AIPS++ Newsletter

April 1998

Figure 1: *Please note Dwingeloo Standard Time (DST) is -2:00 GMT

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

In this issue:

JIVE Use of AIPS++

Huib Jan van Langevelde, Chris Phillips, Harro Verkouter - JIVE, Dwingeloo

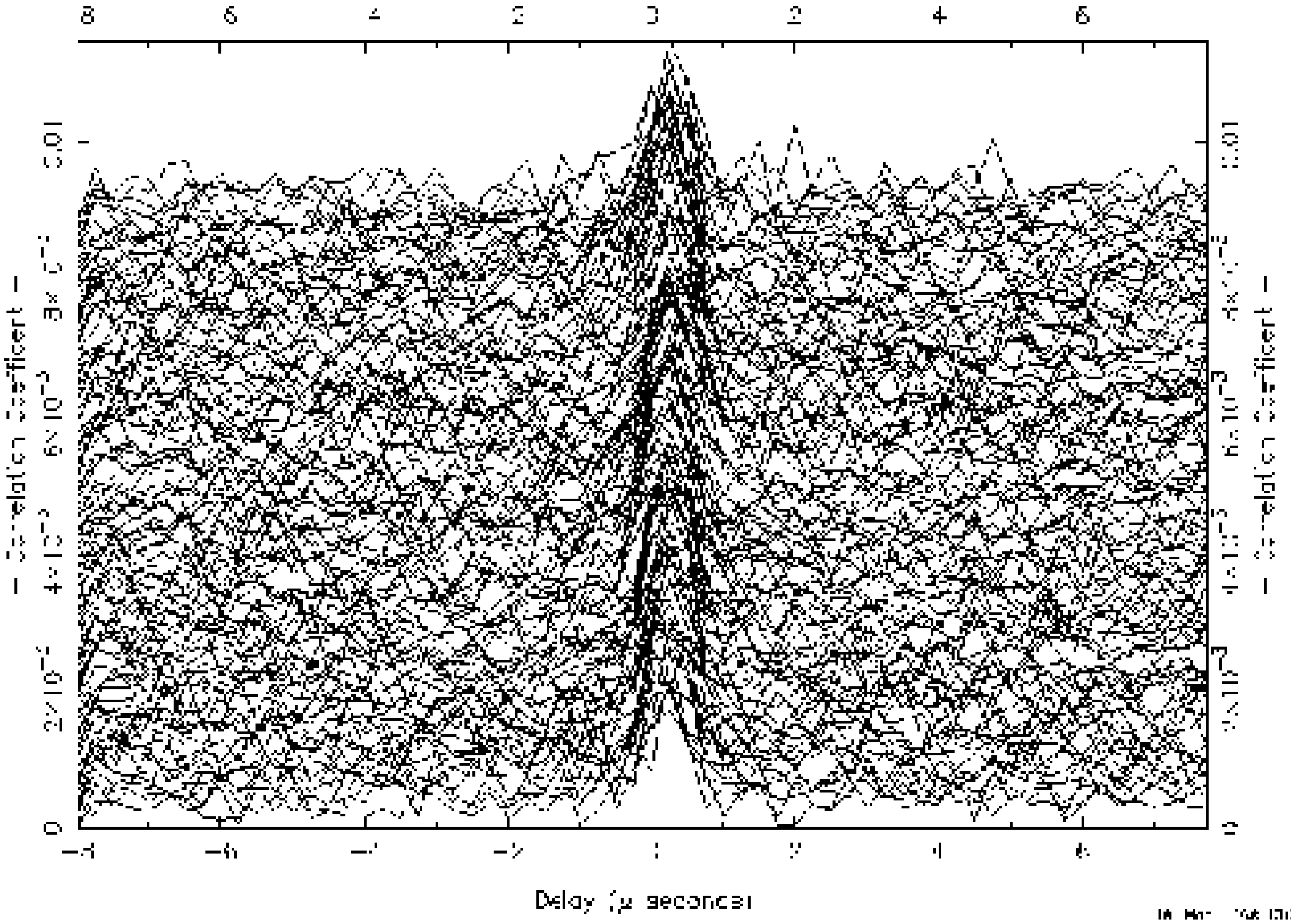
One of the main efforts at the Joint Institute for VLBI in Europe (JIVE) at Dwingeloo in the Netherlands is the construction of a 16 station MkIV data processor for the European VLBI Network (EVN). This correlator will process both European and global VLBI projects recorded in VLBA and MkIV format - with data-rates up to 1024 Mbit/s per station and 4096 spectral channels per baseline. The project is currently in a transition phase between construction and testing; first fringes were obtained in the Summer of 1997 and currently we are integrating hardware prototypes and software. A first imaging test, using the final hardware, is planned for the Summer of 1998.

It was decided to use AIPS++ as the platform for developing all post-processing software. This includes (pseudo) real-time display of results, data quality control, feedback to VLBI telescopes, preparation of calibration data, and formatting the product for the end user. For the latter it is assumed that most astronomical VLBI calibration initially will be carried out in "Classic" AIPS, and therefore users will need their data in `UVFITS` format.

One of the considerations in the decision to develop this "off-line" part of the software in AIPS++ is that we recognize it as the data reduction tool of the future. In particular, the idea that staff astronomers can work and program for the correlator with the same software they use for their scientific research is attractive, but the main consideration in streamlining the correlator output is speed. The expected data-rates are in the range of 50 kB/s to 3 MB/s. We decided to minimize any formatting or sorting operations in the output stage. We also wanted to avoid having to copy the raw data to another format for inspection. This seemed to rule out direct conversion to a standard format, or re-sorting the data for a specific package. AIPS++, on the other hand, offers an interesting capability by creating a special Storage Manager for the correlator output in its raw form.

At the moment we are still defining the correlator output format, ensuring that its content allows a direct mapping to an AIPS++ Measurement Set. This explains our interest in the current effort to come to an MS definition that is capable of completely describing VLBI measurements. As an important step in this project a "JIVE filler" was created that writes the current output data into an MS. This has proven a useful exercise not only to learn how to work in the AIPS++ software environment, but also to discover what data structures are required to create a proper data set.

Furthermore, it has given us a nice tool to inspect the data produced by



the correlator prototype. We have had the capability to study the data with the standard AIPS++ tools from the day after first fringes on 22 July 1997. Using mainly *Glish* scripts for mathematical operations, and plotting, it is possible to analyze the correlator output in the testing stage.

A typical example of such an exercise is displayed in the above figure. Data from a recent test using two separate tapes, played back synchronously for the first time (the first fringes were obtained with two MERLIN stations recorded on a single tape), is displayed as a time series of correlation functions. The absolute value of the correlator response over 64 lags is given, with every two second integration slightly offset in amplitude from the previous one. The data were recorded (with the new MkIV formatters) in

November 1997 and the baseline is between Effelsberg and Medicina. The bandwidth was 2 MHz, the observing band 5 GHz and the source 0016+731.

The data were filled into an AIPS++ Measurement Set which contained the output of 128 correlator chips (one eighth of the final capacity), each with 64 complex lags, integrated over 2 seconds. *Glish* scripts were used to select and display the data. The figure shows that with the dynamic control of the correlator implemented in the control software, we get a nice stable residual delay, which is only 250 nanoseconds from where it was expected. Another important measure in this figure is that the average correlation amplitude is approximately 0.002, which again is as expected.

Other simple diagnostics that are accessible with the standard AIPS++ tools are plots of the residual rate and comparison of the phase response of separate frequency bands. We have been able to quickly produce simple routines that, for example, compensate for the residual rate, average the data in time and Fourier transform to inspect the spectral response. The results can be compared with the autocorrelation spectra, which were also obtained by Fourier transforming the data.

What's New in AIPS++

Tim Cornwell and Brian Glendenning - NRAO, AOC

The following changes have been made to the development version of AIPS++ over the last month:

- The single dish analysis program, `Dish`, can now do regridding (via several different algorithms), and multiple operations can now be sequenced.
- A *Glish*-based visibility visualizer, `mans`, is available. This is currently most useful for WSRT data, but is being extended to work for Measurement Sets from other telescopes.
- `sky` received a number of new additions: deconvolution speed has been improved by using convolutions wherever possible; complex images

(e.g. RR, RL, LR, LL) can now be made; a state window has been added to the GUI; and the documentation has been reworked.

- Component models of the sky brightness can vary in spectral index and frequency.
- The `image` object can now work with transient image expressions.
- To improve on-line help for GUIs, a popup (transient) help mechanism has been defined and is now available from some GUI's, such as the `pgplotter`.
- The preferred plotting tool, `pgplotter`, acquired simple zoom and color map manipulation tools, and the general ability for users to customize a "Tools" menu. In addition, users can specify the minimum number of colors that are needed, and a private color map will be installed if necessary.
- The `measures` GUI has been much improved by a number of additions, including more on-line help (both pop-up and transient); and some new applications such as Doppler-related conversions.
- *Aipsview* has had a number of improvements: vector overlays are now available, the GUI has been improved; and 'Transient" image expressions can now be viewed.
- Some *Glish*-based least-squares fitting (`ls.g`) and general matrix routines (`matrix.g`) are available. *Glish*/`Tk` buttons can now have icons (visible in the `Dish` GUI).
- A flexible mechanism for executing functions periodically in *Glish* is now available (`timer.g`)

The Measures System - Part 1

Wim Brouw - ATNF, Epping

Background

The AIPS++ `measures` module contains two large groups:

- `Quantities`, i.e. values with units attached (10km/s)
- `measures`, i.e. astronomical entities (one or more `Quantities`) with a description of the reference frame they belong to (e.g. an epoch with a reference code 'LAST', an environment frame with the position on Earth, and an offset of the current sidereal date)

Both of these basic container groups can be manipulated and converted to other units and/or another reference environment (e.g. 10km/s to pc/a; the 'LAST' to a 'UTC'). This installment mainly discusses them as used in C++ programs.

Quantity

A `Quantity` originates from a `String` (e.g. `Quantity("10.5km/s")` or from a value and a `String` (e.g. `Quantity(10.5, "km/s")`).The basic structure of the `quantity String` is:

value[[—.—/][**unit**[**signed integer exponent**]]]**...**

A space or "." denotes multiplication, a "/" division. The following would all indicate the same acceleration:

- `5km/s2`
- `5km/s/s`
- `5km.s-2`
- `.005Mm2/Mm.s-2`

Each unit is composed of an optional SI prefix (from Y (yotta) to y (yocto), and u for micro) and a named unit. The basic unit is one of the defining SI units (m, kg, s, A, K, cd, mol, rad, sr) (note that kg is defined, but multiples are expressed in g, e.g. mg) or one of a set of SI and astronomical SI units (like Jy, AU, h), while some special units are available as well (including non-SI ones used in `FITS`). User specified units can be added to the known units as well. All known units can be viewed with `UnitMap::list()`. Some special formats can be used for dates (e.g. 1998/3/2), times (12:13), and angles (2:0:0, 30.0.0, 2h0m, 30d0m).

`Quantities` can be added and subtracted (if their units conform, no adding of apples and pears like s and m), multiplied and divided, and converted to other units with the convert (`to_unit`) member operation. Examples:

- `a = Quantity("10km/s")`
- `cout << a.convert("AU/a") << endl`
- `cout << a.convert("uJy") << endl`

will result in:

- `2.10949527 AU/a`
- `1e+36 uJy.m.kg-1.s`

The full range of possibilities can be viewed using the `measures` module help in the *aips package* at http://aips2.nrao.edu/aips++/docs/ aips/aips.html.

A more general, templated `Quantity` (`Quantum`) is available, allowing such constructs as `Quantum<INT>, Quantum<Vector<Float> >` and also `Vector<Quantum<Double> >`.

Measure

A `Measure` is an astronomical entity consisting of: a value (a `MeasValue` in AIPS++ jargon, e.g. an `MVDirection` containing direction-cosines derived from a right ascension and a declination); a reference environment, representing the `MeasValue`, and, if relevant, in which frame (when, where, in which direction, and at what velocity) it belongs. The environment can also contain an optional offset (the ability to specify only the time part of a full epoch, e.g. midnight today).

The following `Measures` are available at the moment or are being planned:

- - `MEpoch` – an instant in time (UTC, TAI, TDB, ...)

- - `MPosition` – a position on Earth (ITRF, WGS84, shortly more)
- - `MDirection` – a direction in space (J2000, B1950, HADEC, APP, AZEL, SUN, GALACTIC, JUPITER, ...)
- - `MRadialVelocity` – a radial velocity (LSR, GALACTO, GEO, TOPO, ...)
- - `MFrequency` – electromagnetic wave characteristics (LSR, ...)
- - `MDoppler` – radial velocity in special units (RADIO, OPTICAL, ...)
- - `MUVW` – under trial at the moment (J2000, ...)
- - `MEarthMagnetic` – Earth magnetic field (partly available) (IGRF, DIPOLE, MXDIPOLE, ...)

The reference codes are enumerated (e.g. `MDirection::VENUS`). The reference frame is created from `Measures`:

MeasFrame frame(MDirection(ra,dec), MEpoch(54321,92))

The full environment is specified using constructs like:

MDirection::Ref ref(MDirection::GALACTIC, frame)

resulting in a direction:

MDirection dir(long,lat,ref)

`Measures` as described above are basically containers: they can be used to describe the values used in an observation (they can also be stored in Tables for persistence with e.g. a `MeasurementSet`).

Since a `measure` contains all the information to describe it fully (including the, possibly global, frame ('when', 'where', ...)), it can be converted to any other reference code of the same type; e.g., if you know the UTC epoch, and the frame has an observatory position ('where') in it, this UTC can be converted to a LAST with a simple conversion engine:

- **Quantity tim**
- **Quantity::read(tim, ”today”)**
- **MPosition tel = MeasTable::Observatory(”ATCA”)**
- **MEpoch epo(tim, MEpoch::UTC)**
- **MEpoch::Convert engine(epo, MEpoch::Ref(MEpoch::LAST, MeasFrame(tel)))**
- **cout << MVTime(engine().getValue()) << endl**
- **// MVTime for formatting**

The `MEpoch::Convert` creates a conversion engine to convert from the input type to the output type (UTC to LAST). Once set-up it can be reused for similar calculations for speed reasons. Actual conversion is done with the `()` operator (empty argument indicates the input argument) as is shown in the output statement. The `MVTime` (and `MVAngle`) are special formatting (with specifiable format and precision) classes.

`Measure` conversions use the data (e.g. JPL tables, magnetic field components, dUT and polar motion) available from international organizations (e.g. IAU, IERS, IGRF). Standard conversions use the IAU nutation model with interpolation over an interval that can be specified to speed-up processing by default. However, the full JPL DE200 and DE405 tables can be used if higher precision needed. If lower precision suffices, the linear interpolation interval can be changed; IERS tables (with dUT and polar motion) can be bypassed if their precision is not warranted etc. All these selections are steered by the standard AIPS++ `aipsrc` mechanism (see the Reference manual).

To aid the user, an `Observatory` table is available on-line (add your observatory, or improve its coordinates), a source list (with the VLBI reference frame sources) and spectral line list (which at the moment contains only one element): send me your favourite spectral lines. The full capabilities are described in the `measures` module.

Apart from the C++ classes described above, the full capabilities of both the `Quantity` and `Measure` classes can be used from a `measures` GUI (invoked by `dm.gui()` on the *Glish* command line). A description of this GUI will appear in the next Newsletter.

PGPlotter

Brian Glendenning - NRAO, AOC

A new plot GUI has appeared in the last couple of months - `pgplotter.pgplotter` is essentially a replacement for `plotter`. In it's simplest modes it is very easy to operate:

- `include 'pgplotter.g'`
- `pg := pgplotter()` # Make the plot surface
- `x := 2*pi*1:1000/1000` # 0->2*pi
- `pg.plotxy(x, sin(x))`
- `pg.plotxy(x, cos(x), newplot=F)` # Plot on the same graph

This creates a GUI window, and plots sine and cosine curves on the same plot. Besides this simple interface to plotting, complete access to `PGPLOT`* style plotting is available. For those of you unfamiliar with `PGPLOT`, it allows you to:

1. plot lines, errors, and points in a variety of styles and colors;
2. display 2D ("image") data as color rasters or contours;
3. perform flexible annotation of axes or arbitrary plot locations;
4. obtain cursor feedback for creating interactive applications;

5. create multiple plots on a single page; and
6. many, many other displays with options.

The `pgplotter` adds a number of capabilities on top of "native" `pgplot` capabilities.

- Plot commands can be added or changed through a GUI. For example, if you want to change a line's color, or want to add an annotation, you can do this through the GUI. You can do this with plots that you create, or which are created for you by some application such `sky`.
- You can save the plot as a "plot" file, and come back to it in a later session and extend or change the plot. You can have multiple pages in a single plot file.
- You can preview a PostScript version of your plot, and send it directly to the printer from the GUI. The GUI displays the `x,y` value of the plot that the cursor is over,
- You can interactively zoom the plot, and change the color map of a raster display.
- When the `pgplotter` window resizes, the plot is redrawn to fit the new window size.

For *Glish* programmers, the "Tools" menu (which by default has the `zoomer` and `colormap` editor) can be extended. So, for example, it would be straightforward to add a Tool which read some data from a table, fit a line to it, and displayed the fit and the residuals. Also, there is a lower level `pgplotwidget` which has all the capabilities of `pgplotter`, but which can be embedded within a larger control panel.

The main deficiency of `pgplotter` is that we should offer more "high-level" commands, like `plotxy` shown above. I am interested in receiving any suggestions. However, `pgplotter` even now is powerful and straightforward to use. Try it, I think you'll like it.

---

*PGPLOT was developed and is maintained by Tim Pearson at tjp@astro.caltech.edu. PGPLOT is **copyrighted** software; *not* public domain. For further information see the PGPLOT home page at http://astro.caltech.edu/ tjp/pgpl. The AIPS++ PGPLOT widget is a variant of Martin Shepherd's *Tcl*/Tk PGPLOt widget at http://astro.caltech.edu/ tjp/pgplot/tkdriv.html. It was adapted for use with Rivet and AIPS++ through a collaboration between its author and Darrell Schiebel.

Developments in Calibration in AIPS++

Athol Kemball - NRAO, AOC

As part of the synthesis development plan, calibration facilities have been consolidated in a new distributed object (DO) called cal. This DO complements the new imaging capabilities available in sky.

Previously the imager module contained both imaging and calibration functions. The calibration features have been separated partly due to the development of new capabilities in this area, particularly concerning cross-calibration. The term is used here to imply the availability of methods to derive calibration information from a selected subset of u-v data, which can be transferred, interpolated and applied to another selected subset of u-v data. This work involves several steps, including the formalization of calibration table formats, access routines to the new calibration tables, and interpolation methods to be used as these calibration data are applied. These features are being released incrementally at the moment, but all will be available in the next beta release to allow basic cross-calibration.

The design of the calibration system rests on several objectives and principles. It is based on the measurement equation, which has been adopted in the AIPS++ project to model instrumental calibration errors in a general framework. The current design allows both initial and incremental calibration. In addition, a significant effort has gone into general interpolation methods to allow full user control of this important aspect of calibration. Support for different parametrization of the calibration matrices is allowed, as well as specialized solvers (such as for VLBI fringe-fitting). User programmability of the calibration system as a whole is a priority, and has been approached by modular decomposition of the system into structured units. A key problem that has been considered is maintaining history information regarding the origin of calibration tables, their associated initial calibration, and any further history notes recorded by the user.

Work in this area is very active at the moment. Further updates will be provided as these features are released for testing.

The Story of *Glish* - Part 2

Darrell Schiebel - NRAO, CV

For the most part, *Glish* itself, is the work of a very small group of people, two in fact, but it is an amalgamation both in design and in implementation. *Glish* is the *glue* which binds most of AIPS++ together, and it is also an amalgamation of code from several different places. In this installment we emphasize a key feature of *Glish*: the control of processes.

Glish is designed to allow users to tie many processes, called clients, together. Clients are the basis for extending and customizing *Glish*. They allow the AIPS++ group to take the basic generic *Glish* interpreter and create a data analysis system. Many people have worked on developing the clients which make *Glish* useful to astronomers. This flexibility makes *Glish* suitable for many different tasks, from telescope control to data analysis. *Glish*'s ability to act as a *glue* language is one of its key features.

AIPS++ developers use *Glish* to seamlessly integrate the clients they develop into the system. *Glish* is used to start these clients and to orchestrate the flow of information between the clients. The `sditerator` client is an example of how clients are used to extend and specialize *Glish*. This client forms the basis of the single dish package, `dish`, being developed in Charlottesville. The bulk of this application is developed in *Glish*, but much for this *Glish* code in `dish` is based on this `sditerator` client.

Here is an example of how this client is used to step through a data set:

- **iter := sditerator("harv")**

- **while (iter.more()) {**
- **record := iter.get()**
- **iter.next()**
- **}**

The first function called, `sditerator()`, starts the client and defines the interface for the user to communicate with the client. In this case, the data is contained in the `harv` data set stored on disk. The subsequent function calls access information from the client one record at a time.

The *Glish* interpreter and the clients communicate by sending events back and forth. Events are key/value pairs; the key is a string and the value is any *Glish* value, e.g. *float*, *array*, *string*, etc. The interpreter can deal with either synchronous or asynchronous events. Often, however, this event dialog is hidden beneath a function interface, as is the case above. *Glish* is the *glue* which ties all of the miscellaneous clients together.

Glish has benefited greatly from freely available source code. In particular, the regular expressions which were recently added are based upon code taken from *Perl* at http://www.perl.com/, the GUI elements which are part of *Glish* are from `TCL/Tk` at http://www.tcltk.com/, and the command line editing is based on a modified version of a publicly available library, *editline*. The wide availability of source code is simply a tremendous aid in developing systems such as *Glish*.

If you would like more specific information about how *Glish* is used, the *Glish* manual is available on-line at http://aips2.nrao.edu/aips++/ docs/reference/Glish/Glish.html, and there is also a tutorial at http://aips2.nrao.edu/aips++/ docs/notes/195/195.html.

Programmer's Corner - Episode 2

Brian Glendenning - NRAO, AOC

In our last installment you learned how to convolve two arrays. In this installment you will learn how to step through an image of any dimensionality "plane by plane", where the planes can be orthogonal to any cube face (e.g. XY planes, or YZ planes, or ...).

In *Glish*, a `for` loop merely picks one element at a time out of an array. For example:

- `for (i in 1:5) print i` # Prints 1-5
- `for (i in (1:5)*2) print i` # Prints 2,4,6,8,10
- `for (prime in [2,3,5,7]) print prime` # Prints the first 4 primes
- `for (word in ['hello', 'world']) print word` # Prints hello and world

A very useful function is `ind(x)` - it returns an array of indices that match `x`. This is very nearly the same as `1:length(x)`,except if `length(x)`is zero, `1:0` returns `[1,0]` whereas `ind(x)` returns `[]`, the empty array. For example:

- `x := []` # Empty
- `for (i in ind(x)) print i` # Does nothing
- `for (i in 1:length(x)) print i` # prints 1,0

Now then, how do you get all the planes out of an image? Getting a single plane is easy - you can use the `getchunk` method of the `image` object. `getchunk` allows you to specify the `blc` (bottom-left corner) and `trc` (top-right corner) of the pixels you are interested in from an image. You merely have to set `blc,trc` to get the plane you are interested in.

For example, given a `256(RA)*256(DEC)*11(FREQ)*1(STOKES)` image cube, you can get individual RA/DEC or DEC/FREQ planes as follows:

- `myimage := image('myimage')`
- `myimage.shape() [256 256 11 1]` # Length of each axis in image
- `pixels := myimage.getchunk (blc=[1,1,5],trc=[256,256,5])` # XY plane 5

- `shape(pixels) [256 256 1 1]` # Length of each axis in array pixels

- `pixels := myimage.getchunk`
  `(blc=[128, 1, 1],+ trc=[128, 256, 11])` # YZ plane 128

- `shape(pixels) [1 256 11 1]`

Note that in `getchunk` (and `putchunk`) unspecified values of `blc` default to the beginning of the axis (1), and unspecified values of `trc` default to last position of the axis. So, we could get all the XY planes in this image as follows:

- `for (plane in 1:11)` {

- `pixels :=`

- `myimage.getchunk([1,1,plane],[256,256,plane])`

- `... operate on pixels ...` }

While this is fine for stepping through a particular cube, suppose we want to be able to step through a cube of any dimensionality and shape? This is somewhat trickier, and moves us from the "advanced user" realm to the programmer realm. But since this is the "Programmer's Corner" let's press on!

First let us set up some general variables:

- `axis1 := 1; axis2 := 2` # YZ planes

- `shape := myimage.shape()` # Extract the shape from the image

- `numplanes := prod(shape)/(shape[axis1]*shape[axis2])`

The last statement probably requires a bit of explanation. It takes the product of the elements of the shape, in our example `256*256*11*1`, in other words the number of pixels, and divides it by the number of pixels in an XY plane, here `256*256` . This yields the number of planes in the image no matter what the dimensionality is (as long as t is greater than or equal to 2).

So far so good, it seems like we should now be able to iterate through our image plane by plane:

- `blc := array(1, length(shape));` # [1,1,...,1]
- `trc := shape`
- `for (plane in 1:numplanes)` {
- `blc := ????; trc := ????` }

Of course in 3D cases such as we have been using in our examples we have no problem, we merely would have to set the missing axis in `blc/trc` to "plane". However in 4 or more dimensions (e.g., spectral-line ppolarimetry) we need to turn the plane number ("plane") into two or more numbers in `blc/trc`. We will see how we can do this in our next installment.

Project News Summary

Tim Cornwell, AIPS++ *Project Manager - NRAO, AOC*

We are busy preparing the next, third, beta release of AIPS++. Current activities focus around internal testing of the system and writing documentation for users. In both these endeavors, we are aided by our group of hardy testers at the AOC and elsewhere. Our current estimate is that the release will be made in late May. The release will contain a large number of improvements and new capabilities, most of which have been described in the editions of this newsletter.

In other project news, our Scientific and Technical Advisory Group (STAG) met in Socorro on February 9 and 10. This was the second meeting of the group, the first being back in November 1996. The group represents the various organizations comprising the AIPS++ Consortium and is chaired by Robert Braun of NFRA. The report of this meeting and the Project response are to be found as AIPS++ Notes at http://aips2.nrao.edu/aips++/docs/notes/217/217.html and at at http://aips2.nrao.edu/aips++/docs/notes/218/218.html.

AIPS++ is a project of the AIPS+ Consortium at
http://aips2.nrao.edu/aips++/docs/project/consortium.html