the dimensions.  In the current version of
      SciAn, only the axis name and the units are read, and the units are
      ignored unless the coordinate system is polar.*/
    DFSDsetdimstrs(1, "X", "meters", "E10.2");
    DFSDsetdimstrs(2, "Y", "meters", "E10.2");
    DFSDsetdimstrs(3, "Z", "meters", "E10.2");

    /*Set up the dataset information.  In the current version of
      SciAn, only "field" and "cartesian" are read.  Cartesian sets
      up a normal cartesian coordinate system, polar or spherical does
      a Mercator projection*/
    DFSDsetdatastrs("field", "dimensionless", "E10.2", "cartesian");

    /*Set the max and min.*/
    DFSDsetrange(&max, &min);

    /*Finally, put the data*/
    DFSDputdata("scalartest.hdf", 3, dimSizes, array);
}
```

NFF (Neutral File Format)

The NFF reader reads geometric objects from files in the Neutral File Format (NFF) developed by Eric Haines [1987]. The format has been extended with a **t** token for time dependence and an **o** token to offset subsequent objects. Lighting and viewing tokens are ignored, and object materials are only partially supported.

An NFF file is a text file containing a series of entities. Each entity consists of a token, which appears on a new line, followed by a series of arguments, which may appear on the same line or on subsequent lines, depending on the particular entity. There may be blank lines in the file.

The following sections describe all the entities supported by SciAn.

Comment

Syntax:
*text*

Any line beginning with a hash mark (#) is a comment. Comments should only appear between entities.

Cylinder or conical frustum

Syntax:
c
x1 y1 z1 radius1
x2 y2 z2 radius2

This specifies a cylinder or conical frustum starting at *x1*, *y1*, *z1* and ending at *x2*, *y2*, *z2*. *Radius1* is the radius at the first endpoint, and *radius2* is the radius at the second endpoint. If they are the same, the object is a cylinder. If they are different, the object is a conical frustum. Whether the end caps are shown can be determined by a check box in the Geometry control panel.

Polygon

Syntax:
p *nVertices*
x1 y1 z1
x2 x2 z2
...
xn yn zn

Specifies a polygon with *nVertices* vertices. Following the token line are *nVertices* line giving the vertices of the polygon. The direction of the face of the polygon is determined by the right-hand rule.

Polygonal patch

Syntax:
pp *nVertices*
x1 y1 z1 nx1 ny1 nz1
x2 x2 z2 nx2 ny2 nz2
...
xn yn zn nxn nyn nzn

Specifies a polygon patch with *nVertices* vertices. Following the token line are *nVertices* line giving the vertices of the polygon. The *x*, *y*, and *z* give the position of each vertex, and *nx*, *ny*, and *nz* give the x, y, and z components of the vertex normal.

Sphere

Syntax:
`s` *x y z radius*

Specifies a sphere with its center at *x*, *y*, *z* and radius *radius*.

Time step

Syntax:
`t` *time*

This specifies a time, which will affect all subsequent entities in the file until another `t` entity appears. All entities before the first time step are assumed to be eternal. *Time* must be a single real number, specifying a number of time steps.

The colors for the eternal portion of the geometry are kept together at the beginning of the color panel. SciAn tries to reuse the colors in the portion that is time-dependent. The safest way to make the heuristics work best is to use the same color changes in the same order in each time step.

Fill color

Syntax:
`f` *red green blue Kd Ks Shine T refractionIndex*

Red, *green*, and *blue* give the red, green, and blue diffuse color components of subsequent entities. Values are between 0 and 1. All the other parameters are ignored but must be included on the line. It is easiest to make them all 0.

Colors are stored in the color palette for the geometry object, so that they can be edited later via the palette control panel. Normally, every new `f` entity specifies a new entry in the color table. However, when time dependency is used, this behaves a little differently. See the section on the time step entity for more information.

Origin

Syntax:
`o` *dx dy dz*

Offsets subsequent entities to an origin at *dx*, *dy*, *dz*. This is an easy way to provide a simple translation on subsequent entities. An `o` entity cancels the effect of previous `o` entities. To set no offset use `o 0 0 0`.

Writing NFF files

Because NFF is a line-oriented text format, it is simple to write an NFF file using C or FORTRAN. The following C example does a simple simulation of a bouncing ball and writes the output to a file called `example.nff`.

```
/*exampleNFF.c
  Example program to write an nff file

  Compile

  cc exampleNFF.c -o exampleNFF

  Eric Pepke
*/

#include <stdio.h>

#define FLOORMINX     -2.0   /*Min x of floor*/
#define FLOORMAXX     2.0    /*Max x of floor*/
#define FLOORMINY     -1.0   /*Min y of floor*/
#define FLOORMAXY     1.0    /*Max y of floor*/
#define FLOORZ               0.0    /*Min z of floor*/

#define BALLX         -2.0   /*Starting x of ball*/
#define BALLY         0.0    /*Starting y of ball*/
#define BALLZ         2.0    /*Starting z of ball*/
#define BALLRADIUS    0.1    /*Radius of ball*/

#define BALLVX               1.0    /*X starting velocity of ball*/
#define BALLVY               0.0    /*Y starting velocity of ball*/

#define BOUNCE               0.8    /*Bounce coefficient*/

#define AG            10.0   /*Acceleration due to gravity*/

#define TIMESTEP      0.05   /*Time step*/

void Color(FILE *file, float r, float g, float b)
/*Sets to a color*/
{
    fprintf(file, "f %g %g %g 0 0 0 0 0\n", r, g, b);
}

void DrawFloor(FILE *file)
/*Draws the floor*/
{
    /*Color for floor*/
    Color(file, 0.0, 1.0, 1.0);

    /*Now polygon*/
    fprintf(file, "p 4\n");
    fprintf(file, "%g %g %g\n", FLOORMINX, FLOORMINY, FLOORZ);
    fprintf(file, "%g %g %g\n", FLOORMAXX, FLOORMINY, FLOORZ);
    fprintf(file, "%g %g %g\n", FLOORMAXX, FLOORMAXY, FLOORZ);
    fprintf(file, "%g %g %g\n", FLOORMINX, FLOORMAXY, FLOORZ);
}

void DrawBall(FILE *file, float bx, float by, float bz)
/*Draws the ball*/
{
    fprintf(file, "s %g %g %g %g\n", bx, by, bz, BALLRADIUS);
}

void Time(FILE *file, float t)
/*Emits a time marker*/
{
    fprintf(file, "t %g\n", t);
}
```

```
main()
{
    FILE *file;
    float bx, by, bz, bvx, bvy, bvz, t;

    file = fopen("example.nff", "w");
    if (file)
    {
       /*First draw the floor*/
       DrawFloor(file);

       /*Now do the simulation*/
       bx = BALLX;
       by = BALLY;
       bz = BALLZ;
       bvx = BALLVX;
       bvy = BALLVY;
       bvz = 0.0;
       t = 0.0;

       /*Set the color once for all balls*/
       Color(file, 1.0, 0.0, 0.0);

       do
       {
           /*Set the time step.  Omit this line to keep all snapshots
             in one time step to see them all at once*/
           Time(file, t);

           /*Emit the current ball position*/
           DrawBall(file, bx, by, bz);

           /*Go to next time step.  I know this is really cheesy
             integration, but it's just a demo.*/

           t += TIMESTEP;
           bvz -= TIMESTEP * AG;

           bx += bvx * TIMESTEP;
           by += bvy * TIMESTEP;
           bz += bvz * TIMESTEP;

           if (bz < (FLOORZ + BALLRADIUS))
           {
              /*Bounce*/
              bz = FLOORZ - bz + 2.0 * BALLRADIUS;
              bvz = -bvz * BOUNCE;
           }

       } while (bx >= FLOORMINX && bx <= FLOORMAXX &&
                by >= FLOORMINY && by <= FLOORMAXY);


       fclose(file);
    }
    else
    {
       perror("example.nff");
    }
}
```

PLOT3D format

The PLOT3D file reader reads files in the format of the PLOT3D fluid dynamics visualization program [Buning et al., 1990, Walatka, et al., 1991]. This section describes how the file reader works and what subset of PLOT3D files it reads. Some familiarity with PLOT3D commands and data files is assumed.

Because PLOT3D includes file formats written by FORTRAN, SciAn must be compiled using FORTRAN in order for the entire file reader to work. During the installation process (`make INSTALL`) the installer will detect whether a FORTRAN compiler is present on your system. If it is not present, then only the ASCII formats will be readable.

The PLOT3D reader supports files with multiple grids. However, as there is currently no way within SciAn of handling multiple grids as a single dataset [yet! -EMP], the individual segments are treated as separate datasets.

The PLOT3D metafile

When reading files from within PLOT3D, one normally types in a series of commands which assemble the file. This is not the way files are read from within SciAn, so there is a metafile mechanism to read the files.

A metafile is a short ASCII script which contains PLOT3D read commands. You will need to make one metafile for every group of PLOT3D files you wish to read as a single unit. The filename extension of the metafile should be `.p3d`. The lines in the metafile will refer to other files that contain the actual data, and these can be named what you like.

Each line of a metafile is a single PLOT3D `READ` command. A metafile may contain any number of `READ` commands but should at least contain enough to read a single grid (XYZ) file and at least one Q or FUNCTION file. Each read command is of the form:

`READ`/*qualifier1*/*qualifier2...*

The main purpose of qualifiers is to specify the path of data files, data format and operations to be done on the data. To avoid ambiguity, qualifiers describing a file must precede the qualifier that specifies the file path name. At most one qualifier in its group may be used for any data file.

If certain qualifier is not specified for a file, then it inherits the qualifier of the previous file. If this happens to the first file, then the default is assumed.

It is illegal to specify inconsistent qualifiers for a Q or FUNCTION file and its corresponding XYZ file. For example, if a Q file is described as 3D then it is an error to have its corresponding XYZ file qualified as 1D or 2D. It is suggested that all the Q or FUNCTION files using the same XYZ file as grid be placed in the same READ command and the qualifiers common to the data files be placed only once at the beginning of the READ command.

There must be exactly one XYZ file specification in a single READ command. If you need to use another XYZ file, use a separate READ command. There may be any number of Q or FUNCTION files that correspond to the one XYZ file. The Q, FUNCTION, and XYZ files may appear in any order.

If a READ command needs to be separated into two or more lines, a hyphen '-' must follow the end of the previous line to indicate the continuation of the command on the next line.

The reader is not sensitive to case (except in file names) and recognizes abbreviations for each qualifier.

PLOT3D reference

The following is a list of all the qualifiers recognized by the SciAn PLOT3D file reader:

`/1D`
`/2D`
`/3D`

One of these qualifiers specifies the number of topological and spatial dimensions of the dataset. If none is given, `/3D` is assumed.

`/XYZ="`*fileName*`"`

This asks the `READ` command to read an XYZ (grid) file from the file named *fileName*. You can use a complete pathname, or a single name relative to the current directory. The file name must be enclosed in double quotes.

`/Q="`*fileName*`"`

This asks the `READ` command to read a Q file from the file named *fileName*. You can use a complete pathname, or a single name relative to the current directory. The file name must be enclosed in double quotes.

`/FUNCTION="`*fileName*`"`

This asks the `READ` command to read a FUNCTION file from the file named *fileName*. A FUNCTION file contains a single scalar or vector dataset, possibly defined over multiple grids. You can use a complete pathname, or a single name relative to the current directory. The file name must be enclosed in double quotes.

`/MDATASET`

This specifies that there are multiple datasets in a single file. The default assumption is that there is only one dataset per file.

`/MGRID`

This specifies that there are multiple grids in a single file. The default assumption is that there is only one grid per file.

```
/FORMATTED
/UNFORMATTED
/BINARY
```

This specifies the format of subsequent files. `/FORMATTED` is an ASCII file format and is optimal for transportability across machines. It is also the only supported format if the machine on which you compiled SciAn did not have a FORTRAN compiler. `/UNFORMATTED` is a more compact binary file format which typically cannot be transported across machines. `/UNFORMATTED` is the default. `/BINARY` is not supported.

```
/PLANES
/WHOLE
```

This specifies the arrangement of data in subsequent files. `/WHOLE` is the default and specifies that the datasets are stored as a single unit. If `/PLANES` is specified, individual planes across the last dimension are stored subsequently in the file.

```
/CHECK
/NOCHECK
```

This specifies whether Q files should be checked for zero or negative density or pressure while they are being read in. By default, they are checked, and negative values are raised to zero. If there are any zero or negative values, a message will be printed during the reading of the file.

```
/JACOBIAN
/NOJACOBIAN
```

This specifies whether Q variables are scaled by the determinant of the metric Jacobian. It is not implemented.

```
/BLANK
/NOBLANK
```

This determines whether subsequent XYZ files contain a fourth array, an integer IBLANK array, to specify blank locations on the grid. Data which corresponds to an IBLANK of 0 is treated as missing in SciAn.

Example

Here is an example of a PLOT3D metafile:

```
READ/UNFORMATTED/MDATASET/MGRID/XYZ="/user/hwu/xyz.dat"-
    /q="/user/hwu/q.dat"-
    /function="/user/hwu/function.dat"
```

This read command specifies a Q dataset and a FUNCTION dataset with both of them using the same XYZ grid. For attributes not specified, defaults are assumed. These include `/3D`, `/WHOLE`, `/CHECK`, `/NOJACOBIAN`, and `/NOBLANK`.

PDB (Protein Data Bank) format

The PDB (**P**rotein **D**ata **B**ank) file reader reads a subset of the information in files in the Protein Data Bank at Brookhaven National Laboratories [Brookhaven 1992]. It is only partially implemented. We are still working on visualization of chemistry data sets and are still trying to figure out the best way to represent the information internally. For now, the PDB file reader allows the atoms to be read and displayed using the Balls visualization object as colored, sized balls. We are working on ball and stick and ribbon visualization objects, but they are not yet complete. One of the problems that we have is that we are not chemistry experts and find some of the terminology and ideas difficult. If you have expertise in chemistry, we may be interested in collaborating with you.

The default file name extension for Protein Data Bank files is `.ent`, to correspond to the data bank files as distributed by Brookhaven.

A Protein Data Bank file is a series of text lines which resemble the contents of a deck of Hollerith punched cards. Each text contains a single record which gives information on the protein. At the beginning of each line is a six character code which determines what kind of record this is.

Record summary

The following is a listing of all the records recognized by the SciAn protein data bank file reader. All other record types are quietly ignored. Column numbers start at 1 for the first column on the very left of the line.

Record Type	Columns	Value
`ATOM`	7-11	Atom serial number
	13-16	Atom name
	17	Alternate location indicator (ignored)
	18-20	Residue name (ignored)
	22	Chain identifier (ignored)
	23-26	Residue sequence number
	27	Code for insertions of residues (ignored)
	31-38	X orthogonal coordinate in Angstroms
	39-46	Y orthogonal coordinate in Angstroms
	47-54	Z orthogonal coordinate in Angstroms
	55-60	Occupancy
	61-66	Temperature factor
	68-70	Footnote number (ignored)

The atom name is parsed to find the name of the atom and the remoteness indicator. Parsing is done according to the rules in Appendix B of [Brookhaven 1992] with additional heuristics to parse other formats that have been seen in the Protein Data Bank. The name of the atom is used to find the atomic number. The X, Y, and Z values, the atomic number, the occupancy and the temperature factor are used to construct datasets (see “Dataset construction” later in this section). The residue sequence number and remoteness indicator are used to construct residue information, but there is currently no way to visualize residues.

Record Type	Columns	Value
AUTHOR	9-10	Continuation field (ignored)
	11-70	Names of contributors

All the authors in the file are collected and are available in the control panels of all the datasets created from the file.

Record Type	Columns	Value
COMPND	9-10	Continuation field
	11-70	Name of macromolecule

The first COMPND line is used to give the name of the compound. The name is shortened by using only the portion up to the first open parenthesis and is used as part of the name of all datasets produced by the file.

Record Type	Columns	Value
CONECT	7-11	Serial number
	12-16	Covalent bond
	17-21	Covalent bond
	22-26	Covalent bond
	27-31	Covalent bond
	32-36	Hydrogen bond where this atom is an acceptor
	37-41	Hydrogen bond where this atom is an acceptor
	42-46	Salt bridge where this atom has excess + charge
	47-51	Hydrogen bond where this atom is a donor
	52-56	Hydrogen bond where this atom is a donor
	57-61	Salt bridge where this atom has excess – charge

All the CONECT records are used to define all the bonds between atoms. No distinction is made between the kinds of bonds. There is currently no way to visualize bonds, and there will not be until the ball and stick visualization object is complete.

Record Type	Columns	Value
EXPDTA	11-70	Experimental technique

All the EXPDTA lines in the file are collected and are available in the control panels of all the datasets created from the file.

Record Type	Columns	Value
HETATM	7-11	Atom serial number
	13-16	Atom name
	17	Alternate location indicator (ignored)
	18-20	Residue name (ignored)
	22	Chain identifier (ignored)
	23-26	Residue sequence number
	27	Code for insertions of residues (ignored)
	31-38	X orthogonal coordinate in Angstroms
	39-46	Y orthogonal coordinate in Angstroms
	47-54	Z orthogonal coordinate in Angstroms
	55-60	Occupancy
	61-66	Temperature factor
	68-70	Footnote number (ignored)

The HETATM record is used exactly the same way as the ATOM record.

Record Type	Columns	Value
JRNL	11-70	Literature citation

All the JRNL lines in the file are collected and are available in the control panels of all the datasets created from the file. At some point we may make this more intelligent, so that it produces nicely formatted journal references.

Record Type	Columns	Value
REMARK	8-10	Remark number (ignored)
	12-70	Remark text

All the remarks in the file are collected and are available in the control panels of all the datasets created from the file.

Record Type	Columns	Value
SOURCE	9-10	Continuation field (ignored)
	11-70	Text describing source of macromolecule

All the SOURCE lines in the file are collected and are available in the control panels of all the datasets created from the file.

Dataset construction

The information in the file is used to construct a number of datasets.

First, the orthogonal coordinates of all the atoms are used to construct a single 1-dimensional unstructured grid. The grid points correspond to serial numbers of atoms. If an atom is missing, then its grid point is declared as missing data.

If there are no CONECT records in the file, then the bonds are guessed simply by considering any two atoms closer than 1.7 Angstroms to be bonded together. The bond information is stored as the links (or 1-edges) information in the unstructured grid.

Four numerical datasets that share the common grid are produced from each file, each named by the compound name concatenated with another string which describes the dataset. The Atoms dataset contains atomic numbers that correspond with the grid points. It is colored with a color palette that maps atomic numbers onto atom colors. The Radii dataset contains atomic radii for all the atoms. It is set up so that, when the Atoms dataset is visualized using the Balls visualization object, the Radii dataset is used to determine the size of the balls. The Temp Factor dataset contains the temperature factor. The Occupancy dataset contains the occupancy for all the atoms.

Atom parameters

Atom names, weights, atomic radii, and colors are contained in the file `ScianPeriodicTable.h`, which is compiled into SciAn. The names and weights were copied by hand from a printed periodic table. The radii are taken from the CRC manual [Weast 1989]. Some period 0 elements and actinides were missing from the manual. The former were given the same radius as neon, and the latter were given the same radius at plutonium. These radii tend to be a bit on the large side when looking at organic molecules. It is easy to reduce the scaling factor for size from within the Size control panel in the Balls visualization object.

STF (Simple Text Format)

STF (**S**imple **T**ext **F**ormat) is an easy way to write scalar and vector datasets defined over regular or curvilinear grids. Because it is a text format it is not as compact as formats such as HDF, but it can be used without a library, and it can do curvilinear grids, which HDF cannot. By default, STF files are recognized by the `.stf` extension.

An STF file is a text file containing statements separated by newlines. Each statement begins with a single word token at the beginning of the line which gives the purpose of the statement. The statement may have arguments, which appear after the statement token. For most statements, all the arguments must appear on the same line as the statement. Exceptions are noted later. Normally, all of the statements in a single file make up a single dataset or timestep of a dataset. However, you can use the END statement to separate several datasets or timesteps within a single file.

Any line that begins with a hash mark (#) is assumed to be a comment and is ignored.

Background concepts

In order to understand how the STF reader works, it is important to understand some background concepts about grids and datasets. Most of this information is described in the user's manual, but it is summarized here.

The kind of datasets that STF can read are made up of data defined over grids. The grid gives the location of the data points and connectivity between them.

For the purposes of STF, there are two types of grids: regular and curvilinear. In a regular grid, all the cells are the same shape, and the axes of the grid are aligned with the principal axes of the Cartesian space. In a curvilinear grid, each cell can be a different shape, and the axes in a grid can bend in space any way desired.

Grids (or lattices) have two kinds of dimensions: topological and spatial. Some people call topological dimensions "computational" dimensions instead. The topological dimensionality of a grid is given by how many numbers are needed to represent a unique point on the grid. The spatial dimensionality of a grid is given by how many numbers are needed to represent the position of the points in space.

In regular grids, the topological dimensionality is always the same as the spatial dimensionality. In curvilinear grids, it need not be. For example, a grid over a flat sheet of rubber which locates points on the surface of the paper has two topological dimensions and two spatial dimensions. However, if you wrapped that sheet of rubber around a sphere to locate points on the surface of the sphere, you would have a grid still with two topological dimensions but three spatial dimensions.

The data defined over the grid can be scalar or vector. For scalar datasets, such as temperature over an area of terrain, there is just a single real number at each grid point. For vector datasets, such as wind velocity within a cell of air, there are several numbers at each grid point which give components of the vector along the principal Cartesian axes. Individual data points can be missing.

The process of writing an STF file consists of constructing a series of statements which together comprise a dataset. First, the header of the dataset is given. Next, the dataset is defined. Then, additional optional information for curvilinear grids may be added. Finally, the dataset is completed with an `END` statement. (If the dataset is the last one in the file, the `END` statement is not needed.)

Writing the header

The first thing to do in the file is to write the name of the dataset. Do this using a statement of the form

`NAME` *name*

where *name* is the name of the dataset. If the dataset is time dependent, put the timestep here using a statement of the form

`TIME` *time*

where *time* is the time of this time step. If this is a static dataset, leave this out.

Next, specify the rank of the grid using a statement of the form

`RANK` *nDimensions*

where *nDimensions* is the number of topological dimensions. Then give the dimensions themselves, with a statement of the form

`DIMENSIONS` *iDim jDim ...*

where *iDim*, *jDim*, *kDim*, etc. are the first, second, third, etc. topological dimensions. For example, a 20 by 30 dataset would have `RANK 2` and `DIMENSIONS 20 30`.

Next, set the bounds of the grid. The bounds consist of the minimum and maximum **spatial** bounds of the grid. Set the bounds with a statement of the form

`BOUNDS` *xMin xMax yMin yMax ...*

If the grid is a regularly spaced grid, this is all that you have to do to define the grid. Each topological dimension is assumed to correspond to a spatial dimension, and the grid is assumed to be regularly spaced within the bounds.

If you really want a curvilinear or irregular grid, you will need to define the grid later. (See "Writing the grid," below.) In this case, the `BOUNDS` statement is optional.

Writing the data

The individual data points must be ordered linearly in the file, one after the other. Before writing the data, you must specify how they will be ordered. Do this using one of two statements:

```
ORDER COLUMN
ORDER ROW
```

The difference between column and row ordering is which index moves the fastest when stepping through the data points. If the first index moves the fastest, it is column ordering. If the last index moves the fastest, it is row ordering.

If you are writing the loops explicitly to write your data, you can control whether row or column order is used. If you want to let FORTRAN determine the order and not use subscripts at all, column ordering will be used. The following FORTRAN fragments illustrate the difference:

```
      REAL A(20, 30)

C COLUMN ORDER
      DO 10 J = 1, 30
        DO 20 I = 1, 20
          WRITE (*,*) A(I, J)
   20   CONTINUE
   10 CONTINUE

C ROW ORDER
      DO 30 I = 1, 20
        DO 40 J = 1, 30
          WRITE (*,*) A(I, J)
   40   CONTINUE
   30 CONTINUE

C IMPLICIT COLUMN ORDER, CHOSEN BY FORTRAN
      WRITE (*,*) A
```

Next, determine whether the dataset is scalar or vector. If it is scalar, include a single statement like this:

SCALAR

If it is vector, include a statement of the form

VECTOR *nComponents*

where *nComponents* is the number of components in each vector. You will also need to specify how the components are stored, with one of the following two statements:

INTERLACED
NONINTERLACED

The vector components are said to be interlaced when all the components for a vector at a single grid point are stored together. For example, if there are 3 components to each vector, you will write x, y, z triplets one after the other. They are said to be noninterlaced when each component is stored separately. For example, you will write all the x components, all the y components, and so on.

The last thing to do in specifying the dataset is to write all the data points as text numbers in the order specified by the ORDER, INTERLACED, and NONINTERLACED statements. First write a

DATA

statement on a line by itself. On subsequent lines, write each number in any FORTRAN F, E, or G format or any C %f, %e, or %g format. Separate the numbers with any number of spaces, tabs, or new lines. Use as many or as few lines as you like. As long as they are in the correct order and the count is correct, it will work.

You can use the word MISSING or the letter M in place of any individual data point to indicate missing data. If a vector value is missing, all of its components should be listed as MISSING.

Writing the grid

If your grid is curvilinear or irregular, you will need to write out the positions of all the points in the grid. Writing out a grid is very similar to writing out a vector dataset as described in the previous section. The number of components in each vector is the same as the number of spatial dimensions of the grid.

First specify the order with ORDER ROW or ORDER COLUMN. Then specify INTERLACED or NONINTERLACED. Then specify that there is to be vector data with a VECTOR *nSpatialDimensions* statement.

Finally, write out the points of all the grid points as if you were writing a vector field. First use the statement

GRID

on a line by itself to indicate that a grid is to follow. Then, as in the arguments to the `DATA` section above, write the locations of all grid points as if writing a vector field.

If you include a grid, the `BOUNDS` statement in the header is not needed. The bounds will automatically be calculated from the grid.

The grid specification can also appear before the dataset is defined. Either way works.

STF statement reference

Here is a list of all the statements in the STF file format:

`BOUNDS` *xMin xMax yMin yMax ...*

This gives the bounds, or the extent within Cartesian space, of the grid. This can also be thought of as the smallest box that can fit around the grid. There must be as many pairs as the spatial dimensionality of the grid. If a `GRID` statement is used, this is the number of components of the grid. Also, if `GRID` is used, the `BOUNDS` may be omitted, in which case they will be calculated automatically. If `GRID` is not specified, the number of spatial dimensions is assumed to be the same as `RANK` and a regular grid is constructed from `BOUNDS`, `DIMENSIONS`, and `RANK`.

`DATA`

data...

This gives all the data for the dataset in the order defined by `ORDER` and `INTERLACED` or `NONINTERLACED`. The data is freeform text real numbers separated by white space, beginning on the line right after `DATA` and continuing until all the data points have been given. Each data point may be the word `MISSING` or just the letter `M` to specify that the data is missing at that point.

`DIMENSIONS` *iDim jDim ...*

This gives the topological dimensions of the grid. There should be as many dimensions as the number given by `RANK`. For example, a 20 by 50 grid should have `DIMENSIONS 20 50`.

`END`

The `END` statement the end of a dataset. After an `END` occurs, you can start a new dataset. If the dataset is the last dataset in the file, the `END` statement is not needed.

`GRID`

grid data...

The format of this is just like `DATA`, but it gives vector data specifying the positions of all the grid points. This *must* be vector data, and a `VECTOR` statement must precede the `GRID` statement.

`INTERLACED`

This specifies that subsequent vector data in a `DATA` or `GRID` statement is interlaced. In interlaced data, all components for the first grid point appear, then all components for the second, and so on.

`NAME` *name*

This gives the name of the dataset. If it is not present, the name of the previous dataset in the file is used. This is handy for including several timesteps within one file, as the name needs to be specified only once. If the dataset is the first one in the file, the name is constructed from the file name.

`NONINTERLACED`

This specifies that subsequent vector data in a `DATA` or `GRID` statement is not interlaced. In non-interlaced data, all the first components in the dataset appear, then all the second components, and so on.

`ORDER COLUMN`
`ORDER ROW`

These specify column- or row-major ordering for data in subsequent DATA or GRID statements. The default is column-major ordering.

`RANK` *nDimensions*

This gives the number of topological dimensions of the grid. For example, a two-dimensional grid should have `RANK 2`.

`SCALAR`

This specifies that the data after the next `DATA` statement is scalar data. See `VECTOR`.

`TIME` *time*

This specifies that the dataset is a single time step of the whole dataset at *time*, which is a real number. Multiple time steps in separate datasets can be used to make a time-dependent dataset. If there is no `TIME` statement, the dataset is assumed to be static.

`VECTOR` *nComponents*

This specifies that the data after the next `DATA` or `GRID` statement is vector data, with a number of components given by the integer number *nComponents*.

Examples

Here is an example of a C program that generates a data file in the STF format:

```
/*exampleSTF.c
  Example program to write an STF file

  Compile

  cc exampleSTF.c -lm -o exampleSTF

  Eric Pepke
*/

#include <stdio.h>
#include <math.h>

#define XSIZE 50
#define YSIZE 50
#define ZSIZE 50

main()
{
    int i, j, k;
    FILE *outFile;
    int dimSizes[3];
    float array[XSIZE][YSIZE][ZSIZE];      /*Data array*/

    /*Fill array*/
    for (i = 0; i < XSIZE; ++i)
    {
       for (j = 0; j < YSIZE; ++j)
       {
           for (k = 0; k < ZSIZE; ++k)
           {
              array[i][j][k] =  sin(((double) i) / XSIZE * 6.0)
                              + sin(((double) j) / YSIZE * 7.0)
                              + sin(((double) k) / ZSIZE * 8.0);
           }
       }
    }

    /*Open the file*/
    outFile = fopen("example.stf", "w");
    if (outFile)
    {
       /*Print out the name, rank, dimensions, bounds*/
       fprintf(outFile, "NAME field\n");
       fprintf(outFile, "RANK 3\n");
       fprintf(outFile, "DIMENSIONS %d %d %d\n", XSIZE, YSIZE, ZSIZE);
       fprintf(outFile, "BOUNDS %g %g %g %g %g %g\n",
                        0.0, (float) XSIZE - 1,
                        0.0, (float) YSIZE - 1,
                        0.0, (float) ZSIZE - 1);

       /*It will be scalar data, in row order*/
       fprintf(outFile, "SCALAR\n");
       fprintf(outFile, "ORDER ROW\n");

       /*Print the data*/
       fprintf(outFile, "DATA\n");
       for (i = 0; i < XSIZE; ++i)
       {
           for (j = 0; j < YSIZE; ++j)
           {
              for (k = 0; k < ZSIZE; ++k)
              {
```

```
                    fprintf(outFile, "%g ", array[i][j][k]);
                }

                /*Put in a newline every row just to make it easier to
                  read by a person*/
                fprintf(outFile, "\n");
            }
        }

        fclose(outFile);
    }
    else
    {
        perror("example.stf");
    }
}
```

There are several other examples of STF files in the `demo` directory.

SY format

The SY file reader (named for Saul Youssef) provides a simple way of storing 3-D scalar datasets over regular grids. It is recommended that all new work use the STF file reader instead, as it is almost as easy to use and is far more flexible.

An SY file is an ASCII text file containing a series of free form numbers, separated by white space. The first three numbers are the *i*, *j*, and *k* dimensions of the dataset. These three dimensions are aligned with the spatial dimensions *x*, *y*, and *z*.

The next six numbers give the bounds of the dataset in *x*, *y*, and *z*. They appear in the order *xMin*, *xMax*, *yMin*, *yMax*, *zMin*, *zMax*.

The remaining numbers give scalar values at each point in the grid, in FORTRAN column-major order.