Volume 1 - Basic Tools

AIPS++ Group, Editor: Neil Killeen

October 8, 2002

Contents

Chapter 1

Plotting

Jim Braatz

1.1 Summary

This chapter provides an overview of the PGplotter `tool`, primarily by providing several straightforward examples which can be used to learn the basic line plotting capabilities. An example of drawing a simple raster plot is also shown, although images and arrays are better displayed in using the Viewer `tool`.

The aim of this document is to get users started with simple plotting. The User Reference Manual is the primary source of information about `tools`, and it includes comprehensive details on the PGplotter and the PGplotwidget. The Glish User Manual contains details on the native PGPLOT bindings in *Glish*, which implement all of the basic functions available in the PGPLOT code library.

The plotting hierarchy is that the PGplotter `tool` provides a standalone plotting canvas. It is built with the PGplotwidget `tool`, which must be embedded in some parent frame (provided in this case by the PGplotter). In turn, the PGplotwidget `tool` is implemented via the native *Glish* PGPLOT function calls.

Thus, a PGplotwidget `tool` has access to all of the native *Glish*PGPLOT functions. In turn, the higher-level PGplotter `tool` has access to all functions that its PGplotwidget `tool` has.

When using a PGplotter or PGplotwidget `tool`, to access the native *Glish* PGPLOT routines, it is only required that the '->' be replaced by '.', as shown in the examples which follow.

1.2 Creating a PGplotter `tool`

A PGplotter `tool` is created from the *Glish* prompt as follows:

```
include 'pgplotter.g'
pg:=pgplotter()
```

The specific PGplotter `tool`, **pg**, is now ready to accept plotting instructions. The following complete example shows how to make a simple line plot:

```
include 'pgplotter.g'
#
x := 1:100
y := sin(x/5)
pg:=pgplotter()
pg.plotxy1(x,y,'X','Y','Sample')
#
y := cos(x/3)
pg.plotxy1(x,y)             # Adds to plot with new colour
```

1.3 Interacting with the PGplotter

The menu bar on the PGplotter window created by the above example includes the File, Tools, Edit, and Help items.

Under the File item are the selections:

- Save - allows you to save the plot either as a series of PGPLOT commands (these are stored in an AIPS++ table whose name is written to the logger).
- Open - allows you to recreate a plot previously saved as PGPLOT commands with the Save selection. When Open is selected, a file browser will appear. Simply select the desired plot file from the file browser and click Send and Dismiss.
- Print - activates the standard AIPS++ printer GUI. It first saves the file as a postscript file which you can then spool to a printer or preview with *ghostview*.

  If you expect to print only a single page reflecting the current display, it is recommended that you press Clear on the PGplotter, issue the plot commands, and then send the plot to the printer. Otherwise, all plot commands stored in the buffer will be printed, possibly resulting in multiple pages of output.

Under the Tools item are the selections:

- Zoom - Selecting zoom will pop up a new window which gives a Reset Full View button. Press this button to unzoom the plot to its original scale. To zoom the plot, drag out a box with the left mouse button. When the plot is zoomed as required, the zoom `tool` can be dismissed with the button on the zoom popup window.

- Colormap - This selection allows you to modify a color table which has previously been defined by a call to the PGplotter function **ctab** (a native PGPLOT command). If you have not called **ctab** in your plot, this selection has no effect.

- Dismiss - dismisses the GUI (but does not destroy it). The GUI can be recovered with the PGplotter.gui function.

- Done - destroys the GUI and the PGplotter `tool`.

Under the Edit item are the selections:

- Edit - This allows for interactive modification of the displayed plot. Selecting this will create a new GUI for editing the plot commands. The plot created by the above example is drawn with a series of 11 commands which are shown under the Drawlist column. To change the color of the line plot from red to green, for example, find the sci entry closest to the bottom of the drawlist. Select it with the left mouse button and change the ci entry from 2 to 3, then hit return and the plot will be updated. A clipboard is available to simplify any rearrangement or additions to the drawlist commands. To add new commands, the Add to Drawlist and Add to Clipboard options can be used. Simply select the command to be added from the list given and press the appropriate button. The new commands can then be seen at the bottom of the Drawlist or Clipboard list.

Under the Help menubar item are the standard AIPS++ help items, which can be used to point your web browser to appropriate help pages, report defects, and ask questions of the AIPS++ staff.

A one line message area appears just below the menu bar. Instructions for using various PGplotter tools (e.g. zoom) may appear here. Below that is the canvas for the plots.

At the bottom are buttons Save, Print, Clear, Dismiss and Done. The first and last two are short-cuts for the same operations under the menubar item File. The Clear button clears the display and empties the display list. With the Print button, you can type in the name of the file in the widget next to that button.

The PGplotter commands are most easily learned from examples. The following examples show some common plot types. For a comprehensive list of the PGplotter capabilities, see the User Reference Manual and the Glish User Manual.

1.4 Example: Reading Data from an ASCII Table and Making Plots

Suppose you have a file named *SampleData.txt* which contains the following data:

```
0.1  0.248   1.355
0.2  0.596   1.306
0.3  0.634   1.569
0.4  0.877   1.312
0.5  0.874   0.871
0.6  1.182   0.550
0.7  1.088   0.390
0.8  1.169   0.261
0.9  1.157   0.086
1.0  1.075   0.171
1.1  0.935   0.250
1.2  0.798   0.411
1.3  0.805   0.449
1.4  0.504   0.530
1.5  0.307   0.552
1.6 -0.035   1.004
1.7 -0.172   1.317
1.8 -0.232   1.648
1.9 -0.488   1.567
2.0 -0.440   1.291
```

Then the data can be read and plotted using the following script:

```
include 'table.g'
include 'pgplotter.g'
#
t := tablefromascii('sample.tbl','SampleData.txt',,T)
t.colnames()
#<< Column1 Column2 Column3
x  := t.getcol('Column1')
y1 := t.getcol('Column2')
y2 := t.getcol('Column3')
pg := pgplotter()
pg.plotxy1(x,y1)
pg.plotxy1(x,y2)
#
t.done()                     # Destroy tools when finished
pg.done()
```

The **tablefromascii** command (actually a constructor that constructs a Table `tool` called **t**) serves to convert the ASCII data into an AIPS++ table, which is written to disk under the name *sample.tbl*. The disk table can subsequently be reaccessed by connecting it to a Table `tool` via the constructor `t := table('sample')`.

The next *Glish* statement in the example is useful during interactive sessions to list the names of the columns in the table. In this case, table **t** has columns named *Column1, Column2,* and *Column 3* (the output is not shown in the example script), which are the default names given by **tablefromascii** when no header information is provided with the ASCII table.

The first column of the ASCII file is interpreted as the abcissa (and stored in the *Glish* variable `x`) and the numbers in the next two columns are plotted against it.

Both columns from the ASCII table are plotted on a single set of X-Y axes. The function **plotxy1** can be called repeatedly to add new lines on top of the existing plot, and the scaling of the axes will be adjusted accordingly.

To plot data on the same plot but using an alternate scaling on the X-Y axes, the function **plotxy2** can be used. For example, the following lines can be appended to the example to plot new data without altering the original X- and Y-scales.

```
xnew := 1000:2000
ynew := 100*sin((xnew-1000)/100)
pg.plotxy2(xnew,ynew)
```

Several high level PGplotter calls, including **plotxy, plotxy1, plotxy2, timey, and timey2** are intended to provide a way to take a quick look at a user's data. The functions **plotxy1** and **plotxy2** are intended to be used together and may not mix smoothly with **plotxy** or primitive PGPLOT calls in all cases. Similarly, **timey** and **timey2** are meant to work together. When a sophisticated plot is required, it may be necessary to stick to the primitive function calls.

1.5 Example: Using the PGPLOT primitives

Chapter 12 of the Glish User Manual describes using direct PGPLOT bindings with *Glish*. A PGplotter `tool` is not required to use those direct bindings. However, PGplotter `tools` do support those methods. The only difference being that events described in the Reference Manual are replaced by functions when using the PGplotter. In practice this amounts to the '->' being replaced by '.'. For example, to set the color index use `pg.sci(3)` rather than `pg->sci(3)`. The following example shows a simple plot drawn with the primitive PGPLOT commands.

```
include 'pgplotter.g'
#
tmp := -100:100
x:=tmp/10
y:=sin(x)*exp(x/5)
#
pg:=pgplotter()
pg.env(-10,10,-7,7,1,0)
pg.lab('X Axis','Y Axis','A Clever Plot')
pg.sci(3)
```

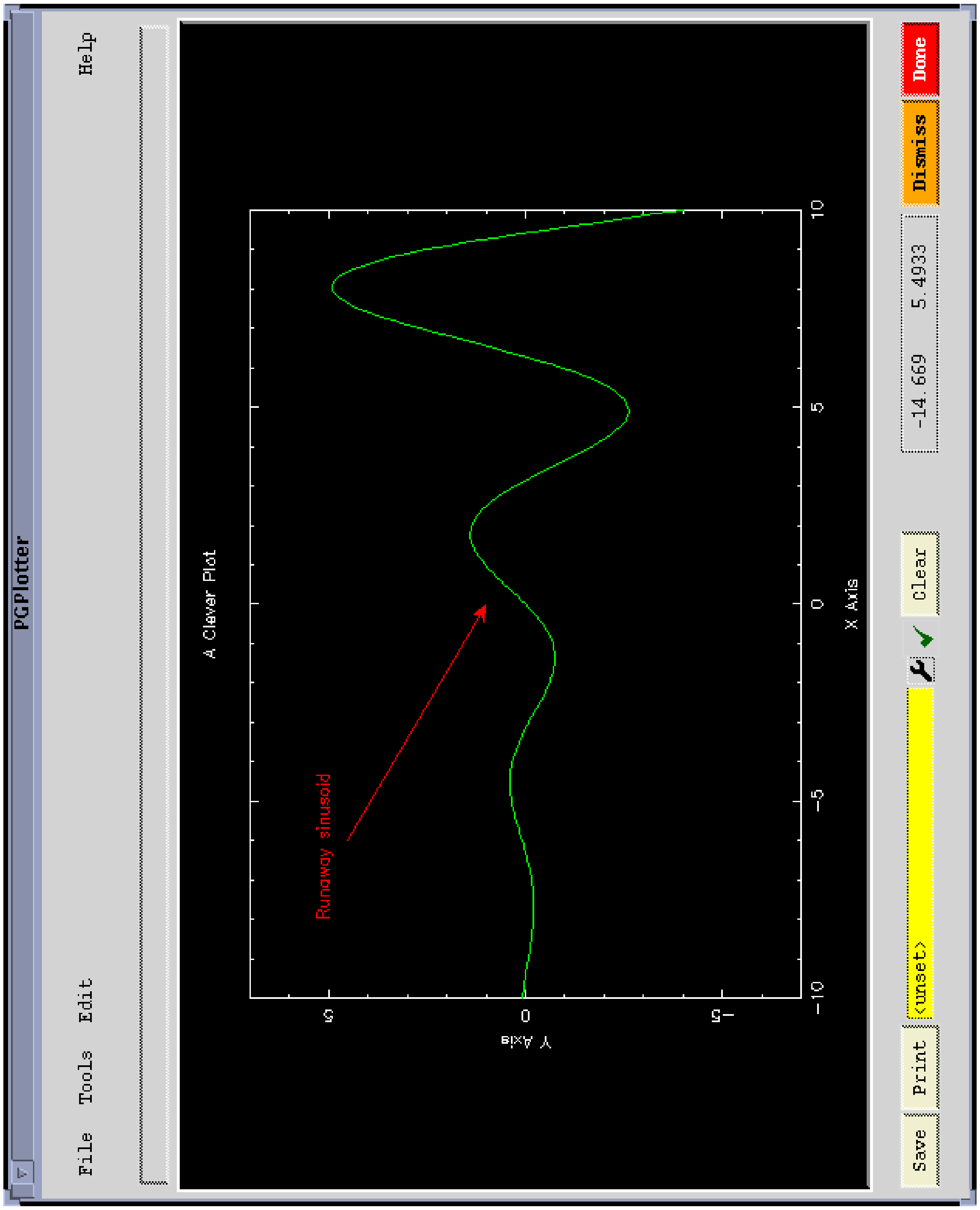
```
pg.line(x,y)
pg.sci(2)
pg.ptxt(-8,5,0,0,'Runaway sinusoid')
pg.arro(-6,4.5,0,1)
```

Consult the Glish User Manual for the comprehensive list of PGPLOT routines available.

1.6 Example: Plotting to Several Panels in a PGplotter Window

The following example shows how several plots can be drawn to individual panels on a single page. It also illustrates a simple method of plotting an image (represented as a simple *Glish* array) to the PGplotter window.

```
#
# Create some data to plot:
#
x1 := 1:100; y1 := (x1/10)^2
x2 := 1:50; y2 := log(x2)
x3 := -1000:1000
y3 := sin(x3/100)*cos(x3/100)
im := array(0,100,100)
for (i in 1:100)
 for (j in 1:100)
  {
  im[i,j] := i/50+sin(j/10)+random(0,10)/10
  }
#
# and plot the 4-panels:
#
include 'pgplotter.g'
pg := pgplotter()
pg.subp(2,2)
pg.env(min(x1),max(x1),min(y1),max(y1),1,0)
pg.line(x1,y1)
pg.env(min(x2),max(x2),min(y2),max(y2),0,0)
pg.line(x2,y2)
pg.env(min(x3),max(x3),min(y3),max(y3),0,0)
pg.line(x3,y3)
pg.env(1,100,1,100,1,0)
pg.gray(im,-1,4,[0,1,0,0,0,1])
```


PGPlotter
File
Tools
Edit
Help
A Clever Plot
Runaway sinusoid
X Axis
Y Axis
-14.669
5.4933
Save
Print
<unset>
Clear
Dismiss
Done

Chapter 2

Processing Tabular Information

Tim Cornwell

2.1 Summary

This chapter provides an overview of how to process tabular data with standard `tools` and *Glish* capabilities.

The major modules and `tools` that are required are:

- *Glish* - as the command line processor, for reading and writing files, and for various processing operations.

- Table `tool` - is for reading, writing, accessing and browsing tables. Conversions can be made to and from ascii formats, and from FITS binary tables. The Table `tool` supports selections and queries using the Table Query Language (TaQL).

- Pgplotter `tool` - is used to plot data using an interface to the PGPLOT graphics package.

- Measures & Quanta `tools` - are useful for various conversions.

- mathematics module - has various `tools` for mathematical operations such as statistics, fitting, fourier transforms, etc.

The User Reference Manual contains the primary documentation for all these `tools` and their functions. The aim of this document is to provide pointers to the relevant chapters and examples.

2.2 The basics

Most data in AIPS++ are stored in tables. A table consists of an unlimited number of columns of data, with optional column keywords and optional table keywords. See the table module documentation for details.

There are different flavours of tables, e.g. Measurement Sets holding visibilities, Images, calibration tables and so on. The flavour difference simply means that these tables have (different) specific required entries and there are high-level `tools` which understand these differences. But fundamentally, all these tables use the same underlying data structure, which can be accessed via a generic Table `tool`. A Table `tool` provides you with direct access to a table, no matter what flavour it is. You can recover, manipulate and display the data with it.

As well as using a Table `tool`, one can also manipulate tabular information (however it is stored) with *Glish* itself. Examples are given below.

2.2.1 Reading and writing

You have a number of options for reading your data into AIPS++.

- Use the intrinsic *Glish* input and output operations to read either ASCII or binary files.
- Use the Table `tool` constructors tablefromascii or tablefromfits to read into a Table `tool`.
- There are specialized readers for certain formats and purposes. For example, to read NVSS catalog information from the NVSS catalog search pageinto a componentlist, you can use the constructor asciitocomponentlist. There are similar readers for FIRST and WENSS.

Reading into a table is often the best route either if you have lots of data or you want to do things like selections and queries on the table. Reading directly into *Glish* variables is adequate otherwise, and is more convenient if the data needs some preprocessing.

To write information out in an ASCII file, the basic route is to use the table.toascii function.

2.3 Examples

2.3.1 Example 1: Reading from an ASCII file into a Table

As our first example, let's read from an ASCII file into a table. Our ASCII data file, called *SampleData.txt* contains three columns:

```
0.1  0.248   1.355
0.2  0.596   1.306
```

```
0.3  0.634   1.569
0.4  0.877   1.312
0.5  0.874   0.871
0.6  1.182   0.550
0.7  1.088   0.390
0.8  1.169   0.261
0.9  1.157   0.086
1.0  1.075   0.171
1.1  0.935   0.250
1.2  0.798   0.411
1.3  0.805   0.449
1.4  0.504   0.530
1.5  0.307   0.552
1.6 -0.035   1.004
1.7 -0.172   1.317
1.8 -0.232   1.648
1.9 -0.488   1.567
2.0 -0.440   1.291
```

We can read these into a Table `tool` according to:

```
include 'table.g'
tbl := tablefromascii('sample.tbl','SampleData.txt', autoheader=T)
tbl.browse()
#
tbl.done()               # Destroy tool and free up resources when finished
```

Here we have created a AIPS++ table on disk called *sample.tbl* and filled it with the data. Note the use of the `autoheader` function argument to instruct the system to automatically work out the data type of each column. We then invoked the **browse** function of this `tool` to look at it. The figure below shows you what you would see.

There are other features of the **browse** GUI which you can use to examine the data. For example, under the Table item of the menu bar you will find a Plot selection which enables you to make simple plots of the data.

2.3.2 Example 2: Reading, fitting and plotting from an ASCII file

This example shows how to read the file containing the number of lines of code in AIPS++ and converts it into a plot. The file containing the lines of code is at ftp://aips2.nrao.edu/../linecount/source_lines.

This example uses basic *Glish* I/O rather than a Table `tool`, and includes writing some ad-hoc support *Glish* code. This is to show you the sort of flexibility *Glish* offers for this kind of processing.

The first 10 lines are:

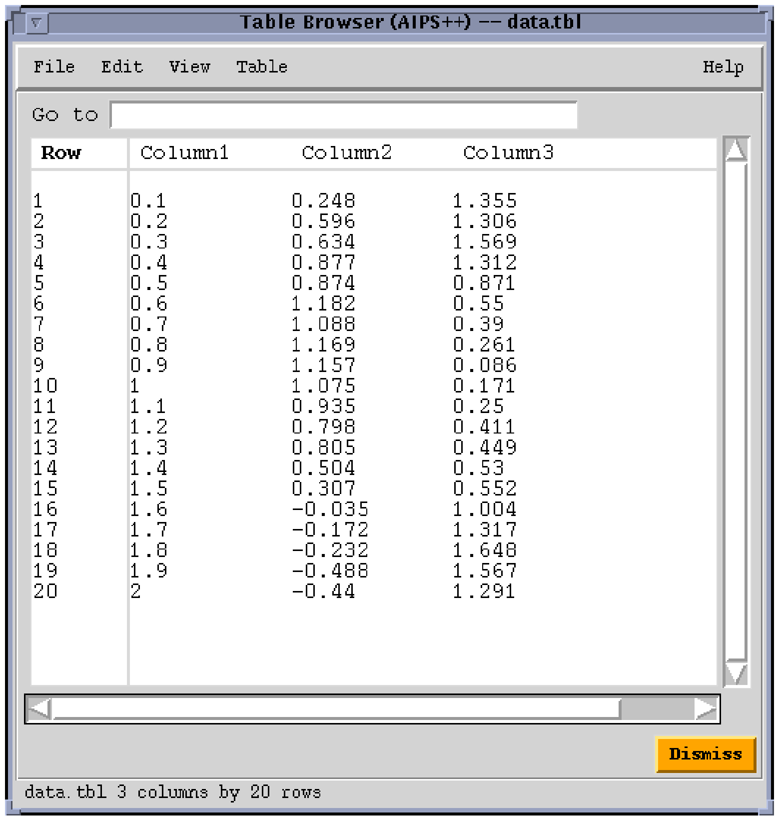


Figure 2.1: The Table browser display

```
Wed Nov 25 14:41:51 GMT 1992 29830
Tue Dec 1 01:08:02 GMT 1992 36557
Tue Dec 1 12:55:01 GMT 1992 36699
Wed Dec 2 12:55:01 GMT 1992 36951
Thu Dec 3 12:55:01 GMT 1992 36965
Fri Dec 4 12:55:01 GMT 1992 36965
Sat Dec 5 12:55:01 GMT 1992 36966
Sun Dec 6 12:55:01 GMT 1992 36966
Mon Dec 7 12:55:00 GMT 1992 36966
Tue Dec 8 12:55:00 GMT 1992 36966
```

First we need to define a function which will convert date strings (1999, Sept, 12) to a fractional numerical date 1999.69589.

```
include 'quanta.g'
toyears := function(yy, mm, dd)
{
```

```
  s  := spaste (dd, mm, yy)
  d  := dq.quantity(s)                      # Date
#
  s  := spaste ('01Jan', yy)
  d0 := dq.quantity(s)                      # Reference date
#
  offset := dq.getvalue(d) - dq.getvalue(d0)    # Numerical offset
  nyear := as_float(yy) + offset/365.0          # Result
#
  return nyear
}
```

Now we can read the file and apply some rudimentary checks as we go along:

```
parts := split(dq.time('today', form='dmy'), '-')
today := toyears(parts[3], parts[2], parts[1])

# We need some variables to hold the results
loc := [=]
loc.date := []
loc.size := []
nlines := 0

# Open the file containing the source lines and read a line
f:=open('< source_lines')
line := read(f)

# While there is something in the line, process it
while(sum(strlen(line)) > 0) {
  nlines +:= 1
  parts := split(line)
  loc.date[nlines] := toyears(parts[6], parts[2], parts[3])
  loc.size[nlines] := as_float(parts[7])

  # Simple post-hoc editing
  if((nlines>1)&&(loc.size[nlines]<0.5*loc.size[nlines-1])) nlines -:= 1
  if(loc.date[nlines]>today) nlines -:= 1
  if((parts[2]==19)&&(parts[3]==Jun)&&(parts[6]==1999)) nlines -:= 1
  # Get the next line
  line := read(f)
}

# Sort sizes on dates, and sort dates
rec.y := sort_pair(loc.date, loc.size)
rec.x := sort(loc.date)
```

To make a plot, we can use a Pgplotter `tool`:

```
include 'pgplotter.g'
p:=pgplotter()
p.plotxy(rec.x, rec.y, T, T, 'Date', 'Size (Lines of LOC)', 'Lines of Code in AIPS++')
```

Suppose that we now want to make a fourth order fit and plot the result. We'll fit to the logarithm of the lines of code to ensure that the prediction is positive. We use a Polyfitter `tool` to do the fit and the evaluation of the fit for the given dates.

```
include 'polyfitter.g'
pf:=polyfitter()
coeff:=[]; coefferrs:=[=]; chisq:=0; ypred:=[=];
#
pf.fit(x=rec.x, y=ln(rec.y), coeff=coeff, coefferrs=coefferrs, chisq=chisq,
         sigma=0.01, order=4)
pf.eval(x=rec.x, coeff=coeff, y=ypred)

p.sci(3)                        # Green
p.pt(rec.x, exp(ypred), -1)

# Destroy tools to free up resources when finished
pf.done()
p.done()
```

The plot looks like this:

2.3.3 Example 3: Reading, fitting and plotting from an ASCII file

Each developer in AIPS++ has a list of software targets which is tracked through the development cycle. In this example, we read a list of developer target dates into an AIPS++ table.

Here are a few lines from such a list. Each entry is a string.

```
"Filler for GBT spectral processor" "Bob Garwood" "1999/09/10" "1999/09/10"
"SD imaging via imager" "Bob Garwood" "1999/09/13" "1998/08/17"
"Initial calibration in dish" "Bob Garwood" "1999/09/13" "1999/03/01"
"Tests of mosaicing in imager" "Mark Holdaway" "1999/09/13" "1998/12/31"
"Lattice review" "Peter Barnes" "1999/09/13" "1999/02/01"
"Review Coordinate classes" "Tim Cornwell" "1999/09/13" "1999/02/01"
```

We can convert this file to an AIPS++ table in the same way as the first example. We instruct the constructor to determine the header information (*e.g.* type and name of each column) automatically.

```
include 'table.g'
tbl := tablefromascii(tablename='aipspptd.tab', asciifile='aipspptd.csv',
                      autoheader=T)
tbl.browse()
```

When we browse the table, we see that it has four columns called Column1, Column2, Column3, Column4. To give these better names, we can write a short header file, let's call it *aipspptd.hdr*.

```
Target Person TargetDate OriginalDate
A A A A
```

If we now do the following, we'll get the desired names:

```
include 'table.g'
tbl := tablefromascii(tablename='aipspptd.tab', asciifile='aipspptd.csv',
                      headerfile='aipspptd.hdr')
tbl.browse()
```

This screen shot shows what the browser would show you:

For more information on the capabilities of the constructor see tablefromascii.

We can now use the Table Query Language (TaQL - see also Note199 and a GUI interface via taqlwidget) to select on the values in the columns. For example to see all entries that belong to 'Tim Cornwell', we do:

```
subtbl1 := tbl.query('Person==\'Tim Cornwell\'')
subtbl1.browse()
```

Note that we had to "escape" the single quote around the string 'Tim Cornwell' to prevent it being interpreted by *Glish* as the end of string.

To get all targets whose TargetDate has already passed:

```
subtbl2 := tbl.query('TargetDate<DATE()')
subtbl2.browse()
```

The TaQl `DATE()` function returns the current date when given no arguments. One could also have said:

```
subtbl2 := tbl.query('TargetDate>DATE(12Sep1999)')
subtbl2.browse()
```

to find all the target dates after 12Sep1999 (all but the first target).

To plot the time delays between the original target completion date and the current target completion date, we need to get the two date columns from the table, convert to days, and subtract. We can then use the histogram capability of a Pgplotter `tool`to display the histogram:

```
tdates := tbl.getcol('TargetDate')
odates := tbl.getcol('OriginalDate')
delay := []
for (i in 1:length(tdates)) {
   t1 := dq.getvalue(dq.totime(tdates[i]))         # Convert date strings to numeric time in days
```

```
   t2 := dq.getvalue(dq.totime(odates[i]))
   delay[i] := t2 - t1
}
#
include 'pgplotter.g'
p := pgplotter()
p.env(min(delay), max(delay), 0, 20, 0, 1)
p.hist(delay, min(delay), max(delay), 30, 1)
p.lab('Delay in days', 'Number', 'Histogram of target delays')
#
# Destroy tools to free up resources when done
#
p.done()
subtbl1.done()
subtbl2.done()
tbl.done()
```

2.3.4 Example 4: Editing Tables From AIPS

In this example, a gain table associated with a visibility data set is written out of AIPS and read into AIPS++ for inspection and manipulation.

1. Calibrate the data set in AIPS.
2. Write gain table out of AIPS as an ASCII file.
3. Read ASCII gain table into AIPS++.
4. Manipulate the gain table in AIPS++.
5. Read ASCII gain table back into AIPS.

Begin by calibrating the data in AIPS (using the AIPS task **CALIB**) and obtaining a calibration table (SN or CL). There are two tasks in AIPS to write out AIPS tables, **PRTAB** and **TBOUT**. **PRTAB** outputs a limited number of significant digits and should *not* be used; **TBOUT** allows for the output of double precision numbers and should be used. The **DOCRT** verb, which specifies the maximum width of the column in characters should be set to something large, such as 10000. Execute the **TBOUT** task to create the output ASCII table file and then copy the file into a directory where you will run AIPS++. The following is a typical input to **TBOUT**:

```
AIPS 1: TBOUT:  Task to Write AIPS tables to text files.
AIPS 1: Adverbs         Values           Comments
AIPS 1: ----------------------------------------------------------------
AIPS 1: USERID       0                    Image owner ID number
AIPS 1: INNAME     '10/11/96    '         Image name (name)
AIPS 1: INCLASS    'X BAND'               Image name (class)
```

```
AIPS 1: INSEQ         1                     Image name (seq. #)
AIPS 1: INDISK        4                     Disk drive #
AIPS 1: INEXT       'SN'                    Extension type
AIPS 1: INVERS        1                     Extension file version #
AIPS 1: OUTFILE    'DR21_SN.ASCII           External text file name.
AIPS 1:
AIPS 1:               '
AIPS 1: DOCRT         1                     Max width of output (char)
AIPS 1: BCOUNT        1                     Begin Table record to Write
AIPS 1: ECOUNT        0                     End Table record to Write
```

After `TBOUT` has completed, start up AIPS++ and read the ASCII table, using the Table `tool` **tablefromascii** consructor.

```
include 'table.g'
tbl := tablefromascii('DR21_SN.tbl', 'DR21_SN.ASCII', readonly=F)
```

Assume that antenna 5 has significant fluctuations in its gain and you want to replace the determined gains with a 5th order polynomial fit to smooth out the variations. The Table module allows for data to be read into and out of *Glish* variables. The Mathematics module contains the functionality to carry out numerical operations, such as polynomial fits (Polyfitter.fit) as well as evaluation the resulting fits (Polyfitter.eval). The following session shows how to carry out the procedure described above.

```
include 'polyfitter.g'
tbl2 := tbl.query("ANTENNA==5")               # Creates a reference table holding only ANTENNA 5
#
x := tbl2.getcol("TIME")                      # Get data
yin := tbl2.getcol("REAL1")
#
pf:=polyfitter()                              # Make fitter and fit
local coeff, coefferrs, chisq0, ypred, errmsg, yout
pf.fit(coeff, coefferrs, chisq, errmsg, x, yin, order=5)
pf.eval(yout, errmsg, x, coeff)
#
ok := tbl2.putcol("REAL1", yout)              # Put data back into reference and hence parent table
#
tbl2.done()
tbl.done()
pf.done()
```

You can look at the gains as a function of time (both raw data and polynomial fit) by using the **browse** function.

The modified table can now be read back into AIPS using the AIPS task **TBIN**.

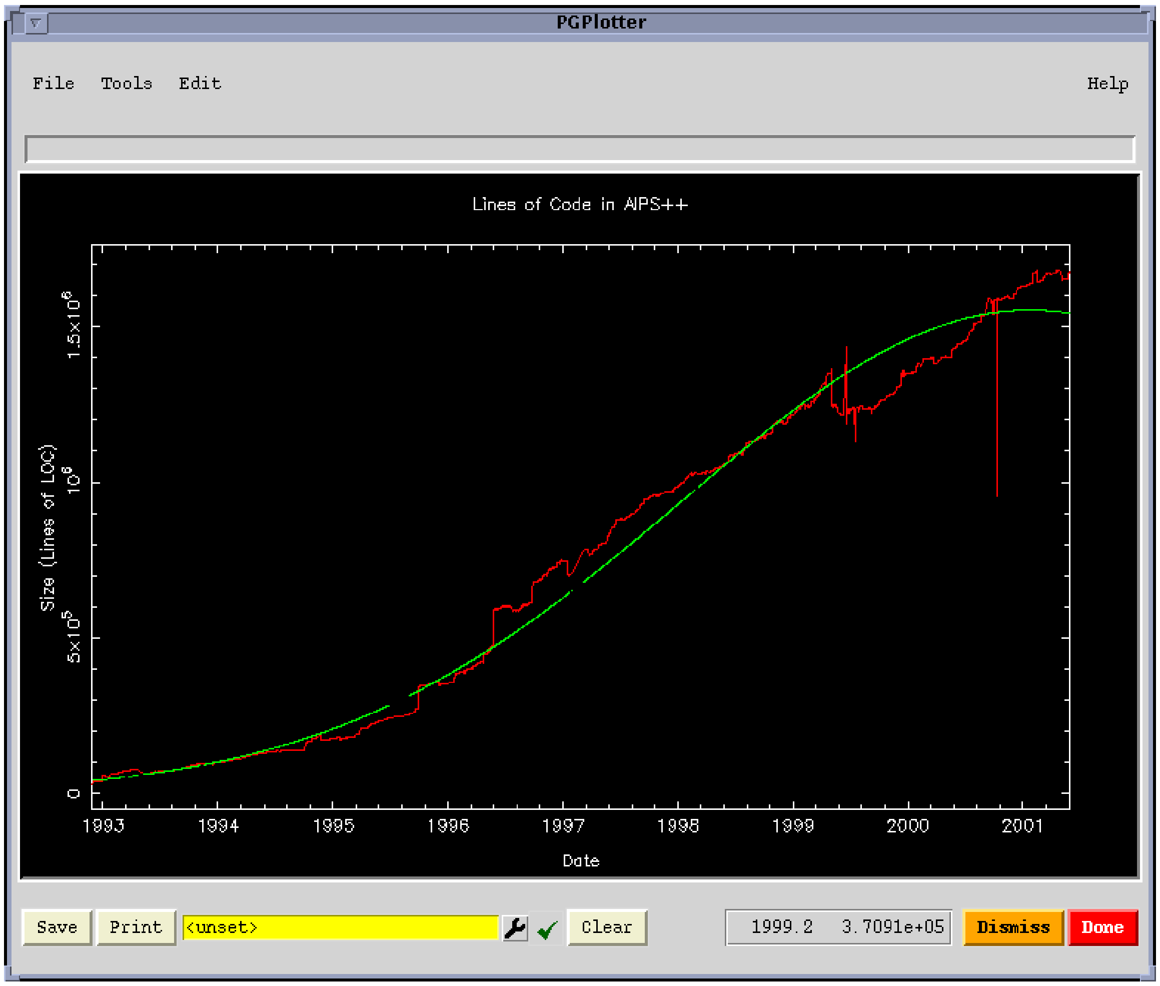
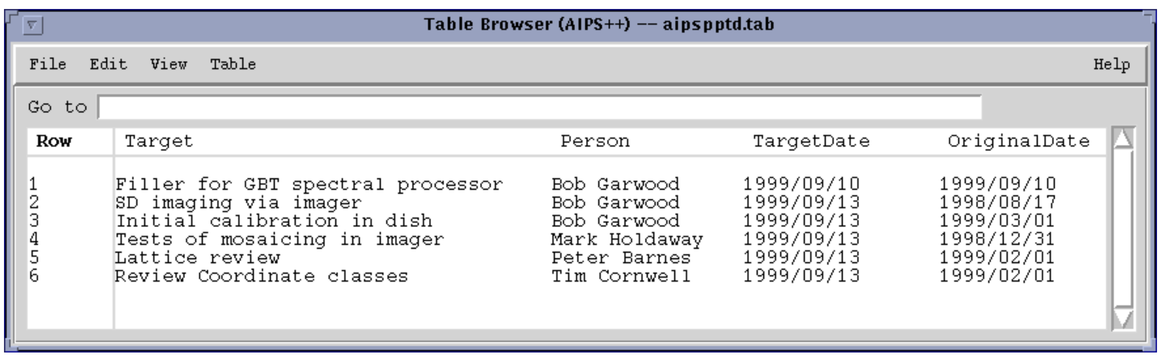


Figure 2.2: Lines of code in AIPS++



Row	Target	Person	TargetDate	OriginalDate
1	Filler for GBT spectral processor	Bob Garwood	1999/09/10	1999/09/10
2	SD imaging via imager	Bob Garwood	1999/09/13	1998/08/17
3	Initial calibration in dish	Bob Garwood	1999/09/13	1999/03/01
4	Tests of mosaicing in imager	Mark Holdaway	1999/09/13	1998/12/31
5	Lattice review	Peter Barnes	1999/09/13	1999/02/01
6	Review Coordinate classes	Tim Cornwell	1999/09/13	1999/02/01

Chapter 3

Display of Data

David Barnes, Malte Marquarding & Neil Killeen

3.1 Summary

This chapter describes how to display data with the Viewer `tool`. You can display images, *Glish* arrays, measurement sets, and Tables holding source catalogs in a specific format. In the future, display of generic Tables will also be available.

If you are impatient and don't want to know anything about the Viewer, you should look at this 3.13.

There are three different ways to view data using the Viewer `tool`. The first approach is strictly using graphical user interfaces (GUIs). The second approach shows you how to create Viewer based applications via *Glish* scripting. The third approach is a combination of these two using the Image `tool`.

The Viewer `tool` is available from the *Glish* command line interface (CLI) and from the Toolmanager GUI. Detailed documentation describing the full interface of the Viewer `tool`is given in the Viewer module of the Reference Manual. The Viewer `tool` is made available in *Glish* with the command

```
include 'viewer.g'
```

It will be assumed in all of the code fragment examples that follow, that you have issued this command.

For working through the procedures in this chapter without an image or a *Glish* array at hand, you may use the following code to generate an example of each:

```
ar := viewermaketestarray()
im := imagemaketestimage('testimage.im')
```

Now you have a *Glish* array in `ar` and an Image `tool` in **im**. If you are interested in a sample source catalog to overlay on your images, you can open the `Sources` table used in the Measures tool as follows:

```
msc := vieweropentestskycatalog()
```

Now you have a table of sources accesses via the Table `tool`, **msc**.

3.2 Viewing a raster map of an image or *Glish* array

It is straightforward to display a raster map of an image or *Glish*array. A raster map represents pixel intensities in a two-dimensional cross-section of gridded data with colors selected from a finite set of (normally) smooth and continuous colors—a colormap.

3.2.1 Procedures

Because of the toolkit approach, there are a variety of ways to accomplish our display goals.

Displaying an image via an Image `tool`

First let us see the simplest way to display an image. Do this from an Image `tool`. If the image you wish to display is already present in *Glish* as an Image `tool`, (say, **im**) then all that is necessary to display the image is:

```
im.view()
```

If you haven't yet constructed an Image `tool` for your image, then do that first, e.g.

```
include 'image.g'
im := image('testimage.im')       # Native aips++ disk image
im.view()
```

Shortly a display of your image will appear, adorned with a GUI containing a menu bar and various buttons, whose functions are described here. Right now, if you're keen to alter some of the settings of the display, just press the Adjust button near the bottom left of the display! [1]. The Image.view function uses its own Viewer `tool` to display the raster map.

Using the Image.view function, while straightforward, imposes some limitations on the flexibility of the display.

We can also easily display a FITS image.

[1] Note that it can take a little time for the Adjust window to appear on your screen—please be patient and wait, rather than repeatedly pressing the Adjust button!

```
include 'image.g'
im := image('/fits/3c273.fits')    # Detects whether aips++ or FITS image
im.view()
```

The performance of this native-FITS access display will be fine provided you don't reorder the display axes. If you want to reorder axes in the display, then convert the image to native AIPS++ format first.

```
include 'image.g'
im := imagefromfits('/data/3c273.im', '/fits/3c273.fits')
im.view()
```

A more flexible way to display a raster map

This describes how to use the general Viewer GUI, step by step.

1. Include support for the Viewer `tool`, and display its basic GUI

   If you are starting from the Unix prompt, you can type:

   ```
   viewer
   ```

   this line is just a wrapper which starts *Glish* and executes the code in the example immediately below. Otherwise, if you are already in the *Glish* command line, type:

   ```
   include 'viewer.g'
   dv.gui()
   ```

   You will see a GUI which consists of two main windows, entitled viewer - Data Manager (AIPS++) and viewer - Display Panel (AIPS++). The `tool` **dv** is made for you when the `viewer.g` script is loaded. It stands for 'default viewer'. It's always there and generally that's the Viewer `tool` to use (see below for examples of making your own). Here are the GUI descriptions of the Viewer Data Manager

2. Make a Viewer DisplayData from your data

   A Viewer DisplayData is itself a `tool` which holds both data and functions which can display those data. A Viewer DisplayData for a raster map, for example, has attributes describing what colormap to use, what data range to display, etc.

   - Data loaded into *Glish* via a `tool` or global variable

     In the bottom left box called Tool Name of the viewer - Data Manager (AIPS++) window, click on the array (or `tool`) you want to display.

     If the dataset you created on the command line is not visible click the Update button and the list should update, and in it you should see the name of the data you prepared for display. You are now given a list of available display types on the right hand side. Clicking on the button Raster Map will create a Viewer DisplayData of this type.

or

- Image data on disk

  In the viewer - Data Manager (AIPS++) file browser window, select a filename. After clicking on it, you are presented with the available display types for these data. Clicking on the button Raster Map will create a Viewer DisplayData of this type. Follow this link to see a detailed description of the Viewer Data Manager GUI.

3. If the Autoregister button on the Viewer Data Manager wasn't checked, register the newly created DisplayData for display on the provided Viewer DisplayPanel: in the viewer - Display Panel (AIPS++) window, select the DisplayData you have just created from the Register... submenu of the DisplayData menu. Shortly thereafter, a raster map of your image or *Glish* array should appear on the Viewer DisplayPanel.

The result of this procedure, for the test *Glish* array generated with the `viewermaketestarray` function, is shown in Figure 3.1. The array is displayed as a greyscale raster map in the viewer - Display Panel (AIPS++) window. The DisplayData menu is extended in the figure to show that the DisplayData `array:mar(Raster Image:1)` has been registered for display (i.e. the checkbox is lit).

In Figure 3.1, the main parts of the Viewer DisplayPanel GUI are visible: the menubar along the top; the control buttons down the left side, which are used to assign functions to up to three mouse buttons; the animation buttons down the right hand side which are disabled in this case because the array has only two dimensions; the button bar along the bottom, and just above that, the position information box. These are described in detail here.

Command line procedure

This final offering of how to viewer a raster map of some appropriate data, is wholly command-line based. Thus it forms a nice introduction to actually doing some more sophisticated things with the Viewer `tool`, like using it in your own *Glish* application.

First, construct a Viewer `tool`:

```
mv := viewer();
```

See the Viewer documentation in the Reference Manual for a description of the parameters that can be supplied to the Viewer `tool` constructor. Rather than constructing your own Viewer `tool`, you could also use the default Viewer `tool` (**dv**) that we have seen earlier. It is constructed upon inclusion of the `viewer.g` *Glish* script. Now construct a Viewer DisplayPanel in which the raster map will be displayed:

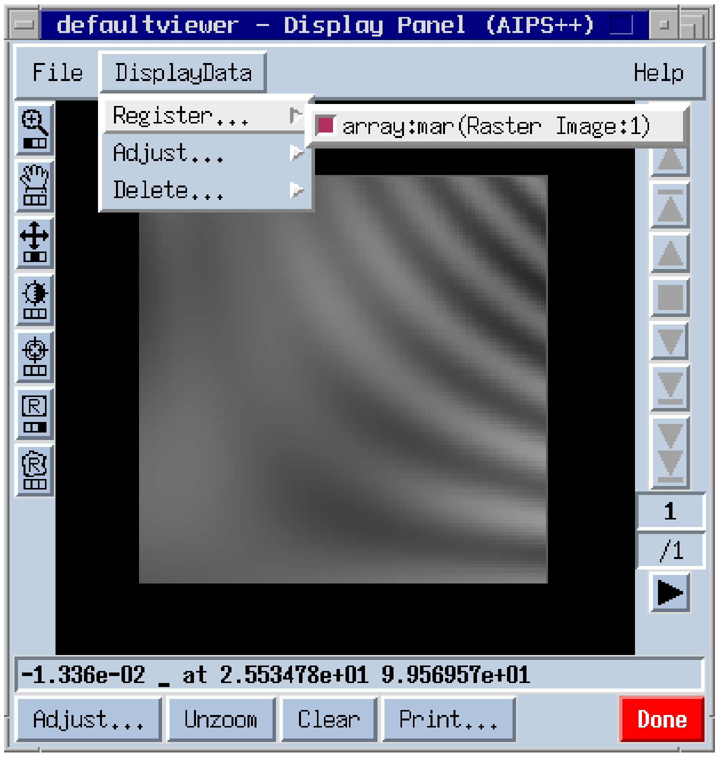


Figure 3.1: Registering a DisplayData on a Viewer DisplayPanel

```
mdp := mv.newdisplaypanel();
```

At this point, you will have a Viewer DisplayPanel on the screen, ready to display data. The `tool` **mdp** can be used to control this Viewer DisplayPanel. For example, you could temporarily remove the Viewer DisplayPanel from the screen by entering `mdp.unmap()` at the command-line. Once again, see the Viewer.newdisplayapanel documentation for the available parameters for the **viewer.newdisplaypanel** function, and the Viewer displaypanel documentation for the functions available in the newly created Viewer DisplayPanel.

If you have an image to display, then load the data as follows, substituting *'/data/3c273.im'* for the name of your `image file`. The quotes are required, and you may need to specify the path to the image if it is not in the current directory.

```
mdd := mv.loaddata('/data/3c273.im', 'raster');
```

If instead you have a *Glish* array to display, then load the data as follows, substituting `data` for the name of the global *Glish* variable in which the array is stored (note this time *without* quotes)

```
mdd := mv.loaddata(data, 'raster');
```

Note that your image or array should have at least two dimensions.

Once you have completed the above steps, **mdd** is a Viewer DisplayData, which needs to be registered on a Viewer DisplayPanel (e.g. **mdp**) so that it can be displayed. This is done like this:

```
mdp.register(mdd);
```

and after a brief delay, the length of which is in some way proportional to the size of your image or array, you should see an initial raster map of your image or *Glish* array.

3.3 Viewing a contour map of an image or *Glish* array

With only very slight modifications to the above procedures, it is possible to display a contour map generated from an image or *Glish* array. A contour map shows lines of equal pixel intensity (e.g. flux density) in a two dimensional cross-section of gridded data. Contour maps are particularly useful for overlaying on raster images, so that two different measurements of the same part of the sky can be shown simultaneously.

3.3.1 Procedures

Simple ways to display a contour map of an image

As in the case of a Raster Image, select the image `tool` in the Tool Name listbox or the image name in the file browser. Now click on Contour Map instead of Raster Image. Register the Viewer DisplayData on the Viewer DisplayPanel as before using the Register... submenu of the DisplayData menu.

Alternatively, for an image already present in *Glish* as an image `tool`, (say, **im**), use the contour option of Image.view on the command line, i.e.

```
im.view(contour=T);
```

This will register the image for you automatically.

Other possibilities

The procedure via the Viewer `tool` GUI is easily modified to produce a contour map display: simply select Contour Image where previously you selected Raster Image.

The command-line procedure for creating a raster map display can be modified to create a contour map display by simply replacing the line:

```
mdd := mv.loaddata('/data/3c273.im', 'raster');
```

with:

```
mdd := mv.loaddata('/data/3c273.im', 'contour');
```

3.4 Viewing a vector map of an image or *Glish* array

Consider a Complex image or array. At each pixel, this represents an amplitude and phase (position angle). A common method of displaying amplitude and position angle data is via vectors (line-segments) where the length of the vector is proportional to the amplitude. For example, it is usual to view linear polarization data in which the fractional polarization is the vector amplitude and the the linearly-polarized position angle is the vector position angle. This may be overlaid on total-intensity raster displays. Another example might be the display of fluid-flow.

Since amplitude and phase can be conveniently represented as a Complex number, the data source for this kind of vector display is either a Complex image or a Complex *Glish* array (later we will describe how you might create such data).

However, you can also provide Float images and arrays, in which case the data values specify the position angle and the amplitude is assumed to be unity. The angular units should be specified by the brightness unit of the image. If none, degrees are assumed.

The position angle is measured positive North through East when you display a plane holding a celestial coordinate (the usual astronomical convention). For other axis/coordinate combinations, a positive position angle is measured from +x to +y in the absolute world coordinate frame.

The procedures already described for raster and contour displays are appropriate as well for vector displays.

3.4.1 Procedures

How to display a vector map with Float images

At present, Image tools can only be associated with Float images. You will get an error if you attempt to associate an Image tool with a Complex image disk file. This will be rectified in the future.

For a Float image, you can make a vector map display via the Image tool function Image.view.

```
include 'image.g'
im := imagefromshape('zz', [64,64])    # Make empty image
im.set(20)                             # Set to 20 degrees
im.view(vector=T, axislabels=T)        # Display
```

For Complex images (and optionally for Float images) you must use the native Viewer interface as described below. This also gives you more flexibility in combining displays (raster, contour, vector etc).

A more flexible way to display vector maps

The procedure via the Viewer `tool` GUI is easily modified to produce a vector map display. After selecting an image, Image tool or *Glish* array (Complex or Float), simply select Vector Map where previously you selected Raster Image.

The command-line procedure for creating a raster image display can be modified to create a vector map display by simply replacing the line:

```
mdd := mv.loaddata('/data/3c273.im', 'raster');
```

with:

```
mdd := mv.loaddata('/data/3c273.im', 'vector');
```

although in this case drawing vectors of the intensity of 3C273 wouldn't be very illuminating ! You might try

```
im := imagefromshape('zz', [128,128])
im.set(45.0)
im.setbrightnessunit('deg')
im.done()
mdd := mv.loaddata('zz', 'vector')
mdp.register(mdd);
```

which will make an image with value 45 deg and then display lots of vectors all of the same length aligned in the same direction.

Making Complex images and arrays

Complex images are not yet fully supported in AIPS++. Most notably, the Image tool cannot open a Complex image which is unfortunate. This will be rectified in a future release.

However, selected functions of the Imagepol tool do write out Complex disk images which can be used as a data source for the vector displays.

These functions are:

- complexlinpol - generates Complex linear polarization $Q + iU$
- complexfraclinpol - generates Complex fractional linear polarization $(Q + iU)/I$
- fourierrotationmeasure - generated Complex linear polarization $Q+iU$ as a function of Rotation Measure
- makecomplex - generates a Complex image from either Float amplitude and phase images, or real and imaginary images

3.5 Viewing a marker map of an image or *Glish* array

A marker map represents pixel intensities by a particular marker shape and fill style. For example, the marker shape may be a square. The size of the square is proportional to the absolute pixel intensity. If the pixel is positive the square is filled, if the pixel intensity is negative, the square is open.

This kind of display is useful for quantities such as the Rotation Measure. The RM may be positive or negative with a distribution that may well be quite discontinuous. Displaying RM images as rasters is not very useful because the strongly negative values (which are equally important as the strongly positive values) will come out dark. Contouring usually fails because of continuity issues. With the marker map display you are biased against seeing small absolute values (square size becomes smaller). So you really have to decide what you are interested in and what display is most useful.

3.5.1 Procedures

Simple ways to display a marker map of an image

As in the case of a Raster Image, select the image `tool` in the Tool Name listbox or the image name in the file browser. Now click on Marker Map instead of Raster Image. Register the Viewer DisplayData on the Viewer DisplayPanel as before using the Register... submenu of the DisplayData menu.

Alternatively, for an image already present in *Glish* as an image `tool`, (say, **im**), use the contour option of Image.view on the command line, i.e.

```
im.view(marker=T);
```

This will register the image for you automatically.

Other possibilities

The procedure via the Viewer `tool` GUI is easily modified to produce a contour map display: simply select Contour Image where previously you selected Raster Image.

The command-line procedure for creating a raster map display can be modified to create a contour map display by simply replacing the line:

```
mdd := mv.loaddata('/data/3c273.im', 'raster');
```

with:

```
mdd := mv.loaddata('/data/3c273.im', 'marker');
```

3.6 Viewing a source catalog overlay

The Viewer can be used to plot overlays of sources from catalogs stored in AIPS++ tables, on raster images or contour maps. Note though, that it is only an *overlay* feature—you **MUST** have another DisplayData already registered on a Viewer DisplayPanel in order for the catalog sources to be drawn.[2]

Creating a Skycatalog The Viewer module comes with a skycatalog tool to convert from ASCII tables and componentlists to the skycatalog table format. Most commonly you will have an ASCII file e.g. *mydata.txt* like this:

NED	16	19:44:47	-14:46:51	NGC6822	159.8
NED	22	01:04:46	+02:07:04	IC1613	328.1

The best way to prepare this file for import is to prepend a header with information about what is in the columns. (Note that this is not necessary and the skycatalog tool can automatically determine the column data and name the columns Column1..ColumnN if no header is present.)

So after inserting the header into your file it looks like this:

Name	No	RA	DEC	NED-ID	mom0
A	A	A	A	A	R
NED	16	19:44:47	-14:46:51	NGC6822	159.8
NED	22	01:04:46	+02:07:04	IC1613	328.1

The first line specifies the names you would like to use to identify the columns. The second line indicates what data type your column holds. Most commonly you'll only need these:

A = ASCII(String)
R = Real(Float)

See tablefromascii for more on table headers.

All you need to do now is:

```
include 'skycatalog.g'
sca := skycatalog('myscat.tbl')
sca.fromascii(asciifile='mydata.txt',hasheader=T, # tell the tool which columns
              longcol='RA',latcol='DEC')          # are longitude and latitude
sca.browse()                                      # invokes table browser
sca.done()
```

You can re-access this skycatalog table in the Viewer Data Manager via the table *myscat.tbl*.

Look at the skycatalog tool in the reference manual for more details on how to use this `tool`.

3.6.1 Procedures

As for viewing raster and contour maps, viewing catalog overlays is pretty straightforward, and can be accomplished a number of ways:

[2]This is necessary because the sky catalog overlay facility is *passive*, that is it will not set up a coordinate system on the Viewer DisplayPanel for itself to use.

1. Use the viewer GUI: click on a a suitable table in the Data manager window, and the Display Types will show SkyCatalog Overlay. If you click on this a DisplayData will be created. Then in a Viewer DisplayPanel which already has a raster and/or contour map registered, register the newly made DisplayData for display. Any sources in your catalog table, which are in the region of sky currently displayed in the Viewer DisplayPanel, will be drawn with crosses.

2. Use the command-line interface as follows:

   ```
   mdd := dv.loaddata('myscat.tbl', 'skycatalog');
   mdp.register(mdd);
   ```

   In this example, **mdp** is an existing Viewer DisplayPanel with a raster or contour map already registered.

Once you have a sky catalog overlay registered on a Viewer DisplayPanel you can adjust a number of parameters to obtain the desired display.

3.6.2 Notes

- For a source table to appear in the Datasets listbox of the viewer - Data Manager (AIPS++) window, it must be available as an AIPS++ table
- There are no sources in the 'Sources' table provided in AIPS++ which lie in the field of the test image provided in AIPS++. So to see any catalog sources, you either need to prepare your own catalog source table, or view an image which contains a bright continuum source, for example a synthesis imaging calibrator source.

3.7 Viewing a measurement set

Measurement sets can now be displayed and flagged directly from the viewer. The interface is fairly basic at present, but further refinements are in progress. From the Viewer Data Manager window, simply choose your measurement set file, then click 'Raster Image'.

You could do the same thing from the command line with:

```
mv := viewer();
mdp := mv.newdisplaypanel();
mdd := mv.loaddata('mydata.ms', 'raster');
mdp.register(mdd);
```

You should use the filename rather than the variable name of an ms `tool`. You should also assure that no other Display Data is registered on the Viewer DisplayPanel.

The only type of display currently available is 'raster', which is similar to `msplot`'s 'Display data as an image' mode (or to TVFLG). A plotting-style Display Data for measurement sets is planned for the future.

Usually, the measurement set data is loaded automatically when the Viewer DisplayData is created. Loading data for large measurement sets can take some time. Look at the AIPS++ console window (rather than the logger) to monitor progress.

For *very* large measurement sets, you must press 'Apply' on the 'Adjust' panel to load the data yourself. This gives you the opportunity to choose something other than Observed Amplitude, for example, before a time-consuming data load.

A complete description of the 'Adjust' panel interface for controlling display and editing of the measurement set can be found at this link.

3.8 Operating the Viewer DisplayPanel

The Reference Manual gives a complete description on how to operate the Viewer DisplayPanel GUI. Just follow this link

3.9 Adjusting display parameters

On the Viewer DisplayPanel, there is a button at the bottom left labelled Adjust.... If you click it, then for each DisplayData that is currently registered on the Viewer DisplayPanel, a window labelled Adjustment (AIPS++) (and prefixed by the name of the DisplayData so that you know which is which) will appear (a little patience is sometimes needed). Each of these adjustment GUIs consists of a series of roll-up windows, which can be exposed or hidden by clicking the text next to the up or down arrow. The options available for adjustment depend on the type of display. The settings common to *raster*, *contour* and *vector* and are described here Each of them have specific settings as well:

- raster
- contour
- vector
- marker
- skycatalog
- measurement set

At the bottom of the adjustment window, are a couple of buttons with which you can save current settings, and restore previously saved settings. Just type a unique name which you will remember for this DisplayData, and hit save or restore. The new parameters (if they exist) will be installed and the displays updated accordingly. **Note: You have to be in the directory you started the viewer in to be able to retrieve the parameters.**

3.10 Advanced, worked examples

In this section, a few fully-worked examples are given which cover quite advanced uses of the viewer. The examples are generally accomplished from the command-line so that you can easily replicate them yourself.

3.10.1 Overlaying contours on raster maps

A common operation is to overlay contours of one image on a raster map of another image to show, for example, the HI emission in relation to the starlight in a galaxy. The Viewer `tool` can easily produce such a display, provided the two images have the same coordinate type, epoch, projection and projection parameters. Yes, that's a long list of requirements, but most of them will be relaxed in the near future. Since there is only one test image available however, in this example we will overlay contours from the test image on a raster map of the very same image. Let's begin by creating the test image and making a DisplayPanel and two DisplayData:

```
include 'image.g';
im := imagemaketestimage('test.im');
#
mdp := dv.newdisplaypanel();
mdd1 := dv.loaddata(im.name(), 'raster');
mdd2 := dv.loaddata(im.name(), 'contour');
```

Immediately we can see the overlay by simply registering the two DisplayData on the DisplayPanel. For a single refresh cycle, we wrap the registration between a **hold** and **release** pair:

```
dv.hold();
mdp.register(mdd1);
mdp.register(mdd2);
dv.release();
```

There are plenty of ways to improve the display, by adjusting various parameters for the raster and contour maps. This can be done by pressing the Adjust... button on the Viewer DisplayPanel, and modifying the parameters in the GUI. The GUI can also be displayed for each DisplayData by doing:

```
tmp := mdd1.gui();
tmp := mdd2.gui();
```

Why don't you do this now and start playing away ?

3.10.2 Displaying complex data

In this example, we generate a complex data array, and use some special features of the Viewer `tool` to visualize the data. Let's commence by taking the standard Viewer `tool` test array data, and Fourier transform it to obtain a complex array:

```
myrealdata := viewermaketestarray(180);
include 'fftserver.g';
mfft := fftserver();
mycxdata := mfft.realtocomplexfft(myrealdata);
```

At this point, `mycxdata` is a complex array, storing the Fourier transform of the real array `myrealdata`. We can immediately view the complex array with the Viewer `tool` as follows:

```
mrdp := dv.newdisplaypanel();
mrdd := dv.loaddata(mycxdata, 'raster');
mrdp.register(mrdd);
```

The image has a rather high dynamic range, so let's ramp up the "magnifying" power of the colormap, and zoom in by a factor of three:

```
mrdd.setoptions([powercycles=-3.0]);
mrdp.zoom(3);
```

Now, the default behavior of the raster map, when provided with complex data, is to shade pixels according to the magnitude of the pixel values. Valid alternatives are the phase, real and imaginary components. Select which you like according to the following examples:

```
mrdd.setoptions([complexmode='phase']);     # display phase
mrdd.setoptions([complexmode='real']);      # display real component
mrdd.setoptions([complexmode='imaginary']); # display imaginary component
mrdd.setoptions([complexmode='magnitude']); # back to magnitude display
```

Make sure you execute at least the last line here, so that at this point you are back viewing the magnitude of the data. Note that if you have the adjustment panel open for this DisplayData as you are making changes from the command line, you will see the adjustment panel update in response to your changes!

Let's get a little more sophisticated. We can actually display two (or even three) components of the complex data at once, using a multi-channel raster map. Such a map has different data being mapped to either the red, green and blue components of each pixel on the screen, or instead the hue, saturation and value components of each pixel on the screen. First we shall avail ourselves of a different type of Viewer DisplayPanel, an *RGB* DisplayPanel, and make two new DisplayData:

```
mcxdp := dv.newdisplaypanel(maptype='rgb');
mcxdd1 := dv.loaddata(mycxdata, 'raster');
mcxdd2 := dv.loaddata(mycxdata, 'raster');
```

Now we will set the `mcxdd1` DisplayData to display the real component of the pixels, and place this on the red channel, and set `mcxdd2` to display the imaginary component via the green channel:

```
mcxdd1.setoptions([complexmode='real', colormode='red']);
mcxdd2.setoptions([complexmode='imaginary', colormode='green']);
```

Now we should be able to make a really cool display by registering both these DisplayData on the one Viewer DisplayPanel. We make use of the **hold** and **release** functions of the Viewer `tool` to do this in one refresh cycle:

```
dv.hold();
mcxdp.register(mcxdd1);
mcxdp.register(mcxdd2);
dv.release();
```

Again, let's just improve things by spreading out the colors, switching to a smoother resampling mode, and zooming in:

```
dv.hold();
mcxdd1.setoptions([powercycles=-1.0, resample='bilinear']);
mcxdd2.setoptions([powercycles=-1.0, resample='bilinear']);
mcxdp.zoom(5);
dv.release();
```

Finally, we can allocate more colors to the red and green channels by taking them away from the blue channel. Find out the current color cube dimensions:

```
print mcxdp.status().pixelcanvas.colorcubesize;
```

Chances are the result will look something like `[7 6 6]` if you are using an 8-bit screen, or `[256 256 256]` if you are using a TrueColor screen. If the numbers are low, you can rearrange them as follows:

```
dv.hold()
tmp := mcxdp.setoptions([colorcubeblueaxislength=1]);
tmp := mcxdp.setoptions([colorcuberedaxislength=12]);
tmp := mcxdp.setoptions([colorcubegreenaxislength=12]);
dv.release();
```

If you are on a TrueColor (24bit) screen, you cannot modify the color allocations at this point.

In the final display, you can now see the areas where the real component is dominant (red-dominant pixels), areas where the imaginary component is dominant (green-dominant pixels), and the in-between areas where the real and imaginary components have roughly the same magnitudes (yellow colored pixels).

3.11 3D slice display of data cubes

The Viewer `tool` also provides a slightly modified Viewer DisplayPanel, the Viewer Slice DisplayPanel. It has most of the functionality of the Viewer DisplayPanel. You might notice that some of the tools in the control box don't appear. These tools will be available soon. In addition the Viewer Slice DisplayPanel provides an extra tool, the Multi panel crosshair. This controls the slicing action. Dragging it over one of the three different displays of the data will control the animation of the other two. The top-left display will show a XY slice, the top-right a ZY and the bottom-left panel a XZ slice of the data. **Note that only a Viewer DisplayData with at least three axes and more than one pixel along the third axis will register on the Viewer Slice DisplayPanel.** The slices probably will not fill the panels evenly. This is because the ratio of the pair of displayed axes

The button bar at the bottom of the Viewer Slice DisplayPanel shows a Preload... button. It executes an animation loop on each of the slices of data. By doing this, the display caches all views of the data and will improve the speed of the Multi panel crosshair tool. **Note: This can take a while for large data cubes, although it is recommended to use this if you chose to have axis labels.**

Note: Don't use FITS images on this DisplayPanel. Convert them to native AIPS++ images first. The access of FITS images with reordered axes is very slow.

3.12 Displaying channel maps of a data cube

The viewercanvasmanager gui accessible via the `File` menu of Viewer DisplayPanel, gives you control over the number of channels in a data cube you want to display at the same time - so called channel maps. The `Layout` rollup drives this application. Select the number of views you want to see in x and y and the animator will take care of the distribution. These will be arranged from top-left to right-bottom, starting at the frame selected in the animator. **Be careful here: If your margins are too big this can get out of control. Always reduce the margins in the `Geometry` rollup first.**

3.13 A step-by-step recipe to make a post-script plot of your image

This section gives a detailed description on how to get from your image on disk to a publishable PS file just using the GUI.

Let's say we have an image called `ngc253.fits`:

1. **Starting the viewer**

   To start the viewer type `viewer` at your Unix command line.

2. **loading the image into the viewer**

   - In the data manager window select the file `ngc253.fits`.
   - On the right-hand side click on `Raster image`. The image will now be displayed on the display panel.
   - Click on the `Done` button to finish with the data manager.

3. **Adjust (fiddle) the colormap**

   The displayed might not show a nice contrast. You can use the colormap fiddling tool to adjust this.

   - This is by default associated with the middle mouse button, so move your cursor onto the image and drag the cursor across the display while holding the middle mouse button down. Release when you are happy with the result.

4. **Turning on axis labels**

   - Click on the `Adjust...` button at the bottom-left of the display panel.
   - click on the roll-up `Basic settings` to roll it up.
   - click on the roll-up `Axis labels` to roll it down
   - press the menu button next to `Axis Labelling & annotation` and select `True`. The axis labels will appear on the display panel.
   - press the `Dismiss` button at the bottom of the adjustment gui to hide this window.

   Your axis labels might be cut off right now or you think that there is too much space around them. In this case you should go to point 4, otherwise continue skip this.

5. **Adjusting the axis label space**

   - On the display panel select `Canvas manager` from the `File` menu
   - press on `Geometry` to roll this rollup down.
   - use the sliders to remove excess or add extra space.
   - press the `Done` button at the bottom of the canvas manager gui to remove this window.

6. **Printing the image/ save it to a PS file**

   - press the `Print...` button at the bottom of the display panel.

   Now either save it to a file:

- enter a filename in to the first field (`Output file`) of the canvas print manager AND(!) hit return.
- press the `Save PS` button at the bottom of the gui.

or print it directly:

- press the `Print` button at the bottom of the gui.
- in the new window select either `Ghostview` or a `Printer` with name.
- if you want to delete the underlying postscript file after printing check the box `Remove after printing`
- press the `Print` button

7. press the `Dismiss` button on the canvas print manager window.

That's it you have successfully create a post-script file of your image. You might have noticed all the other options you have in the Adjust panel. Just play around and look what they do.

Chapter 4

Image Analysis

Neil Killeen

4.1 Summary

This Chapter provides an overview of image analysis functionality. The term 'image analysis' has a rather broad meaning. It basically refers to functionality for processing images, but excludes creating them from visibility data (both imaging and deconvolution). These are parts of the imager `tool` described in a separate chapter of this document.

All the `tools` discussed in this chapter are available from both the *Glish* command-line interface (CLI) and from the Toolmanager graphical user interface (GUI).

The `tools` that are currently available for image analysis can be found in the Images module. Refer to this module's documentation to find out about basic concepts common to the `tools` in this module. Refer to the specific `tool` documentation to learn details about individual `tools`. Access to all the `tools` in the Images module can be gained from *Glish* via the command

```
include 'images.g'
```

Alternatively, you can include the appropriate *Glish* script for the desired `tool` as specified below. Also, you should specifically know about one other `tool` which is not a part of the Images module. This is the Viewer `tool`. This `tool` is used to display data, including images. Some of the services in the Images module use the Viewer.

The `tools` available for image analysis are:

Image is the basic `tool` type used to manipulate AIPS++ images. Access with:

```
include 'image.g'
```

An Image `tool` is a straightforward `tool` that provides a range of uncoupled services. What this means is that unlike a `tool` such as Imager, which has a lot of state and required functions, an Image `tool` just has a lot of functions that do a certain thing to the image. Current functionality ranges from data conversion (e.g. to and from FITS), inquiry, pixel access, coordinate manipulation, convolution, statistical analysis, source finding and fitting, regridding, mask handling, and some useful utility services.

Regionmanager This `tool` enables you to create and manipulate regions-of-interest (or more simply, regions) in images. Access with

```
include 'regionmanager.g'
```

Generally, one only needs to use the pre-made, default Regionmanager `tool` called **drm**, to create and manipulate regions (this is often referred to as 'The Regionmanager'). These regions are used to specify portions of an image that you are interested in for some astrophysical reason (e.g. what is the integrated flux density around this source). Refer to the Regionmanager documentation for all the details on regions.

Regions are in fact themselves `tools`; they are created and managed by the Regionmanager. But regions are rather simple `tools`; they are just used as containers to pass the region information around. You don't need to use any of their functions. Region `tools` are often arguments to Image `tool functions`.

The Regionmanager has a custom GUI interface which is preferred over the command-line interface (which is somewhat cumbersome for world regions).

Coordsys This `tool` enables you to create and manipulate Coordinate Systems. Access with

```
include 'coordsys.g'
```

Generally, a Coordinate System is associated with an actual data set, such as an image. However, they can exist in their own right and can be used to do coordinate conversions. The Coordsys `tool` can be used to create and manipulate Coordinate Systems. A Coordinate System that is associated with an image can also be recovered into a Coordsys `tool` via the Image `tool` function Coordsys.function.

Imagefitter This high-level `tool` bundles together many services (e.g. the Viewer and Image `tools`) to offer an interactive application enabling you to fit models to sources in images. Access with

```
include 'imagefitter.g'
```

Imageprofilefitter This high-level `tool` bundles together many services (e.g. the Viewer and Image `tools`) to offer an interactive application enabling you to fit models to 1-D profiles extracted from images. Access with

```
include 'imageprofilefitter.g'
```

Imagepol This `tool` provides polarimetric analysis of images and offers easy access to the different Stokes parameters in an image, computation of the standard secondary (e.g. polarized intensity) and tertiary images (e.g. fractional polarization) and their errors. There are also methods for deriving the Rotation Measure.

```
include 'imagepol.g'
```

4.2 Creating an Image tool

You can make an Image `tool` in a variety of ways. This `tool` is used to gain access to the data stored in the image; these might be the pixel values, the pixel mask(s), or perhaps the coordinate information. You will typically make a lot of Image `tools`, whereas you might make only one or two Imager `tools` (create images from visibilities).

4.2.1 From a built-in test image

Built into the system is a test FITS image. You can convert it to an AIPS++ image with:

```
im := imagemaketestimage('myimage')       # Convert to disk file called 'myimage'
                                          # and associate with tool 'im'
im.view()                                 # Display it via tool function view
```

4.2.2 From existing AIPS++ image files

```
im1 := image('x.app');                                              # Access disk file 'x.app'
r := drm.quarter()
im2 := imagefromimage(outfile='x.app.copy', infile='x.app', region=r)  # Makes a disk file copy of x.app
im3 := imageconcat('x.concat', infiles="x*.app", axis=3, relax=T)      # Concatenate files
```

The first example is the most common way to make an Image `tool`. It just associates the `tool` with a pre-existing AIPS++ disk image file. The second example makes a copy of a subset (via a region) of the input image; the Image `tool` **im2** then gives you access to it. The third example concatenates images along a specified axis (in this case axis 3).

4.2.3 From foreign formats (e.g. FITS)

Presently, there is capability to

- Convert FITS image files to AIPS++ format

  ```
  im := imagefromfits('x.app', 'X.FITS') # Convert from FITS
  im.view()                              # Display
  #
  im2 := imagefromfits(infile='X.FITS')  # Output Image tool is virtual
  im2.view()
  ```

  In the second example, we didn't write the AIPS++ image out - it's stored in memory. This is called a virtual image (see below).

- Access FITS image files directly

  ```
  im := image('X.FITS')                  # Native FITS access
  im.view()                              # Display
  ```

  This example shows how the `image` constructor can also be used to gain *native* access to the FITS file. No format conversions occur and the data are sourced directly from the FITS file. Performance will be good unless you start to try to access the data in non-native order. For example, if with the Viewer, you reordered the display axes from their native order, performance would be poorer. Always convert to an AIPS++ image if you need to do this.

- Access Miriad image files directly

  ```
  im := image('hcn.mir')                 # Native Miriad access
  im.view()                              # Display
  ```

  The same comments above about native FITS file performance apply here as well.

- Convert Miriad or GIPSY image files to AIPS++ format via FITS and an installation of that package

  ```
  im := imagefromforeign('hcn.app', 'hcn', package='gipsy')
  im.view()                              # Display
  ```

  This constructor relies on the nominated package being installed. It is used to write out a temporary FITS file which is then converted to AIPS++ format.

4.2.4 From simple shapes or Glish arrays

```
im1 := imagefromshape('test1.data', [64,128])   # Image has value 0
pixels := array(10.5, 32, 64)                   # Glish array (value 10.5)
im2 := imagefromarray('test2.data', pixels)     # Image has value 10.5
```

4.2.5 From image expressions

The Image `tool` offers a sophisticated Image calculator. You can give it complex expressions of images to manipulate. See § 4.11 for more information on the image calculator. You can both create images and change images with the calculator. Here are a couple of simple examples.

```
im1 := imagecalc(outfile='image.out', pixels='image1 + min(image2)')   # Writes a disk file
im2 := imagecalc(pixels='image1 + min(image2)')                        # A read-only image
im2.calc (pixels='2.0*$im2')                                           # Double values
```

In these examples, the AIPS++ image disk files *image1* and *image2* pre-exist. In the first example, an expression is evaluated and the new file *image.out* created. In the second example, an output disk file is never created. The expression is re-evaluated every time the pixels of the **im2** Image `tool` are accessed. This is a read-only image. In the third example, we replace the pixel values with the result of the expression. Note the use of the $im2 syntax which means use the pixel values of the image associated with the **im2** Image tool (it does not have a disk file).

4.3 Virtual Images

We also have Image `tools` that are not associated one-to-one with disk files; these are called "virtual" images (see the article in the AugustNewsLetter). For example, you saw above how with the image calculator, imagecalc, one can create an expression which may contain many images. You can write the result of the expression out to a disk `image file`, but if you wish, you can also just maintain the expression, evaluating it each time it is needed - nothing is ever written out to disk in this case. There are other Image constructors and functions like this (the documentation for each one explains what it does). The rules are:

- If you specify the `outfile` argument, then the image is always written to the specified disk `image file`.
- If you leave the `outfile` argument unset, then if possible, a virtual image will be created. Sometimes this virtual image will be an expression as in the example above (i.e. it references other images) or a temporary image in memory, or a temporary image on disk. (the Image.summary function will show the type of image you have). When you destroy that Image tool, the virtual image will be destroyed as well.

- If you leave `outfile` unset, and the function or constructor cannot make a virtual image, it will create a disk file for you with a name of its choice (usually input plus function name).

- You can always write a virtual image to disk with the Image.subimage `tool function`.

Here are some more examples:

```
im1 := imagemaketestimage()                    # A temporary image
im2 := imagefromshape(shape=[128,128])         # A temporary image
im3 := im2.subimage(region=drm.quarter())      # A reference to a temporary image
im4 := imagecalc(pixels='2*$im1')              # An expression image referencing a temporary image
im5 := imageconcat(infiles="z2 z3 z4", axis=1)  # References given disk files
```

4.4 Summarizing the Image

Having created an Image `tool`, a very common thing is to want to know something about its basic attributes. You can use the Image.summary function of your Image `tool` for this.

```
im := image('x1.app')
im.summary()
```

which lists, to the logger, the output

```
Image name       : test
Image type       : PagedImage
Pixels mask(s)   : mask0 [mask1, mask2]
Region(s)        : None
Image units      : JY/BEAM

Direction reference : J2000
Spectral  reference : LSRK
Velocity  type      : RADIO
Rest frequency      : 1.42041e+09 Hz

Telescope        : ATCA
Observer         : NEBK
Date observation : 1997/02/121/03:03:26


Axis Coord Type      Name             Proj Shape Tile   Coord value at pixel    Coord incr Units
------------------------------------------------------------------------------------------------
1    1     Direction Right Ascension   SIN  512   10  00:00:00.000     6.00 -6.000000e+01 arcsec
2    1     Direction Declination       SIN  512   20 +00.00.00.000    11.00  6.000000e+01 arcsec
3    2     Spectral  Frequency             1024   30  1.415000e+09    16.00  1.000000e+03 Hz
                     Velocity                         1.140944e+03    16.00 -2.110611e-01 km/s
```

You can see that the image is of a type called a “PagedImage”. This is the basic disk-based AIPS++ image. It has three masks, of which “mask0” is active (it’s not in square brackets). The image has three pixel axes associated with two coordinates. The first coordinate, a Direction Coordinate is associated with the first two axes of the image. The third axis of the image is associated with a Spectral Coordinate; the listed rest frequency is used to convert the coordinate attributes to velocity in the listed velocity definition (RADIO).

There are many other functions which allow you to find out basic things about your image. See the Overview section of the Image `tool`.

A simple statistical summary of your image can be obtained with

```
im := imagemaketestimage()
im.statistics()
```

which lists to the logger the output

```
Selected bounding box [1, 1] to [113, 76]
Creating new statistics storage lattice of shape [9]

Number points =   8.588000e+03       Sum      =   1.813081e+03
Flux density  =   8.395378e+01 Jy
Mean          =   2.111179e-01
Variance      =   3.891285e+00       Std dev   =   1.972634e+00
Rms           =   1.983785e+00

Minimum value  -1.865986e+00 at [79, 63] (23:59:47.733, +00.05.00.000)
Maximum value   1.943354e+01 at [55, 32] (00:00:00.533, -00.01.12.000)
```

4.5 Display

There are a few ways to display an image.

4.5.1 Function view

This is the most straightforward; just invoke the Image.view function of the Image tool. This uses the Viewer tool.

```
im := imagemaketestimage('zz')
im.view(axislabels=T)
```

By default, the image is displayed as a raster. You can also see it as a contour image (or both). See the documentation for more details and other arguments.

4.5.2 From the catalog

The Catalog `tool` can also be used to view your image. To get the catalog GUI running do

```
include 'catalog.g'
dc.gui()                        # Default catalog tool
```

Then click on an image name (the disk file name) to select it and then press the View button to display it. Note that you don't get access to the Image `tool` that Catalog makes in order to do this. When you you press the done button of the image display, the temporary Image `tool` that it makes is destroyed.

4.5.3 From the spanner (wrench) menu

When using the Toolmanager GUI, you will find that virtually all of the many entry widgets have a spanner ('wrench' for yo'all Americans) menu associated with them. If you are dealing with an image entry widget, you can find an item in the spanner menu called 'view'. If you select that, in a way similar to the Catalog `tool` view, your image will be displayed. Again, when you you press the done button of the display, the temporary Image `tool` that was made is destroyed.

4.5.4 With the plain Viewer

You can of course also use the Viewer in its plain form, rather than using it packaged up for you in the **`view`** function. See the `tool` documentation for details (see also the Display chapter in this document). But all you really need do is

```
im := image('x1.app')
dv.gui()
```

and then, in the Data Manager GUI, select the `image file` or Image `tool` you wish to display. Then select how you would like to see it (from the Display-Data Type list, and your image will appear on the Display Panel.

4.6 Coordinates

4.6.1 General

The Coordinate System defines the world (physical) coordinates associated with some observation.

A Coordsys `tool` can create Coordinate Systems (independent of any image) and manipulate that Coordinate System (change values, reference frames etc). The Coordinate System can also be retrieved from an image via the Image.coordsys function and put back the Image.setcoordsys function.

There is also functionality to inter-convert pixel and world coordinates (functions Image.topixel and Image.toworld). The same functionality is available in the Image `tool` (functions Image.topixel, Image.toworld, and Image.coordmeasures) but really, behind the scenes, the Coordsys `tool` is doing the work.

We saw in the previous section how the Image.summary) function lists the Coordinate System. The function also operates on Coordsys `tools`:

```
im := image('x1.app')      # Image tool
cs := im.coordsys();       # Fish out Coordsys tool from image
cs.summary()               # List Coordinate System
```

In this case (listing not shown), the listing would have looked similar, but the non-Coordinate System things (image name, masks, units, image shape and image tile shape) would be missing.

4.6.2 Retrieving and changing the Coordinate System

This is done with a Coordsys `tool`. Here is a simple example.

```
im := imagemaketestimage()                # Create Image tool
cs := im.coordsys()                       # Get Coordsys tool
cs.referencepixel()                       # Print reference pixel
#<< [56 38]
cs.setreferencepixel(value=[40,60])       # Set new reference pixel
cs.referencepixel()
#<< [40 60]
im.setcoordsys(cs)                        # Set modified Coordinate System into image
cs.done()                                 # Destroy tools
im.done()
```

Because the Coordinate System can be retrieved from an image, and stored in its own Coordsys `tool`, it can be passed around to other `tools` and functions. For example, when making world regions with the Regionmanager, the Coordinate System is required.

4.6.3 Coordinate conversion

You may inter-convert between pixel and world coordinates with the Image `tool` functions Image.topixel and Image.toworld. These just invoke Coordsys `tool functions` of the same name.

Allowed formats are numeric vectors, quantity vectors, measures and strings (see function documentation).

```
im := imagemaketestimage();
print w := im.toworld([10,10], 's')       # Formatted as strings
#<< 00:00:24.533 -00.05.36.000
#
```

```
im.topixel(w)
#<< [10.0007122 10.0000115]          # Precision lost during string formatting
w := im.toworld([10,10], 'm')        # Record of measures
im.topixel(w)
#<< [10 10]                          # No precision lost
```

4.7 Mask handling

AIPS++ images may contain zero or more pixel masks (see the images module documentation and the image `tool` documentation). Pixel masks indicate which pixels are good (mask T) or bad (mask F). The good ones should be used for calculations (e.g. find statistics of image).

You can see the names of any pixel masks with the Image `tool` Image.summary function. Here is the beginning of such a listing

```
Image name       : hcn
Image type       : PagedImage
Image mask       : mask2 [mask0, mask1]
Image units      : JY/BEAM
```

This image has three masks. The mask named 'mask2' is currently the default mask (applied by applications). Any mask in square brackets (and it could be all of them) is not presently active.

You can change the values of the image pixel mask(s) via three Image `tool` functions.

The function Image.set is the simplest; you can set the mask to T or F in a specified region.

```
r := drm.quarter()                   # Quarter by area
im.set (pixelmask=F, region=r);      # Set mask to F in region
```

The function Image.putregion lets you replace the mask with a boolean array,

```
im := imagemaketestimage('zz')
local p,m
im.getregion(p,m)                   # Get pixels and mask
m[p<0] := F;                        # Set mask to F when pixels negative
im.putregion(pixelmask=m)           # Replace mask
```

In this example, to start with there is no mask in the image, so when you recover it, it will be all good (T). We then set some if it to bad (F) and replace it.

Finally the function Image.calcmask offers you the chance to change the mask via LEL expressions.

```
im := imagemaketestimage('zz')
im.calcmask('zz>0')
```

This example is the equivalent of the one above. The mask is set to the result of evaluating the boolean expression. Therefore, it is set to T when the image is positive.

In addition to this, the Image `tool` function Image.maskhandler enables you to select which (if any) pixel mask you wish to apply. You can also copy masks within an image and between images, delete and rename masks.

This function also has a custom GUI interface available via the Toolmanager and also via the Image `tool` function Image.maskhandlergui from *Glish*.

```
im := imagemaketestimage('zz')
mg := im.maskhandlergui()
```

See the Image.maskhandlergui documentation in the Reference Manual for a description and picture.

4.8 Utility

The Image `tool` has a collection of useful utility functions. See the Image `tool` function for details. Here are a few things that are handy.

4.8.1 Deletion and renaming

Since AIPS++ uses the host operating system for its file systems, you may use the appropriate operating system command to delete an AIPS++ image disk file (if it has an Image `tool` associated with it at the time you get into trouble of course).

You may also delete an AIPS++ disk image file from *Glish* with the Image `tool` function Image.delete.

```
im := image('g1.app')
im.delete(done=T)       # Destroy image tool as well
```

Renaming is suported with the Image `tool` function Image.rename.

```
im := image('g1.app')
im.rename('g2.app')
im.name()
#<< /DATA/ELARA_1/aips++/adata/g2.app
```

4.8.2 History

Currently, AIPS++ does not fully support history tracking (i.e. recording everything you do to a file in a history). With AIPS++ images, partial support is available. If a FITS file contains a history, and you convert it to an AIPS++ image, then the history is stored with the AIPS++ image. You can browse the history (via the Table browser in reality), direct the history to the logger, or save it in a *Glish* variable with the Image `tool function` Image.history.

You can also add your own history to the image if you wish with the function Image.sethistory. If you write the image out to FITS again, the history will be safely stored.

4.8.3 Locking

There is an automatic file locking system in AIPS++ so that processes are able to lock and unlock files as appropriate. You may need some control over file locking if you happen to write a *Glish* application which has multiple processes trying to access the same image (the imagefitter `tool` is an example of this type of script).

There are functions Image.lock and Image.unlock with which you may lock and unlock the image for the current process.

4.9 Regions of interest

The Regionmanager is used to create and manipulate image regions-of-interest. These are used to specify which section of an image you would like acess to (e.g. find the statistics in this region). Although regions are themselves (simple container) `tools`, all you need do usually is pass them around.

The Regionmanager has a custom GUI; you can access this (using the default Regionmanager `tool` **drm**):

```
include 'regionmanager.g'
drm.gui()
```

The GUI interface is much better than the command-line interface for creating world regions, as it effectively hides the Coordinate System. It also makes operations like storing and recovering regions from Images simple (you can store regions in Images).

With the Regionmanager you can create simple pixel and world regions, as well as compound regions from simple world regions (e.g. unions, intersections, complements etc.).

The Regionmanager GUI can also be used to interactively create regions from an image display. From the Regionmanager's list of available images you select the image of interest, and then you select 'interactive' from the list of available region types to create. The Image `tool` function Image.view is then automatically used to display the image. With this display you can interactively create a region (a box or a polygon, double clicking within the region to signify its ready - actually this emits and event) and it is then captured by the Regionmanager for further distribution. Some discussion on this can also be found below in the section on communications, § 4.15.

4.10 Pixel Access

4.10.1 Getting and putting pixels

There is functionality to get and put pixel values from an image. The exchange medium between the image and *Glish* is a *Glish* array. Since a *Glish* array is an in-memory object, make sure you don't put the entire image (if it is large) into it ! If this is a problem, iterate through your image in tile-shaped chunks (see the Image.summary listing to get the tile shape of an image).

There are pairs of functions Image.getchunk & Image.putchunk and Image.getregion & Image.putregion. The first pair only gives you access to the pixel values in regularly shaped regions (i.e. hyper cubes). The second pair gives you access to the pixel values, the mask values (see Images module documentation and Image `tool` documentation) and enables you to access the values from more complex regions-of-interest. The first pair of functions is substantially faster, so use it for simple regions if you are only interested in the pixel values.

4.10.2 Set a scalar value

There is an Image `tool` function called Image.set with which you can set the values of the image pixels and/or mask within a specified region to one scalar value. It's efectively a simplified interface to the image calculator. Note that the scalar can be the result of an expression involving images. Some examples are:

```
im := imagefromshape('xx', [10,20]);
r1 := drm.box([2,2],[6,8]);                # Make a pixel box region
im.set(pixels=1.0, region=r1);             # Set all pixels to 1 in the region
im.set(pixels='min($im)', region=r1);      # Set pixels in region to minimum of image xx
im.set(pixelmask=T);                       # Set default mask to all T
im.set(pixels=0, pixelmask=F, region=r1);  # Set pixels and mask in region
```

4.10.3 Replace masked pixels

There is a handy function called Image.replacemaskedpixels which replaces the values of all pixels in the specified region whose mask is bad (F) with the specified value. The mask is left unchanged.

```
im := imagemaketestimage();          # Test image
r := drm.quarter();                  # Quarter of image
im.set (pixelmask=F, region=r);      # Set mask to F in region
im.replacemaskedpixels(pixels=0.0)   # Replace pixels masked bad with 0
```

See Image.replacemaskedpixels for more details.

4.11 Image Calculator

4.11.1 Introduction

There is a sophisticated system in AIPS++ called the Lattice Expression Language (LEL). It allows one to compute expressions involving Lattices (images are just lattices with Coordinate Systems). When the expressions are computed, the images are accessed in tile-sized chunks and iterated through. In addition, sub-expressions whose result is a scalar are pre-computed. This ensures good efficiency and no large use of in-core memory.

There are two notes in the AIPS++ documentation system containing information about LEL. First, note 223 gives a full description of LEL's capabilities for both the *Glish* and C++ user. This is required reading for the user of the image calculator. It is the ultimate repository of information about the capabilities of LEL (including tables of all its functions and operators).

If you are having trouble sleeping, then you could admire note 216 which describes the C++ design and implementation (this note is a little out of date now, but the basic concepts are still valid).

4.11.2 Basic Access

The LEL system is used to provide an image calculator to the end *Glish* user. There are a few ways you can access this system.

- Image `tool` constructor imagecalc

  With this you can construct an image from expressions involving other images. For example,

  ```
  im1 := imagecalc(outfile='x3', pixels='sin(x1) + min(x2) ')
  ```

  where the image disk files *x1* and *x2* must pre-exist and the result is the image disk file called *x3*.

  You can also construct a read-only image from expressions involving other images. The output image is never created, it is just evaluated each time you use it.

  ```
  im2 := imagecalc(pixels='pi() - min(x2) + sqrt(x1)')
  ```

  where the image disk files *x1* and *x2* must pre-exist. Note that by simply excluding the output file name (argment `outfile`) you have created the read-only image.

- Image `tool` function Image.calc With this you can change the pixel values of an image by replacing them with the result of the expression.

  ```
  im1 := image('x2')
  im2 := im1.calc('abs(x2) / min(x2)')
  ```

where the image disk file *x2* must pre-exist. Because the scalar expression is evaluated first, the images in the expression can be the same as the image which is being overwritten.

4.11.3 Special characters in file names

Note that for image file names with special characters in them (like for example), you should (double) escape those characters or put the file name in ouble quotes. E.g.

```
- im1 := imagecalc(pixels='test\\-im')           # Note double escape
- im2 := imagecalc(pixels='"test-im"')
```

4.11.4 Conformance of expressions

The basic rule of LEL is that the images in your expression should conform; they should have the same shape and Coordinate System.

However, there are some exceptions to this.

First, if you reduce an image to a scalar (e.g. via the `min` function) then conformance of that image is not required.

Second, LEL will try, if it can, to extend one image to make it conformant with the other. It can do that if one image is a true subset of the other (thus if one image has all the coordinate axes of the other image and if those axes have the same length or have length 1). If so, LEL will add missing axes and/or stretch axes with length 1. For example, in this way you could subtract a 2D image from a 3D image (see continuum subtraction example below).

If your images are not conformant, then you need to regrid them to the same Coordinate System and shape via the Image `tool` function Image.regrid.

4.11.5 Tool substitution in expressions

When using LEL, you can use either image disk file names, or an associated Image `tool` name in the LEL expression. For example,

```
im1 := image('im1')
im2a := imagecalc('im2a', '2*im1')     # Use disk file names
im2b := imagecalc('im2b', '2*$im1')    # Use tool name
```

To include virtual images (which are not stored in disk) in LEL expressions you must use this feature.

4.11.6 Image Pixel Masks

See the Image `tool` and § 4.7 for basic information on pixel masks. See note223 for all the details of how image pixel masks are handled in LEL.

Here we just provide some basic knowledge. LEL, like other AIPS++ image handling software, applies the default mask if there is one. You can over-ride this in LEL expressions if you wish.

```
im := imagecalc('x1:nomask')     # Use no mask
im := imagecalc('x1:mask3')      # Use mask3 stored in image x1
```

You can also make what is called a transient conditional mask. Take the sub-expression `sum(x1[x1<5])`.

Let us say that image *x1* has no pixel mask to start with. The square brackets indicate an expression mask. When the result of that condition is True (effectively creating a transient mask) then the rest of the expression is evaluated. Thus, whenever a pixel in *x1* is less than 5, the value of that pixel is added to the sum.

Extending this slightly,

```
im := imagecalc('x2',  'x1[x1 < 5]');
```

In this case, the output image *x2* must be created with a pixel mask. It will be T for all pixels in *x1* whose value is less than 5. Otherwise it will be F. The mask given to *x2* is a physical mask stored with the image, whereas the expression mask is transient.

4.11.7 Regions

Like masks, image regions can be stored permanently in images (the Regionmanager does this). These regions can be accessed from within LEL expressions. The basic syntax is '[tablename::regionname]' (there are some shortcuts, see note 223). Thus

```
im := imagecalc('x1 + sum(x1[x2::region0])')
im := imagecalc('x1[region0]')
```

In the first example, a region stored in image *x2* called 'region0' is applied to image *x1* and the sum is found in that region and added to the values in image *x1*. In the second example, the region is looked up in image *x1* (a short cut).

It is also possible to use a mechanism similar to the `tool` substitution method to access regions made in *Glish* (i.e. ones not yet stored in an image). This is very convenient. For example,

```
im := image('z1')
im.shape()
#<< [64 64 128]
ndim := length(im.shape())
blc := array(1, ndim)
trc := im.shape()
trc[ndim] := 1;
r1 := drm.box(blc, trc)
im2 := imagecalc('z2',  '$im[$r1]')
```

will create an output image which is just the first plane of the input image. Note we have used the $ substitution syntax for both the image and the region.

4.11.8 Example - continuum subtraction

As an example of some of the Image `tool` functionality, let's write some *Glish* to mask some bad channels and then subtract the average of some continuum channels from all the channels in an image using the Image `tool` calculator.

```
include 'image.g'
include 'regionmanager.g'
include 'os.g'
#
infile := 'hcn'
subtfile := spaste(infile, '.subtracted')
contfile := spaste(infile, '.continuum')
dos.copy(infile, subtfile, overwrite=T)      # Copy as we will modify image insitu
#
im := image(subtfile)                        # Construct Image tool
cs := im.coordsys()                          # Get coordinate system
drm.setcoordinates(cs, verbose=F)            # Tell defaultregionmanager to use them
cs.done()
#
shp := im.shape()                            # Image shape
axis := 3                                    # Define spectral pixel axis
n := length(shp)
#
r1 := drm.wrange(pixelaxis=axis,             # Define region for first 5 spectral pixels
                 range="1pix 5pix")
if (is_fail(r1)) fail
x := ""
x[1] := spaste(n-4, 'pix')
x[2] := spaste(n, 'pix')
r2 := drm.wrange(pixelaxis=axis, range=x)    # Define region for last 5 spectral pixels
if (is_fail(r2)) fail
#
ok := im.set(pixelmask=F, region=r1)         # Mask first 5 spectral pixels
if is_fail(ok)) fail
ok := im.set(pixelmask=F, region=r2)         # Mask last 5 spectral pixels
if is_fail(ok)) fail
r1.done()                                    # Destroy region tools
r2.done()
#
cont := [=]                                  # Define continuum range
cont[1] := "1pix 20pix"
cont[2] := "55pix 64pix"
r := [=];                                    # Make a region for each range
for (i in 1:length(cont)) {
   r[i] := drm.wrange(pixelaxis=axis, range=cont[i])
   if (is_fail(r[i])) fail
```

```
}
union := drm.union(r)                        # Union of ranges
if (is_fail(union)) fail;
#
dos.remove(contfile)
ok := im.moments(outfile=contfile, moments=-1,   # Generate average of image
                 axis=axis, region=union)        # in specified contimuum ranges
if (is_fail(ok)) fail
#
expr := spaste('$im - ', contfile)           # line - continuum
ok := im.calc(expr)                          # apply expression in situ
if (is_fail(ok)) fail
#
im.view()                                    # display
```

4.12 Analysis

There are a number of dedicated mathematical analysis functions in the Image `tool` at this point. These are

4.12.1 Statistics

The Image `tool` function Image.statistics computes a range of statistics from the pixel values in the image. You can then plot them, list them and retrieve them (into a Glish record) for further analysis.

To evaluate the statistics over the whole image:

```
im := image('z2')                  # A 2-D image
im.stats()
```

giving the output

```
Number points =   6.422000e+03       Sum      =  -1.695736e+03
Flux density  =  -7.852021e+01 Jy
Mean          =  -2.640511e-01
Variance      =   2.712798e-01       Std dev  =   5.208453e-01
Rms           =   5.839183e-01

Minimum value  -1.865986e+00 at [79, 63] (23:59:47.733, +00.05.00.000)
Maximum value   1.478747e+00 at [29, 59] (00:00:14.400, +00.04.12.000)
```

This function also lets you evaluate the statistics over some repeating chunk of the image. For example, you can ask for your statistic(s) to be computed for each XY plane of a cube as a function of Z, or for each X-axis profile, and so on.

```
im := image('z3')                       # A 3-D image
im.stats(axes=[1,2])                    # Evaluate over XY planes as a function of Z
im2.stats(axes=[1,2], plotter='1/glish', plotstats="mean stddev")
```

gives the output:

```
Selected bounding box [1, 1, 1] to [155, 178, 64]
```

Plane	Freq	Npts	Sum	Mean	Rms	Std dev	Minimum	Maximum
1	1.413350e+09	2.759000e+04	2.168769e+02	7.860706e-03	9.708145e-02	9.676444e-02	-5.872678e-01	5.634090e-01
2	1.413370e+09	2.759000e+04	3.294856e+02	1.194221e-02	1.010458e-01	1.003395e-01	-4.426085e-01	7.099073e-01
3	1.413389e+09	2.759000e+04	1.082081e+02	3.922004e-03	1.055577e-01	1.054867e-01	-4.100212e-01	6.766760e-01
4	1.413409e+09	2.759000e+04	6.575802e+01	2.383400e-03	1.756554e-01	1.756424e-01	-9.973667e-01	8.662508e-01
5	1.413429e+09	2.759000e+04	4.585344e+02	1.661959e-02	1.878055e-01	1.870721e-01	-1.035953e+00	1.018446e+00
6	1.413448e+09	2.759000e+04	1.473713e+02	5.341474e-03	1.351079e-01	1.350048e-01	-5.234021e-01	1.195704e+00
7	1.413468e+09	2.759000e+04	2.559742e+02	9.277789e-03	1.433194e-01	1.430214e-01	-6.152341e-01	1.135976e+00
8	1.413488e+09	2.759000e+04	4.356140e+02	1.578883e-02	1.542441e-01	1.534367e-01	-6.993687e-01	1.215297e+00
9	1.413508e+09	2.759000e+04	3.723920e+02	1.349736e-02	1.642182e-01	1.636655e-01	-6.909095e-01	1.388082e+00
10	1.413527e+09	2.759000e+04	5.345861e+02	1.937608e-02	1.724699e-01	1.713811e-01	-6.358010e-01	1.335077e+00
11	1.413547e+09	2.759000e+04	7.577886e+02	2.746606e-02	1.788728e-01	1.767547e-01	-6.653469e-01	1.341522e+00
12	1.413567e+09	2.759000e+04	1.106117e+03	4.009122e-02	1.880124e-01	1.836915e-01	-7.188058e-01	1.282076e+00
13	1.413586e+09	2.759000e+04	1.329324e+03	4.818138e-02	1.962451e-01	1.902420e-01	-6.393265e-01	1.191303e+00
14	1.413606e+09	2.759000e+04	1.411702e+03	5.116716e-02	2.052504e-01	1.987740e-01	-5.953373e-01	1.225875e+00
15	1.413626e+09	2.759000e+04	1.354188e+03	4.908257e-02	2.138623e-01	2.081575e-01	-5.535692e-01	1.475392e+00
16	1.413645e+09	2.759000e+04	1.264175e+03	4.582004e-02	2.176246e-01	2.127501e-01	-6.773824e-01	1.470784e+00
17	1.413665e+09	2.759000e+04	1.298691e+03	4.707106e-02	2.112372e-01	2.059297e-01	-7.119259e-01	1.496961e+00
18	1.413685e+09	2.759000e+04	1.341463e+03	4.862133e-02	2.125130e-01	2.068799e-01	-6.245704e-01	1.562188e+00
19	1.413704e+09	2.759000e+04	1.442007e+03	5.226557e-02	2.084975e-01	2.018440e-01	-6.114445e-01	1.554983e+00
20	1.413724e+09	2.759000e+04	1.582090e+03	5.734287e-02	2.001675e-01	1.917816e-01	-5.846441e-01	1.583541e+00
21	1.413744e+09	2.759000e+04	1.501028e+03	5.440480e-02	1.893141e-01	1.813316e-01	-6.073643e-01	1.503856e+00
22	1.413763e+09	2.759000e+04	1.269058e+03	4.599702e-02	1.701235e-01	1.637903e-01	-4.517879e-01	1.371341e+00
23	1.413783e+09	2.759000e+04	7.299262e+02	2.645619e-02	1.468132e-01	1.444124e-01	-5.721810e-01	1.026282e+00
24	1.413803e+09	2.759000e+04	6.232603e+02	2.259008e-02	1.331807e-01	1.312533e-01	-6.960571e-01	7.298118e-01
25	1.413823e+09	2.759000e+04	5.981219e+02	2.167894e-02	1.249476e-01	1.230547e-01	-5.943727e-01	7.063599e-01
26	1.413842e+09	2.759000e+04	4.736431e+02	1.716720e-02	1.137065e-01	1.124051e-01	-6.665137e-01	6.953478e-01
27	1.413862e+09	2.759000e+04	4.054529e+02	1.469565e-02	1.171430e-01	1.162197e-01	-6.885164e-01	5.358027e-01
28	1.413882e+09	2.759000e+04	3.413342e+02	1.237166e-02	1.213791e-01	1.207491e-01	-7.589104e-01	6.029773e-01
29	1.413901e+09	2.759000e+04	2.388184e+02	8.655977e-03	1.278192e-01	1.275281e-01	-8.899862e-01	6.607218e-01
30	1.413921e+09	2.759000e+04	2.580464e+02	9.352897e-03	1.297307e-01	1.293954e-01	-9.948431e-01	8.899596e-01
31	1.413941e+09	2.759000e+04	2.157427e+02	7.819597e-03	1.258456e-01	1.256047e-01	-9.192608e-01	7.220768e-01
32	1.413960e+09	2.759000e+04	2.848281e+02	1.032360e-02	1.173345e-01	1.168816e-01	-8.530115e-01	8.228853e-01

4.12.2 Histograms

The Image `tool` function Image.histograms computes histograms from the pixel values in the image. You can then plot them, and retrieve them (into a Glish record) for further analysis.

This function lets you evaluate the histograms over some repeating chunk of the image. For example, you can ask for your histogram(s) to be computed for each XZ plane of a cube as a function of Y, or over the whole image, or for each X-axis profile and so on.

This example makes a histogram from the entire image.

```
im := image('z2')
im.histograms(plotter='1/glish')
```

4.12.3 Moments

The Image `tool` function Image.moments offers traditional image moment analysis (weighted sums of profiles) of images. It offers a wide range of moments, and a range of methods with which to calculate them.

Some of the methods are interactive; be aware that they are very user intensive, and really only practical for smallish numbers of spectra.

The command-line interface for this function is rather cumbersome. Therefore, a custom GUI interface has been created as well. This GUI can be accessed via the Toolmanager (if you ask Toolmanager to show you the function arguments for this function, by default it builds the custom GUI) and also from Glish:

```
im := image('mycube')
mg := im.momentsgui()
```

See the Image.momentsgui documentation for details. The idea of this GUI is that it is context sensitive; it disables parts of itself depending upon what you choose to do so that you are (hopefully) constrained to do something useful.

4.12.4 Convolution

There is a range of convolution `tools`.

2D convolution

You can convolve specified axes by a 2-D kernel via the function Image.convolve2d.

```
im := image('xfy')                                                  # RA/Freq/DEC
im2 := im.convolve2d(outfile='xfy.con', axes=[1,3], type='gauss',
                     major='20arcsec', minor='10arcsec', pa=45);
#
im := image('xyf')                                                  # RA/DEC/Freq
im2 := im.convolve2d(axes=[1,3], type='gauss',                      # Output is virtual
                     major='20pix', minor='10pix', pa=45);
```

In the first example, we convolve the RA/DEC plane of the image (which happens to be in order RA/Freq/DEC) by a gaussian. In the second example, we convolve the RA/Freq plane of the image by a gaussian. Because we are convolving unlike axes, we must specify the width of the kernel in pixels.

Separable convolution

This is available via the function Image.sepconvolve. You specify the axes to convolve, and the convolution model (from Gaussian, Hanning, Boxcar presently) and parameters. There is also a custom GUI interface provided via Image.sepconvolvegui. This convolution is Fourier based.

```
im := image('mycube')
im2 := im.sepconvolve (outfile='mycube.sepcon', axes=[1,3],
                        types="gauss boxcar", widths="10arcsec 20pix")
```

In this example we convolve axes 1 and 3 with a gaussian and boxcar respectively, and express the x and y kernel widths in world and pixel units, respectively.

If you only want Hanning convolution of one axis, then a separate function Image.hanning is also available. This convolution is direct (not Fourier based) and is faster (and more accurate) than single axis Hanning convolution via the function Image.sepconvolve.

```
im := image('mycube')
im2 := im.hanning (outfile='mycube.hann', axis=3)
```

Convolution by an array

You may convolve your image by any kernel by providing the convolution kernel in a *Glish* array to the function Image.arrconvolve.

```
im := image('myimage')
kernel := mykernel(...)         # Your code to generate the kernel
im2 := im.arrconvolve(outfile='myim.con', kernel=kernel)
```

4.12.5 Fourier Transform

With the Image `tool` function Image.fft you can Fourier Transform the image. It will automatically do just the sky-plane(s) by default, or you can specify which axes to transform.

```
myim := image('gc')
myim.fft(amp='amp.gc', phase='phase.im')     # Save amp and phase
amp := image('amp.gc')                       # Make Image tool
amp.view()                                   # Display
```

4.12.6 Regridding

With the Image `tool` function Image.regrid you can regrid the image to a specified Coordinate System. This supports reference frame changes (e.g. LSR to TOPO, J2000 to B1950 etc) as well as projection changes for the sky.

```
im1 := image('radio.image')
im2 := image('optical.image')
cs2 := im2.coordsys();
im3 := im1.regrid(outfile='radio.regridded', csys=cs2, shape=im2.shape());
```

In this example, we regrid a radio image onto the grid of an optical image - this probably (if the optical FITS image was correctly labelled !!) will involve a projection change (optical images are usually TAN projection, radio usually SIN).

```
im1 := image('radio.image')
cs1 := im1.coordsys();
cs1.referencecode('dir')
#<< J2000
cs2.setreferencecode(type='dir', value='B1950', adjust=T)
cs1.referencecode('dir')
#<< B1950
im3 := im1.regrid(outfile='radio.regridded', csys=cs1)
```

In this example, we regrid a radio image from J2000 to B1950. The Coordinate System is recovered from the image into a Coordsys `tool`. It is then used to change the Direction Coordinate reference to B1950 (adjusting the reference value as well). This Coordsys `tool` is then given to the regridding function as the template. Note that changing the Coordsys `tool` has no effect on the Coordinate System still in place in the original image.

4.13 Source/profile Fitting, Finding and Modifying

Current support is for the fitting of 2D models (currently only Gaussians are supported) to the sky. This is provided at two levels.

The Imagefitter `tool` is an interactive application with which you fit models to sources. Your image is displayed, and you interactively fit the specified region with the desired model. Multiple simultaneous models are supported. The source fits are stored in a Componentlist `tool` for further use.

This `tool` is based around the Image `tool` function Image.fitsky which fits the specified model to the specified region of the image and returns the fit in a Componentlist `tool` for further use.

```
fitter := imagefitter('myimage')    # Starts fitter GUI
cl := fitter.componentlist()        # Recover fits when done
```

Together with fitting there is a function which automatically finds sources called Image.findsources. This function is tuned to find strong point sources. Its job is not to find extended sources or weak sources. The result is also a Componentlist `tool` which can be used as an initial estimate for the fitting function Image.fitsky.

```
im := imagemaketestimage('zz')
est := im.findsources(nmax=3)
local pixels, mask, converged
fit := im.fitsky(pixels, mask, converged, estimate=est, models="gauss gauss gauss")
im.modify(model=fit, subtract=T)
```

In this example, we create the test image. Then we find an estimate of the strongest three (point) sources. We use this estimate to fit three simultaneous gaussians. Finally we subtract the fit from the image.

As well as fitting 2-D models of the sky, you can also fit 1-D models to profiles exctracted from an image. The Imageprofilefitter `tool` is an interactive application with which you fit models to 1-D profiles. Your image is displayed, and you interactively specify where the profile should be extracted from and then interactively fit it. You can also automatically fit many profiles in a region and save the fits in an image.

```
fitter := imageprofilefitter('myimage')    # Starts fitter GUI
r := fitter.getfit()                        # Recover fit into a record when done
```

4.14 Polarimetry

4.14.1 General

Polarimetric analysis of images is handled via an Imagepol `tool`. This tool is created from an AIPS++ image (like the Image `tool` is) and is specialized for polarimetric analysis of images. The input image must contain a Stokes coordinate with some combination of Stokes I, Q, U, and V.

The `tool` offers easy access to the different Stokes parameters in an image, computation of the standard secondary (e.g. polarized intensity) and tertiary images (e.g. fractional polarization) and their errors. There are also methods for deriving the Rotation Measure.

```
ip := imagepol('myim')              # Construct Imagepol tool from image
si := ip.stokesi()                  # Get Stokes I Image tool
si.view()                           # Display it
flp := ip.fraclinpol()              # Fractional linear polarization Image tool
tpi := ip.totpolint()               # Total polarized intensity Image tool
tpi.statistics()                    # Get statistics
tpi.view ()                         # View
imagedones()                        # Destroy all image tools
```

In this example, we construct and Imagepol `tool`. We then recover the Stokes I image, and compute the fractional linear polarization image and the total polarized intensity image. You get access to these via Image `tools` which are returned to you by the Imagepol `tool`. Therefore you can use all the Image `tool` functions like **statistics** and **view** on them.

```
lpa := ip.linpolposang()              # Linearly polarized position angle Image tool
lpaerr := ip.sigmalinpolposang()      # Error in linearly polarized position angle Image tool
lpa.view(mask='$lpaerr<5]')           # Display and mask bad when position angle error > 5 degrees
```

Here we compute the linearly polarized position angle and error images and return them as Image `tools`. Rather than mask the position angle image based on its error when the position angle image is created, we prefer to defer that to when the error image is actually used. This gives you greater flexibility. In this case, we use the `mask` argument (an 'on-the-fly' mask) with the **view** function so that pixels whose position angle error is greater than 5 degrees are masked bad and not displayed.

4.14.2 RotationMeasure

There are two methods for dealing with the generating the Rotation Measure. A traditional approach and a new Fourier approach.

```
ip := imagepoltestimage('iquv.im', rm=350)              # Makes test image and Imagepol tool from it
ip.rotationmeasure(rm='rm', rmerr='rmerr', rmmax=400)   # RM (traditional)
rm := image('rm')                                       # Image tool associated with RM
rmerr := image('rmerr')                                 # Image tool associated with RM error
#
rm.statistics(stats, list=F)
print stats.mean, ', ', stats.sigma
#<< 349.9826, 0.7228746

#
rm.view(mask='$rmerr<0.725')                           # Display and mask bad when error > 0.725rad/m/m
ip.done()                                              # Destroy this Imagepol tool
#
ip := imagepoltestimage('iquv.im', nx=8, ny=8, bw=8e6, nf=512, rm=1e6)   # Makes test image
ip.fourierrotationmeasure(amp='amp');          # RM (Fourier)
amp := image('amp')
amp.view(order=[3,1,2])                                # Display with RM along x axis
```

In this example, we first use the alternative Imagepol `tool` constructor **imagepoltestimage** which generates an IQUV cube for the specified Rotation Measure. Then we use the traditional method and a new Fourier-based method to recover the RM.

The test image has constant Stokes I and V (plus noise) and a sinusoidally (because of the RM) frequency dependent Q and U (plus noise). The traditional method recovers the RM and error images and we display the RM when the error is less than the specified (contrived) value. You can see from the statistics that the correct value is recovered.

The Fourier method recovers the linearly polarized intensity (we save the amplitude) as a function of Rotation Measure which we also display. We display the image (cube) with the RM along the X axis of the display to see the recovered signal.

4.14.3 Vector overlay

A traditional method of viewing linear polarization is to overlay vectors whose length is proportional to the fractional linear polarization, or total linearly polarized intensity and whose position angle is the linearly polarization position angle. This display is done with the Viewer `tool` as follows:

```
include 'imagepol.g'
include 'viewer.g'
#
p := imagepol('iquv.im')  # Make Imagepol tool
p.complexlinpol('clp')    # Make complex image of linear polarization disk file
dv.gui()                  # Start data manager to give access to Complex image
```

Now select the image *clp* with the Viewer's data manager GUI, click Vector Map in the DisplayData Type selection area and the vector display will appear on the display. You can choose to also display raster or contour maps of whatever you like to complete the display. The Viewer's Vector map display capability also offers you amplitude noise debiasing and the On-The-Fly mask through the Adjust GUI.

4.15 Communications

The communications of AIPS++ are handled by *Glish*, which has the capability to send and receive events. These events contain information which is then used by the process capturing the event. This is the way in which we allow `tools` to communicate with each other. One can get terribly clever and make very complex connections, rapidly causing confusion.

In the context of image analysis, there are some communications between `tools` which you should understand. We have kept them reasonably simple so that they are clear, although sometimes too cumbersome. Presently, these communications are mainly to do with transmitting regions.

4.15.1 The Regionmanager and region widgets

When you use the Toolmanager, the `tool` function arguments requiring an image region-of-interest are managed by a specialized region-entry widget. Like all of our entry widgets, this one has a spanner (wrench) menu. One of the menu items activates the default Regionmanager (**drm**) GUI. With this you can create a region and then send it to the region entry widget (this process is described in more detail in the Regionmanager Regionmanager.gui `tool` function documentation. When you send the region to the widget, you can specify whether the connection between the the region widget and the Regionmanager remains open or not (also see above documentation).

You can make a simple test with

```
include 'guientry.g'
ge := guientry();
r := ge.region(allowunset=T)
```

and then use the spanner (wrench) menu to start the regionmanager GUI.

Note that the communications are designed so that only one region entry widget at a time can be connected to the Regionmanager GUI (otherwise there would be confusion about which regions are going where).

4.15.2 The Regionmanager and images

It is possible to request 'interactive' region creation from the Regionmanager GUI (select desired Image `tool` from Regionmanager list and click 'interactive' from list of possible region types).

This action will start the Image.view function of the specified image (the 'break' button on the display panel will be active - normally it is inactive) and the Regionmanager will collect the interactive regions that you make (by double clicking inside the delineated region - see also Image.view documentation for instructions on how to make them). You could then send this region along to a region-entry widget for example.

Although we could connect a region-entry widget directly to the image display (cutting out the Regionmanager), it is currently felt that until we have a real communications manager in AIPS++, that such clever connections will be too unclear (the user must have some way of knowing what is connected to what). Thus we persist with the slightly less sophisticated model of the Regionmanager always being the hub for region communications.

4.16 Error Messages

The Images module is consistent in its approach to error handling. All `tool` functions return a fail condition if an exception (unrecoverable error) is generated. Thus, if you are writing *Glish* scripts, you can do robust error checking. The documentation for each `tool` function tells you if a fail is a possible return. The error messages for these exceptions are posted to the logger. If the logger GUI is running, you will see a Red Box of Death with the error message in it.

```
const mystats := function (name)
{
  im := image(name)
  if (is_fail(im)) fail;
  return im.statistics()
}
```

In this example, we make a function which takes a string and tries to create an Image `tool` from it. If it fails, the error condition is propagated out so that the caller of the function can check for the fail as well.

4.17 References

Here are some useful documentation links:

Imager is the synthesis imaging module.

Images is the Images module.

Image is the Image `tool`.

Regionmanager is the Regionmanager `tool`.

Imagefitter is the Imagefitter `tool`.

Imageprofilefitter is the Imageprofilefitter `tool`.

Imagepol is the Imagepol `tool`.

Viewer is the Viewer display `tool`.

User Reference Manual is the User's Reference Manual

Glossary is the Glossary

Copyright

The AIPS++ consortium may be reached by email at aips2-request@nrao.edu. The postal address is: AIPS++ Consortium, c/o NRAO, 520 Edgemont Rd., Charlottesville, Va. 22903-2475 USA.