Bima Memo 98 &
ATA Memo 58

J-MIRIAD: Java Wrappers for MIRIAD Methods

G. R. Harp[1], and M. C. H. Wright[2]

1. Allen Telescope Array, SETI Institute
2. Radio Astronomy Laboratory, UC Berkeley

2003-09-26

Abstract

A new set of Java wrappers for the MIRIAD input-output routines provides access to MIRIAD data from Java and Jython (Jython is a dialect of Python). Originally developed as part of the correlator for the Allen Telescope Array, these wrappers are general purpose and are available for application in other software projects where Java or Python is used. In this article we describe the architecture of these wrappers and demonstrate their use with some examples.

Introduction

The radio-astronomical community is currently faced with the difficult challenge of upgrading its data manipulation software. Powerful, public-domain analysis packages (MIRIAD[1], AIPS, GILDAS, etc.) developed over 20+ years in procedural languages such as FORTRAN and C represent a valuable legacy. Yet new telescopes are controlled by object-oriented software written in languages like Java or C++. The challenge is to bridge the gap between the legacy code and new software systems.

One approach is to transliterate or rewrite the old packages using new techniques. This approach has several disadvantages: It discards 20 years of productivity by some of the best minds in the field, it is costly (in dollars or hours), the completion date may be far in the future (especially if developed “for free”), and it is followed by a long debugging period since years may pass before users exercise every aspect of the code.

The alternative is to integrate the legacy software into our new systems. This also has disadvantages: The legacy software must be maintained requiring developers skilled in procedural languages, the compilation of the complete system is more complex, and it is inherently inelegant. The last point should not be disregarded, as it represents a motivational barrier for developers who take pride in their work.

We acknowledge that there is no correct choice, but a cost/benefit analysis adds perspective. With so few software developers in our community, we must place high value on their time. We believe this time better spent creating new functionality to address modern astronomical problems rather than reproducing functionality that already exists.

In line with this, several groups are working to expose the capability of legacy software to modern systems. In one such effort[2,3] Python scripts allow invocation of MIRIAD programs and GILDAS routines from the Python command line. The aim of these efforts is to give Python control of the complete data processing chain at the ALMA millimeter array under construction in Chile. On the MIRIAD side, this effort has focused on high level MIRIAD programs rather than low-level subroutines. However, to generate fresh MIRIAD data files from scratch, one requires low-level access. Recently, Pound[4] has developed Java wrappers for some low-level MIRIAD routines.

In a parallel effort we have developed Java wrappers for the I/O routines in MIRIAD. These wrappers (dubbed J-MIRIAD) were developed for the Allen Telescope Array (ATA) which is currently under development by the SETI Institute and the U. C. Berkeley Radio Astronomy Laboratory[5]. J-MIRIAD allows the ATA control software, written in Java, to write MIRIAD files using output from the ATA correlator. Presently J-MIRIAD includes routines for reading and writing headers, history files, UV data and image data. In this memo we describe the architecture of J-MIRIAD and demonstrate its application.

A side benefit of J-MIRIAD is that it exposes MIRIAD I/O to Python automatically, if one uses Jython. Jython is a "super-dialect" of Python; it is compatible with ordinary Python but also gives access to Java classes and methods from the command line. We demonstrate this behavior with an example.

Architecture

While choosing a design for J-MIRIAD, we decided to maintain a 1:1 correspondence between J-MIRIAD routines and MIRIAD routines. This shortens the learning curve for users already familiar with MIRIAD, and simplifies transliteration from FORTRAN to Java.

We reap some of the benefits of our object-oriented language by putting methods into classes that keep track of values maintained between method invocations. For example, most MIRIAD I/O routines take an integer argument that identifies the MIRIAD file. We hide this argument from the J-MIRIAD user by storing it in a private variable. This simplifies the interface and additionally allows error detection and prevention (e.g. file not open, invalid file number, etc.).

In a more detailed example consider the call for opening a MIRIAD image dataset in C:

```
/* C version */
void xyopen_c(int *thandle, Const char *name, Const char *status,
int naxis, int *axes){…}
```

In J-MIRIAD, this is replaced by two methods (one of a few exceptions to the 1:1 correspondence rule):

```
/* Java version */
```

```
public void xyopen(String name, int[] axes) throws IOException{…}
public int[] xyopen(String name) throws IOException{…}
```

Taking each parameter in turn, notice that the C-parameter `thandle` disappears because it is transformed to a private field of the XYIO class (see below). The `name` parameter is preserved, but the `status` parameter disappears. In C, `status` is "`old`" or "`new`", but in J-MIRIAD this is implied by which version of `xyopen` that is called. Since "`new`" files require that you specify the `axes`, one version takes an `axes` parameter and creates a new file. Instead "`old`" files obtain `axes` from the file itself, and this is returned from the version that opens an old file. Finally, since Java arrays know their own length, there is no need to specify `naxis` as in C.

From the Java perspective, J-MIRIAD consists primarily of 3 classes (fig. 1): An abstract superclass, MirIO, which contains header and history I/O, and two concrete classes UVIO and XYIO for reading/writing uv data or image data, respectively. UVIO and XYIO each derive from MirIO and so incorporate its functionality.

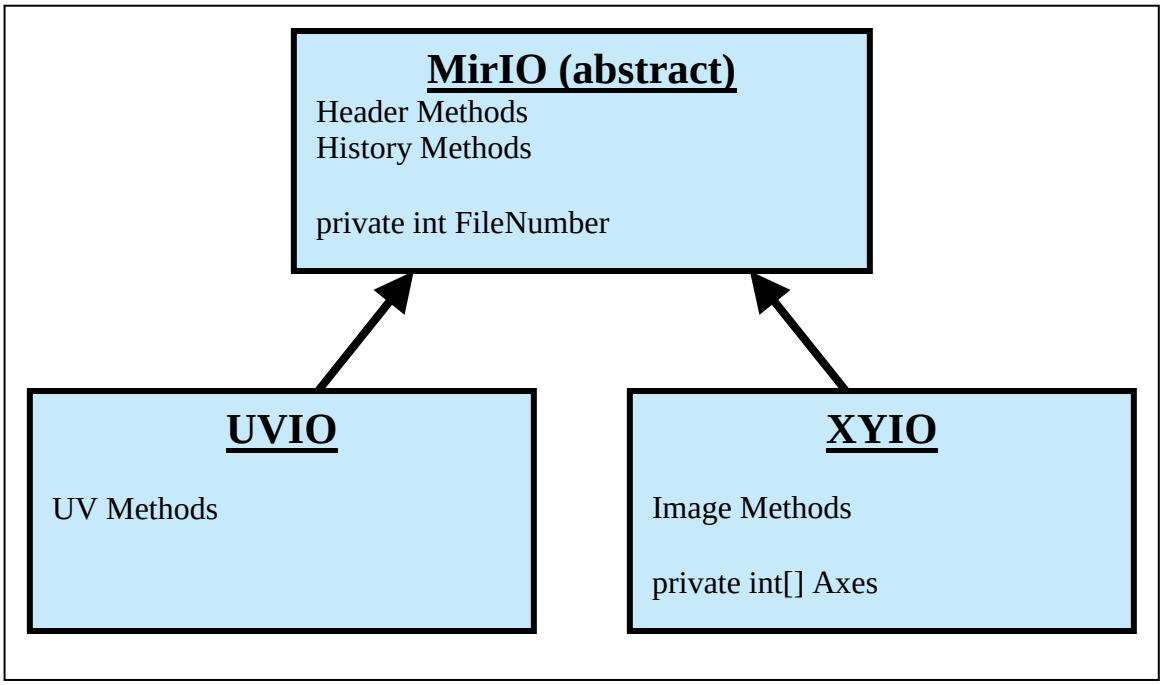

Figure 1: J-MIRIAD class inheritance.

Implementation

Now we come to the "inelegant" part of J-MIRIAD, where the Java methods interface with C. We use the Java Native Interface (JNI) technology which is built in to Java. Inside of our Java classes we declare "native" methods, e.g.

```
/* Java native method declaration */
private native int xyopen(byte[] name, byte[] how, int[] axes);
```

Sun provides a utility “javah” which takes the compiled XYIO class and generates a header file from it. Inside this header file is a C-style declaration for each native method, in this case

```
JNIEXPORT jint JNICALL Java_XYIO_xyopen
      (JNIEnv *, jobject, jbyteArray, jbyteArray, jintArray);
```

To complete the connection between Java and MIRIAD, we must provide an implementation of this function. This is tedious but straightforward. The only twist is that we must be aware that the Java Virtual Machine (JVM) automatically calls `malloc` for array type arguments. Following best-practice, our implementation wraps each array in a “smart pointer” (`SafeJXArray`) at the beginning of the function, e.g.

```
JNIEXPORT jint JNICALL Java_XYIO_xyopen
      (JNIEnv * env, jobject obj, jbyteArray filename,
      jbyteArray how, jintArray axes)
{
      SafeJBArray safe_filename (env, filename);
      SafeJBArray safe_how (env, how);
      SafeJIArray safe_axes (env, axes);
      int filenum;
      xyopen_c(&filenum,
            char*)safe_filename.Get(),
            (char*)safe_how.Get(),
            safe_axes.Length(), safe_axes.Get());
      return filenum;
}
```

The `SafeJXArray` types are a set of classes we wrote for wrapping JNI arrays. They make sure that the JNI arrays are `free`’ed when they go out of scope and expose the underlying C-arrays through the `Get()` and `Length()` functions.

Once the C++ code is written, it is compiled into a shared object library (`libjmir.so`) and made available to the JVM at run time. The compiled MIRIAD C-routines must be similarly made available. Initially we sought to use the shared object libraries produced by the standard MIRIAD distribution. However, this was inconvenient due to co-dependencies between libraries. We hope to work with the MIRIAD development team to find a simpler library structure, but for the time being we simply compiled the necessary MIRIAD files directly into `libjmir.so` .

One outstanding issue with J-MIRIAD deserves mention. The original MIRIAD code was developed prior to widespread adoption of “Exception” technology for error handling. Hence when MIRIAD encounters a “fatal” error, the program simply aborts. Again, we hope to work with the MIRIAD development team to insert a mechanism for generating exceptions instead.

Demonstrations

The main scripting language chosen for ATA control is Jython, a dialect of Python. Jython has the power to import Java classes analogously to Python scripts. Since the ATA control system is written in Java, Jython gives us powerful control of the ATA from the Jython command line *without writing any Python wrappers*. This property is beneficial to users of the scripting interface, since they use exactly the same interface as program developers and only one set of interfaces need be maintained.

Jython is also a powerful development tool. As you develop any new piece of Java code, Jython can be used to exercise the code as you write it. This approach was used in development of J-MIRIAD. The appendix shows a by product of J-MIRIAD development, a Jython script to write a simple MIRIAD UV file (of a point source). This file now serves as an example for Python users who wish to manipulate MIRIAD data at the ATA.

For sophisticated operations, it makes sense to use Java rather than Python. After all, Python does not offer many of the benefits of modern methods (e.g. strong typing, data hiding, etc.) for which we wrote the wrappers in the first place. To more fully demonstrate J-MIRIAD's capabilities, we have written two Java programs "UVFromImage" and "XYFromImage." Each program takes a graphics file as input (*.gif, *.jpg, etc.) and generates a MIRAID file (UV or Image, respectively) from it.

Figure 2 is an image generated from UVFromImage. Here 350 antennas (61,000 baselines) were assumed using positions from the proposed complete ATA. The full width of the image corresponds to the primary FOV of the ATA antennas, assumes a sky frequency of 1420 MHz, and simulates an 8 hour observation. The source structure is obviously artificial, but highlights the unique combination of FOV and resolution that will be available from a single ATA observation.

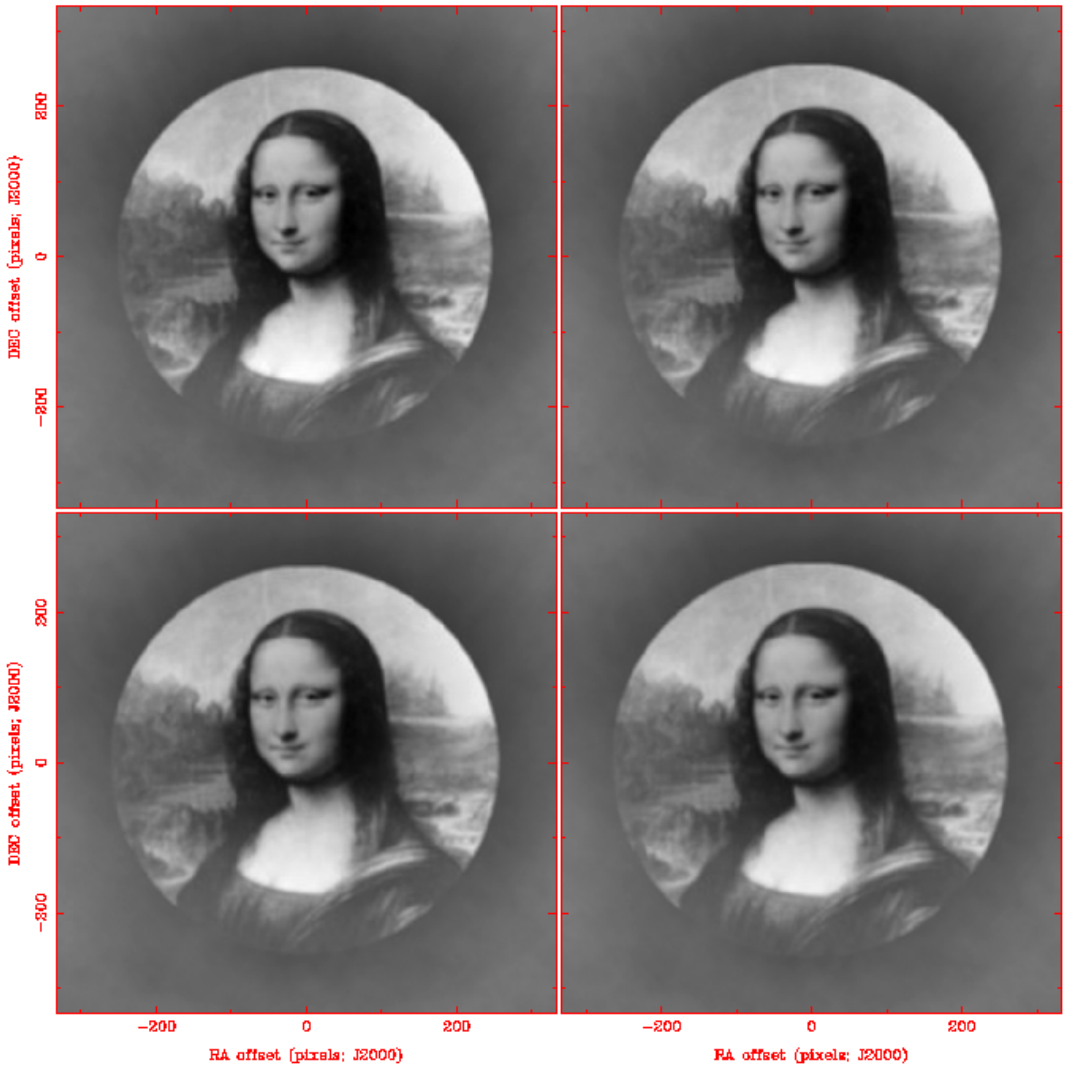

Figure 2: “Dirty” image from a UV file generated by J-MIRIAD. Four channels are displayed out of 1024 frequency channels produced by the ATA correlator over 100 MHz BW and centered at 1421 MHz. The full width of each image corresponds to the FWHM of the ATA antenna primary beam.

Conclusion

J-MIRIAD provides access to a subset of MIRIAD C routines. This subset fulfills the baseline needs of the ATA correlator, since they permit generation of MIRIAD UV files (as output from correlator) and MIRIAD Image files (as output from preliminary data analysis). In addition, both Java and Jython provide methods to invoke self-contained MIRIAD programs at the operating system level (e.g. well developed imaging, self-calibration and deconvolution programs such as INVERT, SELFCAL and MOSMEM). Thus J-MIRIAD already allows the creation of sophisticated data analysis programs that leverage the power of MIRIAD.[6]

Here we have justified our choice, mentioned in the introduction, to re-use legacy code rather than re-implement it with new techniques. With only a few man-weeks of effort, we have brought a substantial fraction of MIRIAD's capabilities into the 21st century.

We hope that these efforts will stimulate similar activity in other software development groups. To this end, we will make J-MIRIAD source code available to interested parties (please contact authors). Although J-MIRIAD was developed in the context of the ATA control system, it would be straightforward to convert it to a stand-alone product. If other groups desire to adopt this code, we would be interested in cooperatively developing a generic version of J-MIRIAD.

Appendix: Jython Script for Simple MIRIAD UV File

```
from  java.lang   import      *
from  ata.miriad  import      *
from  ata.util    import      *
from  ata.math    import      *

def uvgen(outfile):
      io = UVIO()
      io.uvopen(outfile, "new")
      io.hisopen("write")
      io.uvset("preamble", "uvw/time/baseline", 0, 0.0, 0.0, 0.0)
      io.uvset("data", "wide", 0, 1.0, 1.0, 1.0)
      io.hiswrite("Jython: Miriad")
      io.wrhda("obstype", "crosscorrelation")
      io.uvputvra("source", outfile)
      io.uvputvra("operator", "Jython")

      sra = 0.0
      io.uvputvrd("ra", sra)
      io.uvputvrd("obsra", sra)
      sdec = 30.0/180.0*Math.PI
      cosdec = Math.cos(sdec)
      sindec = Math.sin(sdec)
      io.uvputvrd("dec", sdec)
      io.uvputvrd("obsdec", sdec)
      io.uvputvrd("lo1", 1.421 - .19 - 0.01)
      io.uvputvrd("lo2", 0.19)
      io.uvputvrd("freq", 1.421)
      io.uvputvrd("freqif", 0.01)
      io.uvputvrr("pbfwhm", 8000.0)
      io.uvputvrr("inttime", 600.0)

      now = 106290400679200000OL
      timeout = 2452889.6293378705
      lst_rad = 0.5646596920389076
      twopi = Math.PI * 2.0
      along = sra - lst_rad
      alat = Constant.HC_lat / 180.0 * Math.PI
      coslat = Math.cos(alat);
      sinlat = Math.sin(alat);
      io.uvputvrd("latitud", alat)
      io.uvputvrd("longitu", along)
      telescop = "hatcreek"
      mount = UVIO.ALTAZ
      evector = 0
      evector = 0.0
      io.uvputvri("mount", mount)
      io.uvputvrr("evector", evector)
      io.uvputvra("telescop", telescop)

      io.uvputvrr("jyperk", 150.0)
      io.uvputvrr("vsource", 0.0)
      io.uvputvrr("veldop", 0.0)
      io.uvputvrr("epoch", 2000.0)
```

```
nant = 3
io.uvputvri("nants", nant)
io.uvputvri("ntemp", 0)

io.uvputvri("npol", 1)
io.wrhdi("npol", 1)
io.uvputvri("pol", UVIO.PolXX)
io.uvputvri("nwide", 1)
io.uvputvrr("wfreq", 1.421)
io.uvputvrr("wwidth", 0.02)
io.uvputvrr("wsystemp", 300.0)
tpower = [400.0, 400.0, 400.0]
io.uvputvri("ntpower", nant)
io.uvputvrr("tpower", tpower)

antpos = [0.0,0.0,0.0, 0.0,100.0,0.0, -100*Math.sin(alat), 0.0,
100.0*Math.cos(alat)]
io.uvputvrd("antpos", antpos)

ha = 1.0
lst = ha * Math.PI / 12.0 + sra
io.uvputvrd("ut", lst)
io.uvputvrd("lst", lst)
sinha = Math.sin(lst)
cosha = Math.cos(lst)
sinq = coslat * sinha
cosq = sinlat * cosdec - coslat * sindec * cosha
psi = Math.atan2(sinq, cosq) + evector
io.uvputvrr("chi", psi)

n=1
while (n < nant):
      m = 0
      while (m < n):
            bxx = antpos[3*n] - antpos[3*m]

            byy = antpos[3*n+1] - antpos[3*m+1]
            bzz = antpos[3*n+2] - antpos[3*m+2]
            bxy = bxx * sinha + byy * cosha
            byx =-bxx * cosha + byy * sinha
            p0 = bxy
            p1 =  byx * sindec + bzz * cosdec
            p2 = -byx * cosdec + bzz * sindec
            p3 = timeout + 365.25 / 366.25 * ha / 24.0
            p4 = UVIO.antbas(m, n)
            preamble = [p0, p1, p2, p3, p4]
            wcorr = [ComplexF(1.0, 0.0)]
            wflags = [1]
            wviz = Visibility(preamble, wcorr, wflags)
            io.uvwrite(wviz)
            m=m+1
      n=n+1

io.hisclose()
io.uvclose()
return
```

References

[1] Sault, R. J., Teuben, P. J., and Wright, M. C. H., 1995, in Astronomical Data Analysis Software and Systems IV, ed. R. Shaw, H. E. Payne, and J. J. E. Hayes, ASP Conf. Ser., 77, 433.
[2] Pety, J., Gueth, F., Guilloteau, S. Teuben, P., Wright, M., et al., to be presented at ADASS, 2003.
[3] Pety, J., Cosson, F., Gueth, F., Guilloteau, S., Lucas, R., Teuben, P., and Wright, M., ALMA memo 465: "Case for interoperability as an ALMA off-line model."
[4] Pound, M., unpublished.
[5] DeBoer, D.R., Dreher, J.W., ATA memo 23: "A System Level Description of the ATA."
[6] This is made possible partly because MIRIAD was well designed at the outset. MIRIAD consists of relatively small, self-contained programs that can be invoked in any sequence. If MIRIAD were designed as an integrated system, the present efforts at code re-use would be complicated.