CSImage Programming Guide

Michael Ochs[1] and Joseph Callahan[2]
[1]Bioinformatics and [2]NMR and Medical Spectroscopy
Fox Chase Cancer Center
Philadelphia, PA 19111
27 April 2000

Description

This programming guide provides information to allow users to build their own format descriptor files (fdf's), add data fields, and create operational classes (*visitor* classes). The fdf files define how to load external data formats into CSImage. Examples are located in CSImage/fdf directory (windows users should substitute \ for / throughout this document). Each fdf file contains a list of variables and their locations and encodings within the external file. In addition to the built-in routines for Big and Little Endian byte order and VAX and IEEE float encodings, there are two source files, UserInt.java and UserFloat.java, located in the CSImage/java directory which permit the user to define nonstandard byte ordering and float encoding conversion routines. Data fields are added to the UserVar class, located in the CSImage/csi/data directory. The file UserVar.java is placed in the CSImage/java directory when you unpack the CSImage.jar file, so modifications can be made. Visitor class files are placed in the CSImage/csi/visitor directory. The files ParameterVisitor.java and VisitorTemplate.java are placed in the CSImage/java directory when you unpack the CSImage.jar file, and provide examples for building additional visitors.

Format Descriptor Files (fdf)

The fdf file is used when you import data into CSImage from another format. CSImage can import ASCII and Felix data (old format) automatically, however it is also capable of importing any external file. A sample file is given below for data generated by a Siemens magnetom system running on a VAX/VMS computer.

The file begins with comment lines. You can use as many comment lines as you wish, all initiated with the usual Java comment symbol //. The comment lines in this fdf file essentially describe the file in detail and it is recommended that you copy these lines into all fdf files. In addition lines are allowed to have comments at the end, indicated by the // characters. After the comment lines there is the line specifying which data class the external data should be read into (*e.g.* csi.data.Spec. csi.data.Image2D, *etc*.). The next line gives the byte order of the input data, LittleEndian, BigEndian, or Custom). Custom byte ordering allows users working with data which has odd byte orders (PDP for instance) to write their own class to handle reordering the input bytes. This class should reside in the CSImage/csi/io directory and be named UserInt.class. The next line specifies the float encoding used on the input file, either VAX, IEEE, or Custom. As with byte ordering, the Custom choice allows the user to create a class UserFloat.class in the CSImage/csi/io directory to handle unusual encodings. The following line needs to be left unused, then the specification of data location begins.

The following lines tell the program how to read the data file. The colon character, :, is used as a field separator. The command skip is used to skip the specified number of bytes in the file. The following line shows a typical line for reading. The method in the data file (here csi.data.Image2D) is specified, followed by the number of bytes in the data file used for this

```
// FDF File for Siemens Magnetom Image Data Set
//
// FDF Descriptor
//      Style of fdf file is comments, class line, byte order line,
//      float encoding line, blank line, field lines with format
//           skip: number of bytes OR
//           variable name: size in bytes: external type: internal type
//      a line beginning with DATA, then data format as
//           R/I order: R size: I size: external type: [spatial ordering]
//      the R/I order is how real and imaginary are ordered
//      the R size is the size in bytes of the real variable
//      the I size is the size in bytes of the imaginary variable
//      the external type is the data type used in the external file
//      the spatial ordering is the order of the time and spatial
//              dimensions in the external file
//
Class: csi.Data.Image2D         // name of the class it should find
Byte Order: LittleEndian        // byte order used in file
Float Encoding: VAX     // the type of float encoding
                   ---- LEAVE THIS LINE UNCHANGED
skip: 3072
setPatient: 26: String: String
skip: 2
setOperator: 12: String: String
skip: 572
setSeqFile: 64: String: String
skip: 8
setPulseType: 8: String: String
skip: 88
setTR: 4: float: float
skip: 120
setSliceThickness: 4: float: float
skip: 4
setFovX: 4: float: float
setSliceOffset: 4: float: float
skip: 112
setTE: 4: float: float
skip: 136
setAutoScale: 4: float: float
skip: 264
setFreq: 4: int: float
skip: 2278
setSizeX: 2: int: int
setSizeY: 2: int: int
skip: 54
userData.setDirInt: 2: int: int
userData.setOrientInt: 2: int: int
userData.setXaxisInt: 2: int:int
userData.setYaxisInt: 2: int: int
skip: 425
setDate: 9: String: String
update: 1
skip: 898
DATA        *** DO NOT CHANGE THIS LINE ***
RealOnly: 2: 2: int: xy:
```

variable, the way the variable is stored in the external file, and the way the variable is stored in CSImage. The program handles automatic conversions wherever possible and has a mechanism for conversions which cannot be done automatically (*e.g.* int to String where some type of coding is in use). The fdf file continues with these specification lines until all the parameters are completed. Of special interest are the lines beginning with userData. These inform the program to invoke the methods specified in the UserVar class which has been instantiated in the data class. For example, the line

```
userData.setDirInt: 2: int: int
```

tells the program to take the next two bytes of the input (read as an int with proper byte swapping if needed) and convert it to a Java int, myInt, then invoke

```
UserVar.setDirInt(myInt)
```

to write that variable into the data object. The program also permits a single call of update with an int specifier. If this line is present, the method update in the UserVar class is called. This method is passed the data object and the int specifier, allowing the user to handle complicated conversions of variables (the provided UserVar class used at Fox Chase gives an example of conversion of the dirInt, orientInt, and axisInt variables into Strings appropriate for the application).

After the parameters are finished and the program has had the input file set to the proper location (here with the final skip command), the actual spectrum data, image data or multidimensional data is read in. The program is notified that the actual image or spectral data is beginning by the line beginning with DATA. The following line specifies the format of the data. Note that for a csi.data.Image2D object there should be the number of points specified by the variables set by the setSizeX and setSizeY lines, while for the csi.data.Spec object there should be the number of points specified by the setNumPts line. The program will get these numbers using the csi.data.spec.getNumPts() and csi.data.Image2D.getSizeX() and csi.Data.Image2D.getSizeY() methods, so the user can use the update method to set the variables if necessary.

The line specifying the data begins with the order of the data. The choices are RealOnly, ImaginaryOnly, allRthenI, allIthenR, RthenImixed, and IthenRmixed which specify that only real data is present, that only imaginary data is present, that all real values are given before the imaginary values, that all imaginary values are given before the real values, that each point is given with its real value followed by its imaginary value, and that each point is given with its imaginary value followed by its real value respectively. In addition, the encoding type Custom is permitted. The user is then responsible for implementing the method getData in the classes UserReadImageData and UserReadSpecData in the CSImage/csi/io directory to handle reading the actual data. The following fields give the size of the real and imaginary variables in bytes. The next field gives the encoding of these variables in the external file. The final field, present only for multidimensional data, gives the ordering of the points with the directions listed slowest changing to fastest changing (e.g. in this file the x variable changes the slowest).

The UserVar Class

This class, located in CSImage/csi/data, allows the user to insert new variables and associated methods into the system. When coupled to the visitor classes, it also permits manipulation of these variables in any manner the user wishes to implement. There are few limitations on this class, however certain routines must be present.

The UserVar class is responsible for writing and reading its own variables during Save and Open operations. As such, two methods must be present (though they are not required to perform any action unless persistence of UserVar variables is required). The routines are writeVariables(DataOutputStream) and readVariables(DataInputStream). These routines are called by csi.io.WriteFile.writeData() and csi.io.ReadFile.readData() and they are passed the appropriate data streams. The user should call the appropriate write and read routines from these stream classes to write persistent variables (e.g. writeDouble, readFloat, etc.). As a note, Java includes serialization which could automatically handle the writing and reading of these files. Unfortunately, tests at Fox Chase Cancer Center showed a large hit in speed (an order of magnitude) and size (factor of two) when using serialization.

It is important to make sure that all changes to these routines are made prior to saving large amounts of data. The reason is that if there are mismatches between the data files and the routines to read them, the program will be unable to handle the data files. The user can write routines to modify the files for the new code, however if planning can avoid this, it is well worth the effort. The UserVar file also has a public static String version so your write and read routines could check for appropriate version numbers and handle them automatically if desired. Also, CSImage has built in functions for writing and reading data without the UserVar variables. Since these variables are written at the end of the file, you can always recover the data structures built-in to the program using the Import and Export feature. These features also allow users who have different UserVar classes to share data.

The Visitor Classes

The greatest extensibility feature of CSImage is the use of the Visitor design pattern for operations. This design pattern permits the easy addition of new operations to the program. The coupling of this design pattern to Java's reflection mechanism also permits the new functions to be added automatically at run time to CSImage's User menu. There are two pieces to making this addition.

First, edit the CSImage/strings/OperString.java file to add the name of the new operation. The final String array in this file is

```
public static String[] UserItems = {
    "Parameter"};
```

where a new example operation has been added. Add the new operation *Name*s after "Parameter" (at this time there can be no field separators in the name, *i.e.* white spaces). When CSImage is next started, the new menu items will appear under the User menu.

Second, create the class file with the name *Name*Visitor.class where *Name* is the same String used for the menu item name. A template, *template.java*, is provided in the CSImage/java directory. Essentially a csiVisitor class contains a series of Apply(csiObject) methods, one for each type of data for the operation. Specifically, the class *NameVisitor* should have methods such as

```
public void Apply(csi.data.Spec theData)
```

and

```
public void Apply(csi.data.Image2D theData).
```

Neither routine is necessary if the operation is not appropriate for that type of data. The UserVar variables are also available to the Apply method through the line

```
UserVar myVar = theData.getLocalVar();
```

where *theData* is the data object passed to the *Apply* method.

Finally, once the operation has been performed, the *Apply* method must append a description of its operation to the history of the data object. Screen updating following an operation depends on there being a new entry in the history of the data object. Prior to the return statement, adding the line

```
theData.addHistoryLine(myString);
```

will ensure proper updating of the display. The variable *myString* should be a description of the operation. The HistoryWindow handles the display of this information and looks for two types of field separators. A colon, :, indicates the break between the highest level tree element and other elements while commas indicates breaks between lower level tree elements.

When the class file is created, it should be moved to the CSImage/csi/visitor directory. The file *ParameterVisitor.java* which was used to create the *ParameterVisitor.class* file is included as an example of adding a new operation to the program.