AIPS++ is a project of the AIPS++ Consortium

AIPS++ Newsletter

Greenwich	GMT	+ 0:00
Australia	EAST	-10:00
Netherlands	MET	- 1:00
Charlottesville	EDT	+4:00
Urbana-Champaign	CDT	+5:00
Socorro	MDT	+6:00

Project Manager: Tim Cornwell
Managing Editor: Bob Hjellming
Production Editor: Kate Weatherall

August 1998

In this issue:

Pulsar Data in AIPS++

Rick Fisher - NRAO, Green Bank

The AIPS++ tool-box offers the possibility of analyzing pulsar data for time of arrival, flux density, profile shape, dispersion, and other parameters of interest. I have used it look at pulsar data from the 140-ft spectral processor using *Glish* routines and my own solar system ephemeris client at http://www.gb.nrao.edu/~rfisher/Glish/solar_system.html (also see The *Glish* Software Bus in this issue of the Newsletter to see an example of how to implement *Glish* clients).

This note is a brief description of how you can build your own pulsar analysis scripts in *Glish*. Some of the functions that I originally wrote in *Glish* are now available as faster **C++** modules in AIPS++. This is often a good migration path for new routines because *Glish* is a reasonably powerful mathematics engine by itself. Eventually, we'd like to add pulsar analysis to the single dish part of AIPS++ with a proper GUI and other support.

In pulsar timing mode the spectral processor's output is a set of 2-D matrices of intensity as a function of sky frequency and pulsar phase with one matrix for each receiver channel. Feel free to contact me for the details on how to create and AIPS++ table from the spectral processor FITS files, but for this example let's assume that one matrix and the necessary header information has been loaded into *Glish* variables as follows:

```
  data        matrix of 256 frequency channels by 128 time bins
  utcstart    UTC time of the leading edge of the first time bin
              in seconds
  mjd         modified Julian date that goes with utcstart
  phasetime   spacing or width of time bins in seconds
  freqres     spacing of frequency channels in Hz
  ctrfreq     sky frequency of the center of frequency channel 129,
              where the channel numbers run from 1 to 256
  dispmeas    pulsar dispersion measure in parsec/cm^3
  rfsideb     R.F. sideband, 0 = upper, 1 = lower
  ifsideb     I.F. sideband, 0 = upper, 1 = lower
  template    an array of 128 bins which contains the pulsar's
           profile template
```

First, we do a rough normalization of the data by dividing each frequency channel column by the average power in that column. This removes the receiver passband shape to first order.

```
- for (ch in 1:data::shape[1]) {
-     avg := sum(data[ch,]) / data::shape[2]
-     data[ch,] /:= avg
- }
```

Now, if you display the normalized 2-D array, you will see the pulse in the frequency/pulse phase plane. If there is much dispersion at the observed frequency, the track will be a diagonal across the image.

```
- dd.array(data)
```

To dedisperse the pulse each frequency column must be shifted in time to line up with the center frequency column. The amount of shift depends on the dispersion measure, center frequency, frequency offset, and time bin width. The shift operation is done with the AIPS++ mathematics module FFT server's shift function. This shift is actually an array rotation, which is what we want.

```
- fftserv := fftserver()
- dm_coef := 0.0041494 * dispmeas
- one_over_cf_sq := 1.0e18 / ctrfreq^2
- freq_step := freqres
- if (rfsideb != ifsideb) freq_step := - freq_step
- for (ch in 1:data::shape[1]) {
-     ch_offset := as_double(ch - (data::shape[1] / 2) - 1)
```

```
-     freq := ctrfreq + (freq_step * ch_offset)
-     time_shift := dm_coef * ( 1.0e18 / (freq^2) - one_over_cf_sq)
-     bin_shift := -time_shift / phasetime
-     data[ch,] := fftserv.shift(data[ch,], bin_shift)
- }
```

Now, to look at the pulse profile all we need to do is sum the frequency bins in each time bin row.

```
- profile := array(0.0, data::shape[2])
- for (bin in 1:data::shape[2]) {
-     profile[bin] := sum(data[,bin]) / data::shape[1]
- }
- dp.ploty(profile, style_='lines')
```

The pulse arrival time can be found by cross-correlating the dedispersed profile with the profile template.

```
- cc := fftserv.crosscorr(profile - 1.0, template)
- dp.ploty(cc, style_='lines')
```

Then solve for the time of the cross-correlation peak offset by fitting a y-axis parabola through the three points nearest the peak

```
- nch := length(cc)
- peak_ch := order(cc)[nch]
- if ((ch := peak_ch - 1) &lt; 1) ch := nch
- y1 := cc[ch]
- y2 := cc[peak_ch]
- y3 := cc[(peak_ch % nch) + 1]
- k := (y3 + y1 - 2.0 * y2) / 2.0
- x0 := (y1 - y3) / (4.0 * k) + peak_ch - 1.0
```

and then add that time offset to the UTC of the beginning of the data array to get the pulse time of arrival.

```
- toa := utcstart + x0 * phasetime
- if (toa = 86400.0) {
-     toa -:= 86400.0;
-     mjd +:= 1;
- }
- print 'TOA: MJD', mjd, 'UTC', toa, 'seconds'
```

This time of arrival may then be corrected to the solar system barycenter for comparison with other measurements of the same pulsar. The JPL solar system ephemeris will soon be part of the AIPS++ `measures` module with a function that returns the barycenter delay. In the meantime you are welcome to use the *Glish* client `ephemeris`, that I wrote for this purpose, as described in http://www.gb.nrao.edu/~rfisher/Glish/solar_system.html

I have also written a *Glish* script that elaborates on this code using two AIPS++ tables containing pulsar data. located at http://www.aoc.nrao.edu/~rhjellmi/aips++/newsletter/aug98/psrexample.g

Project News Summary

Tim Cornwell - NRAO, Socorro

The next, third, beta release continues to consume much of our time. We know that users are eagerly awaiting this release, and we'll be pleased to finally get it done. The response to the first release in February 1997 was substantial and we learned a lot about how users viewed the system. The second beta release in September 1997 elicited less comment, presumably because the incremental changes were small. Hence the main goal for the third release is to make substantial improvements in user interface, functionality and documentation. Much of the necessary work has been completed and tested but we have still to complete the following:

- Improved documentation, based on terminology and concepts acceptable to our users. Astronomers from our testing group at the AOC are helping write this documentation.
- Improved calibration tools, including some tools for data access and calibration transfer.
- Testing of some functionality.

We expect these to be largely completed in August or September 1998, and the third beta release will occur soon after. If no large new changes are required, a fourth beta will follow within 3-4 months, and then a first public release some time in the first half of 1999. Our main concern is with the user interface. If this is seen to require more work, then the first public release will be further delayed.

Programmer's Corner - Part 3

Brian Glendenning - NRAO, Socorro

We now will finish our example of using *Glish* to convolve planes of any orientation from an image of any shape. This chapter is somewhat more involved than the previous segments (Part 1, Part 2). To recap, how do you loop over all planes of the image (that is, set up `blc` and `trc` appropriately so that we can call the `getchunk` method of `image`)?

Let's suppose for the sake of argument that you have a cube with a shape [512,512,4,128]. A cube like this might be a spectral line image cube with full polarization, i.e. RA,DEC,POL,FREQ, and 128 frequency channels. If you are interested in getting at the RA,DEC planes from the image, then you need to generate the following series of `BLC` and `TRC` values:

```
BLC (RA,DEC,POL,FREQ) | TRC (RA,DEC,POL,FREQ)
============================================
1,1,1,1               | 512,512,1,1
1,1,2,1               | 512,512,2,1
1,1,3,1               | 512,512,3,1
1,1,4,1               | 512,512,4,1
1,1,1,2               | 512,512,1,2
...                   | ...
1,1,3,128             | 512,512,3,128
1,1,4,128             | 512,512,4,128
```

How can you do this? First lets note that you can index into a *Glish* array with both a scalar and array. For example:

```
- a := 1:5
- a[1] := 0 # Index with a scalar
- a
[0 2 3 4 5]
- a[[1,2,3]] := 9 # Index with the array [1,2,3]
- a
[9 9 9 4 5]
```

Indexing with an array is just like looping over all indices in that array, however it is considerably faster and more concise.

Besides indexing with an array of "indices", you can also index with a `mask`, which is an array of `booleans`. Array values are kept only where the mask is `True`. For example:

```
- a := 1:5
- a[[T,F,F,F,T]]
[1 5]
```

Since the logic of incrementing an index is kind of complicated, let's put it into a function called `bump`:

```
bump := function(where, shape, axis=1)
{
    where[axis] +:= 1;
    if (where[axis]  shape[axis] &amp;&amp; axis &lt; length(shape)) {
&#9;# If we are beyond the end of the current axis, set it to one and
```

```
        # ancrement the next axis.
&#9;where[axis] := 1;
&#9;return bump(where, shape, axis+1);
    } else {
        return where;
    }
}
```

In the example, `shape` is [4,128] and `where` runs from [1,1] to [4,128]. Note that this is an example where it is somewhat easier to write the function using recursion.

Now you can finally write a loop to go through all planes of the image:

```
while (!any(blc[mask] &gt; lengths[mask])) {        #1
    pixels := im.getchunk(blc,trc);              #2
    # ... convolve as before ...                 #3
    blc[mask] := bump(blc[mask], lengths[mask]); #4
    trc[mask] := blc[mask];                      #5
}
```

The following steps are accomplished with this *Glish* code.

1. Loop until `blc[mask]` has gone "past the end", i.e. given the way `bump` is implemented it will stop at [5,128]. `lengths` is the "shape" of the image, in this case [512,512,4,128].

2. Get the pixels from the image.

3. Convolve the pixels as before.

4. Increment the parts of the `blc` that change to the next plane. In the example, after this is called for the first time, `blc[mask]` is [2,1]).

5. Copy the changed values from `blc` to `trc`.

That's it for this article and series. I had intended to move next into GUI programming and other advanced topics. However, as was announced in the last AIPS++ Newsletter, I have moved to the MMA project, so I will bring this series to a halt, at least for the moment.

The *Glish* Software Bus

Darrell Schiebel - NRAO, Charlottesville

Glish's vector oriented command language is very powerful. A large portion of AIPS++ is written in *Glish*. The language has arrays, records, and all of the basic types. However, it is not the *Glish* scripting language which makes *Glish* unique, but rather the underlying "software bus". I would like to close out this series of articles on the "Story of *Glish*" by discussing *Glish*'s software bus in a little more detail.

Because of its roots as software for controlling distributed physics experiments, *Glish* has a very flexible `event transport` layer integrated into the language. This integration of distributed control makes *Glish* unique. Few other languages attempt this level of process control. CORBA (Common Object Resource Broker Architecture) and other libraries are now attempting to make this sort of control functionality available, but they were not available when AIPS++ adopted *Glish*.

Glish can start programs, called clients, on either the local host or remote hosts, and then interact with these programs. Clients are "plug-compatible" because of the way they can accept and generate events. *Glish* controls clients by sending events to them. An event is simply a name plus a value. By limiting the interaction between *Glish* and the clients to a series of discrete events, it is easy to substitute one client for another. As long as the new client accepts the same events, the substitution can be made without affecting *Glish* or the other clients. The clients are an encapsulation of a particular functionality.

For AIPS++, clients are typically written in **C++**. The following is a simple example which returns the current time. The source code for this client looks like:

```
int main( int argc, char** argv )
    {
    Client c( argc, argv );
    timeval tval;

    for ( GlishEvent* e; (e = c.NextEvent()); )
        {
        if ( ! strcmp(e-Name(),"time") )
            {
            if ( gettimeofday( &amp;tval, 0 ) <0 )
                c.Error( "couldn't get current time" );
            else if ( e->IsRequest() )
                c.Reply( ctime( &amp;tval.tv_sec ) );
            else
                c.PostEvent( "result", ctime( &amp;tval.tv_sec ) );
            }
        else
            c.Unrecognized( );
        }

    return 0;
    }
```

The Client class is the connection which **C++** clients use to communicate with the *Glish* interpreter. The `NextEvent()` function is used to get events as they are sent from the interpreter. In this case, the client has a loop for accepting events. `NextEvent()` returns a null pointer when the client should exit. `Reply()`, `PostEvent()`, `Error()`, and `Unrecognized()` are all member functions of client, and they are used to send events back to the *Glish* interpreter.

In *Glish*, the use of this client (if compiled into an executable called `time_client`) looks like:

```
- x := client('time_client')
- print x->time()
Tue Aug 4 10:40:16 1998
- x->time()
warning, event echo_client.result (Tue Aug 4 10:40:36 1998) dropped
```

With the print statement, `Reply()` is invoked in `time_client` because the *Glish* interpreter is waiting for a reply. In the next statement, another `time` event is sent to the client. In this case no result is needed, so the interpreter continues, and later receives the `result` event generated in the `time_client` by the call to `PostEvent()`.

The `Error()` member function of client is to indicate that an error has occurred, in this case due to a problem with `gettimeofday()`. The `Unrecognized()` member function is called to indicate that the event received has an unrecognized event name, in this case any event name other than `time`.

Once created, clients greatly enhance the capabilities of *Glish*. This is the mechanism through which much of the functionality of the AIPS++ libraries, written in **C++**, is made available to *Glish*. These clients can run transparently on local or remote machines. The resources provided by each of the clients can be utilized and combined in *Glish* scripts to solve complicated problems in ways unanticipated by the authors of the individual clients. This sort of flexibility and plug-compatibility is achieved because each client has a well-defined interface. Separate processes do a nice job of enforcing encapsulation. Existing stand-alone programs can be made into *Glish* clients by adding the **C++** code necessary to generate the events by which *Glish* and the "program" initiate and return results.

Recipe of the Month Glish Matrix Operations Doing Least Squares Fitting with Error Analysis

Bob Hjellming - NRAO, Socorro

Some of the advanced mathematical operations in *Glish* are in packages like `matrix` and `mathematics`. In this recipe we show how you can read in a table of (x,y) data and do least squares fitting to these data with complete error analysis including not only estimation of the errors in the polynomial coefficients, but also computation of the covariance matrix and correlation coefficients.

This is also an example of using a *matrix* object to carry out matrix operations.

Recipe: Glish Matrix Operations Doing Least Squares Fitting with Error Analysis

Goals: Read an ASCII table of (x,y) data, fit a polynomial to the data, perform error analysis on the fit, and plot data and fit.

Using: *t*ablefromascii function, *T*able, and *M*atrix objects

Results: Polynomial fit parameters, their errors, related Covariance and Correlation Coefficient matrices, and plots data and fit.

Assume: You have an input file of ASCII data in columns, and a header description file. In this recipe the ASCII data file named `begdata.txt` has two columns of numbers. The ASCII header file describes the data columns in the data file; in this case a header file `begdatahdr` assigns names and data types to each column, following the rules described in documentation of `header file contents`. Both input and header files must be in the directory from which you started AIPS++.

Script: A *G*lish script executes this recipe.

AIPS++/Glish commands and results	***Purpose and Background***
`tablefromascii('begtable', 'begdata.txt','begdatahdr')`	Put ASCII table of data into an AIPS++ table using the AIPS++ `tablefromascii` function. For details on what you can do with AIPS++ tables see the AIPS++ Reference Manual. The result of invoking `tablefromascii` is an AIPS++ Table named *begtable* is written to disk.
`xydata := table('begtable')`	Read this Table into a *Glish* object named `xydata`.
`xydata.getkeywords()`	Show keywords defined in the `gbihdr` file.
`[XDATA=Independent Variable, YDATA=Dependent Variable]`	List keywords.

`x := xydata.getcol('XDATA')` `y := xydata.getcol('YDATA')`	Extracts columns from the table and assign them to 1-D arrays using *getcol* on the *Glish* table object called *xydata*.

`N := length(y)`
`PolynomialOrder := 2`
`M := PolynomialOrder+1`
`ydata := array(y,N,1)`
	Extract number of (x,y) points
	Select polynomial fit to order 2
	Order-dependent matrix parameter
	Need y in array type for matrix input.

`mx := matrix_functions()`	Make matrix object named *mx*.

`A := array(0,N,M)`
`A[,1] := 1`
`for (i in 1:N) for (j in 2:M)`
`A[i,j] := x[i]^(j-1)`
	Build $A = 0$ matrix of right shape and put correct values into A.

`Atran := mx.transpose(A)`
`G := mx.mult(Atran,A)`
`Ginv := mx.invert(G)`
	Transpose A, then multiply $Atran$ by A to get G matrix
	Invert G.

`Coef := mx.mult(Ginv,mx.mult(Atran,ydata))`	Polynomial fit to $y(x)$.
`ysoln := mx.mult(A,Coef)`	Compute y solution from fit.

`errsq := N*mx.mean((ydata - ysoln)^2)/(N-1)`
`rms := errsq^0.5`
`Cov := errsq*Ginv`
	Compute rms error and Covariance matrix or (j in 1:M) sigma[j] := Cov[j,j]^0.5 Compute errors for each coefficient.

`r := Cov`
`for (j in 1:M) for (k in 1:M)`
`r[j,k] := Cov[j,k]/(Cov[j,j]*Cov[k,k])^0.5M`
	Compute r = correlation coefficient matrix.

`Coef`
`rms`
`Cov`
`sigma`
`r`
	Print out the polynomial coefficients and the various error parameters.

```
pl :=
pgplotter(background="white")
red := 2
blue := 4
pl.sci(blue)
pl.env(min(x),max(x),min(y),max(y),0,1)
pl.lab("x","y","Plot Data
(Red) and Second
Order Polynomial Fit to
Data (Blue)")
```

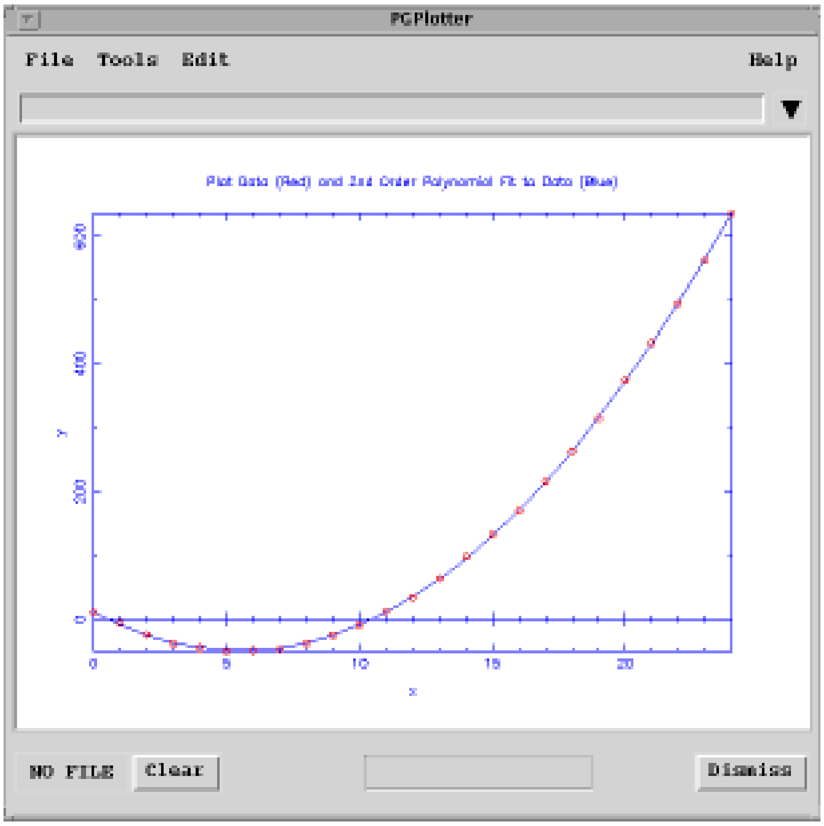
Make a pgplotter object named *pl*, set background to white, open plot window on the screen Set color values for plot. Set plot maxima and minima and label plot and axes.

```
pl.line(x,ysoln)
pl.sci(red)
pl.pt(x,ydata,4)
```

Plot polynomial fit as blue line and data point with red symbols. Final plot shown below.

In the *Pgplotter* window, *File* can be used to print and save the plot(s).

Contributions of recipes or other material for a prototype AIPS++ Cookbook are welcome. Please send to rhjellmi@nrao.edu.



PGPlotter
File Tools Edit
Help
Plot Data (Red) and 2nd Order Polynomial Fit to Data (Blue)
NO FILE
Clear
Dismiss

What's New in AIPS++?

Athol Kemball and Tim Cornwell - NRAO, Socorro

The following personnel changes have occurred in the AIPS++ Project:

- Jeff Uphoff (NRAO) has left to join TransMeta Corp. (where he will be working with Linus Torvalds (creator of Linux), among others!); and
- Joe McMullin (NRAO) has moved from Green Bank to Charlottesville.

The following changes have been made in AIPS++:

- A new script, `visplot.g`, is available which allows general display of visibility data. It has been extensively revised to incorporate user feedback regarding both capabilities and layout.
- A GUI utility, `simpleimage.g`, has been added to provide a semi-automated interface to the synthesis imaging capabilities in the *s*ky package. This is a first attempt to provide and evaluate a dedicated synthesis GUI.
- The display utility, *A*ipsview, has been modified to use 0 for the default, starting slice for the Z-axis when the axis name is `STOKES`. This should simplify start-up.
- The `measures.g` script has been split into `measures.g` and `measuresgui.g` for faster initialization. The underlying Measures code has been rationalized by placing Quantum and Units classes in a separate code module. Support for ITRF and topocentric frames has been added to Direction measures.
- The keywords `AS` and `IN` have been added to the table query language (*T*aQL) to mirror their use in *S*QL.
- There has been continuing work on support for source component models, with the introduction of `componentmodels.g`, containing `componenteditor.g` and `componentlist.g`. Support for elliptical disks has been added, as well as reference frequency manipulation, along with a range of other capabilities.
- A mechanism to verify AIPS++ distributions has been implemented, which allows AIPS++ updates to be verified as correctly passing a set of tests before they are distributed to other sites.
- Pop-up help utilities have been modified to minimize the problem of left-over help windows.
- A *Glish* utility, `regionmanager.g`, has been implemented to allow the manipulation of image regions of interest. This has included the development of underlying code in the library to provide support for image region selection and specification in general.

- A holography application, *holog.g*, has been implemented for WSRT data.
- The support for the Cygnus *e*gcs compiler and the SGI native compiler has been extended, with small changes throughout the system to provide compatibility on points of syntax.
- The flags `-help`, `-version` and `-info` have been added to *Glish*. Documentation changes have been made in preparation for *Glish* v2.7, along with other general updates. These include the rationalization of `gmisc.g`, which will be removed shortly.
- The new table browser has been revised to support new features, including th capability to plot rows versus columns.
- The UVFITS writer, *U*VFitsWriter, has been modified to optionally write system temperature (TY) and gain curve (GC) tables. This allows the export of AIPS++data to other packages earlier in the data reduction sequence.
- The new AIPS++ web page has been adopted, and extensively revised to support new features and incorporate user comments regarding the layout.
- There has been work on NFRA *Glish* data display utilities, including `datalinetool` and `mstool`. These extend or replace previous *Glish* data display utilities. More general utilities have been updated including `buttonscript.g`, (which generates *Glish* scripts from GUI button presses), *glishelp.g* (GUI *Glish* help) and `inspect.g` (interactively inspect a *Glish* variable). A new application, `j2convert`, has been added to read and convert the uvw coordinates of a WSRT measurement set.
- Code for a command line parameter-setting shell has been checked in, which will be available shortly for general use as `app.g`.
- A *T*cl/Tkbased widget client has been checked in to the system. A first version of a PGPlot driver for the WorldCanvas has also been implemented.
- The application `fits2table` now converts SDFITS files accurately into an AIPS++ table for correct handling by the DISH package.
- GBT spectrometer test data can be imported into an AIPS++ table.

Dan Briggs - *A Professional Remembrance*

Tim Cornwell - NRAO, Socorro

We were shocked and saddened by the tragic news that Dan Briggs of the NCSA/BIMA AIPS++ group was killed in a sky-diving accident on July 4. Dan was well-known to many of us as a colleague and friend. Dan's many friends have put together a memorial page that you can find at http://www.nmt.edu/~pio/dbriggs/. This memorial page describes Dan's attitude towards life and many personal accomplishments. Here I'd like to say something about my memories of his professional career.

I'd personally known Dan for a long time in a number of different roles: as a stilt-walker in Socorro parades, as a student at New Mexico Tech, as my Ph.D. student in the early 90's, and then as a collaborator and colleague within AIPS++.

He came to Socorro as a graduate student in physics after completing degrees in physics and mathematics at Caltech. My first contact with him was to see him walking on stilts in a parade in Socorro, a picture that is accessible from his memorial page. In the late eighties, he began spending time at NRAO, working with Rick Perley and then Craig Walker on different projects. He eventually came to me to ask about doing a Ph.D. on radio-astronomical imaging, something that I readily agreed to once I realized his great range of talents. For his Ph.D., and then later on in his career, he worked as what I would call a technical radio-astronomer, interested more in the techniques of astronomy rather than the results. Dan had an excellent combination of attributes for a researcher: great basic intelligence, varied technical skills, persistence, a judicious amount of perfectionism, and an enormous appetite for hard work. His Ph.D. from New Mexico Tech, which was overseen by myself and Jean Eilek, showed the quality of work that he was capable of producing (his thesis is available on the web at http://www.aoc.nrao.edu/ftp/ dissertations/dbriggs/diss.html). It covers the image processing of moderately resolved sources observed with radio-interferometric arrays. It's a stunningly complete piece of work, in which each substantive assertion is carefully described, analyzed and demonstrated. It contains two things that deserve to be called by his name. First, an analysis of the effect whereby deconvolution errors on moderately resolved sources can masquerade as calibration errors. Second, a new form of weighting, which he called robust weighting, whereby resolution and sidelobe-level could be traded one against the other, thus obtaining a compromise of improved resolution and signal-to-noise which optimizes the use of telescope time. Robust weighting was immediately recognized by others as a very useful technique and has been adopted in many reduction packages. In recognition of Dan's contribution, it has been suggested that the term Briggs' weighting to be used in place of robust weighting.

After his Ph.D. was completed in Socorro, Dan moved to Washington, D.C., to work at the Naval Research Lab as a postdoc, first on the Big Optical Array, and then on low-frequency imaging with the VLA (in collaboration with Namir Kassim, Kurt Weiler and others). In the latter work, Dan developed an existing, limited package for wide-field imaging from VLA B-configuration 327 MHz data into a well-engineered, very capable package for A-configuration data, that could run in a reasonable amount of time on a parallel machine. This exploited many of his skills: understanding of the physics and mathematics, expertise at meeting the complicated computing requirements, and appreciation of the resulting science.

Following his time at NRL, he moved on to work with the NCSA/BIMA group on parallelization of code, and he was just starting to implement wide-field imaging in AIPS++. In addition to the personal tragedy, his death is a great loss for radio astronomy. Dan was only getting started on what I'm sure would have been a very interesting and productive career. I know that I'll miss his quick mind, willingness to get involved, and high standards – qualities that also showed up in his personal pursuits. One very typical memory that I have of him professionally is the Friday afternoon meetings on radio-astronomical imaging that we used to have at the Array Operations Center in the early nineties. In a very short time, he could pick up a new idea, reflect on it, fit it into his understanding, and then offer insightful comments, all while leaning laconically against the whiteboard. I miss the "to and fro" of the discussions in those meetings.

Dan, we'll all miss you.