Getting Results with AIPS++

AIPS++ Group, Editor: Neil Killeen

June 21, 2002

Contents

1 Plotting **9**
- 1.1 Summary . . . 9
- 1.2 Creating a PGplotter `tool` . . . 9
- 1.3 Interacting with the PGplotter . . . 10
- 1.4 Example: Reading Data from an ASCII Table and Making Plots 11
- 1.5 Example: Using the PGPLOT primitives . . . 13
- 1.6 Example: Plotting to Several Panels in a PGplotter Window . . 14

2 Processing Tabular Information **17**
- 2.1 Summary . . . 17
- 2.2 The basics . . . 18
  - 2.2.1 Reading and writing . . . 18
- 2.3 Examples . . . 18
  - 2.3.1 Example 1: Reading from an ASCII file into a Table . . . 18
  - 2.3.2 Example 2: Reading, fitting and plotting from an ASCII file . . . 19
  - 2.3.3 Example 3: Reading, fitting and plotting from an ASCII file . . . 22
  - 2.3.4 Example 4: Editing Tables From AIPS . . . 24

3 Display of Data **29**
- 3.1 Summary . . . 29
- 3.2 Viewing a raster map . . . 30
  - 3.2.1 Procedures . . . 30
- 3.3 Viewing a contour map . . . 34
  - 3.3.1 Procedures . . . 34
- 3.4 Viewing a vector map . . . 35
  - 3.4.1 Procedures . . . 35
- 3.5 Viewing a marker map . . . 37
  - 3.5.1 Procedures . . . 37
- 3.6 Viewing a catalog overlay . . . 38
  - 3.6.1 Procedures . . . 38
  - 3.6.2 Notes . . . 39
- 3.7 Viewing a measurement set . . . 39

3.8 Operating the Viewer DisplayPanel . . . 40
3.9 Adjusting display parameters . . . 40
3.10 Advanced, worked examples . . . 41
3.10.1 Overlaying contours on raster maps . . . 41
3.10.2 Displaying complex data . . . 42
3.11 3D slice display of data cubes . . . 44
3.12 Displaying channel maps of a data cube . . . 44
3.13 A step-by-step recipe to make a post-script plot of your image . . . 44

4 Image Analysis **47**
4.1 Summary . . . 47
4.2 Creating an Image tool . . . 49
4.2.1 From a built-in test image . . . 49
4.2.2 From existing AIPS++ image files . . . 49
4.2.3 From foreign formats (e.g. FITS) . . . 50
4.2.4 From simple shapes or Glish arrays . . . 51
4.2.5 From image expressions . . . 51
4.3 Virtual Images . . . 51
4.4 Summarizing the Image . . . 52
4.5 Display . . . 53
4.5.1 Function view . . . 53
4.5.2 From the catalog . . . 54
4.5.3 From the spanner (wrench) menu . . . 54
4.5.4 With the plain Viewer . . . 54
4.6 Coordinates . . . 54
4.6.1 General . . . 54
4.6.2 Retrieving and changing the Coordinate System . . . 55
4.6.3 Coordinate conversion . . . 55
4.7 Mask handling . . . 56
4.8 Utility . . . 57
4.8.1 Deletion and renaming . . . 57
4.8.2 History . . . 57
4.8.3 Locking . . . 58
4.9 Regions of interest . . . 58
4.10 Pixel Access . . . 59
4.10.1 Getting and putting pixels . . . 59
4.10.2 Set a scalar value . . . 59
4.10.3 Replace masked pixels . . . 59
4.11 Image Calculator . . . 60
4.11.1 Introduction . . . 60
4.11.2 Basic Access . . . 60
4.11.3 Special characters in file names . . . 61
4.11.4 Conformance of expressions . . . 61
4.11.5 Tool substitution in expressions . . . 61
4.11.6 Image Pixel Masks . . . 61
4.11.7 Regions . . . 62

4.11.8 Example - continuum subtraction . . . 63
4.12 Analysis . . . 64
4.12.1 Statistics . . . 64
4.12.2 Histograms . . . 65
4.12.3 Moments . . . 66
4.12.4 Convolution . . . 66
4.12.5 Fourier Transform . . . 67
4.12.6 Regridding . . . 67
4.13 Source/profile Fitting, Finding and Modifying . . . 68
4.14 Polarimetry . . . 69
4.14.1 General . . . 69
4.14.2 RotationMeasure . . . 70
4.14.3 Vector overlay . . . 71
4.15 Communications . . . 71
4.15.1 The Regionmanager and region widgets . . . 71
4.15.2 The Regionmanager and images . . . 72
4.16 Error Messages . . . 72
4.17 References . . . 73

5 Synthesis Calibration **75**
5.1 Introduction . . . 75
5.2 Calibration philosophy . . . 76
5.2.1 The Measurement Equation . . . 76
5.2.2 Data representation and the calibration mechanism . . . 79
5.2.3 What are the current calibration capabilities in AIPS++? . . . 80
5.3 Practical use of the calibrater tool . . . 80
5.3.1 Setting up the calibrater tool . . . 80
5.3.2 Initial calibrator models . . . 81
5.3.3 Uv-data selection . . . 81
5.3.4 Calibration table conventions . . . 82
5.3.5 Solving for visibility-plane effects . . . 82
5.3.6 Establishing the flux density scale . . . 82
5.3.7 Correcting the observed data . . . 83
5.3.8 Extending calibration to bandpass, polarization, etc. . . . 83
5.3.9 Self-calibration . . . 85
5.4 References . . . 86

6 Imaging, Deconvolution and Self-calibration **87**
6.1 Overview . . . 87
6.1.1 Required Tools . . . 87
6.1.2 Data Requirements . . . 88
6.2 Setting up Imager . . . 88
6.2.1 Construction . . . 89
6.2.2 Selecting Data . . . 89

6.2.3 Setting the Image Parameters . . . 89
6.2.4 Channel Selection and Combination . . . 90
6.2.5 Weighting . . . 92
6.3 Creating and Deconvolving Images . . . 93
6.3.1 Creating Images . . . 93
6.3.2 Deconvolution . . . 94
6.3.3 Specifying the deconvolution region . . . 95
6.4 Wizard interface to Imager . . . 96
6.5 Component Models and Primary Beams . . . 96
6.6 Multi-scale CLEAN . . . 97
6.6.1 General . . . 97
6.6.2 The `imagermultiscale()` function . . . 98
6.7 Wide-field imaging and Mosaicing . . . 99
6.8 Combining single dish and interferometer images . . . 99
6.9 Self-calibration . . . 99
6.10 Summary of Imager functions . . . 100
6.10.1 Setting/seeing the basic state . . . 100
6.10.2 uv selection and filtering . . . 101
6.10.3 Masking . . . 101
6.10.4 Create images . . . 101
6.10.5 Deconvolution . . . 101
6.10.6 Image Combination . . . 102
6.10.7 Utility . . . 102
6.10.8 Calibration and self-calibration . . . 102
6.11 Examples of Imaging Scripts . . . 103
6.11.1 Multi-frequency synthesis image of one field . . . 103
6.11.2 Multiple regions on the sky simultaneously . . . 103
6.11.3 Spectral-line imaging . . . 104
6.11.4 Using Clean and MEM . . . 105
6.11.5 Deconvolver and multi-scale CLEAN example . . . 106
6.11.6 Imaging and Self-Calibrating . . . 107

7 Wide field imaging 109
7.1 Background . . . 109
7.1.1 Possible solution to the wide-field imaging problem . . . 110
7.2 Available Tools . . . 110
7.2.1 Basic capabilities . . . 111
7.2.2 Faceting size . . . 111
7.2.3 Deconvolution . . . 112
7.2.4 Self-calibration . . . 112
7.2.5 Masking . . . 112
7.2.6 Outlier fields . . . 113
7.2.7 Models . . . 114
7.3 A worked example: VLA 4m imaging of Coma . . . 114
7.4 Special cases . . . 115
7.5 Troubleshooting . . . 116

7.5.1 Insufficient faceting . . . 116
7.5.2 Non-isoplanatism . . . 116
7.5.3 Asymmetric primary beams . . . 116
7.5.4 Clean diverging on the edges of the facets . . . 116
7.6 Bibliography . . . 116

8 Mosaicing (Multi-field imaging) 119
8.1 Mosaicing Background . . . 119
8.1.1 The AIPS++ Mosaicing Solution . . . 120
8.1.2 Advantages of Incremental Deconvolution with an Approximate PSF . . . 121
8.2 Mosaicwizard for Quick, Simple Mosaicing . . . 121
8.3 Fundamental and Necessary Details . . . 121
8.3.1 Set the Data Fields . . . 121
8.3.2 Set the Image . . . 122
8.3.3 Setting the Voltage Pattern (primary beam) . . . 122
8.3.4 Weighting . . . 123
8.3.5 Deconvolving . . . 123
8.4 Advanced Details . . . 123
8.4.1 Controlling the Major Cycles . . . 123
8.4.2 Details with Multi-Scale CLEAN . . . 124
8.4.3 The `imagermultiscale()` Function . . . 125
8.4.4 Details with MEM . . . 125
8.4.5 Using Convolutions Instead of Visibility Subtraction . . . 125
8.4.6 Outlier Fields . . . 125
8.4.7 Component Models . . . 126
8.4.8 Flux Scale Images . . . 127
8.4.9 Masks . . . 127
8.5 Self-calibration . . . 128
8.6 An Example Mosaicing Script . . . 129
8.7 Bibliography . . . 130

9 Single Dish Analysis 133
9.1 Introduction . . . 133
9.2 Getting Data Into Dish . . . 134
9.2.1 MeasurementSets . . . 134
9.2.2 Facility filler (e.g., GBT) . . . 134
9.2.3 SDFITS data file (e.g., Arecibo Observatory) . . . 135
9.2.4 UniPOPS SDD data file . . . 135
9.2.5 Dish demo data . . . 137
9.3 The Dish Graphical User Interface . . . 138
9.3.1 Results Manager . . . 138
9.3.2 Menubar and Message Line . . . 139
9.3.3 Browsing . . . 139
9.3.4 Inspecting . . . 140
9.3.5 Operations . . . 140

9.4 Saving/Restoring State . . . 150
9.5 The Dish Plotter . . . 152
9.6 The Dish Command Line Interface . . . 156
9.7 Recipes . . . 157
9.7.1 Recipe 1: Change syntax to suit . . . 157
9.7.2 Recipe 3: Add a function to DISH (or fun with extensibility) 157
9.8 sditerator . . . 158
9.9 SDFITS . . . 158
9.9.1 EXTNAME keyword . . . 158
9.9.2 Virtual columns . . . 158
9.9.3 The DATA column and the DATA axes . . . 159
9.9.4 CORE keywords and columns . . . 161
9.9.5 SHARED keywords and columns . . . 162
9.9.6 Other columns . . . 164
9.9.7 Multiple SDFITS tables in a single file . . . 164

10 VLA reduction 165
10.1 Introduction . . . 165
10.2 MeasurementSets . . . 166
10.3 Basic initialization . . . 167
10.4 Basic Calibration Fundamentals . . . 171
10.4.1 Summary of Data Reductions Steps . . . 172
10.5 Example data and scripts . . . 174
10.6 Filling VLA data into AIPS++ . . . 175
10.6.1 Filling data from VLA archive format . . . 175
10.6.2 Filling data from UVFITS format . . . 176
10.7 Data examination and inspection . . . 177
10.8 Data editing . . . 177
10.8.1 Interactive editing . . . 180
10.9 Basic Gain Calibration . . . 181
10.9.1 Setting the calibration source model . . . 181
10.9.2 Solving for complex gain, G . . . 182
10.9.3 Bandpass calibration . . . 184
10.9.4 Transfer the flux density scale . . . 185
10.9.5 Correct the data . . . 186
10.10Instrumental Polarization Calibration . . . 187
10.10.1continuum polarimetry case (only) . . . 188
10.11Imaging . . . 190
10.12Self-Calibration . . . 195
10.13Image analysis . . . 200
10.14Cleanup . . . 206
10.15References . . . 207
10.16Appendix 1 . . . 207
10.17Appendix 2 . . . 208

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)
```

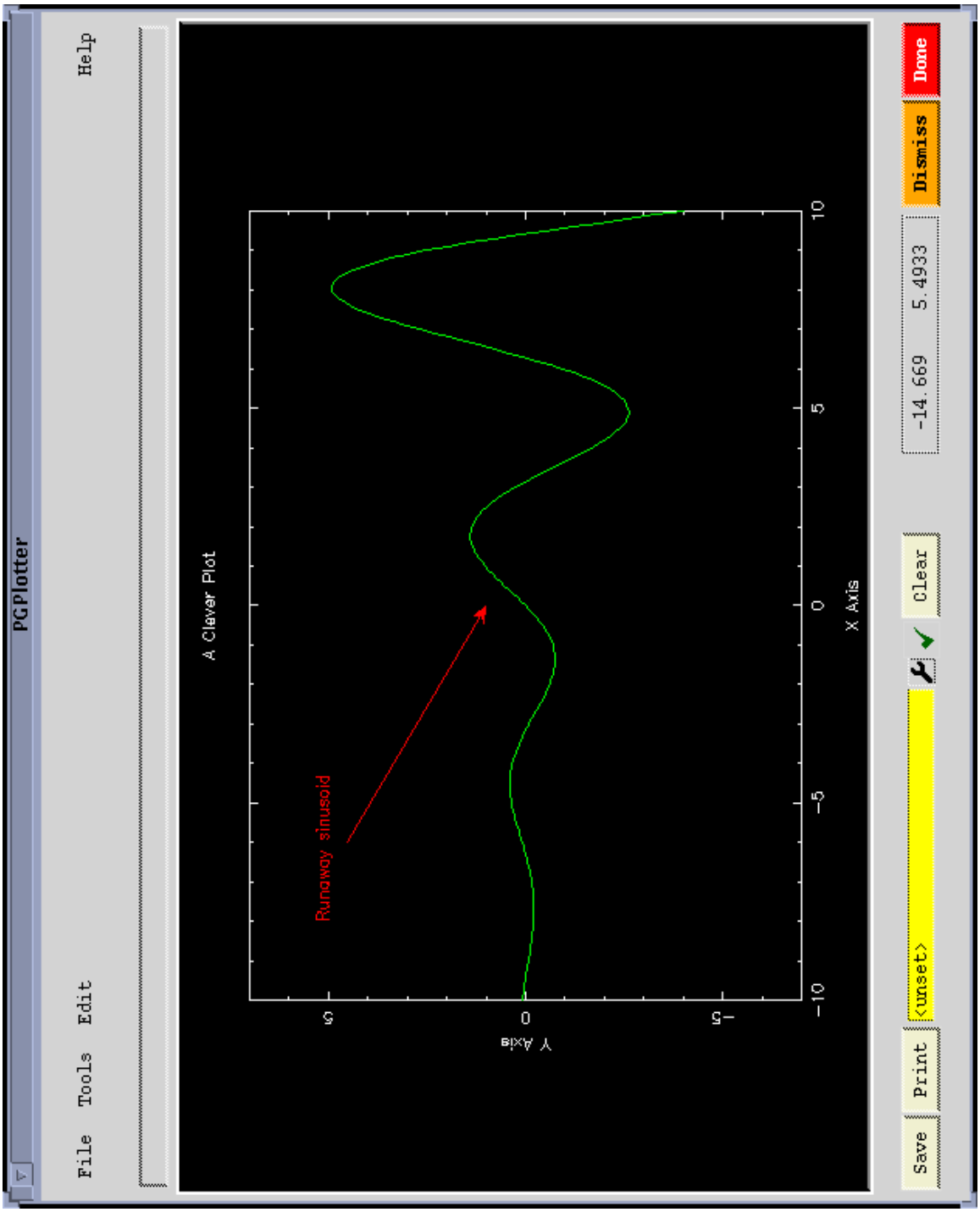
```
pg.line(x,y)
pg.sci(2)
pg.ptxt(-8,5,0,0,'Runaway sinusoid')
pg.arro(-6,4.5,0,1)
```

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
Y Axis
X Axis
10
5
0
−5
−10
−14.669 5.4933
Clear
Dismiss
Done
Save Print
<unset>

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:

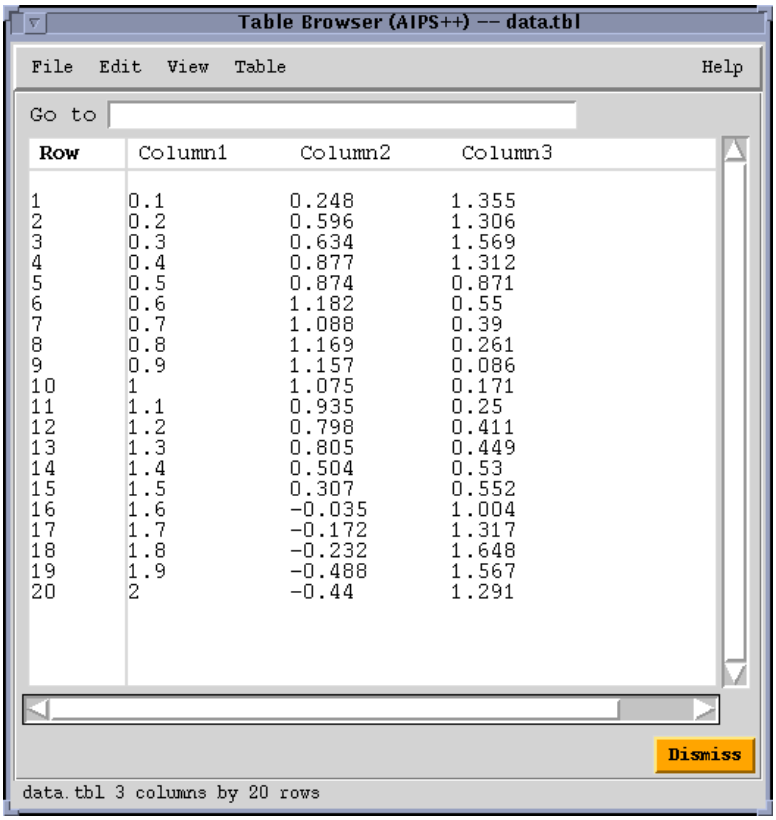


Figure 2.1: The Table browser display

```
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 ta
#
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**.

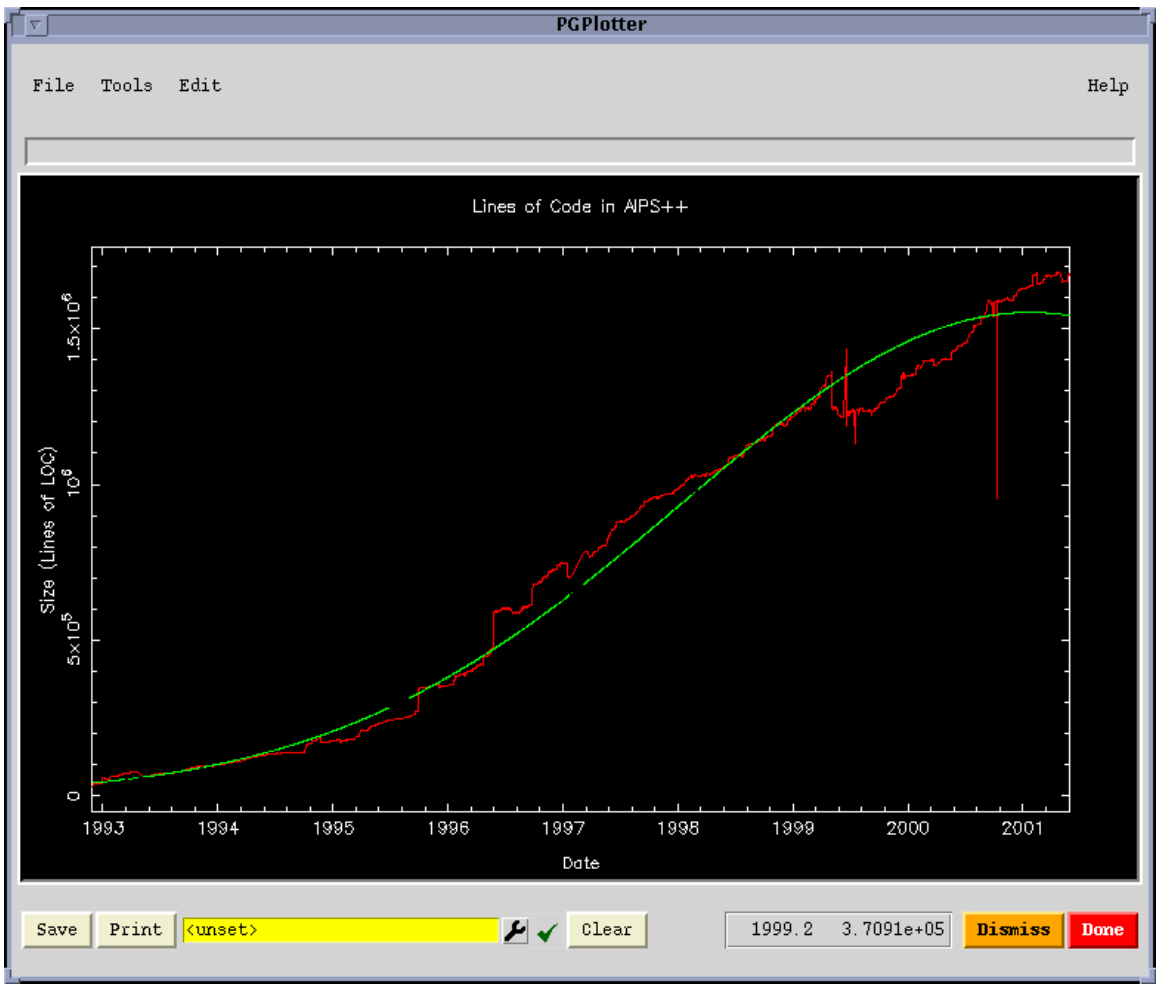


Figure 2.2: Lines of code in AIPS++

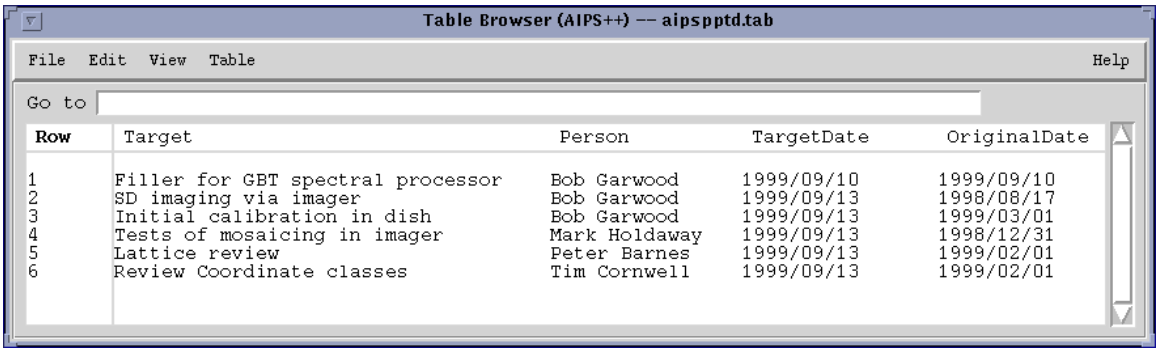


Table Browser (AIPS++) -- aipspptd.tab
File Edit View Table Help
Go to
Row Target Person TargetDate OriginalDate
1 Filler for GBT spectral processor Bob Garwood 1999/09/10 1999/09/10
2 SD imaging via imager Bob Garwood 1999/09/13 1998/08/17
3 Initial calibration in dish Bob Garwood 1999/09/13 1999/03/01
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`. Presently, this is restricted to images, *Glish* arrays and Tables holding source catalogs in a specific format. In the future, display of generic Tables and Measurement Sets 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:

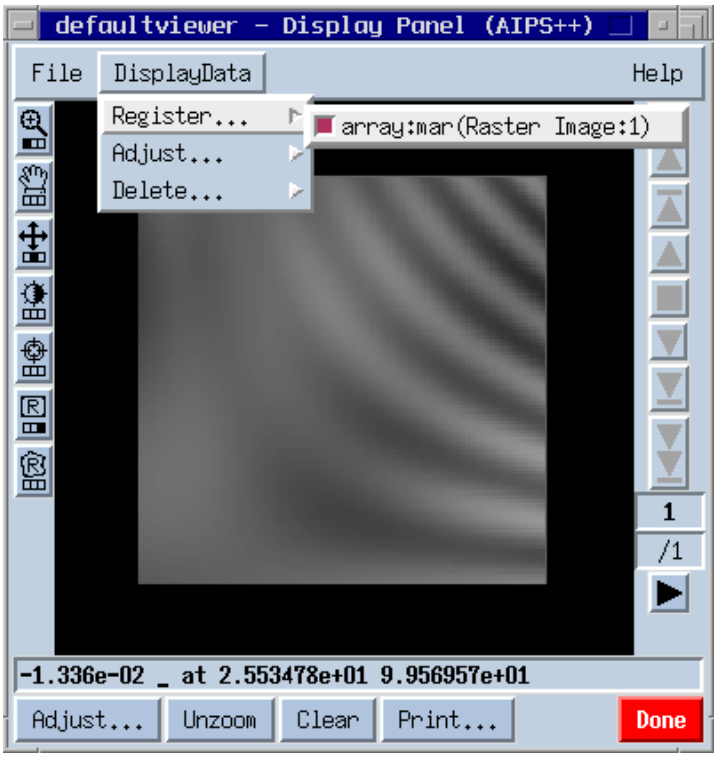


Figure 3.1: Registering a DisplayData on a Viewer DisplayPanel

```
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

Presently, 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('mydata.txt',hasheader=T,      # tell the tool which columns
              'RA','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 presently 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 o
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 te
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
-------------------------------------------------------------------------------------
1    1     Direction Right Ascension   SIN  512   10  00:00:00.000     6.00 -6.000000e
2    1     Direction Declination       SIN  512   20 +00.00.00.000    11.00  6.000000e
3    2     Spectral  Frequency             1024   30  1.415000e+09    16.00  1.000000e
                     Velocity                        1.140944e+03    16.00 -2.110611e
```

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 := imagemaketestimate()
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
1     1.413350e+09 2.759000e+04  2.168769e+02  7.860706e-03  9.708145e-02  9.676444e-02  -5.87267
2     1.413370e+09 2.759000e+04  3.294856e+02  1.194221e-02  1.010458e-01  1.003395e-01  -4.42608
3     1.413389e+09 2.759000e+04  1.082081e+02  3.922004e-03  1.055577e-01  1.054867e-01  -4.10021
4     1.413409e+09 2.759000e+04  6.575802e+01  2.383400e-03  1.756554e-01  1.756424e-01  -9.97366
5     1.413429e+09 2.759000e+04  4.585344e+02  1.661959e-02  1.878055e-01  1.870721e-01  -1.03595
6     1.413448e+09 2.759000e+04  1.473713e+02  5.341474e-03  1.351079e-01  1.350048e-01  -5.23402
7     1.413468e+09 2.759000e+04  2.559742e+02  9.277789e-03  1.433194e-01  1.430214e-01  -6.15234
8     1.413488e+09 2.759000e+04  4.356140e+02  1.578883e-02  1.542441e-01  1.534367e-01  -6.99368
9     1.413508e+09 2.759000e+04  3.723920e+02  1.349736e-02  1.642182e-01  1.636655e-01  -6.90909
10    1.413527e+09 2.759000e+04  5.345861e+02  1.937608e-02  1.724699e-01  1.713811e-01  -6.35801
11    1.413547e+09 2.759000e+04  7.577886e+02  2.746606e-02  1.788728e-01  1.767547e-01  -6.65346
12    1.413567e+09 2.759000e+04  1.106117e+03  4.009122e-02  1.880124e-01  1.836915e-01  -7.18805
13    1.413586e+09 2.759000e+04  1.329324e+03  4.818138e-02  1.962451e-01  1.902420e-01  -6.39326
14    1.413606e+09 2.759000e+04  1.411702e+03  5.116716e-02  2.052504e-01  1.987740e-01  -5.95337
15    1.413626e+09 2.759000e+04  1.354188e+03  4.908257e-02  2.138623e-01  2.081575e-01  -5.53569
16    1.413645e+09 2.759000e+04  1.264175e+03  4.582004e-02  2.176246e-01  2.127501e-01  -6.77382
17    1.413665e+09 2.759000e+04  1.298691e+03  4.707106e-02  2.112372e-01  2.059297e-01  -7.11925
18    1.413685e+09 2.759000e+04  1.341463e+03  4.862133e-02  2.125130e-01  2.068799e-01  -6.24570
19    1.413704e+09 2.759000e+04  1.442007e+03  5.226557e-02  2.084975e-01  2.018440e-01  -6.11444
20    1.413724e+09 2.759000e+04  1.582090e+03  5.734287e-02  2.001675e-01  1.917816e-01  -5.84644
21    1.413744e+09 2.759000e+04  1.501028e+03  5.440480e-02  1.893141e-01  1.813316e-01  -6.07364
22    1.413763e+09 2.759000e+04  1.269058e+03  4.599702e-02  1.701235e-01  1.637903e-01  -4.51787
23    1.413783e+09 2.759000e+04  7.299262e+02  2.645619e-02  1.468132e-01  1.444124e-01  -5.72181
24    1.413803e+09 2.759000e+04  6.232603e+02  2.259008e-02  1.331807e-01  1.312533e-01  -6.96057
25    1.413823e+09 2.759000e+04  5.981219e+02  2.167894e-02  1.249476e-01  1.230547e-01  -5.94372
26    1.413842e+09 2.759000e+04  4.736431e+02  1.716720e-02  1.137065e-01  1.124051e-01  -6.66513
27    1.413862e+09 2.759000e+04  4.054529e+02  1.469565e-02  1.171430e-01  1.162197e-01  -6.88516
28    1.413882e+09 2.759000e+04  3.413342e+02  1.237166e-02  1.213791e-01  1.207491e-01  -7.58910
29    1.413901e+09 2.759000e+04  2.388184e+02  8.655977e-03  1.278192e-01  1.275281e-01  -8.89986
30    1.413921e+09 2.759000e+04  2.580464e+02  9.352897e-03  1.297307e-01  1.293954e-01  -9.94843
31    1.413941e+09 2.759000e+04  2.157427e+02  7.819597e-03  1.258456e-01  1.256047e-01  -9.19260
32    1.413960e+09 2.759000e+04  2.848281e+02  1.032360e-02  1.173345e-01  1.168816e-01  -8.53011
```

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 Imag
lpa.view(mask='$lpaerr<5]')           # Display and mask bad when position angle error
```

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
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
#
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 erro
ip.done()                                               # Destroy this Imagepol tool
#
ip := imagepoltestimage('iquv.im', nx=8, ny=8, bw=8e6, nf=512, rm=1e6)   # Makes test i
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

Chapter 5

Synthesis Calibration

Athol Kemball & George Moellenbrock

5.1 Introduction

This chapter describes how to calibrate synthesis data. At present, this covers data only from connected-element arrays or VLBI data for which fringe-fitting has already been performed elsewhere. The discussion currently covers treatment of calibration effects in the uv-plane; calibration of image-plane effects will be described in a future version of this document.

The major AIPS++ `tools` that one will need to use are:

Calibrater is used for applying or solving for a range of uv-plane calibration effects, and plotting of calibration solutions.

Imager is used for gridding and Fourier transformaton of a MeasurementSet, deconvolution via various methods, and prediction of model visibilities from deconvolved images. Calibration of image-plane effects (those that are a function of direction in the sky) are handled in this `tool`.

Msplot is used for plotting and editing of visibility data, and is useful for investigation and editing during calibration cycles.

Componentmodels are used to represent the sky brightness by discrete components. Such models may be used in imaging and calibration to improve the quality of the results.

In addition, you will need to use the Viewer (see also the Display section of `Getting Results`) to examine your images, and perhaps the MeasurementSet `tool` MS to load and save MeasurementSets.

5.2 Calibration philosophy

As a starting point, it is useful to consider the overall calibration philosophy used in AIPS++. The calibration system has been designed to meet several important goals: i) support for synthesis *and* single-dish radio-telescopes in a generic formalism which minimizes instrument-specific differences; ii) modeling of all relevant physical effects which may corrupt the observed data, in both the uv- and image-plane, such as propagation or instrumental response factors, in a unified framework; and iii) support for arbitrary parametrization of the individual calibration effects.

5.2.1 The Measurement Equation

All radio-astronomical synthesis data reduction packages have an implicit calibration formalism, which includes a set of calibration effects which may be applied or solved for, and a mathematical framework for how these provide a relationship between the observed and corrected data. Even if this mathematical relationship is not given explicilty, it is, of necessity, enshrined in the infrastructure code.

AIPS++ has explicitly adopted the measurement equation (ME) formalism (Hamaker, Bregman and Sault 1996; Sault, Hamaker and Bregman 1996; Hamaker and Bregman 1996, Hamaker 1998, Noordam 1995; and Cornwel 1995) as the underlying calibration framework. It meets the objectives described in the preceding paragraph, and is easily extended to full coverage of a complete set of image-plane calibration effects (see AIPS++ Note 191). The generic nature of the ME, and its independence on the polarization basis for the data or the calibration parametrization are particularly strong advantages.

A misperception of the measurement equation is that it seeks to describe the physical processes or environment affecting the observed data at a needless or semi-infinite level of detail. This is mistaken; the arbitrary parametrization of the individual calibration effects allows the level of detail to be set at an arbitrary level of coarseness, consistent with available data or even expediency.

It is useful to examine a mathematical representation of the calibration formalism, as it has been adopted. Before describing the measurement equation in this form we need to consider some preliminary definitions for this particular formalism. A generic interferometer is considered to measure a four-vector of cross-correlations $\vec{V}$ between two recorded polarizations (p, q) for each of two feeds (i, j) characterizing an individual baseline. This measured vector includes instrumental calibration effects and is denoted by:

$$\vec{V}_{\mathrm{ij}} = \begin{pmatrix} V_{\mathrm{pp}} \\ V_{\mathrm{pq}} \\ V_{\mathrm{qp}} \\ V_{\mathrm{qq}} \end{pmatrix}_{ij} \tag{5.1}$$

The ME relates the measured vector $\vec{V}$ to the true polarized sky brightness,

$\vec{\mathcal{I}}$, which is expressed in a Stokes basis as:

$$\vec{\mathcal{I}} = \begin{pmatrix} I \\ Q \\ U \\ V \end{pmatrix} \tag{5.2}$$

Different instrumental and propagation effects are modeled in the ME as calibration components and take the form of four-by-four matrices, acting on four-vectors of the type given above. The calibration components are subdivided into two general categories based on whether they act in the image- or uv-plane, and they are expressed in the polarization basis (p, q) of the feed, which may typically be circular or linear. Calibration components which are feed-based are constituted as the direct matrix product of separate two-by-two Jones matrices for each feed on the baseline under consideration, as:

$$J_{ij} = \left[J_i \otimes J_j^*\right] \tag{5.3}$$

For a particular calibration component (types listed below), the diagonal elements in the two-by-two Jones matrices describe response to like polarization, and the off-diagonal describe the response to opposite polarization. The direct matrix product of two matrices is defined as:

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \otimes \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}b_{00} & a_{00}b_{01} & a_{01}b_{00} & a_{01}b_{01} \\ a_{00}b_{10} & a_{00}b_{11} & a_{01}b_{10} & a_{01}b_{11} \\ a_{10}b_{00} & a_{10}b_{01} & a_{11}b_{00} & a_{11}b_{01} \\ a_{10}b_{10} & a_{10}b_{11} & a_{11}b_{10} & a_{11}b_{11} \end{pmatrix} \tag{5.4}$$

The direct product of two Jones matrices thus describes the calibration component's impact on all possible correlation combinations between two feeds.

Provision is also made for multiplicative and additive baseline-based calibration components, which are not decomposed into feed-based terms. These take the form of four-by-four and four-by-one matrices, respectively.

The full ME including image-plane and uv-plane calibration effects can be defined in terms of the quantities described above as:

$$\begin{aligned} \vec{V}_{ij} = \quad & M_{ij} \prod \left[{J^{vis}}_i \otimes {J^{vis}}^*_j\right] \times \\ & \sum_k \prod \left[{J^{sky}}_i\left(\underline{\rho}_k\right) \otimes {J^{sky}}^*_j\left(\underline{\rho}_k\right)\right] \times \\ & S\, \vec{\mathcal{I}}_k\, e^{-2\pi i\left(\underline{r}_i - \underline{r}_j\right)\underline{\rho}_k} + A_{ij} \end{aligned} \tag{5.5}$$

where:

- $\left[{J^{vis}}_i \otimes {J^{vis}}^*_j\right]$ and $\left[{J^{sky}}_i\left(\underline{\rho}_k\right) \otimes {J^{sky}}^*_j\left(\underline{\rho}_k\right)\right]$ are individual uv-plane and image-plane calibration components respectively.

- $\underline{\rho}_k$ denotes direction, subscript k indexes directions, and $(\underline{r}_i - \underline{r}_j)$ denotes the interferometer baseline vector.
- The four-by-four matrix S converts the sky brightness Stokes polarization basis to the polarization basis used in the interferometer itself.
- M_{ij} and A_{ij} denote multiplicative and additive baseline-based instrumental effects respectively.
- $\prod$ and $\sum$ denote product and summation, respectively.

The ME thus reflects the standard Fourier transform relationship between the uv- and image-planes, and allows for generic calibration in both domains. Time and frequency averaging are implicit in the ME form given here, and are assumed to be over ranges shorter than the timescale for variation of any of the terms in the equation.

In the case of fields-of-view narrow enough that the $J_i\left(\underline{\rho}_k\right)$ are constant with $\underline{\rho}_k$, these terms may be factored out of the summation and treated uv-plane effects J_i, and the measurement equation may be simplified to (ignoring baseline-based effects):

$$\vec{V}_{ij} = \prod \left[{J^{vis}}_i \otimes {J^{vis}}^*_j \right] \vec{V}^{ideal}_{ij} \tag{5.6}$$

$\vec{V}^{ideal}_{ij}$ represents the visibilities measured by an ideal interferometer, and implicitly contains the Fourier transform of the sky.

The individual calibration effects (in either the uv- or image-plane) are denoted by single letter abbreviations, each representing the two-by-two Jones matrix for the ith feed. These include:

G_i - the composite complex electronic gain of all components located after the feed in the signal path, but excluding the correlator.

T_i - atmospheric tranmission effects which are independent of polarization (i.e., not including Faraday rotation). Useful for modelling tropospheric gain losses.

F_i - propagation effects due to Faraday rotation.

D_i - instrumental polarization response.

$E_i\left(\underline{\rho}_k\right)$ - electric field pattern describing the primary beam response, usually normalized with respect to G_i.

B_i - bandpass response, possibly normalized relative to G_i.

C_i - feed configuration matrix. This describes the nominal feed configuration, including rotation or conversion of linear to circular polarization. This term is known a priori, and handled implicitly.

P_i - parallactic angle. This is known analytically.

Since these matrix components may not commute, the order in which they appear in the measurement equation is important. In general, the correct order is given by the order in which the terms affect the incoming wavefront. Thus, a spectral line polarization observation would be described (ignoring primary beam and direction-dependent Faraday rotation) by the measurement equation as (recalling Equation 5.3):

$$\vec{V}_{ij} = B_{ij}G_{ij}D_{ij}P_{ij}T_{ij}\vec{V}_{ij}^{ideal} \tag{5.7}$$

The measurement equation is implemented in AIPS++ by sub-dividing the machinery dealing with image-plane and uv-plane calibration between imager and calibrater respectively. The remainder of this chapter treats the case illustrated in equation 5.7, i.e., uv-plane effects which are handled by the calibrater tool.

5.2.2 Data representation and the calibration mechanism

AIPS++ stores synthesis uv-data in a MeasurementSet (MS), and it is important to understand how observed and calibrated data are treated in this framework. The observed data are stored in the DATA column in the MAIN MS table; additional columns are added during processing for CORRECTED_DATA and MODEL_DATA. The CORRECTED_DATA column contains the data which result when applying all desired visibility-plane calibration corrections, moving from left to right across the uv-plane portion of the ME. This provides corrected data suitable for imaging.

The MODEL_DATA column, in contrast, is formed by propagating initial source models (perhaps for trial imaging) from right to left through the ME, applying the supported image-plane corrections, and Fourier transforming to the uv-plane. This realizes a best guess for the $\vec{V}_{ij}^{ideal}$ term in equation 5.7. The point at which these two columns meet in the ME allows a residual to be computed and thus an estimate of χ^2. The calibration component(s) at the intersection point in the ME are solved for by computing gradients of χ^2 with respect to the Jones matrices, and performing a standard non-linear least-squares minimization[1].

As an example of the calibration mechanism, we recast equation 5.7 to the implicit form used when solving for D, given an existing solution for G and knowledge of P. We ignore the B calibration component, either because the data is from a continuum observation (only one channel), or because we assume that the bandpass shape is unimportant, at least for this stage of the calibration. Since G and P appear on either side of D in the ME, they are applied to the observed data ($\vec{V}_{ij}$) and model data ($\vec{V}_{ij}^{ideal}$) terms, respectively:

[1] In practice, the CORRECTED_DATA column isn't used until corrected data intended for imaging is desired; the observed data is corrected on-the-fly by the uv-plane solver.

$$G_{ij}^{-1}\vec{V}_{ij} = \left(G_{ij}^{-1}G_{ij}\right) D_{ij} \left(P_{ij}\vec{V}_{ij}^{ideal}\right) \tag{5.8}$$

$$D_{ij} = G_{ij}^{-1}\vec{V}_{ij} \left(P_{ij}\vec{V}_{ij}^{ideal}\right)^{-1} \tag{5.9}$$

$$\left[D^{vis}{}_i \otimes D^{vis*}{}_j\right] = G_{ij}^{-1}\vec{V}_{ij} \left(P_{ij}\vec{V}_{ij}^{ideal}\right)^{-1} \tag{5.10}$$

In equation 5.10, the D calibration is realized as the feed-based (two-by-two) Jones matrices which are solved for in the χ^2 minimization. It is straightforward to see how this algebraic manipulation can be extended to any of the calibration components, and the flexible implementation of the calibrater tool is clearly evident.

5.2.3 What are the current calibration capabilities in AIPS++?

At this time, AIPS++ can solve for a wide range of uv-plane effects, including electronic gain (G_i), mean atmospheric tranmission (T_i), instrumental polarization response (D_i), and the bandpass response (B_i). Parallactic angle effects (P_i), and feed configuration (C_i) corrections are pre-computed. The primary beam voltage and polarization response can be corrected in the image-plane ($E_i\left(\underline{\rho}_k\right)\ D_i\left(\underline{\rho}_k\right)$), but not yet solved for.

Synthesis calibration support is most mature for connected-element observations; fringe-fitting capabilities are currently being developed and added. Both self-calibration and cross-calibration are supported.

Synthesis calibration is under active development in AIPS++ with new features added on a regular basis.

5.3 Practical use of the calibrater tool

5.3.1 Setting up the calibrater tool

A Calibrater `tool` is initialized by specifying an input MS. This is used throughout subsequent calibration. If another MS is to be calibrated at present, a second tool will be required. Note, however, that the facility to append to existing calibration tables allows individual measurement sets to be processed in sequence and does not require the the input data be concatenated.

To construct a Calibrater tool (for a spectral line polarimetry dataset called *polspec.ms*):

```
include 'calibrater.g'
cal := calibrater('polspec.ms')      # Construct tool
```

If the dataset has not previously been used in a calibrater or imager tool, this step will form the CORRECTED_DATA, MODEL_DATA, and IMAGING_WEIGHT columns. The CORRECTED_DATA column will be initialized as a duplicate of the (observed) DATA column. The MODEL_DATA column will be initialized with unit total intensity point source visibilities. The IMAGING_WEIGHT column will be initialized with unit weights, and is described in more detail in imager tool documentation.

5.3.2 Initial calibrator models

The Calibrater tool requires an initial visibility estimate in the MODEL_DATA column before visibility-plane solvers can be used. This can be specified via a known external source model, expressed as a set of images and/or a list of component models, which are then transformed to the visibility-plane using the Imager.ft function. This initial model can also be derived by imaging the observed data directly, without calibration, where this is possible. Please consult the general Imaging chapter for instructions on forming images.

Alternatively, if the calibrator source is sufficiently compact, it can be regarded as a point source in the initial calibration iteration, or indeed throughout calibration. Upon initialization, the MODEL_DATA column already contains unit total intensity point source visibilities. If a subset of calibrators are to be used to set the flux density scale for the observation, the models for these should be set to the proper flux density. A facility exists to set standard source flux densities on the Baars or Perley-Taylor scales using the function Imager.setjy in the Imager `tool`. In this case, if the source is not recognized, it is set to an intial flux density of 1.0 Jy.

```
include 'imager.g'
imgr:=imager('polspec.ms')
imgr.setjy()                  # Set MODEL_DATA column to point-source models
                              # (of correct flux density, if recognized):
```

5.3.3 Uv-data selection

Before using Calibrater to solve for any calibration components, selection can be applied to the MS to restrict the subset of uv-data to be used in the solution. This is specified using the Calibrater.setdata method and the very general `msselect` query argument as follows:

```
cal.setdata(msselect='FIELD_ID IN [1,2,3]')  # select data only for calibrators
                                             # (fields 1, 2, & 3 in this case):
```

Other selection keywords may be included in the msselect parameter, including ANTENNA1, ANTENNA2, SPECTRAL_WINDOW_ID, etc. These are described fully in the MS `tool` documentation. Arbitrary selection may be achieved by combining the selection keywords with the logical && and || operators.

5.3.4 Calibration table conventions

The calibration table definition is available in AIPS++ Note 240. Calibration tables are always cumulative for the specific Jones type they contain. Thus they always refer to the observed DATA column as corrected by Jones matrices of other types. Thus, only one calibration table of each type may be applied to correct the data. Provision for incrementally refining the calibration of a specific Jones type, as in self-calibration, is available.

The calibration tables are not attached to a given MS, but contain references to the MS with which they are associated. Thus they can hold calibration solutions from a collection of different measurement sets, as the capability is provided to append to an existing calibration table. The calibration tables are defined by user-specified text names, which are entirely under user control.

5.3.5 Solving for visibility-plane effects

The Calibrater `tool` contains all the common capabilities required to solve for the calibration components enumerated above. At the start of a cycle to solve for calibration components, a decision needs to be made as to which known components will be applied, which will be solved for, and which will be ignored. These choices are specified using the Calibrater.setapply and Calibrater.setsolve functions. These functions may be run multiple times to arrange to apply and/or solve for sequences of calibration tables of different types. A particular calibration component type may only be specified once in any sequence of setapply *or* setsolve executions.

The actual solution is done by the Calibrater.solve function, which takes no further input parameters. The resultant solutions can be examined using a basic capability in Calibrater.plotcal, or further using the PGplotter or Tablebrowser `tools`.

To obtain standard G (gain) calibration for selected data, with P (parallactic angle) pre-applied anticipating polarization calibration:

```
cal.setapply(type='P',t=5.0)                # Calculate & apply P on 5-second timescal
cal.setsolve(type='G',t=300.0, refant=15,   # Solve for G (gain)
             table='polspec.gcal')
cal.solve()                                 # Solve
```

5.3.6 Establishing the flux density scale

Once an electronic gain solution (G_i) has been performed for all calibrator sources, those secondary sources which were assumed to be of unit flux density can be scaled to a primary flux density scale set by known calibrators included in the run. This is achieved using a utility function Calibrater.fluxscale, which takes as input the electronic gain calibration table, a list of primary and secondary sources, and the name of an optional output electronic gain calibration table. The gain factors in the output table will be scaled correctly once the

function has completed. It will also report the estimated point source flux density for the secondary sources, based on the ratio of the mean gain modulus between the primary and secondary calibrators, as is standard practice in cases when this method can be used.

```
cal.fluxscale(tablein='polspec.gcal',         # Transfer flux density scale from
              tableout='polspec.fluxscale',   # amplitude calibrator (3C286)
              reference='3C286',
              transfer=['cal1','cal2'])
#
cal.done()                                    # Destroy calibrater tool
```

5.3.7 Correcting the observed data

The observed data, which are stored in the DATA column, are corrected by using the Calibrater.setapply function to specify the calibration tables to be used, the interpolation intervals and any selection which is to be applied to the calibration tables as they are read into AIPS++. This function, in conjunction with the selection which can be applied to the MS itself via the Calibrater.setdata, allow cross-calibration. The actual application of the corrections is performed using the Calibrater.correct function, which takes no further parameters. Note that this function must be called explicitly if the corrections are to be applied; this does not occur by default after solving for a set of calibration effects.

To apply the scaled 'G' calibration obtained above:

```
cal := calibrater('polspec.ms')           # Construct
cal.setdata(msselect='FIELD_ID==4')       # Select data to which calibration will be applied:
#
cal.setapply(type='P', t=5.0)             # arrange to apply P (unnecessary if cal tool not reset)
cal.setapply(type='G', t=0.0,             # arrange to apply G table, solutions from source 1:
             table='polspec.fluxscale',
             select='FIELD_ID==1')
#
cal.correct()                             # apply solutions, writing CORRECTED_DATA column:
cal.done();
```

Application of solutions from other calibrators to other target sources would be done by setting the selection parameters in Calibrater.setdata and Calibrater.setapply appropriately. Muliple sources may be specified in either selection.

5.3.8 Extending calibration to bandpass, polarization, etc.

In general, it is a fairly simple matter to extend calibration to additional calibration components like bandpass and polarization. In the following examples, we assume that the corrupting effects of G dominate the observed data, and that this calibration component has been solved for first, as described above. This is

typical of an observation at the VLA using feeds sensitive to RCP and LCP. For arrays using linear feeds (e.g. ATCA), this assumption may be inappropriate for strongly polarized calibrators (see the chapter for ATCA processing). In any case, the general strategy for treatment of several calibration components is to solve for the dominating effects first, and iterate solutions for components which interact significantly.

Extending the examples given above, instrumental polarization (D) calibration proceeds in manner similar to that for G. In this case, particular attention must be paid to the source *polarization* model since this is where the leverage for this solution originates. For an unpolarized polarization calibrator, the total intensity ($Q = U = V = 0$) model used in the G calibration is sufficient. However, if the calibrator is significantly polarized (compared to the expected size of the polarization leakage), then it is important to specify a model for Stokes Q and U. If the calibrator is highly polarized compared to the net polarization leakage (e.g., 3C286 at cm wavelengths observed with the VLA), then the polarization model may be obtained by imaging it without polarization calibration. Consult the Imager documentation for instructions. Once the polarization model image is obtained, the Imager.setjy or Imager.ft function can be used to set the MODEL_DATA column accordingly.

With a polarization model in the MODEL_DATA column, obtaining this calibration is straightforward. Anticipating D calibration, note that the above examples have always included explicit application of the parallactic angle (P) correction[2]. This rotates the parallactic angle variation out of the terms in the ME that contain the nominal interferometer response, and into the terms containing the leakage response for each polarization combination. Since these corrupting terms, which contain the factors being solved for, vary according to the parallactic angle over the course of the observation, and since it is common practice to solve for a single D calibration component per observation (by setting a long solution interval), it is important to set the preavg parameter in Calibrater.setsolve to a timescale shorter than that for significant variation of the parallactic angle. If preavg is not set appropriately, the data will be averaged over the full range of the solution interval, coherence in the corrupting terms will be lost, and the D calibration solution will be incorrect.

```
cal:=calibrater('polspec.ms');
#
cal.setdata(msselect='FIELD_ID==1')      # Select polarization calibrator:
#
cal.setapply(type='P',t=5.0)             # Arrange to apply existing calibration
                                         # before solution is made
cal.setapply(type='G',t=0.0,
             table='polspec.fluxscale')
```

[2]If the dataset has not been observed for polarization imaging, or if polarization imaging is not desired, the application of P can be omitted and the parallactic angle will be implicitly subsumed into the G calibration. In any case, it is important to be consistent with regard to application of P.

```
#
cal.setsolve(type='D', t=86400.0, preavg=60.0,   # Arrange to solve for D
             refant=15, table='polspec.dcal')
#
cal.solve()                                      # Determine solutions:
```

Given the G and D calibration tables obtained above, bandpass calibration proceeds as follows:

```
cal.setdata(msselect='FIELD_ID==1')       # Select data for bandpass calibrator
#
cal.setapply(type='P',t=5.0)              # Arrange to apply existing calibration
cal.setapply(type='G',t=0.0,              # before solution
             table='polspec.fluxscale')
cal.setapply(type='D',t=0.0,
            table='polspec.dcal')
#
cal.setsolve(type='B', t=86400.0,         # Arrange to solve for B
             refant=15, table='polspec.bcal')
#
cal.solve()                               # Determine solutions
#
cal.done()
```

Finally, we wish to apply all calibrations obtained above. The example below treats the case where no detailed selection of solutions and target data is desired. Use `msselect` in Calibrater.setdata and select in Calibrater.setapply (as needed for the different calibration components) to apply calibration to each target source as desired.

```
cal.setdata()                    # Select all data:
#
cal.setapply(type='P', t=5.0)    # Arrange to apply all existing calibration
cal.setapply(type='G',t=0.0,
             table='polspec.fluxscale');
cal.setapply(type='D', t=0.0,
             table='polspec.dcal');
cal.setapply(type='B', t=0.0,
             table='polspec.bcal')
#
cal.correct()                    # Write corrected data to CORRECTED_DATA column
```

5.3.9 Self-calibration

Self-calibration is the process by which calibration components are incrementally improved by alternatly re-imaging and re-solving for the calibration. It depends on the assumption that each succeeding calibration enables generation of an

improved image[3]. The uniform design of the calibrater tool with respect to the different calibration components results in a very flexible means of self-calibrating any and all of the components. One can imagine cases where, for example, the G and D components are each alternately improved along with the image. Depending upon the relative importance of these two components (and any others that may be relevant), the exact sequence of calibration and imaging will vary, but, in principle, the AIPS++ calibrater tool enables the best possible overall calibration for a given dataset.

In practice, self-calibration is achieved exactly as the initial calibrations illustrated above, except that the MODEL_DATA column needs to be filled with the improved model before each round of calibration using the imager.ft function. The calibration tables generated in each round of calibration are always *cumulative*, i.e., they are always to be applied to the (original) DATA column. At this writing, it is necessary to solve for a particular component in the same way each round, i.e., a phase-only G solution cannot be attempted after the amplitude+phase G cross-calibration. This is because it is impossible to apply *and* solve for the same component at the same time[4]

5.4 References

Cornwell, T. J., 1995, *AIPS++ Note 183*.
Hamaker, J.P., Bregman, J.D., and Sault, R.J., 1996, *A&AS*, **117**, 137.
Hamaker, J.P., and Bregman, J.D., 1996, *A&AS*, **117**, 161.
Hamaker, J.P., 1998, *in press*.
Noordam, J., 1995, *AIPS++ Note 182*.
Sault, R.J., Hamaker, J.P., and Bregman, J.D., 1996, *A&AS*, **117**, 149.

[3] The means of forming reliable images from marginally calibrated data involves careful use of deconvolution mechanisms and is described in the Imager documenation.

[4] Ambitious users may wish to avoid this problem in the short term by copying calibrated data from the CORRECTED_DATA column to the DATA column. This is not recommended, in general, since it the original data is lost (unless it is copied to an additional column). It will be unnecessary in the near future when incremental calibration comes online.

Chapter 6

Imaging, Deconvolution and Self-calibration

Tim Cornwell and Kumar Golap

6.1 Overview

This chapter describes how to make and deconvolve images starting from a calibrated MeasurementSet. There is also some discussion of how to do self-calibration.

6.1.1 Required Tools

The major AIPS++ `tools` that you will need are:

Imager is used for gridding and Fourier transformation of a MeasurementSet, deconvolution via various methods, and prediction of model visibilities from deconvolved images. It also has a number of support functions such as those for making mask images for deconvolution, for calculating point spread functions and sensitivity, and for plotting of visibility data.

Deconvolver Instead of working from the visibility data, as Imager does, you may use a Deconvolver `tool` to deconvolve a pre-calculated dirty image and point spread function. This is sometimes preferable if you need to experiment a lot with the deconvolution. The Imager.image function may be used to calculate the dirty image and point spread function.

Calibrater performs the incremental calibration of visibility data that is needed for self-calibration.

msplot is used for plotting and editing of visibility data, and is useful for investigation and editing during self-calibration cycles.

Componentmodels are used to represent the sky brightness by discrete components. Such models may be used in imaging and self-calibration to improve the quality of the results. The interactive Imagefitter is a convenient tool for fitting models to images.

Simulator may be useful if you wish to check your results via simulation.

In addition, you will need to use the Viewer to examine your images, and perhaps an MS `tool` to load and save MeasurementSets.

The User Reference Manual contains the primary documentation for all these `tools` and their functions.

6.1.2 Data Requirements

To make an image, a reasonably well-calibrated MeasurementSet is required. You may either import calibrated data from another system or calibrate the data in AIPS++. See the Calibration chapter for more details on the latter.

If the dataset was calibrated in another system then you'll need to load it from a FITS file.

- To transfer from AIPS, first split the calibrated data using the task SPLAT (thus applying the calibration), and then use FITTP to write the various sources. Do NOT write in compressed format since AIPS++ cannot read it (yet).
- To load a FITS file, use the fitstoms constructor to create an MS `tool`. Use ms.summary to get a summary of the MeasurementSet. Using this summary, check that the various sources and spectral windows have loaded as expected. Also use this as a reference for future selections from the MeasurementSet.

```
include 'ms.g'
x := fitstoms('pks1331-0.ms', 'PKS1331-09.UVFITS')      # Convert from FITS
x.summary()                                             # Summarise to logger
x.done()                                                # Finished with tool
```

- You might wish to use the visibility plotting and editing tool msplot to examine the data.

6.2 Setting up Imager

The `tool` for imaging and deconvolution is Imager. It has been designed to do everything that is required: gridding/degridding of visibility data, Fourier transforms, deconvolution using various methods, *etc.* A complete overview of the capabilities of Imager can be found in the Reference Manual.

There are two sorts of `tool` functions in Imager; passive and active. The passive functions like `setimage`, `setdata` etc. set the state of the Imager `tool`.

Then, active functions such as `makeimage`, `clean` etc. act on the data according to the previously set state.

6.2.1 Construction

To use an Imager `tool` one must first construct it from a MeasurementSet and then configure it (set its "state") via its functions. When you are finished, you destroy the `tool` (which of course does not affect the actual MeasurementSet on disk).

```
include 'imager.g'
imgr := imager('/home/data/mydata.ms')          # Construct
imgr.summary()                                  # Summarize state of tool
imgr.done()                                     # Destroy
```

The `summary` function is always useful to summarize the current state of your Imager `tool`. You may find you have forgotten what functions you ran and need reminding what you state you have set.

6.2.2 Selecting Data

To make an image from the visibilities one needs to select the part of the data from the MeasurementSet that is going to be used (e.g which pointing or channels or spectral window etc. that is going to be imaged). This is done with the Imager.setdata function. If you don't run this function, you get all of the data.

Here are some examples (refer to the Reference Manual for more details).

```
imgr.setdata(mode='channel', nchan=15,                    # Select 15 channels, second
             spwid=[2], fieldid=3)                        # spectral window  and third field
#
imgr.setdata(mode='channel', nchan=256, step=2)           # Select every second channel to make 2
#
imgr.setdata(mode='velocity', nchan=64, mstart='20km/s',  # Select data in radio velocity space
             mstep='-100m/s');
#
imgr.setdata(mode='opticalvelocity', nchan=64,            # Select data in optical velocity space
             mstart='20km/s', mstep='-100m/s');
```

Other functions can also further reduce the data selected. For example, Imager.filter and Imager.uvrange.

6.2.3 Setting the Image Parameters

Now we set the actual imaging parameters via the Imager.setimage function. This is a **required** function in Imager (in fact, the only one presently). This function is passive; it just sets state so that when you tell Imager to do something (like make an image), it knows what to do.

This function controls things like image size, cell size, spectral and polarization selection, sampling, phasecenter, and number of facets (for wide-field imaging). Images constructed afterwards (using *e.g.* the Imager.clean function) will have these parameters.

One important point to note is that the image size does NOT have to be a power of two. Just choose some reasonably composite even number close to the number that you want. **Do not** use a prime number since the Fourier Transform will then be very slow.

```
include 'measures.g'
imgr.setimage(nx=600, ny=600, cellx='0.5arcsec', celly='0.5arcsec',
              fieldid=2, mode='channel', nchan=20, start=10);
```

In this example, the phase center is that of the specified field (source). In addition, each selected channel will be imaged as one channel of the output image.

```
include 'measures.g'
phctr := dm.direction('J2000', '19h50m00', '50d00m00')      # Use default Measures tool
#
imgr.setimage(nx=600, ny=600, cellx='0.5arcsec', celly='0.5arcsec',
              doshift=T, phasecenter=phctr, mode='mfs')
```

In this example, we specify the phase center explicitly and set `doshift=T` to indicate that the phase center must be taken from the `phasecenter` argument (Ed. note - the interface that specifies the phase center needs to be rationalized some).

6.2.4 Channel Selection and Combination

Functions Imager.setdata and Imager.setimage both have arguments `nchan`, `start` and `step`. Function `setdata` is used to select channels. Function `setimage` is used to specify how many **output** channels the image will have, and how the input channels should be combined (averaging, gridding).

You should call function `setdata` first. After you have called this function, subsequent active Imager function calls only have available to them the data that you selected.

Note however, that when you call `setimage`, the specification of the channels is still in an absolute sense. Thus, if you set `nchan=10, start=20` in `setdata`, and you wish to image all of these channels (let us say one image plane per channel), you must **also** set `nchan=10, start=20` in function `setimage`.

You might think that you have already selected the 20 channels you want with `setdata` and therefore in function `setimage` setting `start=1` would select the first of those 20 channels. This is **not** the way it works. If you asked for, say, channel 1 with function `setimage` when you did **not** select channel 1 with function `setdata`, that channel would be imaged but empty in the output image.

You use the channel selection parameters in function `setimage` to specify how the selected channels will be combined and how many output channels the image will have. Basically, there are two sorts of ways that you might like to use the channels that you have selected.

Firstly, in a multi-frequency synthesis image, all of the selected channels are gridded onto the uv plane to reduce band-width smearing (which would be incurred if you averaged the channels and then gridded). In this case, the `step` argument is not generally relevant; leave it at 1. For example:

```
imgr.setdata(mode='channel', nchan=32, start=10, step=1)
imgr.setimage(mode='mfs', nchan=1, start=10, step=1)
```

Here we have selected 32 input channels, starting with channel 10, and gridded them together to form one output image plane.

Secondly, when you make a spectral-line cube, you may wish to select/combine channels in a variety of ways according to the science you want to do. Here are some examples.

```
imgr.setdata(mode='channel', nchan=200, start=10, step=1)
imgr.setimage(mode='channel', nchan=200, start=10, step=1)
```

This example selects 200 consecutive channels starting with channel 10. The image formed has one image plane for each selected channel.

```
imgr.setdata(mode='channel', nchan=200, start=10, step=5)
imgr.setimage(mode='channel', nchan=200, start=10, step=5)
```

This example selects 200 channels by picking every fifth channel, starting with channel 10. The image formed has one image plane for each selected channel.

```
imgr.setdata(mode='channel', nchan=200, start=10, step=5)
imgr.setimage(mode='channel', nchan=100, start=10, step=10)
```

This example selects 200 channels by picking every fifth channel, starting with channel 10. However, the image formed only has 100 channels. Each output image plane (channel) is formed by averaging 2 successively selected channels. Thus, we have selected channels 10, 15, 20, 25, 30. When we image, channels 10 and 15 are averaged to form output image channel 1. Channels 20 and 25 are averaged to form output channel 2 and so on.

Of course you could also use `mode='mfs'` when combining groups of channels if you want an output image of more than one channel. In this case the combination is done by gridding rather than averaging.

6.2.5 Weighting

The above steps show how to set up your Imager `tool` as desiresd. In addition, before imaging one may wish to set some values in the MeasurementSet itself. This is necessary for visibility weighting. The visibility imaging weights are computed prior to the actual imaging and stored in a column of the MeasurementSet, "IMAGING_WEIGHT".

The first time an Imager `tool` is attached to a MeasurementSet it initializes the imaging weights to the natural weighting scheme. Thereafter, whatever weighting scheme you last used, is the one you will get if you don't explicitly run one of the weighting functions.

The values in IMAGING_WEIGHT column are set, changed, and examined by a number of functions.

- weight sets the column using one of natural, uniform, or Briggs (robust) weighting. For the latter two methods, one can specify the field of view over which the minimization of the sidelobes is done (thus achieving what is often called super-uniform weighting).

```
imgr.weight(type='natural')                 # Natural weighting
imgr.weight(type='uniform')                 # Uniform over entire field of view
imgr.weight(type='uniform', npixels=300)    # Uniform over specified size
imgr.weight(type='briggs', rmode='norm', robust=0.5)  # An example of Briggs weigh
```

- filter applies an optimal filter for a given shape (often called "tapering"). In the following example we apply a Gaussian filter.

```
imgr.filter(type='gaussian', bmaj='4.0arcsec', bmin='2.5arcsec',  bpa='60deg')
```

- plotweights plots the imaging weights either as function of uv distance or on a gridded plane.
- sensitivity calculates and returns the sensitivity of the resulting weighting both absolutely and relative to natural weighting.
- fitpsf calculates the dirty point spread function and returns the best fitting Gaussian.

In Imager, we have chosen separate out the calculation of the weighting from the gridding and Fourier transformation to give more control over this important step. Note that this also gives the ability to set the weights by hand using getcol and putcol functions of the table tool. For example, to replace the weights by their square root:

```
include 'table.g'
t:=table('3C273XC1.ms', readonly=F)
wt:=t.getcol('IMAGING_WEIGHT')
```

```
wt[wt>0.0] := sqrt(wt[wt>0.0])
t.putcol('IMAGING_WEIGHT')
t.flush();
t.close()
```

Before performing such a step, one should close the imager and then construct it again afterwards.

WARNING: All the weighting schemes modify the MeasurementSet and thus are conserved even after destroying the Imager `tool` *and may no longer be suitable for subsequent imaging. You can of course reset the weighting scheme with the* `weight` *function.*

6.3 Creating and Deconvolving Images

6.3.1 Creating Images

Now that your Imager `tool` and the MeasurementSet are set up, you can make an image.

1. It may be helpful to use the Imager.advise function to determine the cell size automatically, as well as the number of pixels for a given field of view.

   Imagine we want to make an image over a field of view of 10 arcmin. The `advise` function will return the maximum pixel size required and number of pixels. If the number of facets required is larger than 1 then one needs a wide-field imaging algorithm as described in the wide-field chapter of Getting Results.

   ```
   myimager.advise(fieldofview='10arcmin')
   ```

2. It is always recommended to make a dirty image of the field before deconvolution. Use the Imager.makeimage function to make a dirty image first. There are a number of possibilities for constructing the image. Usually, you will want to make the image from the CORRECTED_DATA column. This is the column that Calibrater writes into when correcting the DATA column. It is also the column that Imager initializes to a copy of the DATA column if the CORRECTED_DATA column does not exist (e.g. filling from a FITS file).

   You might also like to make the point spread function.

   ```
   imgr.makeimage(type='corrected', image='dirty.image')
   imgr.makeimage(type='psf', image='dirty.beam')
   ```

6.3.2 Deconvolution

At this point, you are ready to deconvolve the point spread function (usually called the dirty beam) from the dirty image. The goal of this step is to correct the dirty image for the sidelobes in the point spread function, which arise from the limited Fourier plane coverage in your observation. You have two choices for the deconvolution: use the Deconvolver `tool` on the dirty image and point spread function or use the various deconvolution functions of Imager. In the latter, the dirty image and point spread function are recomputed as needed, whereas Deconvolver works with a fixed dirty image and point spread function. If you want to mosaic, or image wide- or multiple fields then you must use Imager.

The options for deconvolving are :

- CLEAN; Högbom, Clark and multi-scale variants of the CLEAN algorithm are available. Functions that implement CLEAN are Imager.clean, Deconvolver.clean, and Deconvolver.clarkclean.

  If you use a multi-scale algorithm, you should also set up the scale sizes via the Imager.setscales function.

- Maximum Entropy; maximum entropy and maximum emptiness methods are available. Functions that implement MEM are Imager.mem and Deconvolver.mem.

- Non-negative least squares (see Dan Briggs' thesis). Functions that implement NNLSQ are Imager.nnls.

- Pixon methods. Functions that implement Pixon methods are Imager.pixon and Deconvolver.pixon.

  The Pixon algorithm is available via an IDL library courtesy of the Pixon LLC. This means that you must have the library and IDL installed on your computer. The Pixon library is available free of charge for your personal scientific use direct from Pixon LLC. IDL is available commercially from RSI.

For all deconvolution approaches (except Pixons)a Componentmodel `tool` may be specified, in which case it is removed from the data prior to imaging. Thus the resulting deconvolved image does not include the flux modelled by the Componentmodel.

The deconvolution methods do not keep a list of CLEAN components in sequence. Instead the CLEAN components are immediately added to the model image. This allows interoperability of the CLEAN, MEM and NNLS algorithms as well the deconvolution functions in the Deconvolver `tool`. It also allows editing of the resulting images using the Image tool.

The deconvolution functions can return the residual and restored images, as well as the updated model image. In addition, there are Imager.residual and Imager.restore functions that also compute residual and restored images.

The distinction is that in the former, the residual is that calculated by the deconvolution method at the end of iteration, whereas the latter returns the residual calculated by going back to the original visibility data (except for the cases of multi-field and wide-field algorithms).

```
imgr.clean(algorithm='clark', niter=2000, gain=0.2, model='model_2000',
           image='restored_2000', residual='residual_2000')
```

In the above example we run a Clark CLEAN deconvolution with Imager. 2000 CLEAN components with a gain of 0.2 are requested.

If the model image does not pre-exist it will be created and filled with the CLEAN delta function components found. If it does pre-exist (it may be non-empty (perhaps the result of a prior deconvolution) and its Fourier transform is first subracted. This is also how you would continue to CLEAN deeper from a prior run of `clean, mem, or nnls`.

Note that the model image is updated when this function finishes. So if it was non-empty on input, you should save a copy first, if you wish to preserve that model.

6.3.3 Specifying the deconvolution region

In deconvolution, one must often specify the region of the dirty image over which the sky brightness is expected to be non-zero. In CLEAN, possible components can only come from this region. Designation of regions to deconvolve is done via images of the same shape as the dirty image. The value of these mask images ranges from 0 to 1 (but usually we use 0 [no components] or 1 [allow components]). There are a number of ways of making such mask images.

Imager.make Makes an empty image according to the current image size, cell size, *etc.*

Imager.boxmask makes a mask image from a specified BLC and TRC.

Imager.regionmask makes a mask image from an image region.

Imager.exprmask makes a mask image by applying an image Imager.calc to an image, to, for example, set the mask to all points where the brightness is greater than some limit.

The most powerful approach is probably to use Imager.regionmask, since regions have a number of advantages: they can be polygonal, they can be combined in various ways, they can be defined in world coordinates, and they can be saved to a table for later retrieval and use. The Regionmanager `tool` allows these and other operations on regions.

A compound region holding the union of lots of interactively specified simple (boxes or polygons) regions can be made via the Viewer. Please see the documentation in Viewerdisplaypanelviewer:vdpgui to see how to do this.

NNLS requires two such masks images : one for the region where flux is to be allowed and one for the region to be used as constraints. Typically the latter

region will be bigger than the former. The product of the pixels in the two regions should not exceed about 5-10 million.

```
# Imagine you have a Region tool called 'r1' in Glish

imgr.regionmask(mask='mask_image', region=r1)            # Make mask image from region
imgr.clean(algorithm='clark', niter=2000, gain=0.2,
           model='model_2000', image='restored_2000',
           residual='residual_2000', mask='mask_image')  # Use it !
```

6.4 Wizard interface to Imager

For a quick look at an image, one may wish to use the Imagerwizard function. This is built using various `tools` including Imager. It makes many decisions, such as cell size, automatically, and simplifies obtaining an initial imaging.

```
include 'imagerwizard.g'
ok := imagerwizard('data.ms')
```

The wizard leads you step by step through the imaging process via interactive GUIs. It also writes a *Glish* script which you can then look at, edit and re-run if you wish.

6.5 Component Models and Primary Beams

A priori, survey, or model fitting information can provide the user with the basis for a component model of bright, unresolved or somewhat resolved, confusing sources. These sources can be removed from the visibilities prior to imaging mosaicing by setting them in the Imager.clean or Imager.mem functions. These arguments take a Componentlist table name.

However, these components are assumed to be small compared to the primary beam, and only the primary beam value at the center of a component is ever applied to the Gaussian component (i.e. a Gaussian component stays Gaussian after application of the primary beam, even if its wings extend beyond the primary beam pattern). Hence, errors will result if highly extended components are included in the component list. If your application requires a model with extended Gaussian components, for the time being you may want to turn the extended components into an image with the Image tool's Image.modify function.

When you are imaging only single fields (i.e. not mosaicing), as is relevant to this chapter, the Componentlist is *not* a true representation of the sky brightness. The values it holds must include the attenuation of the primary beam. This is because in single field mode, Imager makes no primary beam corrections of any form whatsoever.

If you have a Componentlist which is a true representation of the sky, you can apply a primary beam attenuation to it with the Imager.pb function.

```
imgr.make('empty')
imgr.pb('empty', operation='apply', incomps='cl.sky',
        outcomps='cl.attenuated.sky')
```

Note that you need the input image so that Imager knows what telescope you are using so that it can get the correct model. Alternatively, if you have used Imager.setvp to set a Voltage pattern from an external table, it would use that.

The Componentlists are specified by their disk file names. (the arguments are not Componentlist `tools`).

6.6 Multi-scale CLEAN

6.6.1 General

Delta function CLEAN algorithms, such as the Clark or Hogbom algorithms, must use a modest gain factor (traditionally 0.1) in order to image extended emission in a reasonable manner. Otherwise, emissions and sidelobes get confused and error striping can result.

Multi-scale CLEAN algorithms attempt to recognize that the emission in the sky usually comes on a variety of scale sizes; it decomposes the image into Gaussians of the specified scale sizes. This means it does not spend huge amounts of time CLEANing large extended sources pretending that they are really collections of delta functions.

Function Imager.setscales allows you to choose these scales, either with an automatic method (you say how many scales you want) or by direct specification of the scale sizes in pixels.

Multi-scale CLEAN does not suffer from the same problems as delta function CLEAN and can use a larger value of the gain factor, in the range of 0.4 and 0.8. Sometimes an oscillatory pattern will occur in the total cleaned flux with the peak residuals bouncing back and forth between positive and negative. If this occurs, try reducing the gain factor or try reducing the size of the largest scale.

The progress display for multi-scale CLEAN shows the flux and peak residuals for each cleaning scale. For any extended brightness distribution, there will be initially more flux on the large scales than on the small scales, and this will be reflected by much larger residuals on the large scales. Consequently, the flux is built up much more quickly by CLEANing on these large scales. However, in the early stages of deconvolution, we may never get anywhere close to the stated CLEANing threshold if we are dominated by extended flux. The exact manner in which the larger scales' residuals will be inflated depends upon the details of the brightness distribution.

```
imgr.setscales('nscales, nscales=3)                # Specify number of scales
imgr.setimage(nx=600, ny=600, cellx='0.5arcsec',
              celly='0.5arcsec', mode='mfs')
imgr.clean(algorithm='msclean',                    # Deconvolve
           model='model',
           image='restored',
           niter=100, gain=0.3)
```

6.6.2 The `imagermultiscale()` function

The imagermultiscale function provides a different sort of packaging of multi-scale algorithms and works on mosaicing or single field data.

Conceptually, the `imagermultiscale()` function offers a second order multi-scale method, or doing multi-scale CLEAN as the deconvolution kernel inside of a cascade of multi-resolution CLEANs each working on increasingly larger images with smaller pixels.

The multi-scale CLEAN algorithm addresses an essential problem in imaging: you often have emission on a wide range of scale sizes which must be imaged simultaneously. While it is vastly superior to delta-function CLEAN algorithms at reconstructing extended emission, it is not incredibly efficient as it seeks to deconvolve very large structures with very small pixels.

A simple scripting approach to multi-resolution imaging which overcomes this inefficiency is provided by the imagermultiscale function. The user provides a vector of image sizes and the corresponding pixel sizes, and the function performs a multi-scale CLEAN on the first image (with the largest pixels). When the first CLEAN stops, its model is regridded onto the second image size and is used as a starting model for the next round of Multi-Scale CLEANing. Of course, since the regridded model will have smaller pixels, it will be in error; but the large scale structure will be pretty well in place. So, each round of Multi-Scale CLEANing will concentrate on the small scale structure, though each will also be able to make larger scale adjustments as well.

If the `imagermultiscale` function is too restrictive for your needs, you are encouraged to pull it out of `imager.g` and edit your own function. For example, you might want to have some fine control over which channels to image. This function could also easily be converted to a multi-scale MEM function.

```
include 'imager.g'
#
ok := imagermultiscale(msname='orion.ms', imsizes=[128, 256, 512],
                       cellsizes=[16,8,4], scales=[0, 5, 15],
                       nitermult=0.15, niterpower=1.2, fields=[2],
                       centerfield=[2],
                       spwid=[1])
```

6.7 Wide-field imaging and Mosaicing

Wide-field imaging and mosaicing are covered in separate chapters of this document.

6.8 Combining single dish and interferometer images

Sometimes, the lack of total-power data in an interferometric observation (single- or multi-field) can result in image defects on the size scale of the interferometer baselines. For example, in a maximum entropy algorithm with positivity constraint, the "negative bowl" will manifest itself in the model brightness distribution by a flat region of near zero emission surrounded by a spurious positive "splash" around the bowl; the "splash" artifact sometimes has small scale structure.

Currently, the only method available in Imager to combine total power measurements with interferometer data is via a *feathering* technique.

Since feathering is an intrinsically post-deconvolution method, it cannot help such a case. However, there are two ways to help out the deconvolution algorithms. First, convert the single dish image to the same image grid as the interferometric mosaic by using the Image.regrid function. Then either rescale or deconvolve the image into units of Janskys per pixel. This image can be used as a `prior` image in maximum entropy or maximum emptiness and will prevent small scale image artifacts due to the lack of total power data. However the `prior` image is not elevated to the level of a hard constraint, so a subsequent pass of the interferometer image and the single dish image through the Imager.feather function will ensure that the large scale structure in the final image is consistent with the single dish image.

```
imgr.feather(image='feathered.image',      # Specify the single dish
             highres='casa.vlaonly',       # voltage pattern via
             lowres='casa.sd',             # a table
             usedefaultvp=F,
             vptable='sd.vptable');
```

6.9 Self-calibration

Self-calibration is the process of using a model of the source to improve the calibration of the data. One makes an initial model using perhaps component-models or a deconvolved image. This is then used to predict the visibilities, and the antenna gains are derived by a least squares fit between the observed and modelled visibilities. Once the antenna gains thus derived have been corrected, the model may be rederived and thus improved. This process is iterated until a satisfactory improvement has been acheived.

In AIPS++ this process is done with an Imager `tool` and a Calibrater `tool` working together.

The steps are:

1. Set up a Calibrater tool for the MeasurementSet. Set the data selection to be consistent with that in the Imager `tool`. The type of solution is set in the Calibrater.setapply and Calibrater.setsolve functions. More on this below. Be sure to set the solution time scales appropriately.
2. Make a deconvolution using your favored method.
3. You might wish to clip the low unreliable levels of the model image using Imager.clipimage.
4. Use the Imager.selfcal function to actually perform the gain solution and re-calibration.
5. Remake the deconvolved image.

This process (steps 3 onwards) can be repeated until the image stops improving. Usually it is a good idea to start with phase-only solutions (controlled in Calibrater.setsolve) and then allow amplitude solutions after the phase-only solutions have ceased improving.

The types of solution allowed are:

T Amplitude and phase per antenna

G Amplitude and phase per feed

D Leakage between polarizations

B Bandpass per antenna

More details may be found in the Calibrater documentation.

Diagnosis of the success of the self-calibration is ultimately judged by the improvement in image quality (measured from the off-source noise level). One may also expect that the rms errors per time interval reported by Calibrater.solve will decrease, and that the residual visibility as displayed by Imager:plotvis will diminish. In addition, it is also advisable to examine the gain solutions using Calibrater.plotcal. These should be small if the initial calibration was of good quality.

6.10 Summary of Imager functions

6.10.1 Setting/seeing the basic state

- Imager.advise - give some advice on the image parameters.
- Imager.setdata - select a subset of the data from the MeasurementSet

- Imager.setimage - define the parameters of the image(s) to be created
- Imager.summary - summarize the state

6.10.2 uv selection and filtering

- Imager.weight - set weighting scheme and save it in the MeasurementSet
- Imager.plotweight - plot visibility weights
- Imager.filter - multiply the visibilities (taper) by a specified function (e.g. gaussian)
- Imager.uvrange - select data based on 'uv' distance

6.10.3 Masking

- Imager.mask - make a mask image by thresholding another image
- Imager.boxmask - make a mask image from the blc and trc of a box
- Imager.regionmask - make a mask from a region
- Imager.exprmask - make a mask image from an image LEL expression

6.10.4 Create images

- Imager.make - make an empty image (with the state set by functions `setdata` and `setimage`)
- Imager.makeimage - makes dirty images or point spread function (with the state set by functions `setdata` and `setimage`) from the corrected data, observed data or model data

6.10.5 Deconvolution

- Imager.clean - deconvolve image made from the data using one of the flavors of CLEAN (clark, Högbom, multi-scale)
- Imager.setscales - set the scalesizes if multi-scale CLEAN algorithms is selected
- Imager.mem - decovolve images using a Maximum entropy or emptiness algorithm
- Imager.nnls - deconvolve images using the non-negative-least-squares algorithm
- Imager.pixon - experimental function to deconvolve images using the pixon algorithm

6.10.6 Image Combination

- Imager.feather - combine low resolution (e.g. single dish) and high resolution (e.g. interferometer) images
- Imager.pb - apply a primary beam (Voltage pattern) correction to an image
- Imager.restore - restore model images to one convolved with a Gaussian beam and residuals added in
- Imager.residual - calculate the residual image corresponding to a given model image
- Imager.smooth - convolve a model image with a Gaussian

6.10.7 Utility

- Imager.sensitivity - calculate the point source sensitivity (useful to test different weighting and filtering schemes)
- Imager.setbeam - Define the restoring beam to be used by the deconvolving functions
- Imager.plotuv - plot uv-coverage of the selected visibilities
- Imager.plotvis - plot the selected visibilities
- Imager.plotweight - plots the imaging weights
- Imager.fitpsf - fit a gaussian to the main lobe of the point spread function
- Imager.setoptions - define cache memory needed for gridding, ftmachine to be used for transforming, grid function to use or the amount of padding to reduce aliasing in gridding/FT
- Imager.correct - Apply parallactic angle corrections
- Imager.clipvis - Flag visibilities by residual amplitude. Useful in removing interference from the residual image

6.10.8 Calibration and self-calibration

- Imager.setjy - Set the right amplitude, phases of point sources in MODEL_DATA column. Used prior to point source calibration.
- Imager.clipimage - Zeros image pixels below a threshold. Needed for editing a model image before using it to self calibrate.
- Imager.ft - Fourier Transform model images/components into the MODEL_DATA column of the MeasurementSet. This is used for self calibration.

6.11 Examples of Imaging Scripts

6.11.1 Multi-frequency synthesis image of one field

```
include 'imager.g'
include 'os.g'
#
imgr := imager('mydata.ms')                          # Construct
#
imgr.setdata(mode='channel', nchan=15, start=1,      # Select data (15 channels, 1 field)
             fieldid=3)
#
imgr.setimage(nx=512, ny=512, cellx='2arcsec',       # Set image parameters - grids
              celly='2arcsec',  mode='mfs')          # all selected channels into 1
#
imgr.clean(algorithm='clark', niter=10000,           # CLEAN 10000 components
           model='model', image='restored',
           residual='residual')
#
dos.copy ('residual', 'residual_10000')              # Save copies
dos.copy ('restored', 'restored_10000')
#
imgr.clean(algorithm='clark', niter=5000,            # Restart and CLEAN another
           model='model', image='restored',          # 5000 components. Updates
           residual='residual')                      # images
#
rest1 := image('restored_10000')                     # Display restored images
rest1.view()
rest2 := image('restored')
rest2.view()
#
resid1 := image('residual_10000')                    # Compare statistics of
resid1.statistics()                                  # residual images
resid2 := image('residual')
resid2.statistics()
```

6.11.2 Multiple regions on the sky simultaneously

The following script demonstrate of how we can image multiple regions of the sky simultaneously (in AIPS this is implemented in task MX). This is a technique used to deal with a source which is outside the main field of view being imaged but whose sidelobes are seen in the image we want to make. It is also sometimes useful to remove some kinds of interference which tend to appear as a source at the pole.

It is assumed in this script that there is only one `fieldid` (i.e. one source).

```
imgr:=imager('mydata.ms')                                  # Construct
#
imgr.setdata(mode='channel', nchan=25,                     # Select data: 25 channels,
             start=4, step=1, spwid=[1,2])                 # 2 spectral windows.
#
mydir:=dm.direction('J2000', '19h58m49', '40d45m29')  # Direction of outlier (CygnusA)
imgr.setimage(nx=200, ny=200,                              # Set image parameters for outli
              cellx='19arcsec',celly='19arcsec',
              doshift=T, phasecenter=mydir,
              mode='mfs', spwid=[1,2])

imgr.make('cygA')                                          # Make empty image for outlier
#
#
imgr.setimage(nx=600, ny=600,                             # Set image parameters
              cellx='19arcsec', celly='19arcsec',    # for main field direction
              spwid=[1,2])
imgr.make('mainfield')                                    # Make empty image for main field

#
imgr.uvrange(uvmin=10, uvmax=5000)                        # Select uv range
imgr.weight(type='briggs', rmode='norm',                  # Weighting
            robust=0.0, npixels=300)

imgr.clean(algorithm='mfclark',                           # Deconvolve
           niter=1000, gain=0.05,
           model="mainfield cygA",                        # Vector of strings
           image="mainfield.restored cygA.restored",
           residual="mainfield.residual cygA.residual")
#
```

Note that the combination of `Imager.setimage` and `Imager.make` can be used as many times as is necessary to define all the outlier fields.

We had to explicitly make the empty model images before passing them to `Imager.clean`. Otherwise Imager would recognize only the last call to `Imager.setimage` and would thus fail to handle the multiple fields.

In the running of the `Imager.clean` function we used the Clark algorithm. In addition, we must specify `algorithm='mfclark'` because we are making a multi-field image.

6.11.3 Spectral-line imaging

The following is an example of making a spectral-line image. We demonstrate how `Imager.setdata` and `Imager.setimage` can be used to average data channels to make one of the spectral-line image channels.

The MeasurementSet is created from a FITS file that comes with the AIPS++ installation.

```
include 'imager.g'
include 'ms.g'
include 'sysinfo.g'
#
aipsroot:=sysinfo().root()
fitsfile:=spaste(aipsroot, '/data/demo/BLLAC.fits')
myms := fitstoms('mydata.ms', fitsfile)               # Generate MeasurementSet
myms.summary()
myms.done()
#
imgr:=imager('mydata.ms')                             # Construct Imager
#
imgr.setdata(mode='channel', nchan=40, start=4,       # Select data: 40 channels
             step=1, spwid=[1])                       # from the first spectral window
#
imgr.setimage(nx=600, ny=600,                         # Set imaging parameters. We
              cellx='19arcsec', celly='19arcsec',     # output 20 channels from the 40
              doshift=F, spwid=[1], mode='channel',   # by combining pairs of channels
              nchan=20, start=4, step=2 )
#
imgr.weight(type='uniform')                           # Weighting
#
imgr.clean(algorithm='clark', niter=1000, gain=0.05,  # Deconvolve
           model='model',
           image='restored',
           residual='residual')
#
imgr.done()
```

Note that if the channel images were needed in the velocity domain, then argument `mode` in both functions `Imager.setdata` and `Imager.setimage` must be put to `velocity` and instead of the `start` and `step` parameters, use the `mstart` and `mstep` parameters.

6.11.4 Using Clean and MEM

In this example we assume we have bright point sources and low level broad scale structures to image. It is well known that CLEAN does not do well with large scale structures and that MEM does not do well with bright compact sources in presence of large scale structures. We can CLEAN on the bright point sources followed by MEM on the low level structures. What follows is an example of how to do that.

```
imgr:=imager('mydata.ms')
imgr.setimage(nx=600, ny=600, cellx='2arcsec', celly='2arcsec',
              doshift=F, spwid=[1])
#
# Use clean with a mask image around the bright point sources
# to restrict clean to deconvolve these sources only.  Alternatively,
# if the point sources are brightest, just CLEAN a little bit
# to remove the bulk of them
#
imgr.clean(algorithm='clark', niter=1000, gain=0.1,
           model='cleanmodel',
           mask='pointsource.mask',
           image='clean.restored',
           residual='clean.residual')
#
# Now run MEM on the data. Note that we use the model image that we
# have got from clean, so that the point sources will be subtracted
# from the visibilities and MEM will deconvolve the residual
#
imgr.mem(model='cleanmodel', niter=40, sigma='0.001Jy',
         image='final.restored', residual='final.residual')
#
imgr.done()
```

6.11.5 Deconvolver and multi-scale CLEAN example

The Deconvolver `tool` has functions which have very similar meanings to to those in Imager. However, no MeasurementSet is used with the Deconvolver; it is based on image plane deconvolution only. It needs a point spread function and a dirty image.

```
deco := deconvolver('3C273XC1.dirty', '3C273XC1.psf')
deco.setscales(scalemethod='uservector',     # Scales of delta function,
               uservector=[0, 3, 10])        # 3 and 10 pixels wide
#
deco.clean(algorithm='msclean',              # deconvolve
           model='3C273XC1.clean',
           niter=100, gain=0.3)
deco.done()
```

Note that it is not very useful to use CLEAN components that are wider than the largest structure in the image.

6.11.6 Imaging and Self-Calibrating

The following is a script of a multi step CLEAN and self-calibration. We use here a test MeasurementSet that is in the AIPS++ system.

```
include 'imager.g';
include  'calibrater.g';
include 'os.g'
#
imagermaketestms(msfile='3C273XC1.ms')               # Make MS
imgr:=imager(filename='3C273XC1.ms')                 # Construct Imager tool
imgr.summary()
imgr.setdata(mode="channel" , nchan=1, start=1, step=1, # Data selection
             spwid=1, fieldid=1);
imgr.setimage(nx=600, ny=600, cellx='0.6arcsec',        # Set image parameters
              celly='0.6arcsec', stokes="I" )
imgr.uvrange(uvmin=1.0, uvmax=500000);
#
dos.remove('model0')
dos.remove('restored0')
dos.remove('residual0')
#
imgr.clean(algorithm="hogbom" , niter=1000,          # Deconvolve 1000
           model="model0" , fixed=F,
           image="restored0" , residual="residual0")
#
dos.remove('model0_5000')
dos.remove('restored0_5000')
dos.remove('residual0_5000')
imgr.clean(algorithm="hogbom" , niter=5000,          # Deconvolve
           model="model0_5000" , fixed=F,
           image="restored0_5000" , residual="residual0_5000")
#
cal := calibrater(filename="3C273XC1.ms")            # Construct Calibrater tool
cal.setsolve(type="G" , t=100, preavg=0.0,           # Set selfcal solution parameters
             phaseonly=T, refant=-1,                 # phaseonly
             table="selfcal.gcal",append=F);
cal.setapply(type="G" , t=0.0, table="selfcal.gcal", # Set parameters for application of
             select='');                             # the calibration

# The next step Fourier transforms the model from the CLEAN
# above and makes model visbilities in the MeasurementSet. The
# previously created Calibrater tool fits the model visibilities with
# the observed ones and hence calculates the necessary gains to make
# the best fit. The solution is applied.

ok := imgr.selfcal(caltool=cal, model="model0",  complist='');
```

```
cal.plotcal(plottype="PHASE", tablename="selfcal.gcal" ,           # Inspect solution
            antennas=[], fields=[], polarization=1, spwids=[],
            timeslot=1, multiplot=T, nx=3, ny=3)
#
dos.remove('model1')
dos.remove('restored1')
dos.remove('residual1')
imgr.clean(algorithm="hogbom" , niter=2000,                       # Deconvolve 2000
           model="model1" , fixed=F,
           image="restored1" , residual="residual1")
cal.setsolve(type="G" , t=30,                            # Setup for phase and amplitude
             preavg=0.0, phaseonly=F, refant=-1,
             table="selcal.gcal" , append=F);
#
ok := imgr.selfcal(caltool=cal, model="model1")        # Solve and apply for a gain se
#
dos.remove('model2')
dos.remove('restored2')
dos.remove('residual2')
imgr.clean(algorithm="hogbom" , niter=5000,            # Final deconvolution, 5000
           model="model2" , fixed=F,
           image="restored2" , residual="residual2")
#
r := drm.quarter()
r0_1000 := image('restored0')
r0_5000 := image('restored0_5000')
r1_2000 := image('restored1')
r2_5000 := image('restored2')
#
r0_1000.view(region=r, includepix=[-.003,.003])
r0_5000.view(region=r, includepix=[-.003,.003])
r1_2000.view(region=r, includepix=[-.003,.003])
r2_5000.view(region=r, includepix=[-.003,.003])
```

Chapter 7

Wide field imaging

Kumar Golap and Tim Cornwell

7.1 Background

The relationship between the visibility domain to the image plane is given by the Van Cittert-Zernicke theorem. This can be reduced to a Fourier relationship for planar arrays and also for small field of views for non-planar arrays (Thompson, Moran and Swenson). When this happens the Fourier inversion of observed visibilities gives the true image convolved with the point spread function (PSF). Both the true image and the PSF are functions of 2 variables only (l,m). Deconvolving is conceptually simple.

When visibilities are sampled by a non-planar array (u, v and w components), the relationship between them and the image plane can no longer be represented by a 2-D Fourier transform. The relationship is now a 3-D Fourier transform. Of course our image of the sky is truly 2-D and when deconvolving it, one should ideally use a different PSF for each point in the image. This can be very difficult to achieve if the image has a few million pixels.

It becomes worse when one has to consider that the field of view for deconvolution is much more than the field of view of interest (at most the primary beam of the antennas). Deconvolution may need to consider the sidelobes and at low frequencies the whole visible sky may have to be considered, especially if strong sources like Cygnus-A or Cas-A are above the horizon. The PSF in the sidelobes suffers both distortion from the 'w' term and the amplitude and gain difference difference of the sidelobes of each antenna.

Wide-field imaging is different from mosaicing as the whole observation can be made with a single pointing. Wide-field imaging is needed when the term $w\sqrt{1-l^2-n^2}$ varies significantly over the field of view. Mosaicing is the technique used to combine visibilities (or images) obtained from different pointings; in effect it increases the field of view beyond the primary beam pattern.

An important question is how do you know that you need wide-field imaging

techniques? If your array is planar (e.g. an E-W array) you do not need them. Otherwise, one simple way is to make an image via basic 2-D Fourier techniqes. If point-source responses on the edge of the image have the same shape as those in the center then it is not needed.

A simple criterion (due to Barry Clark) is that:

(Field of view in radians) * (Field of view in beams) ¿ 1

7.1.1 Possible solution to the wide-field imaging problem

There are in fact a number of different possible approaches to solving the wide-field imaging problem. These are reviewed by Cornwell and Perley. Here we give a brief account:

- 3D transform - after a 3D transform of the visibilities from a non-planar array is performed, the image can be recovered only after a 3-D deconvolution is performed. This technique suffers from the fact that it is very expensive computationally (Cornwell and Perley).

- Faceting the image plane - the image plane can be considered as small tangential facets to the Celestial sphere. The size of the facets are chosen such that the PSF is invariant over the each facet. Each facet has one PSF associated with it. In an iterative fashion, CLEAN is used on each facet down to some limit (the *minor* cycle), and then the components found are subtracted from the visibilities (via Fourier transform), and a re-computation of the residual images is made (the *major* cycle).

  Calculation of the facet planes can be handled in two ways:

  1. By phasing the visibility data appropriately, the dirty image and PSF are made for each facet and the above process is applied to the facets. At the end of processing, the final CLEANed images have to be transformed to a single common plane.
  2. By phasing the visibility data and reprojecting the uv coordinates appropriately, the dirty image and PSF are made for each facet onto a common tangent plane, and the above process is applied to the facets. No final reprojection is required.

7.2 Available Tools

The basic `tools` for calibration and imaging are Calibrater and Imager. Combining these is a `tool` called Dragon. This `tool` allows the user to loop through many cycles of wide-field imaging and self-calibration successively with little or no inputs from the user in the process.

7.2.1 Basic capabilities

Imager handles wide-field imaging via the faceting described above. For various reasons, one may like to change the number of facets being used when processing an image. For example, one may wish to have few facets when deconvolving bright sources and have many more facets to have deep CLEANing.

During the self-calibration part of the imaging process (also sometimes called difference mapping), the flux levels where phase self-cal and amplitude self are needed can be specified.

The imaging process can be helped by specifying the bright sources via a Componentlist `tool`.

If there are bright sources outside the field of interest, their sidelobes can be deconvolved from the image of interest by specifying an outlier region around that source. Inherently there are no limits on the number of facets or number of outlier fields to be specified.

Imager and Dragon have a function which advises on the optimal number of facets, tangent point of image, to be used for given desired signal-to-noise ratio and size of image to be made.

7.2.2 Faceting size

The number of facets needed to image a given field of view depends on several factors. Among them are the uv distance, the amount of amplitude loss allowable, and the positional error allowable. The number of facets used ultimately depends on what the user wants. If flux information only is needed the number of facets can be much less than if, say, very accurate positions are needed.

The `advise` function gives both the number of facets needed to get the accurate positions or just accurate flux measurements.

For accurate flux, a gradient in the w term phase is not important. The `advise` function's least squares fit to uvw gives the best fitting plane, dispersion in w' gives the beam size in w. Max amplitude loss plus simple geometry gives the facet size. But for accurate positioning the facet size should be such that even a gradient in the w term is not important for a given amplitude loss.

Further Advice

The advice for accurate flux may sometimes be too small to the extent of causing artifacts. The positional error might be large enough that sources close to the edge of the facets may end up appearing on 2 facets. In general if one requires accurate flux values, then it would be advisable to use a few more facets than the minimum suggested by `advise`. Make sure that there is not any source splitting over the image.

If very accurate positions are needed then either one has to image with the large number of facets suggested or try an alternative solution. That alternative solution is to image with a number of facets which is practical then get the best fit the position of each of the component from the UV-data. The fitting of components in the UV data is yet to be implemented in AIPS++.

7.2.3 Deconvolution

Deconvolution in the wide-field context is presently available only via CLEAN and implemented only for the Hogbom and Clark CLEAN variants (see Imager.clean and Dragon.image).

7.2.4 Self-calibration

If no self-calibration is needed, either Imager can be used directly or else Dragon can be used but with the arguments `levels` and `amplitudelevel` in `Dragon.image` left at `'0Jy'`.

If you do need self-calibration, then it is set to start at an amplitude level where it is expected that the phase or amplitude errors start to become significant. Usually the user will do phase self-calibration only initially, followed later by phase and amplitude self-calibration at a much lower level of residuals.

```
drgn := dragon('data.ms')
drgn.setimage(...)                  # Setup image parameters
#
drgn.image(levels='1Jy 0.3Jy 0.1Jy',amplitudelevel='0.2Jy',
           timescales='200s 200s 100s', niter=50000, gain=0.05,
           threshold='0.05Jy', plot=F, display=F, algorithm='wfclark')
```

In this example, CLEANing will be done and self-calibration will be executed at residual levels of 1Jy, 0.3Jy and 0.1Jy. As `amplitudelevel` is set to 0.2 Jy, the calibration steps at 1Jy and 0.3Jy will be phase calibration and correction only. However, the step at 0.1Jy will be a phase and amplitude phase calibration and correction. The `timescales` argument defines the averaging time period for the calibration solutions.

7.2.5 Masking

Deconvolution converges faster when the algorithm is told where to expect flux. This is especially the case for CLEAN in the presence of bright sources. A description of how to handle this within Imager can be found in the basic Imaging chapter of `Getting Results`.

In Dragon, the argument `maskmodification` in Dragon.image can be used to define masks at each self-calibration step. The argument can take the following: 'none' for no masking, 'auto' for automatic masking and 'interactive' for user defined ones.

When `maskmodification='auto'`, the restored image at the level CLEAN reached is used to make a threshold mask. All regions which are above the flux limit defined in `levels` is set to be used for searching for CLEAN components.

When `maskmodification='interactive'`, a Viewer display panel is launched when CLEAN reaches the flux level defined and the user can define CLEAN regions interactively.

7.2.6 Outlier fields

Outlier fields are regions outside the main image where there are sources that whose sidelobes fall into the field of view. Such regions would be around bright radio sources such as Cyg-A, Vir-A, the Sun etc. Such sources have to be included in the CLEAN and self-calibration loops so as to achieve good dynamic range. A special outlier field is the pole regions where interference tend to pile up in rotation synthesis arrays. This can be treated as an outlier field and the effect of such interference can thus be somewhat reduced from field of view.

The position of the center of the outlier fields can be given manually or can be looked up from a catalog by using the Measures module.

First let us see the steps that one would undertake to handle outlier fields ddirectly in Imager:

```
dir1:=dm.direction('J2000', '19h00m00', '50d00m00')                # Outlier 1
imgr.setimage(nx=200,ny=200,cellx='10arcsec', celly='10arcsec',
              doshift=T, phasecenter=dir1, mode='mfs',
              spwid=[1,2], facets=1)
imgr.make('outlier1');
#
dir2:=dm.direction('J2000', '19h05m00', '55d00m00')                # Outlier 2
imgr.setimage(nx=200,ny=200,cellx='10arcsec', celly='10arcsec',
              doshift=T, phasecenter=dir2, mode='mfs',
              spwid=[1,2], facets=1)
imgr.make('outlier2');
#                                                                  # Main field
imgr.setimage(nx=2000,ny=2000,cellx='10arcsec', celly='10arcsec',
              doshift=F, mode='mfs', spwid=[1,2]
              facets=5)
imgr.make('mainfield')
#                                                                  Deconvolve all together
imgr.clean(algorithm='wfclark', niter=1000, gain=0.1,
           model="mainfield outlier1 outlier2",
           image="mainfield.restored outlier1.restored outlier2.restored",
           residual="mainfield.residual outlier1.residual outlier2.residual")
```

In Dragon to image with outlier fields one would proceed as follows:

```
#                                                                # Main image
drgn.setimage(name='mainfield' nx=2000,ny=2000,cellx='10arcsec',
              celly='10arcsec',
              doshift=F, mode='mfs', spwid=[1,2], facets=5)
#                                                                # Outlier 1
dir1:=dm.direction('J2000', '19h00m00', '50d00m00')
drgn.setoutlier(name='outlier1',nx=200,ny=200,cellx='10arcsec',
                celly='10arcsec',
                doshift=T, phasecenter=dir1, mode='mfs',
```

```
                spwid=[1,2], facets=1)
#                                                                    # Outlier 2
dir2:=dm.direction('J2000', '19h05m00', '55d00m00')
drgn.setoutlier(name='outlier2',nx=200,ny=200,cellx='10arcsec',
                celly='10arcsec',
                doshift=T, phasecenter=dir2, mode='mfs',
                spwid=[1,2], facets=1)
#                                                                    # Image and deconvolv
drgn.image(levels='1Jy 0.3Jy 0.1Jy',amplitudelevel='0.2Jy',
           timescales='200s 200s 100s', niter=50000, gain=0.05,
           threshold='0.05Jy')
```

7.2.7 Models

The process of imaging can be helped and made to converge much faster if a model is passed to Dragon/Imager to start with. The models can either be position, flux and shape of sources (via a Componentlist `tool`) or a model image (or both).

Model images are usually images made of the region but with poorer sensitivity and dynamic range. The model(s) can be put in the model data column of the MeasurementSet by using the `Imager.ft` function. Otherwise these can be passed as the `model` argument for the `Dragon.image` or `Imager.clean` function.

A Componentlist `tool`, as its name suggests, allows one to make a list of source models. Componentlists can be constructed from from standard catalogues (e.g the NVSS, VLA FIRST etc.) via the asciitocomponentlist constructor. Using the catalogues can be very helpful to Dragon or Imager (helps convergence especially in complex regions).

A Componentlist can also be created from an image using the interactive Imagefitter `tool`. This `tool` allows the user to interactively select regions where there are components and estimate its parameters.

7.3 A worked example: VLA 4m imaging of Coma

Here is an example of a *Glish* script which was used to image the Coma cluster at 74MHz with data from the VLA in the B and C configurations.

```
include 'dragon.g'
myms := fitstoms('coma.ms','COMA-4CB-CUT.FITS',T,F)     # Convert from FITS to MS
myms.done()
#
drgn := dragon('coma.ms')                               # Create Dragon tool
drgn.setimage(name='coma',nx=1800, ny=1800,             # Set imaging parameters
              cellx='30arcsec', celly='30arcsec',
              doshift=F,
```

```
             phasecenter=dm.direction('J2000','0deg','0deg'),
             mode='mfs', facets=25)
#
drgn.setoutlier(name='CenA' , nx=400, ny=400,          # CenA outlier field
                cellx='37arcsec', celly='37arcsec' ,
                doshift=T,  mode='mfs',
                phasecenter=dm.direction('J2000','201.3deg','-42.6deg'),
                nchan=1, start=1, step=1, spwid=1, fieldid=1)
#
#setting an outlier field towards Virgo-A; note for direction we have
# made use  of the inbuilt radio catalog
drgn.setoutlier(name='VirA' , nx=400, ny=400,          # VirgoA outlier from builtin catalog
                cellx='37arcsec', celly='37arcsec' ,
                doshift=T, mode='mfs',
                phasecenter=dm.source('VIRGOA'),
                nchan=1, start=1, step=1, spwid=1, fieldid=1)
#
drgn.uvrange(uvmin=0, uvmax=10000000)                  # uv range
drgn.weight(type="uniform")                            # uniform weighting
drgn.setoptions(padding=1.5, cache=0)                  # padding for the Fourier Transform
#
drgn.image(levels='1Jy 0.3Jy 0.1Jy',                   # image and selfcalibrate
           amplitudelevel='0.2Jy',
           timescales='200s 200s 100s',
           niter=50000, gain=0.05,
           threshold='0.05Jy', plot=F, display=F)
#
drgn.done()
```

7.4 Special cases

- Very bright and compact sources - the NNLS algorithm (Briggs) can be used to deconvolve the bright source and then the residual visibility is imaged using wide field imaging.
- Bright source in the sidelobes - this is a nasty problem as the phase and amplitude of sidelobes are very different from the main lobe. No exact cure is yet available but it can be considered as an outlier field problem and the actual widefield imaging will correct an "averaged" quantity. This will reduce the effect of the bright interfering source but may still leave some artifacts.
- Mosaicing and wide-field - this is a very special and rare case when each pointing is wide enough to warrant the usage of widefield imaging and the object under observation is large enough to need many such pointings. Mosaicing and widefield imaging can be done simultaneously.

7.5 Troubleshooting

7.5.1 Insufficient faceting

This problem is detected when all the sources away from the center of the image are broadened. This should not be confused with the effect of bandwidth smearing (decorrelation).

Another effect of too few facets is that sources near the edge of facets appear as double. This is beacause the positional error is large enough to cause sources near the edge of the facets appear on 2 facets.

7.5.2 Non-isoplanatism

The effect of non-isoplanatism is that the phase/gain of the longer baselines gets distorted while the shorter ones are not affected. In the image one would notice broadening of all known point sources and they will have similar structures. This will affect all the sources in the field.

The solution to this is to image with self-calibration corrections starting at much higher flux levels. One should be aware of the time scale of the non-isoplanar changes. If one passes a too low time scale to self-cal, the signal-to-noise ratio of the self-calibration estimates will be worse. If it is much larger than necessary then it's equivalent to no self-calibration.

7.5.3 Asymmetric primary beams

The capability to correct for directional dependent part of phase and amplitude errors is going to be implemented soon. This should be able to take care of assymetric primary beam and effect of the atmosphere/ionosphere which are direction dependent.

7.5.4 Clean diverging on the edges of the facets

The problem is most probably the effect of ringing on the edge of the facets which can be reduced by padding. So in the `Dragon.setoptions` or `Imager.setoptions` function the padding factor can be given a value of more than 1 (usually 1.2 to 1.5) depending on how severe the edge get picked up by clean. The more the padding the longer will FFT take.

7.6 Bibliography

- Thompson, A. R., Moran, J. M. and Swenson, G. W., Interferometry and Synthesis in Radio Astronomy, Wiley, 1986.

- Cornwell, T. J. and Perley, R. A., Radio Interferometric imaging of very large fields, 1992, Astron. Astrophys., 261, 353-364.

- Briggs, D., High Fidelity Deconvolution of Moderately Resolved Sources, Ph.D Thesis.

Chapter 8

Mosaicing (Multi-field imaging)

Mark Holdaway

8.1 Mosaicing Background

The Fourier transform relationship between the Fourier plane and the image plane must include the primary beam:

$$V(u) = \int A(x)I(x)e^{-i2\pi ux}dx \tag{8.1}$$

Hence, given the image, it is trivial to simulate the corresponding data. However, given the data and desiring the image, we have an inverse problem to solve.

Early attempts at mosaicing treated each field independently, deconvolving and self-calibrating each, and then sewing the overlapping fields' images together via the basic mosaicing equation:

$$I(x) = \frac{\sum_f I_f(x)A_f(x)}{\sum_f A_f^2(x)}. \tag{8.2}$$

However, Cornwell (1988) demonstrated that superior results can be achieved via a simultaneous deconvolution of the data from all the fields. This simultaneous deconvolution was achieved by using maximum entropy (MEM) or maximum emptiness as a solution engine to solve the inverse problem. Total power could be added as additional fields with their own primary beam. However, MEM's positivity bias, which is detrimental to low SNR imaging, and its lukewarm reception from radio astronomers led to a search for other algorithms to image multi-field data.

Sault *et al.* (1996) have implemented mosaicing algorithms which can use either CLEAN or MEM for simultaneous deconvolution.

8.1.1 The AIPS++ Mosaicing Solution

Cornwell, Holdaway, and Uson (1994) proposed a novel mosaicing algorithm for the upcoming millimeter array (MMA): generate the mosaic of the dirty images and a single approximate point-spread function (PSF), and then proceed with any conventional single field deconvolution algorithm. For the MMA's high-quality Fourier-plane coverage and similar PSF's for all fields in the mosaic, this approach was not limited by the differences in the approximate PSF and each field's actual PSF until the possible image dynamic range exceed a few hundred to one.

AIPS++ takes this approach to mosaicing a step further: perform an incremental deconvolution of the residuals with the approximate PSF, with an exact subtraction of the cumulative model brightness distribution at the end of each incremental "major cycle" (similar in concept to the major cycles of the Clark CLEAN).

If all of the fields are observed with many short snapshots over and over again (this is the usual way to make a mosaic observation) then each field will have similar Fourier coverage and hence similar synthesized beams. An approximate PSF can be created which is a fairly good match to the actual PSF of each of the fields. Also, if the sky-coverage of the observed fields is Nyquist or better, then the approximate, shift-invariant PSF will be a reasonable match to the actual PSF of sources at various locations across the mosaic. The residual visibilities from each field can be transformed and mosaiced to make a single residual mosaic image. This mosaic image can be deconvolved with the deconvolution method of your choice; for example, with Clark CLEAN, Multiscale CLEAN, maximum entropy, or maximum emptiness.

The deconvolution algorithm cannot deconvolve arbitrarily deeply, because at some level the discrepancies between our approximate shift-invariant PSF and the true PSF at any location in the image will become apparent, and we will start "cleaning" error flux. Hence, we need to stop deconvolving when we have gotten down to the level of these PSF discrepancies. At this point, we take the part of the model brightness distribution we have just deconvolved and calculate model visibilities (using the measurement equation) and subtract them from the (corrected) data visibilities. To the extent that the primary beam and sky pointing are exact, the visibility subtraction is also exact. The residual visibilities can then be re-mosaiced, but the peak residual is at a much lower level. The process of deconvolving with the approximate, shift-invariant PSF then continues, and another increment to the model brightness distribution is formed, removed from the remaining residual visibilities, and added to the cumulative model brightness distribution. Borrowing from the Clark CLEAN's terminology, we call each cycle of incremental deconvolution and exact visibility subtraction a "major cycle".

8.1.2 Advantages of Incremental Deconvolution with an Approximate PSF

If the major cycles are properly controlled, there are potential advantages to incrementally deconvolving with an approximate PSF. For one, we are doing a regular single image deconvolution, so we are free to pick whatever algorithm we like (though NNLS has not been implemented in the multi-field context). Second, we are spending less CPU doing FFTs. In the Cornwell (1988) MEM-based mosaicing approach, each field needed a couple of FFT's for each iteration. In our incremental deconvolution using MEM, the entire image needs a couple of FFTs per iteration (independent of the number of fields), plus a couple of FFTs per field per major cycle. As there are several MEM iterations per major cycle, we usually come out ahead using the incremental deconvolution.

8.2 Mosaicwizard for Quick, Simple Mosaicing

If you've already read enough and don't want any more details, you are reading the right section. The AIPS++ Mosaicwizard will walk a novice user through the details of making a mosaic image. For example, you don't need to know any details about where your pointings are: a simple graphical display of the pointings allows you to select the ones you want. You don't have to figure out how big your image should be, or what cell size to use, or even what deconvolution algorithm to proceed with; Mosaicwizard either figures these details out for you, or gives you sensible defaults which you can change if you want. The Mosaicwizard will even write a script which shows you the most important commands you need for making a mosaic image like the one the wizard made.

Of course, the cost of this simplicity is loss of the detailed control over the mosaicing process, but if you are confused, it is well worth going through the exercise of making a mosaic with the Mosaicwizard. The Mosaicwizard even comes with its own test measurement set, so you can start playing with it right now.

8.3 Fundamental and Necessary Details

There are a number of simple things you need to do so that your mosaic is successful. Its easy to forget them, though, so we'll remind you.

8.3.1 Set the Data Fields

Mosaicing is a time consuming process, so it will be worthwhile to make a restricted version of the mosaic first. For example, you may want to just image a few fields at lower resolution to reduce the number of pixels you are imaging. Eventually, you will want to image most or all of the observed fields.

```
imgr.setdata(fieldid=1:4)            # Select first 4 fields
```

8.3.2 Set the Image

One of the fields must be specified in `tool function` Imager.setimage to provide the direction of the resultant image's reference pixel. For example, with a 25 pointing (5 x 5 raster) observation, field 13 could be the central field:

```
imgr.setimage(nx=256, ny=256, cellx="3arcsec",
              celly="3arcsec", stokes="I", fieldid=13)
```

Or if a given position is wanted as the image center, then the default measures tool can be used in conjunction with setimage as follows:

```
imageCenter := dm.direction('J2000', '19h30m50', '-20d32m45.4')
imgr.setimage(nx=256, ny=256, cellx="3arcsec",
              celly="3arcsec", stokes="I",
              doshift=T, phasecenter=imageCenter)
```

Setting `doshift=T` tells the software to use the value specified in the `phasecenter` argument.

Currently, the deconvolution methods of choice for multi-field applications, namely Multi-scale CLEAN and maximum emptiness or entropy, do not treat Stokes I and V simultaneously.

8.3.3 Setting the Voltage Pattern (primary beam)

Remember to tell Imager to use the voltage pattern (primary beam). If you don't, the image will be horribly confused. This is because we must account for the voltage pattern when imaging the same location on the but from different pointings.

```
imgr.setvp(dovp=T, usedefaultvp=T, dosquint=F).
```

If you don't like the default voltage pattern (provided for a range of telescopes; see Imager.setvp), you can specify your own voltage pattern and bind it to the telescopes in your MeasurementSet by using the Vpmanager (voltage pattern manager). The Vpmanager will produce a table describing the different telescopes' voltage patterns, and this table can be used by Imager via the `setvp tool function`.

```
imgr.setvp(dovp=T, usedefaultvp=F, vptable='MY.VP.TABLE').
```

The beam squint for certain telescopes (like the VLA) has been included in the default voltage pattern models. It is not relevant for other telescopes (like the ATCA). Beam squint can also be adjusted with the Vpmanager. For the voltage pattern application to include the effects of beam squint, reapplied in parallactic angle increments of 10 degrees, try

```
imgr.setvp(dovp=T, usedefaultvp=T, dosquint=T, parangleinc='10deg')
```

Note that the beam squint conventions have not been thoroughly tested, and could be backwards in orientation.

8.3.4 Weighting

If you use "uniform" or "briggs" weighting, the weighting details will depend upon the way the data are gridded. However, if all fields are specified in function `setdata`, then the weights from all fields will be gridded onto a single grid for the purposes of calculating the weights. This is probably not what you want to do. Rather, it may make more sense to weight the data on a field-by-field basis:

```
for (myfield in [1:25]) {
   imgr.setdata( fieldid=myfield )            # Weight each field separately
   imgr.weight(type="uniform")
}
#
imgr.setdata( fieldid=[1:25] )                # Now select all fields
```

8.3.5 Deconvolving

We are finally ready to image and deconvolve. You can use either imager's CLEAN or MEM functions. Only algorithms with the "mf" prefix will perform multi-field imaging correctly (i.e. algorithm "clark" will grid the data from all specified fields onto the same grid, resulting in a very confused image indeed. CLEAN's mosaicing methods include `mfclark`, `mfhogbom`, and `mfmultiscale`, while MEM's mosaicing methods include `mfentropy` and `mfemptiness`.

8.4 Advanced Details

8.4.1 Controlling the Major Cycles

The key to making the incremental deconvolution in AIPS++ multi-field imaging successful lies in controlling just how deeply we deconvolve in the major cycles. The control parameters discussed here can be set with Imager.setmfcontrol. If we deconvolve too deeply with the approximate PSF, we will spin our wheels while adjacent fields with slightly different PSF sidelobes argue over where the low level model flux belongs; the answer will of course be in error, we will mis-subtract from the data visibilities, and we will have to correct our error in the early stages of the next major cycle of incremental deconvolution. If we don't deconvolve deeply enough in each major cycle, there will be more major cycles, which are dominated by the exact model subtraction.

One can see fairly easily if one is cleaning too deeply or not cleaning deeply enough in a given major cycle by looking at the plots given when the argument `displayprogress=T` (in `tool function clean`). When trying to clean too deeply, the peak residual level towards the end of a major cycle will flatten out (or may even increase and diverge) for many iterations. At the start of the next major cycle, the peak residual level will begin at a higher level than at the end of the previous major cycle. (This can be especially true for mosaicing with multi-scale clean, where the approximate PSF may actually be a pretty poor

match for the true PSF's on the larger size-scales). If all is well, this major cycle will still get to a lower peak residual than the previous one. However, stopping the major cycles sooner will prevent us from spinning our wheels, resulting in a comparable deconvolved image in fewer iterations.

Eventually, there will probably be a smarter, more automatic way to determine if we need to stop the current major incremental deconvolution cycle. Right now, the tools are simple, but easy to use. If the progress display indicates that ending the major cycle sooner is appropriate, we can do that in one of two ways:

- by increasing the `cyclefactor`. The major cycle cleaning threshold is a fraction of the peak image residual. That residual fraction is determined by the cyclefactor multiplied by the peak negative sidelobe of the approximate PSF. Stopping the major cycle cleaning sooner can be accomplished by increasing the cyclefactor. Values ranging between 2 and 4 are common.

- by decreasing the `cyclespeedup`. What if we have set the `cyclefactor` too low? The cycle threshold as calculated above can be made to drift upwards by setting the `cyclespeedup`. The threshold will double every `cyclespeedup` iterations in the major cycle until the major cycle stops. If the `cyclespeedup` is less than or equal to 0.0, no adjustments to the calculated cycle threshold will be made.

In addition to these cycle control parameters, which are applicable to mosaicing with CLEAN, Multi-Scale CLEAN, MEM, or Maximum Emptiness, there are two more control arguments set by `tool function imager.setmfcontrol` which are applicable only to Multi-Scale CLEAN. These are `stoplargenegatives` and `stoppointmode`, which are discussed below.

8.4.2 Details with Multi-Scale CLEAN

See the basic Imaging for multi-scale basics.

Sometimes in the first few iterations of Multi-scale CLEAN in the mosaicing context, the largest scale will be dominated by large negative residuals (i.e. this is just the negative bowl, integrated over the area of the largest clean-component scale). One way to fix this is to make the largest scale smaller. Another way is to use a tighter mask which excludes finding large scale components in the bowl region. And a third, ad hoc way is to stop the major cycle when a negative component is found on the largest scale. This allows the exact subtraction to proceed, often resulting in a reduced bowl level. Stopping the major cycle upon encountering a negative component on the largest scale should only be performed on the first few cycles, as during subsequent cycles small amplitude large scale components may be required to make adjustments in the image. The number of cycles for which stopping when a negative component is found on the largest scale can be controlled by the parameter `stoplargenegatives` in `imager.setmfcontrol`. As smaller scales may require negative components

to correct for errors made by "over-cleaning" in the larger cycles, no restriction should be placed on negative components from smaller size scales.

8.4.3 The `imagermultiscale()` Function

This global function is discussed in the basic Imaging chapter of Getting Results. You can also use it in the mosaicing context (just specify all of the fields you are interested in).

8.4.4 Details with MEM

If there are bright unresolved or barely resolved sources in the field, it may be advantageous to perform a Clark Clean down to the level of the peak extended emission, or include a component list in the model, because MEM does not work well on bright point-like sources.

The maximum entropy/emptiness algorithm has been modified to fit into the incremental deconvolution/major cycle framework adopted by mosaicing in AIPS++. These algorithms deal with both the incremental brightness distribution, which it seeks to solve given the approximate PSF and the residual mosaic image, and the cumulative brightness distribution for the calculation of the entropy function and its gradients. When maximum entropy starts up, it typically takes the algorithm four or five iterations to "find itself" and get the control parameters set to achieve the proper balance between the gradients of the entropy and the chi squared. Once a good balance is struck, the algorithm makes marked progress towards convergence. At the end of a major cycle, the relevant control parameters are saved and used for the next major cycle.

8.4.5 Using Convolutions Instead of Visibility Subtraction

If several different models or images are contributing to the model visibilities, the a visibility plane subtraction must be used. However, if a single model image covering all observed fields is used, as is usually the case with mosaicing, then it is more efficient to bypass the degridding and gridding operations and just calculate the residual images for each field by a convolution with each field's true PSF.

8.4.6 Outlier Fields

Sometimes flux from a confusing source will "leak in" through the outer sidelobes of the primary beam. This is a vexing problem, as the outer sidelobes are not in the primary beam model, the confusing source will not be removed from the data or corrected visibilities in the usual mosaicing algorithm, and its sidelobes will persist, possibly spoiling the dynamic range of other regions in the mosaiced image. In addition, each field will experience the outlying confusing source at a different sidelobe level, and therefore at a different flux, and time-dependent

or antenna dependent gains can result as the azimuthally asymmetric sidelobes rotate over the source and pointing errors result in sidelobe jitter.

The ultimate solution to this problem lies in the direction-dependent gain solver, forthcoming in AIPS++ but not yet available, so we must use other means. Simply using an outlier field in addition to the main image will not work, as the value of the primary beam at the location of the confusing source may well be zero. Within AIPS++ current capabilities, one can turn off the application of the primary beam with function Imager.setvp by putting argument `dovp=F`. Then image the confusing regions one field at a time, and subtract the modeled flux from each field's visibilities. The residual visibilities can then be mosaiced with one of the standard multi-field algorithms. The packaging is not ideal at this point and you have to do some adhoc processing with the Table `tool` (which is one of the strengths of AIPS++ - you can do this when you have to !).

Here is an example:

```
opos := dm.direction('J2000', '12h26m33.248000',    # Outlier direction
                     '02d19m43.290000')
imgr.setvp(dovp=F)                                  # Turn off application of primary l
#
for (myfield in 1:25) {                             # For each field image outlier and
  modelName := spaste('model', i)
#
  imgr.setdata(fieldid=myfield)
  imgr.setimage(nx=64, ny=64, stokes='I',
                cellx=0.2, celly=0.2,
                phasecenter=opos)
  imgr.clean(threshold='0.001Jy', model=modelName, ...)  # Image outlier
#
  imgr.ft(model=modelName)                               # Overwrite MODEL_DATA column
}
imgr.done();

# we are about to make an irreversible change in the MS (called MS.TABLE here)
# so you'd better have a copy MS somewhere!


tab := table("MS.TABLE", readonly=F)                # Create table tool from MS
correctedvis := tab.getcol("CORRECTED_DATA")        # Get corrected data
modelvis := tab.getcol("MODEL_DATA")                # Get model data
correctedvis -:= modelvis                           # Subtract model
tab.putcol("CORRECTED_DATA", correctedvis)          # Put subtracted data back in
tab.done();
```

8.4.7 Component Models

These are discussed in the basic Imaging chapter of Getting Results.

Component Models are handled by the Componentlist `tool`.

The only difference in the mosaicing context (from single field imaging) is that the Component Model *must* be a true representation of the sky brightness. Any attenuation for primary beams is handled by Imager.

If you have a Component Model which has been attenuated by the primary beamm you can correct for that with the Imager.pb function.

```
imgr.make('empty')
imgr.pb('empty', operation='correct', incomps='cl.attenuated.sky',
        outcomps='cl.sky')
```

Note that you need the input image so that Imager knows what telescope you are using so that it can get the correct model. Alternatively, if you have used Imager.setvp to set a Voltage pattern from an external table, it would use that.

The Componentlists are specified by their disk file names. (the arguments are not Componentlist `tools`).

8.4.8 Flux Scale Images

When correcting for the effects of the primary beam to achieve an accurate and uniform flux scale across the image (i.e. by dividing by the primary beam in the case of a single field observation), the noise and errors at the edge of the mosaic sky coverage are amplified. The noise amplification distracts from the visual beauty of the image and may complicate the image's display and interpretation.

Sault et al (1996) endorse a different image plane weighting in the mosaicing which results in a constant noise level across the image, but has a variable flux scale.

In AIPS++ we have implemented an image plane weighting similar to Sault's scheme, but the noise goes to zero outside the region covered by the primary beams. The flux scale is position dependent, but it is flat over most of the mosaic sky coverage. The flux scale images can be created by setting the `fluxscale` argument in Imager's `setmfcontrol` function. Regions outside the multi-field primary beam pattern will have a zero value.

8.4.9 Masks

Routinely in single field deconvolution, only the inner quarter of an image is deconvolved so that the sidelobes from this region can be correctly subtracted from the entire image. However, in the multi-field case, such a restriction usually does not exist. The major cycles only deconvolve down to a certain level, fixed by the sidelobe characteristics of the PSF. After that, the exact subtraction of the deconvolved flux is carried out. Typically, the exact subtraction is performed by multiplying the brightness distribution by a field's primary beam, convolving by that field's exact PSF, multiplying by the primary beam again, and subtracting

from the previous cycle's residual mosaic. The two primary beam multiplications ensure that the far out effects of the PSF, which will not be correct due to the full-image deconvolution, will not effect the model brightness distribution.

If no mask is used, we help the single field major cycle deconvolution algorithms out by creating a mask from the generalized primary beam pattern of all observed fields, with zero outside the outermost fields' primary beam main lobes. If you don't want this mask for some reason, you should supply your own mask image.

8.5 Self-calibration

Apart from some rather esoteric details in the solution interval, self-calibration of multi-field data in AIPS++ is operationally identical to self-calibration of single-field data, but with function Imager.dovp activated.

The success of self-calibration is largely dependent on being able to get a reasonable model for the source brightness distribution over some range of Fourier spacings. Synthesis imaging lore emphasizes the uses of self-calibration when bright point sources dominate the emission, but there is no rule that prohibits the use of shorter baselines in self-calibration. In mosaicing, there will generally be much more signal to work with on the shorter baselines, but the extended structure to which these baselines are sensitive is more likely to suffer from imaging errors than the simple point-like sources, especially if total power data is either lacking or questionable quality. Self-calibration on mosaic images is still on open field.

Self-calibration for multi-field observations starts with a model image of the entire mosaiced region, presumably obtained from either the Imager.clean or Imager.mem functions. Then the Imager.dovp function must be run to turn on the primary beam application (it should already be on if you have just made your clean or mem image with this Imager `tool`). Then use the Imager.ft function to calculate the model visibilities, given by the Fourier transform of the model brightness distribution times the primary beam, for each field. Then proceed with self-calibration as usual with a Calibrater `tool`.

The solution interval can be longer or shorter than the time spent on each field. One must consider the variable emission in the different fields. If the solution interval is very short, some fields may not have enough signal for a good solution, and solution intervals and gain solutions might be calculated on a field-by-field basis. If some fields do not have much emission and a short solution interval is desired, it me be desirable to self-calibrate only on those fields with enough emission for a good self-calibration solution. If the solution interval is longer than the integration time per field, then the fields with more emission (and higher signal-to-noise ratio) will dominate the gain solution, helping along the fields with less emission.

8.6 An Example Mosaicing Script

The following script is nothing special, just making an interferometer-only mosaic image from the multi-field test measurementset which is distributed with AIPS++:

```
include 'imager.g'

# Generate MS and construct Imager tool

imagermaketestmfms(msfile='XCAS.ms')
imgr := imager('XCAS.ms')

# Use the first field as the image center

imgr.setimage(nx=256, ny=256, cellx="3arcsec",
              celly="3arcsec", stokes="I", doshift=F,
              mode="mfs", spwid=1, fieldid=1,  facets=1)

# Weight each field individually

for (myfield in [1:7]) {
  imgr.setdata(fieldid=myfield)
  imgr.weight(type="uniform")
}

# Use all the data for the mosaic
imgr.setdata(mode="channel", nchan=1, start=1, step=1,
             spwid=1, fieldid=[1:7])

# Use the voltage pattern (primary beam)

imgr.setvp(dovp=T, usedefaultvp=T, dosquint=F);

# Make a MEM image  (using  mask image)
base := 'mem'
maskname:= spaste(base, '.mask')
modname := spaste(base, '.model')
imgname := spaste(base, '.image')
resname := spaste(base, '.resid')
scalename := spaste(base, '.scale')

reg1:=drm.box([70, 70,1,1], [185, 185, 1, 1])            # define region of image
imgr.regionmask(maskname, reg1)                          # make a mask image to limit search for

imgr.setmfcontrol(cyclefactor=3.0, cyclespeedup=20.0,
```

```
                  fluxscale=scalename)
imgr.mem(algorithm='mfentropy', niter=80, sigma='0.001Jy',
         targetflux='10.0Jy', constrainflux=F,
         displayprogress=T,  fixed=F,
         complist='', prior='',
         mask=maskname,
         model=modname,
         image=imgname,
         residual=resname)

# Make a multi-scale CLEAN image  for comparison

base := 'msclean'
modname := spaste(base, '.model')
imgname := spaste(base, '.image')
resname := spaste(base, '.resid')
scalename := spaste(base, '.scale')
imgr.setscales(scalemethod='uservector', uservector=[0.0, 3.0, 10.0, 20.0])
imgr.setmfcontrol(cyclefactor=3.0, cyclespeedup=200.0,
                  fluxscale=scalename);
imgr.clean(algorithm='mfmultiscale', niter=1000, gain=0.6,
           threshold='0Jy',
           displayprogress=T, fixed=F, complist='',
           mask=maskname,
           model=modname,
           image=imgname,
           residual=resname)

# Destroy Imager tool

imgr.done();
```

8.7 Bibliography

- T.J. Cornwell "Radio-interferometric imaging of very large objects", *Astronomy & Astrophysics*, **202**, 316, 1988.
- T.J. Cornwell, M.A. Holdaway, and J.M. Uson, "Radio-interferometric imaging of very large objects: implications for array design", *Astronomy & Astrophysics*, **271**, 697 (1994).
- M.A. Holdaway and T.J. Cornwell "A simple multi-scale clean deconvolution algorithm", *in progress* (2000).
- R.J. Sault, L. Staveley-Smith, W.N. Brouw, "An approach to interfero-

metric mosaicing", *Astronomy & Astrophysics Supplement Series*, **120**, 375 (1996).

Chapter 9

Single Dish Analysis

R.W. Garwood & J.P. McMullin

9.1 Introduction

DISH is a collection of *Glish* scripts and clients which provide an environment within AIPS++ for single dish radioastronomy analysis.

The latest version of DISH merges development between the single dish and interferometric efforts in AIPS++. DISH is a standard tool and may take advantage of the tool manager interface. DISH also has a new and much simplified Command Line Interface (CLI). Note: If you have used previous versions of AIPS++/DISH, you **must** remove the file containing the previous state information. This is contained in a file called *default* which will be under your home directory in the aips++/dishstate directory:

```
> rm ~/aips++/dishstate/default
```

This will prevent spurious errors from appearing upon startup due to the changes in the software from the last release.

More information is available on DISH in the User Reference Manual.

DISH may be accessed in several ways:

1. After setting up the environment for AIPS++:

   ```
   > . /aips++/release/aipsinit.sh       # bash shell AIPS++ setup
   > source /aips++/release/aipsinit.csh # (t)csh shell AIPS++ setup
   ```

   execute:

   ```
   > glish -l dish.g                     # to load just the single dish analysis
   or
   > aips++ -l dish.g                    # to start AIPS++ with single dish
   ```

2. Once in AIPS++, it can be accessed through the tool manager interface.

   (a) Select on Packages, dish.

   (b) Select on Modules, dish.

   (c) Select on Tools, dish.

   (d) Press create.

   (e) Choose the toolname for the new DISH tool and press "Create".

3. Once in AIPS++, it can be accessed from the command line.

   (a) include 'dish.g'

   (b) d:=dish();

The default is to start the dish tool with the GUI interface as before. However, this may be over-ridden using a local .aipsrc value:

```
#dish.gui.auto:T
dish.gui.auto:F
```

In this release (v1.7), DISH may use AIPS++ flat tables (filled through the fits2table command) or MeasurementSets (either generated naturally or converted via sdfits2ms or through facility fillers (e.g., gbtmsfiller)).

9.2 Getting Data Into Dish

There are several ways to get data into DISH:

9.2.1 MeasurementSets

If you have data produced as a MeasurementSet, it merely needs to be loaded into the system. From the GUI, this entails using the Open option in the File menu (either Read only or Read and Write). This opens up a file chooser GUI which allows you to select the relevant file. From the command line, the command:

```
- d.open('pne.ms');
```

Loads the file into the results manager and is ready for use.

9.2.2 Facility filler (e.g., GBT)

Some facilities have native fillers which directly convert raw data formats into the MS format. An example based on the GBT is provided below:

```
gbtmsfiller project=zeroSpacing01 minscan=401 maxscan=441 msrootname='oriona1_'
 fillrawpointing=True

where project = the GBT project directory
      minscan = first scan to fill
      maxscan = last scan to fill
   msrootname = the name of the output MS; this will have the Backend used
                appended to it.
```

A dish utility function, *import*, streamlines this process, filling the data and doing an *open* on the resulting MeasurementSet.

9.2.3 SDFITS data file (e.g., Arecibo Observatory)

1. Convert to MS

   Use the AIPS++ utility *sdfits2ms* to convert from a fits file following the SDFITS convention to an AIPS++ table. The syntax, from the UNIX prompt, is:

   ```
   sdfits2ms sdfits_name ms_name, for example:

   charybdis<965>$ sdfits2ms n7027.sdfits n7027.ms
   0%....10....20....30....40....50....60....70....80....90....100%
   End Successfully:      34.79 real        5.43 user        5.03 system
   ```

2. Use the AIPS++ utility *fits2table* to convert from a binary table following the SDFITS convention to an AIPS++ table. The syntax, from the UNIX prompt, is:

   ```
   fits2table input=sdfits_file output=aips2_table [which_hdu=#]
   ```

   where "sdfits_file" is the name of the SDFITS file and "aips2_table" is the desired name of the new AIPS++ table created by this step. The third argument, "which_hdu", is optional. Use this to specify a specific FITS header data unit (hdu) other than the first one after the primary hdu. For example, if you have a FITS file which has three tables following the primary hdu and the second table is a valid SDFITS table then you would use `which_hdu=2` to indicate that that hdu is the one to be converted to the indicated output AIPS++ table.

   Note: Arecibo Observatory data may be brought into DISH in this manner.

9.2.4 UniPOPS SDD data file

There are three steps to getting a UniPOPS SDD data file into DISH:

1. Convert UniPOPS SDD data to UniPOPS FITS binary table.

   Use the UniPOPS utility *uni2fits* to convert the SDD data to FITS binary format. To use the program, you must be logged onto a Sun workstation where UniPOPS has previously been installed and have your path include:

```
export popsdir=~unipops/test/
export PATH=$popsdir/sunbin:$popsdir/utilities:$PATH
purgatory<143>$ uni2fits
Starting uni2fits ver 1.3
So you want to write UniPOPS data to FITS tape eh?
 (to abort this dialog, type CNTRL-C)

Do you want output file on disk(D) or tape(T)? [default=D]:
Using default (D).
Please enter name of output FITS file: hbnmeth.fits
Output file will be: hbnmeth.fits

Choose line or continuum (L, C) [default=L]:
Using default (L)

Label scan axis as Frequency(F) or Velocity(V) [default=F]: V

Please type input UniPOPS file name: sdd.www_001
Do you want to choose a scan number range? (y or n) n
-------------------------------------------------------
Ready to make FITS file, name = hbnmeth.fits
 Will use all L scans from sdd.www_001.
 is this ok? (type Y to proceed) Y

Getting initial information from UniPOPS file.
1+0 records in
1+0 records out

Previewing the UniPOPS file.

u2f version 1.5 (22Apr98),  run: Fri Oct 13 14:19:36 2000
12182+0 records in
12182+0 records out
Wrote 0 scans out of 3018; 0 fits records written.

Converting the data to FITS.

u2f version 1.5 (22Apr98),  run: Fri Oct 13 14:20:50 2000
12182+0 records in
12182+0 records out
Wrote 2674 scans out of 3018; 1675 fits records written.
```

```
FITS writing complete.
```

2. Convert UniPOPS FITS to standard SDFITS.

   Use the AIPS++ utility *uni2sdfits* to convert the data file. The syntax is:

```
uni2sdfits input=unipops_fits_file output=sdfits_file
e.g.
charybdis<276>$ which uni2sdfits
/aips++/daily/linux/bin/uni2sdfits
charybdis<277>$ uni2sdfits input=hbnmeth.fits output=hbnmeth.sdfits
uni2sdfits: Version 1999/07/30 BEG/TPPR/PJT/GvD

Scanning all HDUs looking for any non-constant DATA column shapes
Skipping HDU 0 of type : 1
Scanning HDU 1 of type : 4
HDU 1 is a BinaryTable with fixed shape DATA column with TDIM = (128,1,1,1,1)
Primary HDU
HDU # 1
column SERIES renamed to DATA
column BANDWIDT renamed to BANDWID
column PROJECT renamed to PROJID
column VELOCITY renamed to VFRAME
column VCORR renamed to RVSYS
column TOUTSIDE renamed to TAMBIENT
column BEAMWIDT renamed to BMAJ
NMATRIX keyword removed
MAXIS keyword removed
MAXIS2 keyword removed
MAXIS3 keyword removed
MAXIS4 keyword removed
MAXIS5 keyword removed
Removing column 2 : MAXIS1
keyword value set: TDIM195 = (128,1,1,1,1)
done.
```

3. Convert the SDFITS file to an AIPS++ table (either a MS or a flat table). See Section 2.2.

9.2.5 Dish demo data

There is also a utility for loading demonstration data into dish as a check of the functionality. More information is given on this in the Getting Results in AIPS++ Documentation.

```
- include 'dishdemodata.g'
- dishdemodata() # this will create several demo working sets
-   # on disk, which are then available to be loaded and
- # worked upon.
-                # dishdemo1, dishdemo2, dishmopra, dishparkes, dishspecproc
- # each of these can be accessed through the open command, e.g.
- d.open('dishdemo2'); # assumes 'd' is the defined DISH tool.
```

9.3 The Dish Graphical User Interface

In DISH, wherever practical, what you see is what you are operating on. So, for example, when you press the "Apply" button in the "Baselines" graphical user interface (GUI) frame of DISH, a baseline will be fit to whatever is currently plotted in the DISH plotter. Results appear in the results manager and, when appropriate, are immediately displayed on the plotter. Results can be moved between the results manager, the calculator, and the browser through the use of a menu which appears when the user presses the right mouse button in one of these frames. This default behavior can be changed so that selections from the Results Manager will be operated upon. This change is necessary to perform functions on SDIterators (see Bulk Processing).

The current state of DISH (the contents of the results manager, the settings of all of the operations, the contents of the calculator, and any active browsers) can be saved at any time either to the default state file or to a location of the user's choosing. DISH can be restored to a previously saved state. By default, DISH saves its state upon exit from glish and restores itself to that same state when DISH is started in a new *Glish* session.

The following sections detail the major graphical components of DISH.

9.3.1 Results Manager

The Results Manager is the heart of DISH. All results which DISH creates in response to user actions are stored in the results manager and listed in the *Results* window. These results are all available at the *Glish* command line as glish variables of the same name for the user to interact with. DISH immediately plots the currently selected result if it can be plotted. A description is associated with each result. The user can change the description or the name of the result if they choose to do so. Certain results can also be browsed. Since each result is also a glish variable of the same name, new names are limited to be a valid glish variable name.

Results can be copied to and from the clipboard or to the calculator. This Copy/Paste facility is activated by pressing the right mouse button when the mouse pointer is over the *Results* window. A popup menu appears with the three possible options. The current selection is copied to the chosen destination when a copy option is selected. The contents of the clipboard are pasted into the results manager when that option is selected.

Working sets, SDRECORDs, and everything else

Results are either working sets (also known as sditerators), SDRECORD, and anything else.

A working set is a collection of SDRECORDs. Specifically, it is an sditerator tool. When you open a data set through the *File* menu (either a new one or an existing one) the result is a working set which will be shown in the results manager. The other way to create a working set is to use the "Apply" button in the "Selection" operation. When you make a selection this way the result is also a working set. It does not make a copy of the selected data, rather it makes a reference to the selected data in the underlying data. This means that if you change the underlying data, the reference in the selected data is also changed. Working sets can be browsed. See the Reference Manual for more information on the Sditerator `tool`.

An SDRECORD is a *Glish* record which has the structure described below. It is data (generally spectra at this point in DISH development) plus some standard header values, any non-standard header values, and a history of what has been done to that particular SDRECORD. SDRECORDs can be browsed and plotted.

Anything may appear in the results manager. If it isn't a working set or an SDRECORD it can not be browsed or plotted.

9.3.2 Menubar and Message Line

New data files are created, existing data files are opened, the state of DISH is saved, a previously saved state is restored, and DISH is reset to its default initial state through the *File* menu. The GUI panels for each available operation are enabled and dismissed through the *Operations* menu. The *Options* menu has two options which can be turned on or off. The "Write to Script" option, when turned on, causes DISH to echo the underlying glish commands to the AIPS++ defaultscripter (ds) tool when a GUI operation occurs in DISH. This can be a way to learn more about what DISH is doing behind the scenes and write glish scripts to do operations not yet available in DISH. When the "Save when done" option is turned on, the state of DISH is saved to the current state file when the "done" function is invoked (e.g. `dish.done()`). The "done" function is always invoked whenever you exit glish. Because DISH can not be restarted after "done" has been called it is best not to do that unless you are about to exit glish. This deficiency will be corrected in the next release.

The Message Line is the text immediately below the Results Manager. These messages are also echoed to the AIPS++ logger for longer term storage.

9.3.3 Browsing

A working set within the results manager can be 'browsed' or examined by hitting the 'Browse' button. This brings up a frame which lists all of the scan numbers and objects, record by record, within the working set. Each SDRECORD

can also be browsed to examine all of the contents (e.g., all of the header information and data).

When browsing a working set, individual records can be copied to the clipboard or to the results manager. This copy facility is activated by pressing the right mouse button when the mouse pointer is over the browser window. A popup menu appears with the three possible options. The current selection is copied to the chosen destination when one of the copy options is selected.

9.3.4 Inspecting

Inspecting is another way of viewing the values of the results within the results manager. It lists all of the contents of the selected result. These may be accessed via the *Glish* command line by the names shown while inspecting.

9.3.5 Operations

Currently there are eleven operations pre-defined within Dish. These operations are general utilities available from the basic Dish Gui. All operations have two basic buttons: a green Apply button which will execute the given operation and a yellow Dismiss button which will close the operation frame but retain all information in that frame for subsequent use.

Averaging

The Averaging operation offers several options for performing the averaging (Figure 9.1). Alignment which may be either None, By Velocity, or By X-Axis, 2) Rest Frequency, which allows shifting to match the first in the average (for example, for averaging spectra from transients), 3) Weighting which may be None, Tsys and Time or by RMS, and 4) which will use any selections chosen from the Selection Operation prior to averaging.

Baselines

The Baseline operation allows either a polynomial fit or a sinusoid fit. The baseline regions may be typed into the regions entry box at the bottom of the frame, or selected via cursor (the Plot Ranges and Cursor Active buttons should both be checked). The baseline operation frame can be seen in Figure 9.2.

The default action is to calculate the baseline and show it. Once the fit seems satisfactory, the 'Subtract' button must be checked and the fit 'Apply'd. This will remove the baseline and display the resultant spectrum. The baseline fit may be viewed from the results manager. The current ranges selected will be displayed in the entry field. A history of these is preserved within a cache which can be viewed by clicking on the down-arrow button. The RMS of the fit is displayed to the right of the frame. Units for ranges or fit parameters (sinusoid only) may be converted between channels and X-Axis units using the Convert button.

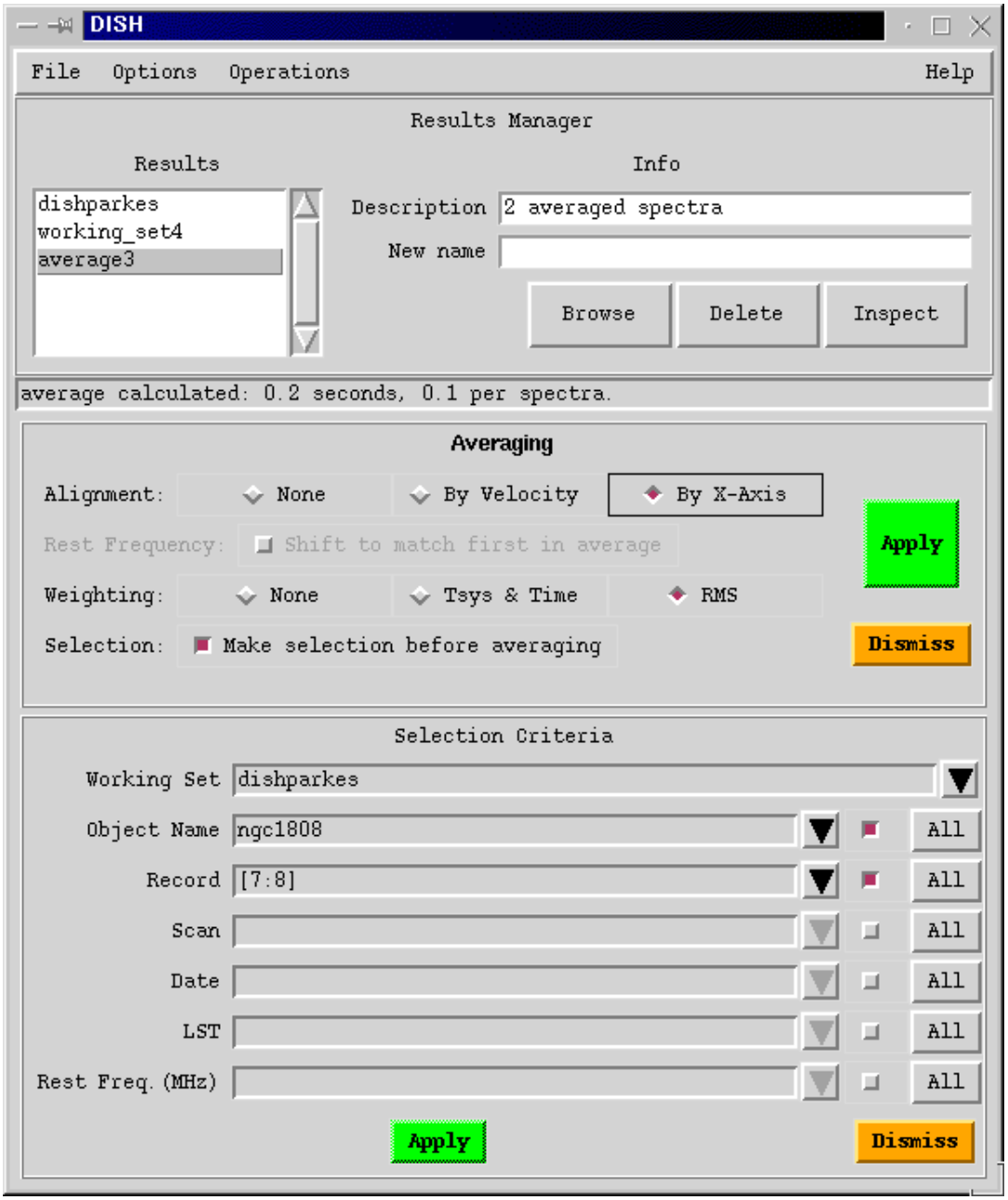


Figure 9.1: Example window showing basic Dish frame with the Averaging and Selection Operation frames open.

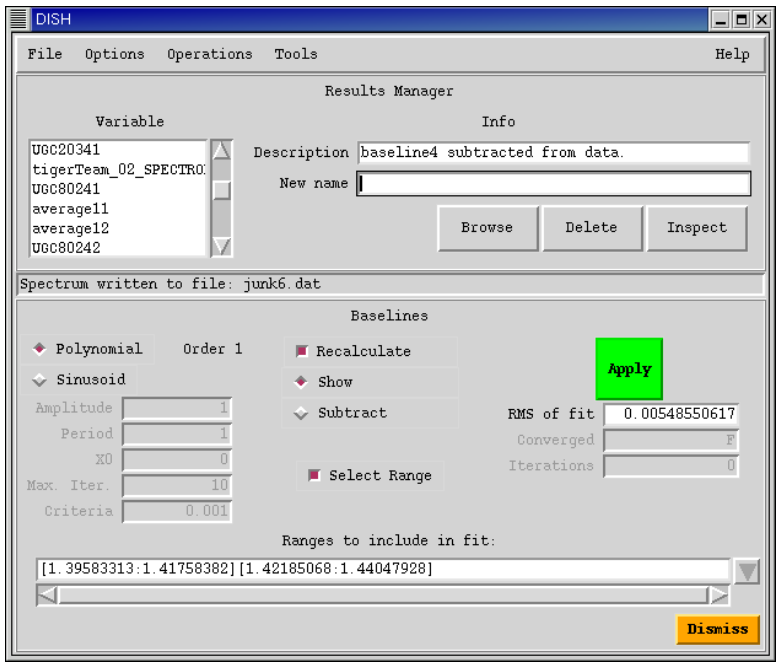


Figure 9.2: Example window showing basic Dish frame with the Baseline Operation frame open. In this example, one baseline region is selected and a 5th order polynomial will be shown overlayed on top of the spectrum.

Calculator

The Calculator operation is a general tool for manipulating spectra in ways not facilitated by the basic set of operations. Spectra may be transferred into the calculator stack through the use of the Copy/Paste to Clipboard facility.

The Copy/Paste facility is activated by the right mouse button. A popup menu appears with a two copy options and a paste option. "Copy to clibpoard" will copy whatever is selected in the Calculator stack listbox to the Clipboard; the Clipboard is simply a convenient virtual storage area. "Copy to results manager" copy whatever is selected in the Calculator stack listbox to the results manager. "Paste from clipboard" retrieves whatever was most recently sent to the clipboard. For example, in Figure 9.3, the Results Manager spectrum "average3" was copied into the Calculator stack. An "_1" designation was added to the label to distinguish it from the same value in the Results Manager. The spectrum was then multiplied by a factor, producing "average3_1_2". The additional id tag is an underscore and the stack number.

The available functions may be seen in the figure. In addition, it works as any standard Reverse Polish Notation calculator. Numbers, vectors and arrays may be typed into the entry field. The calculator will only work with SDRECORDs, numbers, vectors and arrays.

Function on Data

The Function on Data operation is another means of manipulating the displayed data. The string typed in the entry box of this operation is evaluated by glish after any of the standard macros are replaced by their equivalent glish. The result of this evaluation is then reassigned to the data array of the last viewed SDRECORD. The standard macros and their meanings are:

DATA sdrecname.data, the data record of sdrecname.

DESC sdrecname.data.desc, the data description record of sdrecname.

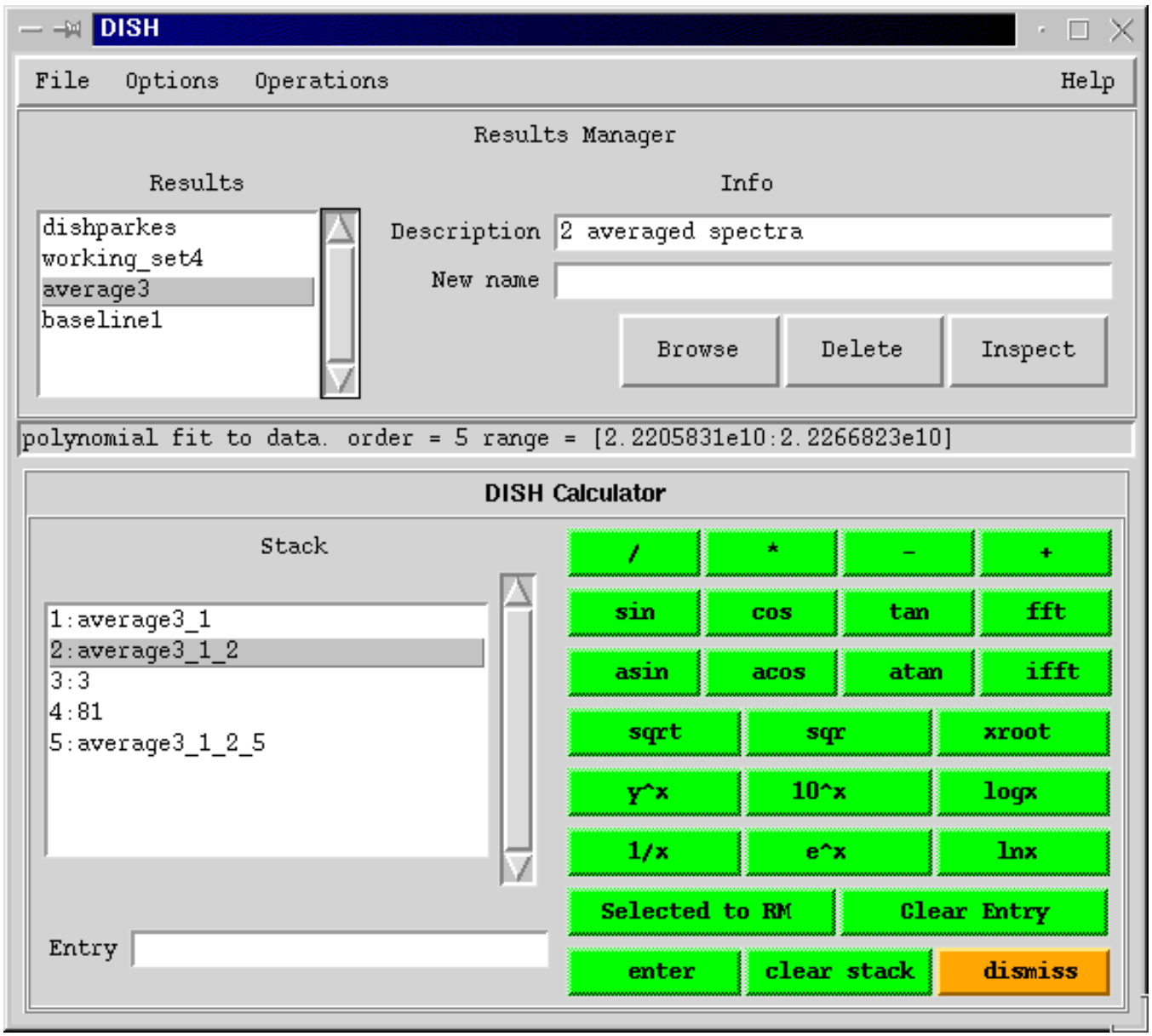


Figure 9.3: Example window showing basic Dish frame with the Calculator Operation frame open

ARR sdrecname.data.arr, the data array.

HEADER sdrecname.header, the header record.

NS_HEADER sdrecname.ns_header, the ns_header record.

In the above definitions, sdrecname is whatever the name of the last viewed data set. For these macro substitutions to work, they must be given as upper case (to distinguish them from their lower case equivalents, in which no substitution would occur).

Since whatever this text evaluates to after macro substituion is reassigned to the data array, it must have the exact same shape as the data array initial had.

Basic scaling or manipulations can be obtained by typing expressions that operate on ARR. For example, a simple scaling can be obtained by typing the function "ARR*1.5", and then pressing "Apply". Any function available from within *Glish* or known at the command line (e.g. a function defined by the user during the run of Dish), can be accessed and used in these expressions.

G Fit

The Gaussian Fitting operation is not contained under the main operation menu. Instead, it is available directly from the plotter. Pushing the 'G Fit' button calls up the interface window. The number of gaussians to fit is set by the "No. components" slider and entry field. The number can either be typed in or the slider can be adjusted to the number of gaussians desired. This number will be reflected in the number of component buttons created below. Depressing a component button makes it active, allowing setting of its properties.

Depressing the green "Estimate" button generates an auto-estimate of a component, generating an approximate flux, pos, and width (height, center, width for the gaussian). Depressing the green "Fit profile" button will generate a fit based on that guess. Interactive estimates for each component can be obtained by pressing the component button followed by the "Inter" button (below the "Estimates" title). This prompts the user to: "Put cursor at line center and click (any)" and then "Put cursor at line HWHM and click (any)". This fills the flux, pos and width fields with the appropriate guess. Continue with the next component and then hit "Fit profile" when all have been defined.

The Plot menu at the top of the frame allows the user to specify which aspects of the fit to view: Estimate, Fit, Residuals. Each of these may be toggled on-off interactively.

Re-gridding

The Re-gridding operation is a more generalized version of the Smoothing Operation as it allows both finer and coarser adjustments of the grids. Currently, it allows Hanning, Boxcar and Gaussian smoothing along with Spline and Fourier Transform Interpolations.

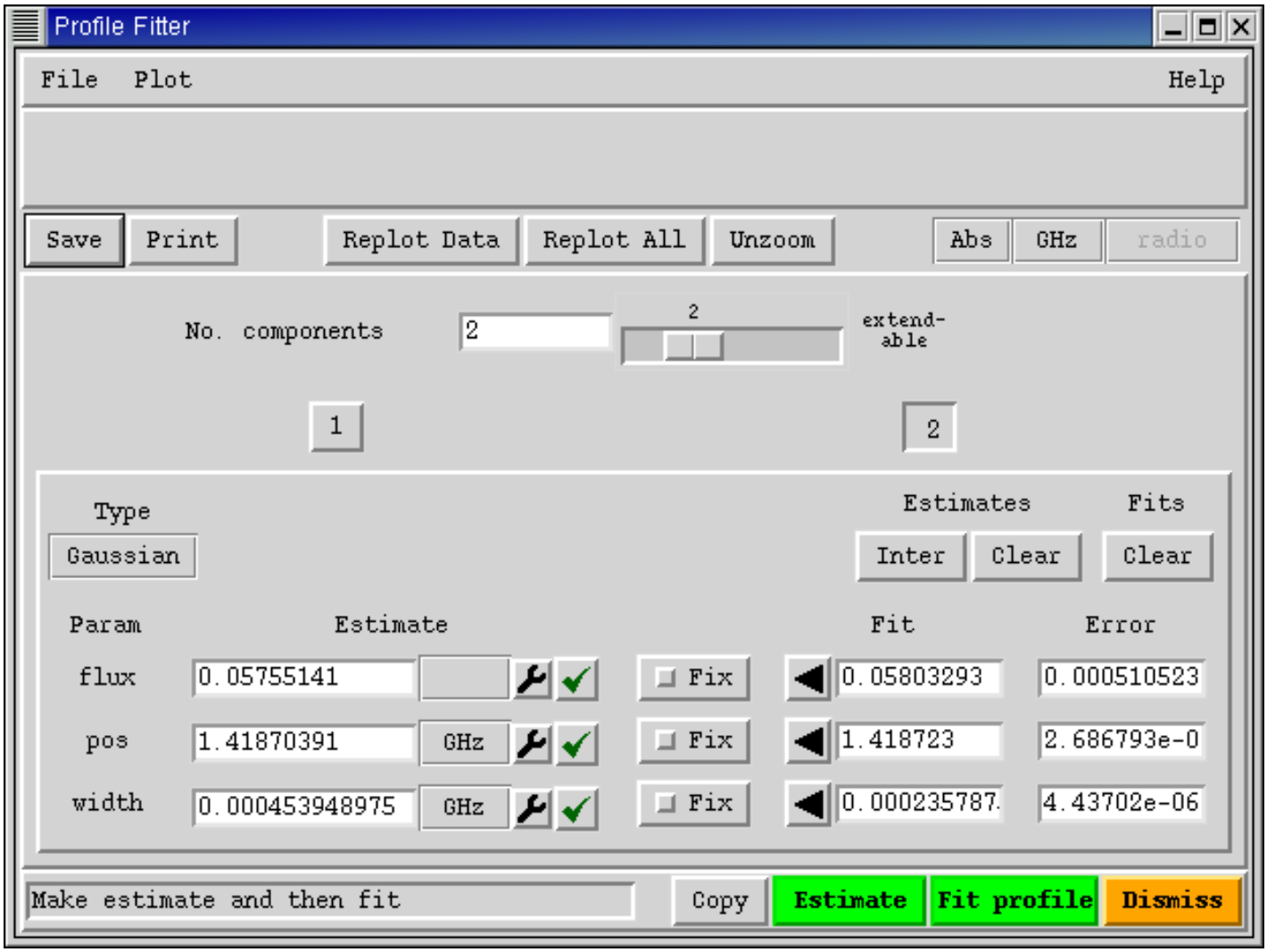


Figure 9.4: Example window showing the GUI gaussian fitting interface

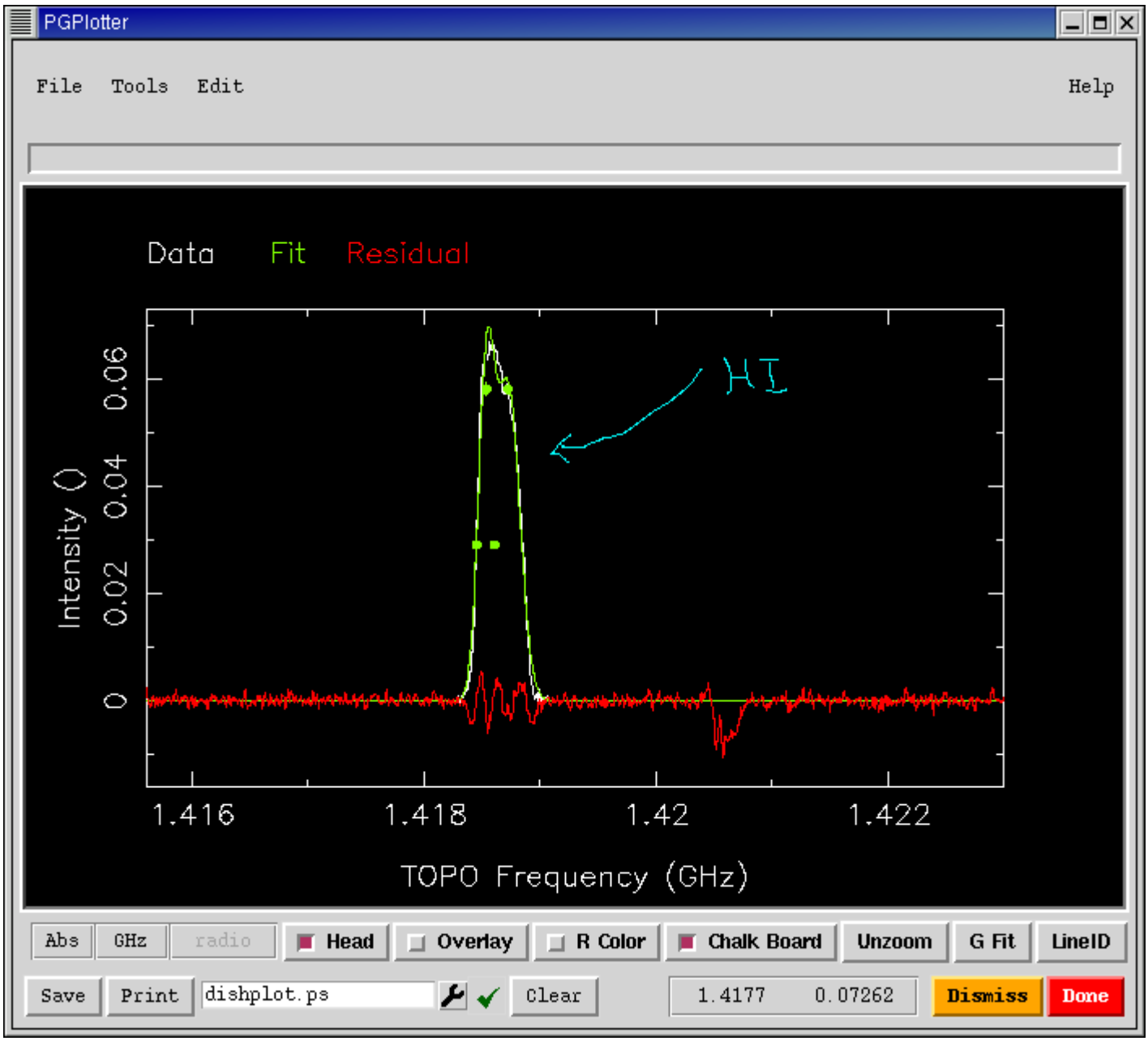


Figure 9.5: Plot showing the results of a 2-component gaussian fit to line profile. The fit and residuals only are displayed. The chalkboard feature has been used to draw a label indicating the line ("HI"). fitting interface

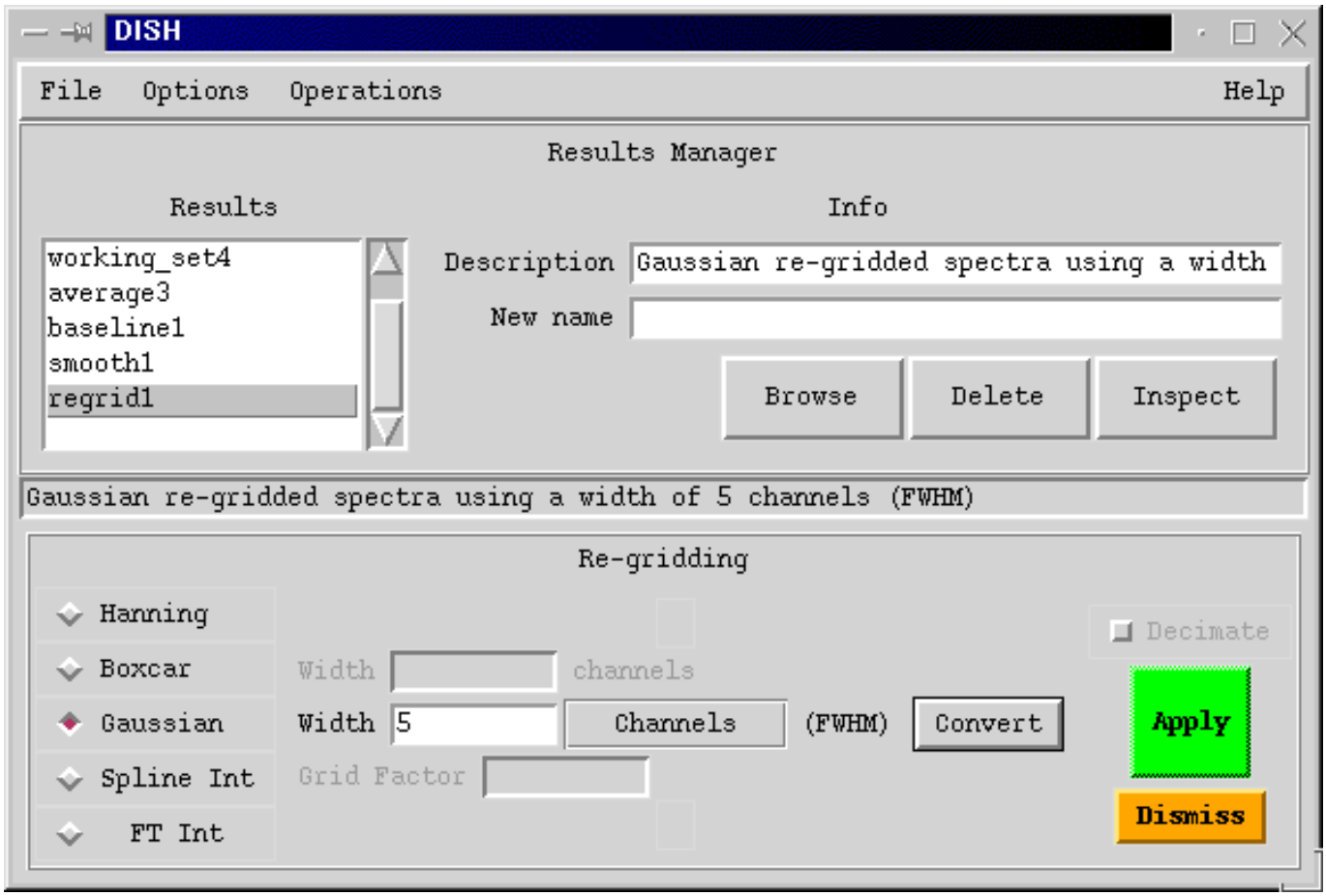


Figure 9.6: Example window showing basic Dish frame with the Regrid Operation frame open

Saving

The Saving operation allows the user to store and organize spectra into named working sets. The working set must first be defined through the File Menu option "Open" (selecting on Read and Write). Once the working set is defined, any displayed spectrum may be written to the working set on disk.

Selection

The Selection operation allows a number of selection criteria to be chosen to create new working sets which are subsets of existing working sets. The recognized terms are: Object Name, Record, Scan, Date, LST, and Rest Frequency. Typing into any of the entry fields is allowed but the selection will only be used if the check box on the right is marked. A history of selected ranges or names will be preserved. For numerical selections, ranges may be made as follows:

```
Record,Scan Entry:
8,9,10 # Select scans 8, 9 and 10
8,[10:13],17 # Select scans 8, 10, 11, 12, 13, and 17
```

The 'All' button will select all relevant values for the given selection criteria.

NOTE: Selection on MSv2 is limited to just row-based selection for AIPS++ V1.4.

Smoothing

The Smoothing operation is a more specific tool than the Re-gridding operation. It offers easy Hanning, Boxcar and Gaussian smoothing.

Statistics

The Statistics operation calculates basic statistics on an interval of the spectrum. The interval must be defined by the cursor (activate the Plot Range and Cursor Active buttons) and then left click on the left and right regions of the spectrum which are of interest. The region will appear in the Range entry field and the Start and Stop fields will be populated by the beginning and end of the range; NOTE: currently, only the first range in the rangebox will be used for the calculation of statistics. If you want changes to the start and stop range for your interval, those changes should be edited or entered at the start/stop entries. The units may be converted between X-Axis units and channels through the convert button (similar to the Baseline operation capability). The Apply button will calculate the Peak value, the location of the peak, the Area over the region, the location of the centroid (half area), the rms, and the minimum. These values may be written as a record to the clipboard through the "Copy to CB" button. The record looks as follows:

```
- statrec:=dcb.paste()
- statrec
```

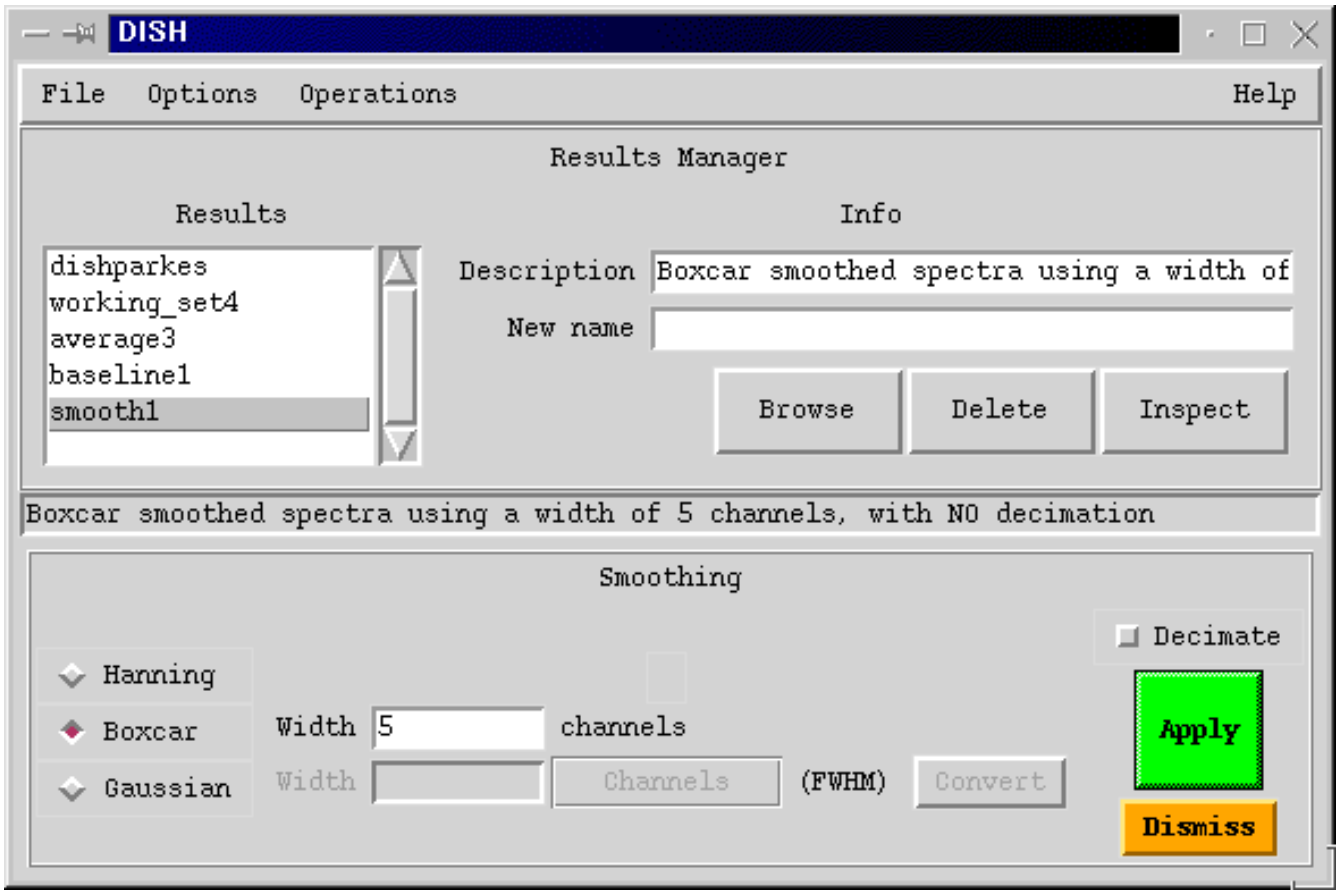


Figure 9.7: Example window showing basic Dish frame with the Smoothing Operation frame open

```
[peak=2.09596825, area=29425582.1, min=-0.26479581, rms=0.580018938, scan=20, ce
ntroid=2.22369707e+10, vpeak=2.22353317e+10]
-
```

In addition, the data may be written to a file through the "Print to File" button. This pulls up a frame which prompts the user for a filename. If the file exists, it will append the values to the end of the file. The file looks as follows:

```
  scan  start          stop           atPeak         Centroid
    20  2.2225160e+10  2.2245390e+10  2.2235332e+10  2.2236971e+10
        Peak           Area           Mininum        rms
        2.0959682e+00  2.9425582e+07 -2.6479581e-01  5.8001894e-01
```

Write to File

This operation allows convenient output to a disk file in the current directory. It automatically defaults to the file name of "scan_number.spc"; for example if you Write to File while looking at scan 20, the file will be "20.spc". The file looks as follows:

```
     FREQ              other axis
       Hz              other unit
22203023360.000000    -3.907563
22203086216.617188    -4.986684
22203149073.234375    -4.020397
22203211929.851562    -4.558375
22203274786.468750    -4.556173
22203337643.085938    -3.856671
22203400499.703125    -4.001695
22203463356.320312    -3.464213
22203526212.937500    -4.438780
22203589069.554688    -3.296077
...etc.
```

9.4 Saving/Restoring State

If you do nothing, a file called *default* will be created within the `$HOME/aips++/dishstate` directory upon exiting from a DISH session. The file is created automatically and dish will be restored to the state saved in that file automatically upon startup. The dish state file contains all of the values known to the results manager as well as the various settings for each GUI element and operation. So, by default a record is kept of the state of DISH after each use of DISH and DISH is restored to this state the next time DISH is used.

There are several ways in which you can alter this default behavior.

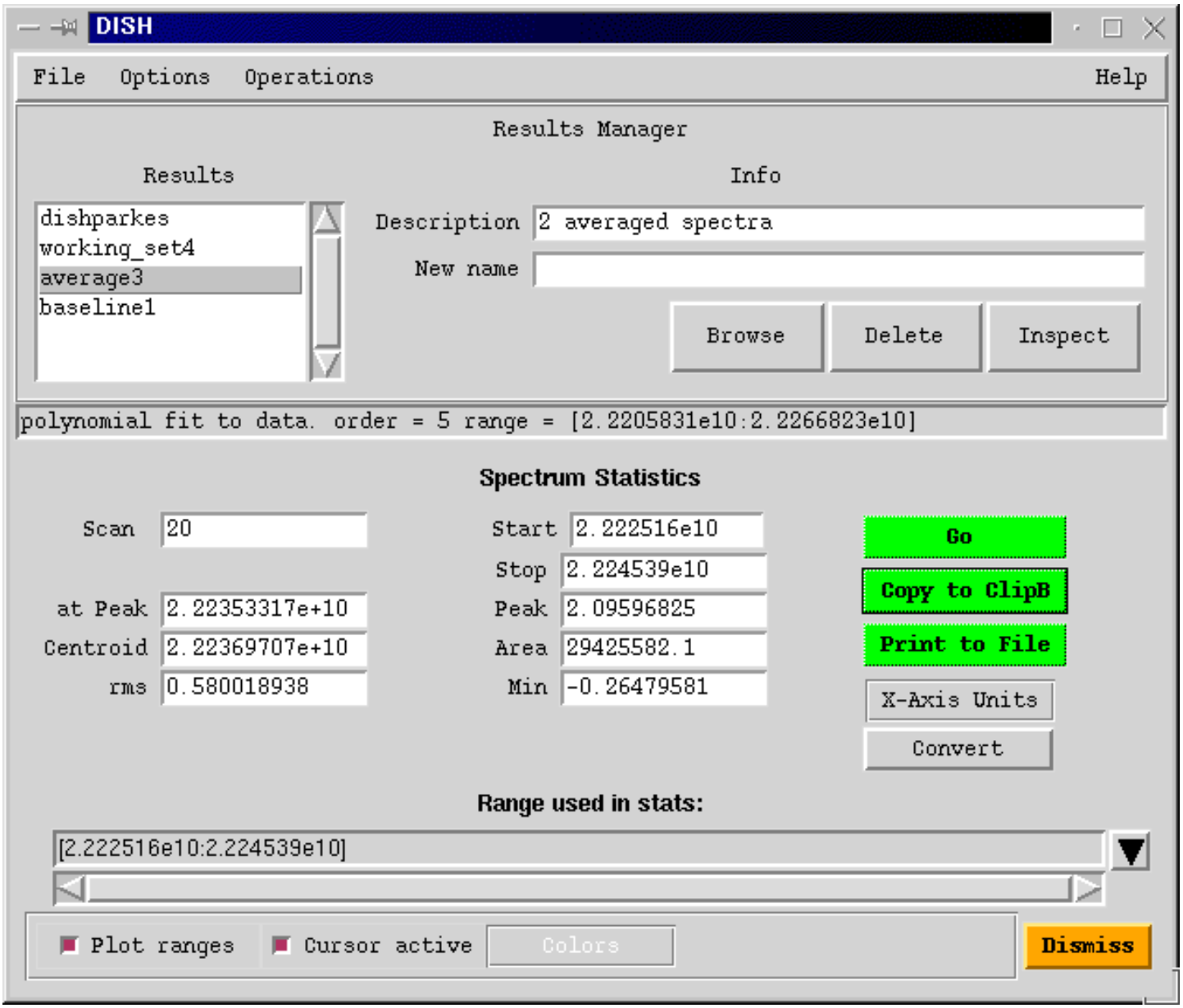


Figure 9.8: Example window showing basic Dish frame with the interval Statistics Operation frame open

- The state can be saved at any time by selecting the *File/Save state* menu item.
- A state can be restored to a previously restored state at any time by selecting the *File/Restore state* menu item.
- The state can be saved to a state file other than the default file at any time by selecting the *File/Save state as ...* menu item. This makes the AIPS++ catalog tool visible and allows you type in or select a file name to hold the state information. Once you have made your selection and the state saved to that file, that file becomes the default state file until changed with *Save state as ...* or *Restore state from ....*
- The state can be restored from a state file other than the default file at any time by selecting the *File/Restore state from ...* menu item. This makes the AIPS++ catalog tool visible and allows you to select a file name to retrieve the state information from. Once you have made your selection and the state retrieved from that file, that file becomes the default state file until changed with *Save state as ...* or *Restore state from ....*
- Restore the state to the default initial DISH state by selecting the *Reset to default state* menu item.
- The default behavior of saving the state when `dish.done()` is invoked (which happens which you exit glish) can be turned off by toggling that option in the *Options* menu.
- The default behavior of restoring from the default state file when DISH is first started can be turned off by setting the aipsrc variable "dish.restorestate" to F (for False). To turn it back on, set this variable to (T) or remove it from your `.aipsrc` file.
- The default state file can be set before DISH starts by setting the aisprc variable dish.statefile to whatever file you wish to designate as the default state file. This is always the state file used when DISH starts up. Its default value is `$HOME/aips++/dishstate/default`

9.5 The Dish Plotter

The Dish Plotter is the familiar AIPS++ pgplotter tool, modified for additional utilities and capabilities. Any activity or task within Dish which produces a plottable result, will automatically have the result displayed in the Dish Plotter; for example, selecting entries within the Results Manger, the SDRecord Browser, or Calculator Operation stack will plot the selection chosen. The plotter features tools for 1) changing the X-axis units, 2) changing the reference frame for the X-axis, 3) toggling header information display, 4) toggling the overlay of subsequent plots, 5) options for a) altering the colors of the plot

(reverse color) and b) allowing a draw facility (chalkboard), 6) toggling the X-axis autoscale, 7) unzoom (incrementally unzooms any zooms), 7) activate the gaussian fit tool, and 8) line identification (marks any molecular lines within the frequency ranges of the displayed plot. The cursor position is continuously displayed in the grooved box at the bottom of the plot whenever the cursor is within the DISH plotter frame. The plotter is arrayed as shown in Figure 9.9.

File Menu This is the standard pgplotter File menu. Open...: This operation will read a plot file on disk written by the Save command. Save: This operation will write a vector plot file of the displayed image to disk. Save will write to the file specified in Feature L (defaults to dish.plot). Print: This operation will print the current spectrum. Dismiss: This closes the plotter window while Done destroys the tool (not recommended!).

Tools Menu This provides a zoom tool and colormap tool (not relevant for single dish case).

Edit Menu This brings up a frame which allows interactive editing (changing of existing commands) or addition of new commands. For example, additional labeling or fiducial lines may be easily added. The frame lists all of the available PGPLOT commands. Selecting one brings up a brief explanation along with the various arguments required. The frame also lists the PGPLOT commands which produced the displayed frame. Again selecting a command brings up a small explanation along with the arguments used in its execution. Cut, Copy, and Paste features are available to duplicate and add additional commands. An example frame is shown in Figure 9.10.

Help Menu Standard pgplotter help menu. The options are: 1) PGPlotter: will drive your Netscape browser to the help section on the PGPlotter. 2) Reference Manual: will drive the browser to the Reference Manual main page. 3) About Aips++...: provides basic information about the AIPS++ version being used.

Plotfile name This is the filename that the Save option in the File menu will save to by default. The default plotfile name is dish.plot.

Clear Plotter Clears all currently displayed plots. Resets most plot parameters to their defaults. The color of displayed data resets to red, but line styles and plot styles previously selected will be retained.

Cursor Position Whenever the cursor is within the Dish Plotter frame, the cursor positions will be continuously read back. The coordinates shown are the X and Y position as defined by the X and Y axes' units.

Dismiss Plotter Dismisses or closes the Dish Plotter frame. The frame will be automatically re-opened if an action occurs which produces a plottable result, i.e., if a spectrum is selected, a new Dish Plotter frame will be initialized to display the selection.

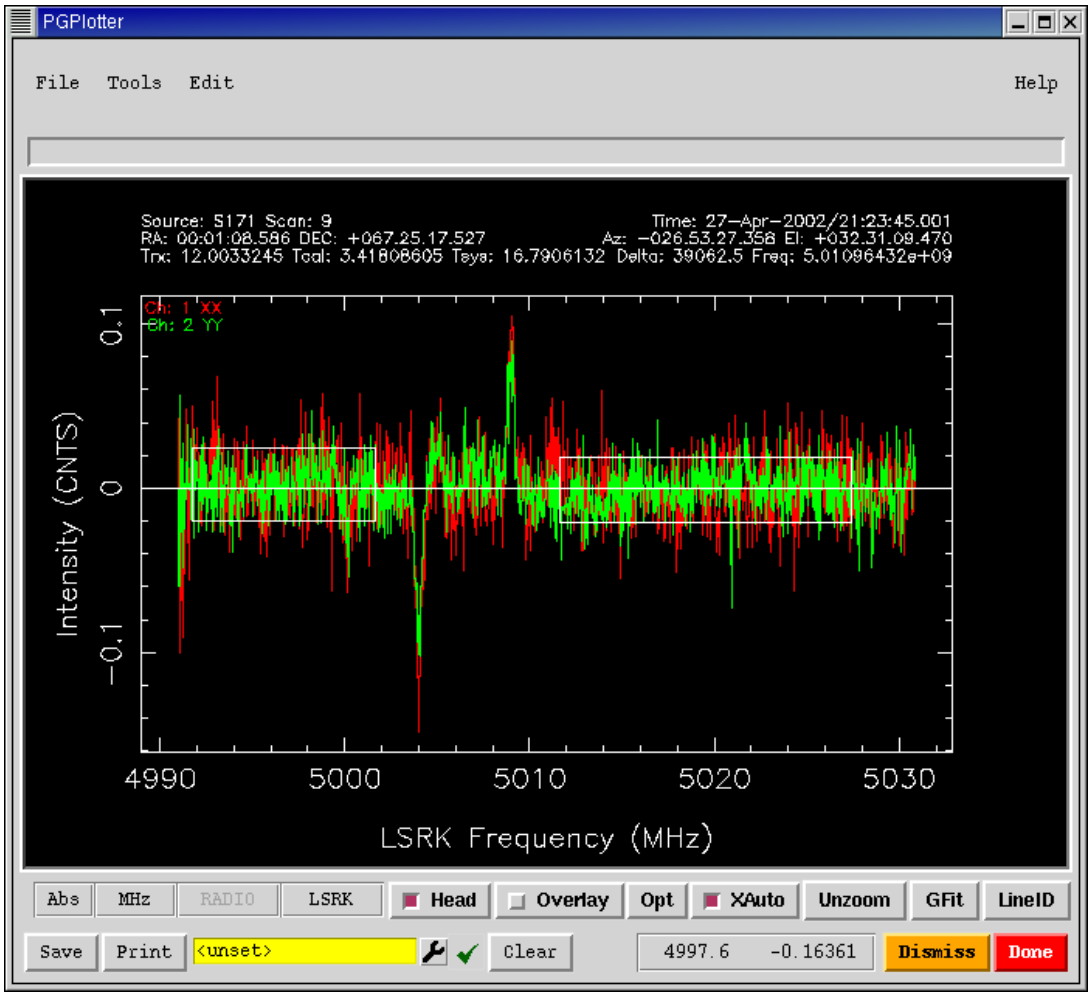


Figure 9.9: Dish Plotter Appearance. Displayed is a baselined GBT frequency switched spectrum. The baseline regions are marked by the white boxes whose height indicates the rms in that region. The different lines are different polarizations as labelled in the upper left.

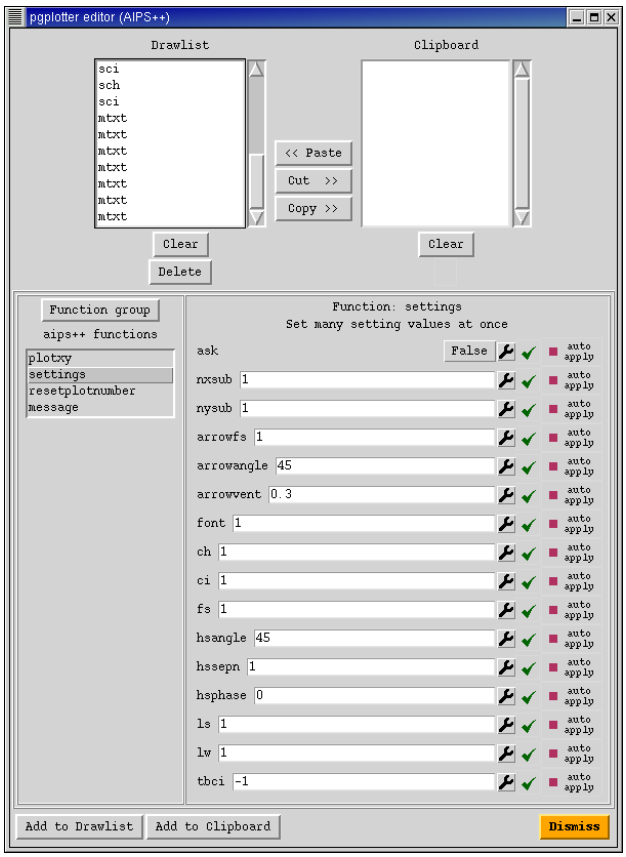


Figure 9.10: Add commands frame for Dish Plotter.

Plot Axes Menu This menu allows selection of the units for the abcissa. The first selection determines whether the units will be absolute ("Abs") or relative ("Rel"). The second selection determines the actual abcissa unit ("GHz, MHz, KHz, Hz, km/s, m/s, or pix"). The third selection determines, for velocity units, whether the radio or optical definition should be used.

Header toggle This determines whether the header information is displayed. If the header button is activated, the scan information listing the source, scan number, Time of observation, RA/DEC, AZ/EL, Trx, Tcal, Tsys, $\Delta\nu$ and Frequency are displayed. If the button is disabled, it only displays the source name.

Overlay Toggle The overlays menu is a simple toggle switch between On and Off. In the Off mode, all new plot results or selections will clear the screen before plotting the new result. The only exceptions to this are the Baseline operation in Show mode, which will overplot the calculated baseline over the spectrum, and multi-channel data which will plot as many channels as found on the same screen. In the On mode, all new selections and results will be overplotted; the axes will not be scaled!

Reverse Color Toggle This reverses the basic foreground and background colors for better displays under different illumination conditions (e.g. projection favors the reversed color display).

Chalk Board Toggle This enables the third mouse button to be used as a piece of chalk on the current display for any annotations, doodles, etc.

X-Axis Autoscale Toggle This is set by default but it can be toggled off to 'keep' the current x-axis range.

Unzoom Allows incremental unzooms from any zooms performed. Zooming is done with the middle mouse button; unzooms may also be done with CNTRL-Middle Mouse Button.

G Fit Button This brings up the Gaussian Fitting interface frame, and initializes the current display for fitting gaussians. NOTE: This button must be depressed each time a gaussian fit is desired.

LineID This tool browses an aips++ table version of the Poynter and Pickett molecular line catalog, and plots any lines found that fall within the frequency range of the displayed plot. Line positions are plotted with short tick marks along the top of the plot while the corresponding molecular name tag is plotted vertically along the bottom.

9.6 The Dish Command Line Interface

The DISH CLI is detailed in the User Reference Manual.

9.7 Recipes

9.7.1 Recipe 1: Change syntax to suit

Finally, any syntax can be roughly approximated by a combination of Glish and the existing tools in DISH. You can always set a new variable to equal an existing function. In this way you can modify and create your own command line reduction language. An example of changing the syntax to approximate a UniPops command would be:

```
UniPops: hanning xx             #this will hanning smooth a spectrum and
                                #redisplay it

- hanning:=function() { d.smooth('HANNING');}
                                #create a new function that automatically does
                                #a hanning smooth; CLI defaults to operate on
                                #the currently displayed spectrum.
- hanning()                     #this now does the equivalent of the above
                                #UniPops command
```

9.7.2 Recipe 3: Add a function to DISH (or fun with extensibility)

Plugins are another standard means of adding functionality to DISH. Any file called mydish_*.gp that lives in the current directory will be picked up when DISH is started. An example plugin file with a simple squaring function is shown here:

```
pragma include once;
note('mydish_example included');
mydish_example := [=];
mydish_example.attach := function (ref public) {
        #new function
#####User adds functions here
        public.square := function(somevalue) {
                wider public;
                print 'the square of ',somevalue,' is: ',somevalue*somevalue;
return T;
        }
# other functions go here
#       public.otherfunction := function(input1,input2,etc) {
#               some code goes here to do something
#               return; # can return a value too if you like
#       }
#       public.etc...
#
#####End user edit
```

```
return T;
}
```

If you then look at the functions contained in your DISH tool, d, it will now have a 'square' function.

Running this operation within dish looks like the following:

```
- field_names(d)
done dismiss gui rm ops addop normalcursor busycursor savestate restorestate debug oper
- d.square(4)
the square of 4 is: 16
T
```

9.8 sditerator

See the Reference Manual for information on the Sditerator `tool` and the `is_sditerator` function.

9.9 SDFITS

SDFITS is a convention for the use FITS binary tables to store single dish data. This appendix will eventually be turned into a full fledged note. At the present time, it is simply a summary of the various rules regarding the interpretation of keywords and columns which, taken together, constute the Single Dish FITS convention.

A FITS binary table is also a SDFITS binary table if it follows these rules.

9.9.1 EXTNAME keyword

```
EXTNAME = 'SINGLE DISH'
```

The EXTNAME keyword must have a value of "SINGLE DISH". A table having this value for EXTNAME is assumed to be fully compliant with this convention.

9.9.2 Virtual columns

Most keywords should be thought of as a column of the same name which has a constant value for all rows in the table.

There are a few exceptions to this rule. FITS keywords which are required to describe the structure of the table are not virtual columns. These include NAXIS, NAXIS1, NAXIS2, BITPIX, XTENSION, EXTNAME, EXTVER, EXTLEVEL, PCOUNT, GOUNT, TFIELDS, TFORM, TTYPE, TUNIT, TNULL, and TDISP. The TDIMnnn keyword is the only table description keyword which may be a column as described elsewhere in this appendix.

DATAMAX and DATAMIN are special keywords. When used as keywords it is most likely that the writer of that table meant them to reflect the range of

all of the data found in that table and not to be used on a row-by-row sense. If the writer feels a need to supply row-by-row values for these two keywords, they should explicitly make them true columns.

When creating a table, a column may be made virtual if it's value can be represented as a single keyword (if it is constant over all rows) and if that constant value is NOT an IEEE special value (it has a standard ASCII representation) and the name of the column is 8 character or less.

If a column exists which has the same name as a keyword, the column values should be assumed to take precedence (writers should attempt to ensure that this confusing situation does not happen).

9.9.3 The DATA column and the DATA axes

The DATA column containsthe primary data array. It is required. Each row of this column contains an n-dimensional array of data. The axis descriptions are handled in a similar way to FITS images.

The TDIMnnn convention This is how the shape of the DATA array should be specified.

`TDIMnnn = '(n1,n2,n3,n4...)'`

- The product of all of the n1, n2, n3, etc., must be $\leq$ the width of the column as given by the TFORMnnn keyword.
- TDIMnnn may either be a keyword (constant shaped columns) or it may itself be a column (variable shaped columns).
- The data may be stored directly - unused elements in a particular row should contain the appropriate value to indicate an undefined value as described in the Binary Table Extension paper (i.e. NaN or use TNULL), Or the DATA column may use the P type of storage (storage on "the heap").

This convention may be used with other columns which do NOT use the same axis description (these will be site specific columns and readers are free to ignore these columns).

Axis description This description applies to any column which has `TMATXnnn = T`. It is assumed that `TMATXnnn = T` for the DATA column even when not present. The use of `TMATXnnn = F` for the DATA column is an error.

Axes descriptor keywords/columns include (not all need be present for each axis):

CTYPEn The type of physical coordinate on axis n.

CRVALn The value of the physical coordinate on axis n at the reference pixel.

CRPIXn The array location of the reference pixel along axis n. CRPIX may be a fractional pixel and/or be outside of the limits of the array. This descriptor is optional for degenerate axes.

CDELTn The increment in physical coordinates along axis n. This descriptor is optional for degenerate axes.

CROTAn The axis rotation

CUNITn The unit of coordinate values along this axis.

Note that these are the original FITS axes description keywords. It is anticipated that when an agreement is reached on the WCS convention that the appropriate keywords specified in that convention will be used to describe the axes here (e.g. CROTA is deprecated in favor of a more general mechanism).

Due to the virtual column convention, the following conflict may arrise:

```
TTYPE10 = 'CDELT4'
TFORM1  = '1E'
TUNIT10 = 'radian'
CUNIT4  = 'degree'
```

Writers should avoid this conflict. Readers encountering such a conflict should report this as an error in the FITS table. However, readers may also choose to proceed by arbitrarily resolving the conflict so that some attempt may be made within the anaysis package to deal with this error. It is recommended that the CUNITxxx specification take precedence over any TUNITxxx specification. In any case, this conflict and the readers resolution of the conflict (if at all) should be reported to the user.

The other standard FITS Image axis keywords may be used here. This will include all of the WCS convention keywords when they become accepted as part of the FITS standard.

Defined axis types (CTYPEnnn values) for this convention (others are allowed but their values may be ignored by readers) are:

frequency-like An 8 character string consisting of the 4 character axis type plus 4 characters describing the reference frame. An axis having this type is required.

axis types
- 'FREQ' - frequency (Hz)
- 'VELO' - velocity (m/s) (radio convention, unless overridden by use of the VELDEF SHARED keyword)
- 'FELO' - regularly gridded in frequency but expressed as velocity in the optical convention (m/s)

reference frames
- '-LSR' : Local Standard of Rest
- '-HEL' : heliocentric (barycentric)

- '-OBS' : the frame of rest of the observer/telescope (topocentric)
- 'LSRK' : LSR as a kinematical definition
- '-GEO' : Geocentric
- 'REST' : rest frequency
- '-GAL' : Galactocentric

Longitude-like 'RA','GLON','ELON' plus an optional WCS projection code (degrees). This axis is required.

Latitude-like 'DEC','GLAT,'ELAT' plus an optional WCS projection code (degrees). This axis is required.

'TIME' The time since DATE-OBS (seconds) If the TIMESYS keyword is present, that keyword defines the time system for this table, including this column, otherwise UTC is assumed. This axis is optional. If not present, a value of 0.0 should be assumed. This axis will often be absent if DATE-OBS contains a time as well as a date.

'STOKES' The Stokes parameter of the data.

- $1, 2, 3, 4 \Rightarrow$ I,Q,U,V
- $-1, -2, -3, -4 \Rightarrow$ RR, LL, RL, LR
- $-5, -6, -7, -8 \Rightarrow$ XX, YY, XY, YX

This axis is optional. If not present, a value of 1 (Stokes I) should be assumed.

'BEAM' Beam ID. This axis is optional. If not present, a value of 1 should be assumed.

'RECEIVER' receiver ID. This axis is optional. If not present, a value of 1 should be assumed.

9.9.4 CORE keywords and columns

These must be provided in all SDFITS tables. They are essential and common to all observations and telescopes. All single dish FITS readers and writers must acknowledge (write and properly interpret) all CORE keywords.

OBJECT A string value giving an object name.

TELESCOP A string value giving the telescope name.

BANDWID The total bandwidth of the backend in units of Hertz.

DATE-OBS A string giving the observation date and optionally the time at the start using the new FITS y2k convention. The TIMESYS keyword may be used to indicate the time system. UTC is assumed if TIMESYS is absent.

EXPOSURE The effective integration time in seconds.

TSYS The system temperature in Kelvin.

9.9.5 SHARED keywords and columns

These have agreeded definitions and interpretions however their presense is optional. These are largely common to all observations and telescopes but not essential. These may be ignored by a single dish FITS reader.

OBSERVER A string giving the observer's name.

OBSID A string describing the observation.

PROJID A string describing the project.

SCAN A scan ID number. Typically this is an identification number given to a chunk of data when the data is taken. Not all telescopes provide a scan ID number.

OBSMODE The type of data and observing mode (8 characters total). The type (LINE, CONT, PULS, etc) + the mode (PSSW, FQSW, BMSQ, PLSQ, LDSW, TLPW, etc). These rules do NOT define these observing modes. Writers are strongly encouraged to use the FITS comments to document these modes.

MOLECULE A string used as a line identifier (with TRANSITI).

TRANSITI A string used as a line identifier (with MOLECULE.)

TEMPSCAL A string describing the scaling applied to reach the output intensity scale ("TB","TA","TA*","TR","TR*").

FRONTEND A string giving the name of the front end device.

BACKEND A string giving the name of the back end device.

TCAL The calibration temp (K).

THOT The hot load temp (K).

TCOLD The cold load temp (K).

TRX The receiver temp (K).

FREQRES The frequency resolution in Hz. This may differ from the channel spacing.

TIMESYS The time system which applies to all time columns and keywords (see the y2k FITS DATE agreement).

VELDEF The velocity definition and frame (8 characters). The first 4 characters describe the velicity definition. Possible definitions include:

RADI radio

OPTI optical

RELA relativistic

The second 4 characters describe the reference frame (e.g. "-LSR", "-HEL", "-OBS"). If the frequency-like axis gives a frame, then the frame in VELDEF only applies to any velocities given as columns or keywords (virtual columns).

VFRAME The radial velocity of the reference frame wrt the observer. $V_frame - V_telescope$.

RVSYS The radial velocity, $V_source - V_telescope$.

OBSFREQ The observed frequency (Hz) at the reference pixel of the frequency-like axis.

IMAGFREQ The image sideband freq (Hz) corresp. to OBSFREQ.

LST The LST (seconds) at the start of scan.

AZIMUTH The azimuth at TIME (deg) (if the TIME axis is non-degenerate, then this is the azimuth at the TIME of the first pixel on the TIME axis.

ELEVATIO The elevation at TIME (deg) (same caveat as for AZIMUTH)

TAU The opacity at OBSFREQ.

TAUIMAGE The opacity at IMAGFREQ.

TAUZENIT The opacity per unit air mass.

HUMIDITY The relative humidity (fraction, 0..1).

TAMBIENT The ambient temp (K).

PRESSURE The atmospheric pressure (mm Hg).

DEWPOINT The dew point (K).

WINDSPEE The wind speed (m/s).

WINDDIRE The wind direction (deg. west of north).

BEAMEFF The main-beam efficiency.

APEREFF The antenna aperture efficiency.

ETAL The rear spillover and scattering efficiency.

ETAFSS The forward spillover and scattering efficiency.

ANTGAIN K per Jy.

BMAJ The major main-beam FWHM (deg).

BMIN The minor main-beam FWHM (deg).

BPA The beam position angle (degrees east of north).

SITELONG The site longitude (deg).

SITELAT The site latitude (deg).

SITEELEV The site elevation (m).

RESTFREQ The rest frequency (Hz).

9.9.6 Other columns

Any additional columns not explicitly mentioned here can be added to an SDFITS table, although, obviously, most readers will not be able to propertly interpret those columns.

9.9.7 Multiple SDFITS tables in a single file

Any number of FITS binary tables can be attached to the same FITS table. This convention doesn't have anything to say about how the tables in such a file might be related. This strategy, attaching multiple SDFITS binary tables to a single FITS file, is one possible strategy for storing variable length DATA. DATA with similar sizes could be stored in separate tables, minimizing the amount of padding required in any one table without resorting to using the table heap convention to store variable length arrays. SDFITS readers should be capable of appending the contents of several SDFITS tables to a single result.

Chapter 10

VLA reduction

Contributors: George Moellenbrock, Debra Shepherd,
Athol Kemball, & the NRAO AIPS++ user group
Version: 2002 June

10.1 Introduction

This document describes how to calibrate and image VLA data using the Astronomical Information Processing System (AIPS++). AIPS++ is a versatile suite of $C++$ software tools that can be run via mouse-based Graphical User Interfaces (GUIs) or an interactive command language called *Glish*.

For fairly standard VLA continuum and spectral-line observations (including polarimetry), this document can stand on its own. Additional AIPS++ documentation consists of:

- *Getting Started* with AIPS++ – an introduction to the AIPS++ environment.
- *Getting Results* with AIPS++ – three volumes are available on:
  - *Basic Tools* (e.g. plotting, data display, image analysis).
  - *Generic Processing* (e.g. data calibration & imaging)
  - *Telescope Specific Processing* (e.g. The Very Large Array, the Green Bank Telescope, and other facilities).
- *User Reference Manual* – complete and detailed software documentation.

For complicated images or unconventional observing strategies, it may be necessary to consult the more detailed AIPS++ documentation listed above. AIPS-aips++ and MIRIAD-aips++ dictionaries are also available which provide mapping of common AIPS and MIRIAD tasks/adverbs to their counterparts in AIPS++.

The data reduction categories covered in this document are those for continuum data, continuum polarimetry, spectral-line data, and spectral-line polarimetry. Individual steps in each data reduction sequence are illustrated using *Glish* script excerpts. These data reduction steps can equivalently be executed from the GUI interface.

The reduction sequences given below use the following AIPS++ `tools`:

vlafiller is used to load VLA archive *uv* data into an AIPS++ format called a Measurement Set (MS). The MS is stored on disk as a directory with a "table" name, e.g. 3C273.ms.

ms is used to access uv-data in a Measurement Set, and list and summarize visibility data.

flagger is used to select and flag *uv*-data in a Measurement Set.

msplot is used for plotting and interactively editing visibility data. It is also useful to investigate and edit the data during calibration cycles.

Calibrater is used to solve for uv-plane calibration effects, apply calibration corrections to the observed data, and plot calibration solutions.

Imager is used to image (e.g. Fourier Transform, deconvolve, restore) the data in the Measurement Set.

Viewer is used to display images, obtain statistics, create CLEAN regions or masks, and create publication quality postscript images.

Image is used for more complex image analysis, e.g. fitting 1–D profiles or 2–D models to images, creating moment maps, regridding images.

10.2 MeasurementSets

Visibility data in AIPS++ are stored in an AIPS++ table known as a MeasurementSet (MS). An MS consists of a main table containing the visibility data and associated sub-tables containing auxiliary or secondary information. The full MS data format is described in AIPS++ Note 229. Tables 1 & 2 identify the commonly accessed components of the MAIN table and list data selection parameters used during typical data reduction sessions.

All AIPS++ data files, including MeasurementSets, are written into the current working directory by default, with each AIPS++ Table represented as a separate sub-directory. MS names therefore need only comply with UNIX file or directory naming conventions, and can be referred to from within AIPS++ directly, or via full pathnames.

Each row of the DATA column in the MS contains a matrix of observed complex visibilities at a single time stamp, for a single baseline in a single spectral window. The shape of the data matrix is given by the number of channels

and the number of correlations (voltage-products) formed by the backend instrument of the telescope (e.g. a correlator for an array, or a spectrometer for a single dish).

Table 10.1: Commonly accessed MAIN Table components (Data, Flags, & Weights *only*)†

Name	Format††	Comments
DATA	Complex(N_c, N_f)	Complex visibility matrix Observed DATA for synthesis arrays
CORRECTED_DATA	Complex(N_c, N_f)	Corrected data created by calibrater or imager tools
MODEL_DATA †††	Complex(N_c, N_f)	Model data created by calibrater or imager tools
WEIGHT	Float(N_c)	Weight for whole data matrix
IMAGING_WEIGHT	Float(N_c)	Imaging weight, e.g. uniform, natural, robust Created by calibrater or imager tools
FLAG	Bool(N_c, N_f^*)	Cumulative data flags

† Additional data attributes are described in AIPS++ Note 229 †† N_c= number of correlators, N_f= number of frequency channels.
† † † The MODEL_DATA column is created with each value having AMP = 1 & phase = 0°.

Table 10.2: Common Data Selection Parameters

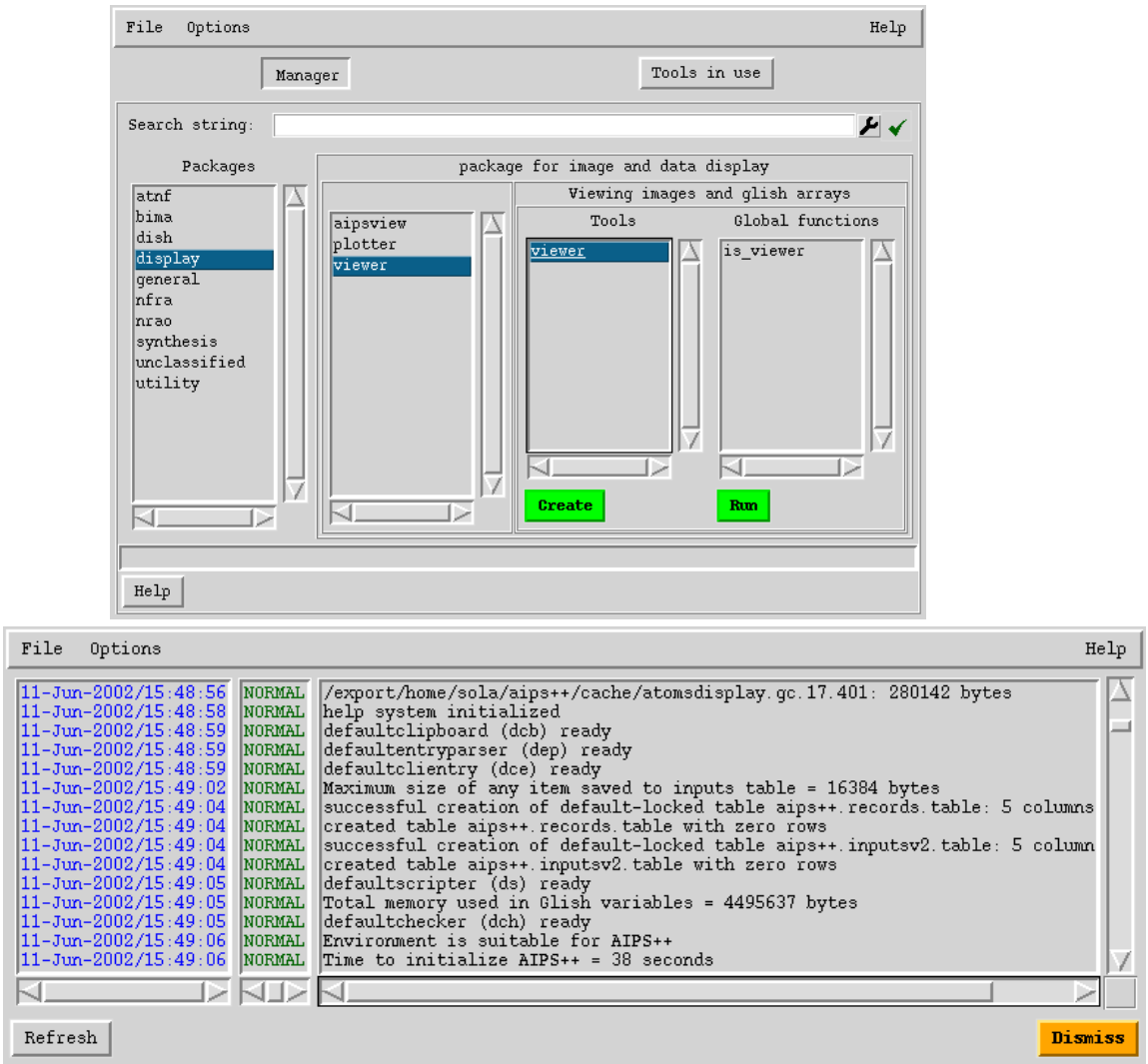
Parameter	Contents
ANTENNA1	First antenna
ANTENNA2	Second antenna
FIELD_ID	Field (source no.) identification
SPECTRAL_WINDOW_ID	Spectral window number (IF no.)
ARRAY_ID	Subarray number
OBSERVATION_ID	Observation identification
POLARIZATION_ID	Polarization identification
SCAN_NUMBER	Scan number
TIME	Integration midpoint time
UVW	UVW coordinates

Caveats as of June 2002:

Because of independent development, selection of data from an MS currently appears in different guises in the various synthesis tools. An effort to reconcile these differences is underway.

10.3 Basic initialization

This section assumes that AIPS++ has been installed on your UNIX or LINUX system. First, create a .aipsrc file in your home directory which will set some



defaults and optimize run time. See Getting Started for a description of how to set up your .aipsrc file. An example .aipsrc file is given in Appendix 1.

To start AIPS++, cd to the directory you want to work in and type in an xterm window:

```
source /aips++/stable/aipsinit.csh     # source the aipsinit.csh file
                                       # that specifies setup parameters
aips++                                 # start aips++ (stable version)
```

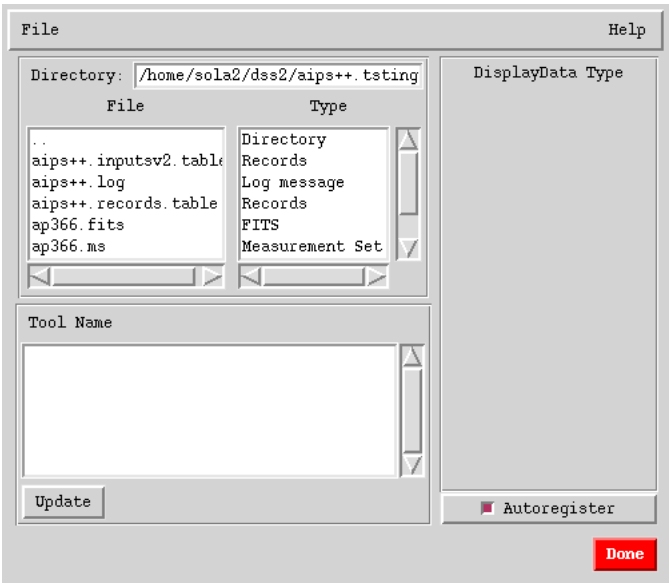
You should get a *Glish* command prompt in the xterm window, a `Tool Manager` GUI window, and a `Logger` GUI window (see figure, above).

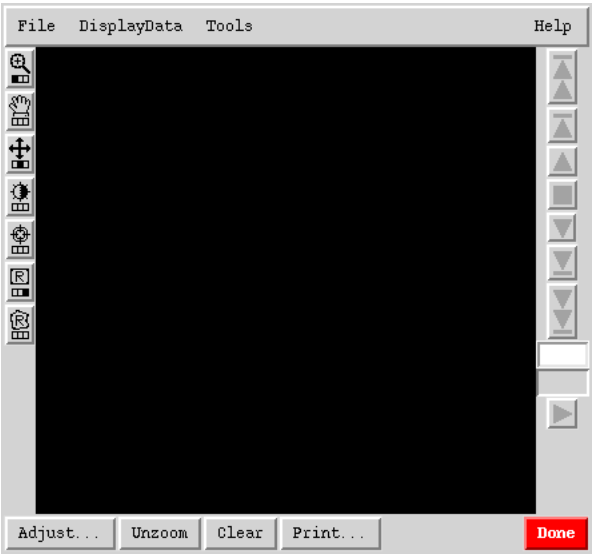
Check with your system administrator if you cannot locate the aipsinit.csh file.

It is also convenient to start up a viewer `tool` to view and analyze images easily. To start the default viewer GUI from the *Glish* command line type:

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

Or, to start the default viewer from the tool manager window click on the `Tools in use` button. Click on the `viewer tool` and then the `Show` button. You will get a **Data Manager** tool which you can use to select any image and a viewer **Display Panel**:





You can put these off to the side until you are ready to use them. Note: once the aipsinit.csh file has been sourced, you can start the `Viewer` directly from the UNIX prompt by typing `viewer`.

The remainder of this document uses *Glish* script sequences to execute data reduction steps. The *Glish* sequences can also be used as a guide for operation using the Tool Manager GUI interface. All parameter names and settings correspond exactly to those listed in the GUI interface (in general, only the immediately relevant parameters are listed in the glish sequences; others may be safely left set to default values). Consult the Getting Started document for general tips on the use of the Tool Manager.

An important difference between working with the GUI interface and the *Glish* command line interface is that a *Glish* command to run a task or function begins with all values set to their default values. Thus, the user need only specify values that will be changed from their defaults. However, the GUIs remember the state of each input. Thus, if you change an input, it will be remembered the next time you use the GUI. If you are unsure of the default value for a parameter in a GUI window, use the "Wrench" icon to the right of the input line to reset the value to its default. Also, GUI inputs do not require quotes around inputs while *Glish* does.

Note: From the Tool Manager window, you can click on Options: 'Copy

commands to scripter' and Options: 'Show scripter' to write equivalent *Glish* commands of your GUI-based actions to a Scripter GUI window. This is useful because you can copy *Glish* script into a file and use this as a record of your data reduction session (or just save the Scripter commands directly with the 'Save' button at the bottom of the Scripter GUI). You can also modify scripts to reduce similar data in a semi-automated fashion.

When working from the *Glish* command line in AIPS++, basic tools for VLA data reduction and imaging must be initialized first. Type at the *Glish* command prompt:

```
include 'vlafiller.g';      # if filling from the VLA archive
include 'ms.g';             # if filling from a UVFITS file
include 'flagger.g';
include 'imager.g';
include 'msplot.g';
include 'calibrater.g';
include 'image.g';
include 'imagepol.g';
include 'viewer.g';
```

This is not required when working entirely from the GUI interface because the tools are loaded automatically when they are selected.

What happens if something goes wrong?

First, check that your inputs are correct. If you are working with the GUI interface, there is 'pop-up' help available on most inputs – simply put the cursor on an item and a short help statement will pop-up on the screen.

You can get help from the Tool Manager window by clicking on the 'Help' button and choosing the 'Reference Manual' option. This will drive your browser to the AIPS++ detailed documentation on each tool. From this web page, you can also reach the main AIPS++ documentation site (click on Documentation in the upper left of the web page) or you can run a Search on key words you are interested in.

Also, you can get help on each `tool` and the `tool` inputs by clicking on the 'Help' button in the GUI `tool` screen.

If your inputs are correct and you are still having problems, you may have discovered a software bug. If you find a bug during your data reduction session, type at the *Glish* command line prompt:

```
bug()
```

Or, using the Tool Manager 'Help' button, choose 'Report a bug' option. This will bring up a bug report GUI that you can fill in and submit automatically within AIPS++.

If you manage to find yourself in a hung state (don't worry, this is rare), you can kill AIPS++ from the LINIX or UNIX command line with:

```
LINUX:   ps -aux | grep aips++
 UNIX:   ps -aef | grep aips++
```

```
Note the process ID numbers listed and kill them with the command:

            kill -9 process_id
```

You are now ready to restart AIPS++.

10.4 Basic Calibration Fundamentals

Calibration and imaging responsibilities are divided between the Calibrater and Imager tools. The Calibrater `tool` handles visibility-plane calibration while the Imager `tool` deals with image formation, deconvolution and image-plane calibration.

In a typical synthesis observation, a number of calibrators of known structure (usually point sources, and at least one of known flux density) are observed in addition to the scientific target(s) in order to obtain a measure of the systematic properties of the instrument and, if relevant, the corrupting media above it. The effects of these systematic properties are then removed from the data for the scientific target. This technique is often referred to as *cross-calibration* and its success depends implicitly on the relationship between: 1) the true brightness distribution of the observed radio sources (what we are after); 2) the formal response of the instrument and its variety of imperfections of varying importance (what we are calibrating); 3) and the data that is actually recorded (what we have measured).

The AIPS++ Measurement Equation, originally defined by Hamaker, Bregman, & Sault (1996a, 1996b), is an explicit description of this relationship. The Measurement Equation for a VLA spectral line polarization calibration is given by

$$\vec{V}_{ij} \;=\; B_{ij}\; G_{ij}\; D_{ij}\; P_{ij}\; T_{ij}\; \vec{V}_{ij}^{\,\mathrm{IDEAL}} \tag{10.1}$$

where:

$\vec{V}_{ij}$ = cross-correlations between two polarizations for each of two feeds (i, j) which characterize an individual baseline (e.g. the measured visibilities).

$\vec{V}_{ij}^{\,\mathrm{IDEAL}}$ = visibilities measured by an ideal interferometer (e.g. no instrumental errors or calibration effects). $\vec{V}_{ij}^{\,\mathrm{IDEAL}}$ = is directly related to the Fourier Transform of the true polarized sky brightness, $\vec{I} = (I, Q, U, V)$.

T_{ij} = Complex gain effects which are polarization-independent, e.g. tropospheric effects, high-frequency opacity corrections, antenna gain as a function of elevation, baseline corrections.

P_{ij} = Paralactic angle.

D_{ij} = Instrumental polarization response. "D-terms" describe the polarization leakage between feeds (e.g. how much the R-polarized feed picked up L-polarized emission, and vice versa).

G_{ij} = Electronic gain response due to components in the signal path between the feed and the correlator. This complex gain term G_{ij} can include phase and amplitude corrections due to changes in the atmosphere along with a scale factor for absolute flux calibration.

B_{ij} = Bandpass response.

The order of the terms from right-to-left, for the most part, is the order in which the errors affect the incoming wavefront. However, it is often best to begin the calibration process by determining solutions for those terms which affect the data most. Thus, the user would normally determine gain solutions (G_{ij}) first (applying pre-computed paralactic angle corrections P_{ij} if doing polarization calibration), then bandpass solutions (B_{ij}) and/or polarization solutions (D_{ij}). Errors which are polarization-independent (T_{ij}) can be determined at any point in the calibration process. T_{ij} has nearly identical form to G_{ij} but it located at a different point in the Measurement Equation. A practical advantage to this factorization is the ability to store polarization-independent effects which are characterized by a certain timescale or a particular parametrization (e.g., tropospheric opacity) as a separate calibration factor, thus isolating them from receiver gain effects which may vary on a different timescale and/or with different systematics. Although most data reduction sessions will probably proceed in this order, the user has complete freedom and flexibility to determine the order in which calibration effects are solved for. Self-calibration is also not limited to just the gain corrections; e.g.: solutions of, say, the bandpass, can be applied to the data and the resulting model used to obtain a better gain calibration.

Once the calibration solutions have been determined, the AIPS++ software applies the solutions according to the order defined by the Measurement Equation. The individual calibration components represented in the Measurement Equation are described in detail in the Synthesis Calibration chapter of `Getting Results`.

The observed data, $\vec{V}_{ij}$, are stored in the DATA column in the MAIN table of the MS; these are the raw data as loaded by the filler `tool` or imported from a UVFITS file. Associated with the DATA column are related columns to hold the most recent version of the calibrated data (CORRECTED_DATA), and the latest visibility plane representation of the source or field model, $\vec{V}_{ij}^{IDEAL}$, (MODEL_DATA). The latter two columns are filled in by the calibrater and imager tools, respectively. The actual calibration information is stored in separate calibration tables. The observed DATA column does not change during reduction, but the related columns do. When plotting the data using a tool such as msplot, the data sub-types (`observed, corrected, and model`) can be selected individually.

10.4.1 Summary of Data Reductions Steps

Listed below are the basic steps for typical continuum and spectral line data reduction sessions which solve for and apply the standard (instrumental/atmospheric)

complex gain (G), the polarization leakage (D), and the relative bandpass (B).

Since reduction of continuum, spectral-line, and polarimetry data in AIPS++ differ essentially only in the addition or omission of certain steps, and in the details of a few parameter settings, only one pass through the basic reduction path is described here. In the remainder of this document, annotated sequences of glish for both continuum polarimetry and spectral-line reduction are included in parallel. Some actions apply to only one type of reduction, but all methods are instructive for either type of observation.

The basic reduction sequence is:

1. Import the data. Data may be imported from either a VLA archive tape using the vlafiller `tool`, or from a UVFITS disk file (e.g., as written by AIPS) using the ms.fitstoms constructor. The result is a Measurement Set table (the AIPS++ internal data format). The ms `tool` is used to obtain a summary of the dataset.

2. Examine and edit the data using the Flagger `tool` for basic data editing, the Autoflag `tool` for automatic data editing, and the msplot `tool` for interactive editing.

3. Set the initial calibration models. By default, all sources have unit flux density point source models. Use the Imager function Imager.setjy to set (point-source) flux densities for the absolute flux calibrators.

4. Obtain the complex gain (G) calibration for calibrators using the Calibrater `tool`. If planning to do instrumental polarization (D) calibration, application of parallactic angle (P) corrections is required at this point.

5. Obtain the bandpass calibration (B) (spectral-line observations only) for bandpass calibrator(s) using the Calibrater `tool`. The G calibration obtained above (and P, if necessary) is applied on-the-fly to ensure that the B calibration is normalized.

6. Transfer the flux density scale derived from the absolute flux calibrator(s) using the Calibrater `tool`. Only the G calibration solutions obtained for the amplitude calibrators are properly scaled in amplitude. The solutions for the other calibrators are rescaled with the Calibrater.fluxscale function by demanding that the mean gain amplitude be the same for all calibrators.

7. Apply the derived calibration to the source data.

8. Obtain the instrumental polarization calibration (D) (polarization observations only). Use a full-polarization model for a calibrator (obtained by imaging[1], and set with the function Imager.setjy). All previous calibration factors are applied before the D-terms or "leakage terms" are determined.

[1] Other means are in the works, stay tuned!

9. Apply all calibrations to the target source data. The Calibrater `tool` Correct function applies calibration to the observed data and writes the result in the CORRECTED_DATA column in the MeasurementSet.

10. Image the target source. The imager `tool` is used to form images from the calibrated data, deconvolve these images, and restore the residual errors to the model image.

11. As necessary, self-calibrate the target source data and reimage. New and improved calibration may be obtained by repeating the initial calibration step(s), this time using the model image obtained in the first round of imaging. Re-image.

12. Repeat the self-calibration as necessary. Self-calibration need not be limited to only the G calibration factor; any solvable calibration component (G, T, B, and/or D, in the present example) may be self-calibrated.

Keep in mind that these examples are basic ones. More sophisticated sequences are possible. For example, it may be desirable to obtain bandpass calibration from an optimum calibrator, and to apply this on-the-fly during full-out G calibration on a different calibrator. This operation can be performed in the G and B calibrations steps listed above for the bandpass calibrator only, then repeating the G calibration step for the other calibrators with the B calibration applied. In fact, any sequence of operations of this nature is possible; the fundamental requirement is that calibration components that significantly influence the one being solved for be applied prior to the solution.

10.5 Example data and scripts

The two sample datasets used in the examples below are available from the AIPS++ Data Repository. The Data Repository is usually created in the directory /aips++/data/ when AIPS++ is installed on a UNIX or LINUX machine (check with your AIPS++ system administrator if the data are not here). You can create AIPS++ MeasurementSets from this data directly as described below using the `fitstoms tool`.

The *Glish* sequences listed below describe calibration and imaging steps and are available as full end-to-end scripts in the AIPS++ Recipe Repository. Scripts in the Repository can be easily modified to reduce and image similar data sets.

The continuum polarimetry dataset is a full-polarization, VLA A-configuration observation at 5 GHz (one spectral window) of the gravitational lens 0957+561. A nearby point-like calibrator (0917+624) has been observed alternately with the target source. 3C286 is used for both flux density scaling and instrumental polarization calibration. The *Glish* script is available here: continuum polarimetry.

The spectral-line dataset is a short VLA D-configuration observation of HI (redshifed to 1413 MHz) in NGC5921. This dataset was observed in total intensity only (RR & LL), with 63 channels in a single spectral window. A nearby

phase calibrator (1445+099) was observed, as well as 3C286 for flux density calibration. The *Glish* script is available here: spectral line.

For clarity in distinguishing operations for each of the sample datasets, tools associated with the continuum polarimetry reduction have names ending in 'C' (imgrC, calC, etc.), and those associated with the spectral-line reduction have names ending in 'S' (imgrS, calS, etc.).

10.6 Filling VLA data into AIPS++

There are two paths for importing VLA data into AIPS++: directly from a VLA archive or distribution tape, or from a UVFITS disk file generated by the AIPS task FITTP. Both methods are described here. The example data sets from the Data Repository are UVFITS files that have been created using the AIPS task FITTP.

10.6.1 Filling data from VLA archive format

VLA data in on-line archive format are read into AIPS++ from disk or tape using the vlafiller `tool`. Tape drives can be located on the local system or on a remote system accessible on the network. Use of a remote tape drive requires knowledge of the network host name (e.g. `gibbon.cv.nrao.edu`, or `gibbon`), and the tape device name (e.g. `/dev/rmt/0ln`). This can often be determined from the physical drive itself, or by logging on to the remote host and using a UNIX utility such as `sysinfo` or the UNIX command `ls -lut /dev/r[ms]t*` to list the device name.

Remote tapes need not be explicitly mounted from AIPS++ as the vlafiller itself will be run on the remote system directly. If you encounter problems in using the filler on a remote host, make sure that the AIPS++ initialization command `source /aips++/release/aipsinit.csh` is included in your `.cshrc` file (or the `sh` equivalent).

To run the vlafiller using a local tape drive, use the vlafillerfromtape function, e.g.:

```
vlafillerfromtape(msname='cals00feb.ms',       # Load project TESTT from tape to
                  device='/dev/rmt/0ln',       # MeasurementSet cals00feb.ms
                  project='TESTT');            # in current directory
```

To run the vlafiller using a remote tape drive:

```
vlafillerfromtape(msname='/home/cals00feb.ms',   # Load project TESTT from remote
                  device='/dev/rmt/0ln',         # tape on host 'gibbon' to MS
                  project='TESTT',               # cals00feb.ms in /home directory
                  host='gibbon');                # on remote host
```

Note that it is necessary to specify a full path name (recognizable by the remote host) for the output file when filling from a remote tape drive.

The vlafiller `tool` allows selection on frequency band, calibrator name, frequency range, project, time, subarray and source, and supports concatenation to an existing uv-data set.

It is usually desirable to obtain a summary of the dataset's properties after filling. This can be done by constructing an ms `tool`, and running its `summary` function as follows:

```
mset:=ms(filename='cals00feb.ms');      # Construct the ms tool for the data
mset.summary(verbose=T);                # Write a summary to the logger window
mset.done();                            # Finish ms tool
```

Copy the summary information into a file for later use (source or FIELD_ID numbers and the times different sources are observed are particularly useful).

10.6.2 Filling data from UVFITS format

Data which have been exported to a disk file from AIPS using the task FITTP can be read into AIPS++ using the general UVFITS filler `ms.fitstoms`, which itself creates an ms `tool`. This is achieved as follows:

continuum polarimetry case

```
msetC:=fitstoms(msfile='ap366.ms',
                fitsfile='/aips++/data/demo/AP366-09C1.fits');
                                        # Construct the ms tool for
                                        #  the Continuum data set located
                                        #  in the AIPS++ Data Repository

msetC.summary(verbose=T);               # Write a summary to the logger window

msetC.done();                           # Finish ms tool
```

The detailed summary of the example data set is given in Appendix 2 for reference.

spectral-line case

```
msetS:=fitstoms(msfile='ngc5921.ms',
                fitsfile='/aips++/data/demo/NGC5921.fits');
                                        # Construct the ms tool for
                                        #  the spectral line data set located
                                        #  in the AIPS++ Data Repository

msetS.summary(verbose=T);                    # Obtain summary

msetS.done();                                # Finish ms tool
```

The detailed summary of the example data set is given in Appendix 2 for reference.

10.7 Data examination and inspection

Once the data have been filled to a MeasurementSet, they can be examined or displayed using various `tools`. The Catalog tool provides an overview of AIPS++ data files at the highest level, as individual AIPS++ tables of different types (e.g. images, MS, etc) in a given directory. The Catalog tool can be started from Glish as:

```
include 'catalog.g'
dc.gui()
```

Here, **dc** is the default catalog tool. It can also be accessed via the Toolmanager interface from the `Tools in Use` button. A mouse click on an individual MS listed in the catalog GUI followed by a click on the Summary button, will write a summary of the data in the MS to the logger (including frequency and polarization settings and fieldnames) to the logger. This information is useful during calibration.

When you click on the View button (to view the presently selected catalog item) you will launch the Tablebrowser, which provides facilities to browse and examine all data fields in the MS main and sub-tables.

Note that indices in the MS, such as FIELD_ID, are stored as integers beginning with zero when viewed at the low level provided by the Tablebrowser. All AIPS++ synthesis `tools` correct for this artifact of c++ and present selection of indices beginning at one.

The primary graphical visibility data visualization `tool` in AIPS++ at present is msplot. Msplot allows raster and line plots of the observed, corrected, and model data against a range of MS indices and parameters, as well as associated plots such as uv-coverage. Facilities are provided to support versatile data selection, range setting and plot iteration. Interactive editing is also supported, as described in the next section. It can be started from *Glish* via:

```
include 'msplot.g'
p := msplot('atca.ms', edit=T);     # Allow editing as well as inspection
```

This will launch a gui for plot type and data selection. It can also be launched from the Toolmanager via the `Synthesis` package, and msplot module.

Consult the `msplot` documentation for more information on using this tool.

10.8 Data editing

Command-based, interactive and automated editing capabilities are supported in AIPS++.The `flagger tool` supports the editing.

To start the flagger tool, type the following:

```
fg:= flagger('mydata.ms')          # start the flagger tool
```

Direct (command-based) editing

The Flagger `tool` offers several directed editing (flagging) operations. In general, this involves selecting the data to be flagged (based in indices, timeranges, etc.), then running a flagging operation. The flagging commands all have the option to run them on a trial basis.
To flag data from antenna=1, spw=1, field=2, within a timerange:

```
fg.setantennas(ants=5)                          # Select data including antenna 5
fg.setids(spectralwindowid=1,fieldid=2)         # Select spectral id and field id
fg.settimerange(starttime="24-FEB-2000/10:00:05",  # Select timerange
                endtime="24-FEB-2000/10:09:37");
fg.state();                                     # Review data selection
fg.flag(trial=T);                               # Trial run to see how many rows
                                                #   will be flagged
fg.flag();                                      # Flag it
```

To reset the data selection, type:

```
fg.reset();
```

To flag data from baseline 3-4, spw=2, field=1, within ±10s of a timestamp:

```
fg.setbaselines(ants=[3,4])                     # Select data for baseline 3-4
fg.setids(spectralwindowid=2,fieldid=1)         # Select spectral id and field id
fg.settime(centertime="24-FEB-2000/10:00:05",  # Select time and range
          delta="10s");
fg.state();                                     # Review data selection
fg.flag();                                      # Flag it
```

The logic in the data selection options is as follows. The two time-related data selection options (set by `settimerange` and `settime`) are logically OR'd together, and the two antenna-related data selection options (set by `setantennas` and `setbaselines` are also logically OR'd. The net time selections, net antenna selections, and all other selections are then AND'd together. Note that the data selection persists after the flagging operation (`flag`) is executed. To clear the selection, execute the `reset` function, or execute individual selection commands without arguments to clear only that particular selection. For example, to flag all data to antenna 11 for the same spectral window, field, and timestamp as for baseline 3-4 above:

```
fg.setbaselines();              # Clears baseline selection
fg.setantennas(ants=11);        # Select antenna 11
```

```
fg.state();                         # Review data selection
fg.flag();                          # Flag it
```

To unflag data based on a selection, use the `unflag` function. For example, to restore data for baseline 11-15 (that was flagged above):

```
fg.setantennas();                   # Clears antenna selection
fg.setbaselines(ants=[11,15]);      # Select baseline 11-15
fg.state();                         # Review data selection (note
                                    #  other selections still in
                                    #  effect)
fg.unflag();
```

Note that use of `unflag` may restore data that had been flagged by other means (e.g., by the on-line system, or by other criteria in previous use of `flagger` or `msplot`). In the near future, a flexible means of manipulating flagging versions will be made available in AIPS++.

Consult the `flagger` documentation for more information on the details of data selection.

Heuristic editing

Several simple heuristic editing methods are available in `flagger`. These methods act on the data selected as described above, but add an additional heuristic constraint.

For example, it is usually desirable to flag the auto-correlation data from the VLA (otherwise, the `imager` tool may attempt to use it in image formation as if it were single dish data). To do this:

```
fg.reset();                     # Reset the data selection
fg.flagac();                    # Flag the auto-correlations
```

It is often the case that the VLA data near the beginning of scans (e.g. after field or frequency band changes is bad. To flag such data, use the `quack` function. For example, to `quack` 20 seconds of data at the beginning of scans (defined by gaps larger than 30 seconds) on field 3:

```
fg.reset();                     # Reset the data selection
fg.setids(fieldid=3);           # Limit quack to fieldid=3 only

fg.quack(scaninterval='30s',  # Trial run of quack
```

```
        delta='20s',
        trial=T);

fg.quack(scaninterval='30s',  # Quack it
        delta='20s');
```

Automated editing methods are available in the Flagger and Autoflag `tools`. These include clipping, median flagging and related methods.

For example, to clip data outside of a specified amplitude range with the Flagger `tool`:

```
fg.setids(fieldid=2, spwid=[1,2]);    # Select field, spw

fg.setpol(pol=[1,4]);                 # Select polarizations 1,4
                                      #  (e.g., RR and LL)

fg.filter(column='DATA',              # Flag data if, DATA column
          operation='range',          #   Amplitudes are outside
          comparison='Amplitude',     #   1.12-1.34
          range=['1.12Jy','1.34Jy'],
          fullpol=T,                  # Extend flags to pols and
          fullchan=T);                #  and chans that weren't
                                      #  tested
```

The `autoflag` tool has more sophisticated algorithms. Examples of its use will be included here in the near future.

Finishing `flagger`

To finish the `flagger` tool:

```
fg.done();     # Done with flagging
```

10.8.1 Interactive editing

Interactive editing is possible on almost all graphical views of the data which are possible in msplot. To enable the editing option, it is necessary to set parameter `edit=T` when the msplot `tool` is created. The current set of flags will be saved (for possible restoration later). Select and plot data as in non-editing mode, then draw boxes with the mouse to select regions to flag and press the `Flag` button to flag it. As many flag boxes as desired may be drawn before executing the flags. Use the `Locate` button to obtain a listing (in the `logger`) of the data points within the box(es).

At this writing the channel and polarization flagging scope selections in `msplot` will only work if you have elected to plot all channels and/or polarization. This constraint will be relaxed in the near future.

Raster displays of visibility data are possible also, and maybe useful for ATCA line data editing.

10.9 Basic Gain Calibration

10.9.1 Setting the calibration source model

When solving for visibility-plane calibration components, the Calibrater `tool` compares the solutions with the MODEL_DATA column. The first time that an Imager or Calibrater `tool` is constructed for a given MS, the MODEL_DATA column is created and initialized with unit point source flux density visibilities (unpolarized) for all sources (e.g. AMP=1, phase=0°). The Imager.setjy function can then be used to set the proper flux density for flux density calibrators. For sources that are recognized flux density calibrators (listed below), Imager.setjy will calculate the Perley-Taylor (1999) flux densities, Fourier transform the data and write the results to the MODEL_DATA column. For the VLA, the default source models are customarily point sources defined by the Baars or Perley-Taylor flux density scales, or point sources of unit flux density if the flux density is unknown. The MODEL_DATA column can also be filled with a model generated from an image of the source (e.g. the Fourier transform of an image generated after initial calibration of the data).

Recognized Flux Density Calibrators		
3C Name	*B1950 Name*	*J2000 Name*
3C286	1328+307	1331+305
3C48	0134+329	0137+331
3C147	0538+498	0542+498
3C138	0518+165	0521+166
−−	1934−638	−−
3C295	1409+524	1411+522

continuum polarimetry case

Set 3C286 (field 11) model according to Perley-Taylor (1999):

```
imgrC:=imager(filename='ap366.ms'); # Start imager tool

imgrC.setjy(fieldid=11);            # Calculate & set flux density for
                                    #  1328+307 (3C286)
```

spectral-line case

Set 3C286 (field 1) model according to user-specified value (if you don't want to use the standard values):

```
imgrS:=imager(filename='ngc5921.ms'); # Start imager tool

imgrS.setjy(fieldid=1,
            fluxdensity=[14.8009,0,0,0]);
                                      # Set flux density of 1331+305
                                      #  (3C286) to 14.8009 Jy
```

At this point, the source model and observed data exist for all calibrators at all uv-points, and it is possible to begin solving for the gains.

10.9.2 Solving for complex gain, G

It is almost invariably the case that the instrumental and atmospheric amplitude and phase gains (hereafter "complex gains") are the dominant calibration error in VLA data. Therefore we solve for this factor first.

For the standard cross-calibration scheme described here, only one term (G) will be used to describe the net complex gain (and so it will include atmospheric effects). One complex gain, as a function of time, will be determined for each spectral window, polarization, and antenna in the MS.

Note: Well-behaved antennas that are located near the center of the array are chosen as Reference antennas for each dataset. For non-polarization datasets, reference antennas do not need to be specified although you can if you want to (AIPS++ will determine the optimum relative phases between antennas adequately). For data that requires polarization calibration, it is best to choose a reference antenna that has a constant phase difference between the right and left polarizations (e.g. no phase jumps or drifts). This is because, AIPS++ solves the right and left polarizations independently and the resulting R-L phase difference that is an average over the array. If a phase jump exists in one antenna, this would affect the value determined for the R-L phase difference and corrupt the polarization calibration.

To solve for G, first choose a solution interval which samples the time scale over which you think the instrument or atmosphere is changing. Time scales depend on the array configuration and the observed frequency. Here, a time scale of 60 sec is chosen for the 6 cm continuum polarimetry data while 300 sec is chosen for the 21 cm spectral line data.

continuum polarimetry case

```
calC:=calibrater(filename='ap366.ms');      # Create Calibrater tool

calC.setdata(msselect='FIELD_ID IN [9,11]');
                                            # Select calibrator sources
```

```
                                        #  in MS fields 9 & 11

calC.setapply(type='P',                 # Arrange to apply parallactic
              t=5.0);                   #  angle correction (for polarization
                                        #  only).  Choose a 5 sec timescale.
                                        #  The P-table corrections are
                                        #  computed in AIPS++.

calC.setsolve(type='G',                 # Arrange to solve for G
              t=60.0,                   #  on a 60.0 sec timescale
              refant=3,                 #  Choose reference antenna 3:
              table='ap366.gcal');      #  a well-behaved antenna
                                        #  near the center of the array.

calC.state();                           # Review setapply/setsolve settings

calC.solve();                           # Solve for the net complex gains
                                        #  and write solutions to the table
                                        #  ap366.gcal located on disk.

calC.plotcal(tablename='ap366.gcal');   # Inspect solutions
```

Note that all calibrater.setapply and calibrater.setsolve executions prior to the `solve` execution will be in effect when obtaining the solutions. Thus, the parallactic angle correction, P, is applied before the G solutions are obtained. The parallactic angle correction is required for VLA polarimetry reduction, and may be omitted otherwise because all VLA antennas have the same parallactic angle. Note: for other arrays which have antennas which are not aligned, the parallactic angle correction should be applied. If data imported from AIPS already had the parallactic angle correction applied using the AIPS task CLCOR, then the `Calibrater.setapply` step should be omitted (it is the user's responsibility to keep track of this).

spectral-line case

```
calS:=calibrater(filename='ngc5921.ms'); # Create calibrater tool for
                                         #  spectral line data set

calS.setdata(msselect='FIELD_ID <= 2';   # Select data for calibrators
             mode='channel';             #  (Fields 1 & 2)
             start=3;                    #  and drop the outer channels
             nchan=55);                  #  that may bias the gain solution.

calS.setsolve(type='G',                  # Arrange to solve for G on
              t=300.0,                   #  a 300-second timescale
              refant=15,                 #  Choose reference antenna 15:
              table='ngc5921.gcal');     #  a well-behaved antenna
                                         #  near the center of the array.
```

```
calS.state();                              # Review setsolve settings

calS.solve();                              # Solve for the net complex gains
                                           #  and write solutions to the table
                                           #  ngc5921.gcal located on disk.

calS.plotcal(tablename='ngc5921.gcal');  # Inspect solutions
```

Note that we have omitted application of 'P' since there is no polarization data.

A Note on Calibrater Tool Operation

As illustrated above, the Calibrater `tool` has three 'set' functions (`setdata`, `setapply`, and `setsolve`) and two 'operational' functions (`solve` and `correct`). The `setdata` function selects data and the `setapply` and `setsolve` functions set up the state of the calibrater `tool`. Both `setapply` and `setsolve` can be executed more than once (for different calibration components) prior to `solve` or `correct` (e.g., `setapply` will typically be executed for both G and P before solving for D). Use the `state` function to review the current state and the `reset` function to clear all or part of the `setapply`/`setsolve` state.

The `solve` function will apply the calibration components (specified with `setapply`) to the *observed* DATA, and solve for the components specified with `setsolve`. Note that is is possible to specify more than one component for solving. In this case, the solutions are done serially (not simultaneously) in the order prescribed by the Measurement Equation formalism (e.g., G before D), with each newly-solved component applied before solving for the next (e.g., the obtained G solutions will be applied before solving for D).

The `correct` function applies a set of calibrations to the observed data and saves it for imaging in the CORRECTED_DATA column of the MeasurementSet. Anything set by `setsolve` is ignored by `correct`.

10.9.3 Bandpass calibration

In the case of spectral-line data, it is necessary to remove gain fluctuations that are a function of frequency. This is achieved by using the Calibrater `tool` to solve for B calibration, in the present example, using 1331-305 as the bandpass calibrator. The G calibration obtained above is applied to ensure that the B calibration will be normalized. The solution interval is set very large so only one B solution (in time) is obtained.

spectral-line case only

```
calS.setdata(msselect='FIELD_ID==1');      # Select bandpass calibrator
                                           #  (Field 1 = 1331+305)

calS.reset();                              # Reset apply/solve state
                                           #  of the calibrater tool
```

```
calS.setapply(type='G',                  # Arrange to apply G solutions
              t=0.0,                     #  from the table ngc5921.gcal
              table='ngc5921.gcal');

calS.setsolve(type='B',                  # Arrange to solve for a single
              t=86400.0,                 #  bandpass solution for the
              refant=15,                 #  entire observation.
              table='ngc5921.bcal');

calS.state();                            # Review setapply/setsolve settings

calS.solve();                            # Solve for the bandpass solutions
                                         #  and write them to the table
                                         #  ngc5921.bcal located on disk.

calS.plotcal(tablename='ngc5921.bcal',   # Inspect the solutions
             plottype='AMP',
             multiplot=T);
```

10.9.4 Transfer the flux density scale

For point sources of unknown flux density, but for which valid solutions have been derived against a unit point source model, the absolute flux density can be established by scaling the mean gain amplitudes against those of the known amplitude calibrators. This scaling is achieved using the Calibrater.fluxscale function as follows:

continuum polarimetry case

```
calC.fluxscale(tablein='ap366.gcal',     # Transfer the flux density scale
               tableout='ap366.fluxcal', #  from 1328+307 (B1950: 3C286)
               reference='1328+307',     #  to the gain calibrator 0917+642.
               transfer='0917+624');     #  Write the solutions to
                                         #  the table ap366.fluxcal located
                                         #  on disk.

calC.plotcal(tablename='ap366.fluxcal'); # Inspect the solutions
```

spectral-line case

```
calS.fluxscale(tablein='ngc5921.gcal',       # Transfer the flux density scale
               tableout='ngc5921.fluxcal',   #  from 1328+307 (J2000: 3C286)
               reference='1331+30500002',    #  to the gain calibrator
               transfer=['1445+09900002']);  #  1445+09900002.  Write the
                                             #  solutions to the table
                                             #  ngc5921.fluxcal located on disk.

calS.plotcal(tablename='ngc5921.fluxcal',    # Inspect the solutions
```

```
            plottype='AMP',
            multiplot=F);
```

In both cases, the amplitude scale is set with 3C286 as the reference flux density calibrator. The output calibration table contains the scaled G calibration factors for all of the transfer sources, and will be used in subsequent reduction steps. Proper scaling is indicated in the `plotcal` plots by smoothly varying (ideally constant) amplitude gains for each antenna, *despite* source changes. At high frequencies ($\nu > 15$ GHz), this scaling step may be biased by variation of the atmospheric contribution to the system noise temperature. Methods for treating this case properly will be available in the near future.

10.9.5 Correct the data

Before imaging, the data need to be corrected by applying the calibrations determined above. The Calibrater.setdata function selects which data are to be corrected, and the Calibrater.setapply function selects which solutions are to be applied. The `setapply` function may be run several times (in any order) to specify simultaneous application of different calibration tables. The tables will be applied in the appropriate order according to their order in the Measurement Equation. In AIPS++, calibration tables are always cumulative, and so are always applied to the observed data in the DATA column. The corrected data are written to the CORRECTED_DATA column in the MS, which will be used directly in imaging. Note: the observed DATA is *NOT* changed. If you want to change the solutions you have applied, simply create new calibration tables and correct the data again – the CORRECTED_DATA column will be over-written.

continuum polarimetry case

```
calC.setdata(msselect='FIELD_ID IN [9:11]');  # Select sources in MS fields
                                              #  9, 10 & 11 to which calibration
                                              #  is to be applied

calC.reset();                                 # Reset setapply/setsolve

calC.setapply(type='P',                       # Arrange to apply parallactic angle
              t=5.0);                         #  correction on 5 sec intervals

calC.setapply(type='G',                       # Arrange to apply flux-scaled G
              t=0.0,                          #  solutions from the ap366.fluxcal
              table='ap366.fluxcal');         #  table.

calC.state();                                 # Review setapply settings

calC.correct();                               # Apply solutions and write the
                                              #  CORRECTED_DATA column in the MS.
```

spectral-line case

```
calS.setdata(msselect='FIELD_ID IN [2,3]');  # Select sources in MS fields
                                             #  2, & 3 to which calibration
                                             #  is to be applied

calS.reset();                                # Reset setapply/setsolve

calS.setapply(type='G',                      # Arrange to apply flux-scaled G
              t=0.0,                         #  solutions (from field 2) from
              table='ngc5921.fluxcal',       #  the ngc5921.fluxcal table.
              select='FIELD_ID==2');

calS.setapply(type='B',                      # Arrange to apply B solutions
              t=0.0,                         #  from the ngc5921.bcal table.
              table='ngc5921.bcal',
              select='');

calS.state();                                # Review setapply settings

calS.correct();                              # Apply solutions and write the
                                             #  CORRECTED_DATA column in the MS.
```

To calibrate field 1, as well, `setdata(...)` and `setapply(type='G',...)` should be executed again, selecting FIELD_ID==1 in both, followed by `correct()`.

At this point, the data for both continuum and spectral-line are sufficiently calibrated to attempt initial total intensity imaging.

10.10 Instrumental Polarization Calibration

For the case of continuum polarimetry, it is necessary to calibrate the instrumental polarization (D) before high-quality polarization images can be formed. Accurate D calibration depends on an appropriate *polarization* model for the calibrator. If the calibrator is unpolarized (Stokes Q = U $\sim$ 0), and point-like[2], the simple total intensity model (non-zero Stokes I only) used in the G calibration is appropriate. In fact, the D calibration could have been obtained at the same time as G by issuing an additional Calibrater.setsolve execution before the solve was executed.

If the instrumental polarization calibrator is significantly polarized (as in the case of 3C286) a non-zero model for Stokes Q and U must be determined. This model can be obtained by imaging in polarization prior to D calibration, as described below. In the near future, a function for simply summing the polarization visibilities will be available to provide an initial polarization model estimate.

[2]Usually, the instrumental polarization calibrator is one of the phase calibrators used for cross-calibration of G, which are ideally point sources for that purpose.

In this example, the polarization model of 3C286 is obtained by imaging the data after applying only the G (and P) calibration. Since 3C286 is virtually a point source at this frequency, the Stokes I, Q, and U values are obtained using the Image.statistics function in the Image `tool`. This operation is illustrated in the continuum polarimetry script.

The Calibrater `tool` operations used in solving for D are similar to those used to solve for G. However, note that one additional parameter—`preavg`—is used explicitly in the `setsolve`execution. Since only one D solution is desired for the entire observation, the solution interval is set very long, and the data for each baseline within the solution interval would nominally be averaged entirely before the solution is determined. However, the parallactic angle varies on timescales much shorter than the solution interval[3]. Thus, the `preavg` parameter is set to prevent averaging of the data beyond an acceptable timescale for the parallactic angle.

10.10.1 continuum polarimetry case (only)

First make IQUV images of the polarization calibrator 3C286 to determine the polarization model (a more detailed description of imaging is given in the next section).

```
imgrC.setdata(mode='none',                  # Select continuum data for
              fieldid=11);                  #  for 3C286.

imgrC.setimage(nx=512,                      # Set up imaging parameters.
               ny=512,
               cellx='0.1arcsec',
               celly='0.1arcsec',
               stokes='IQUV',
               fieldid=11);

imgrC.weight(type='uniform');               # Choose weighting
                                            #  e.g. uniform, natural.

imgrC.makeimage(type='corrected',           # Make an image using the
                image='3C286.cal1.im');     #  CORRECTED_DATA column
                                            #  in the MS.  Write the image
                                            #  3C286.cal1.im to disk.
```

The resulting image, 3C286.cal1.im, written to disk, has 4 planes, with I, Q, U, and V images. Using the Viewer, obtain statistics on each plane of the IQUV image (see the Image Analysis section of this document for details about Viewer operation). Record the maximum OR minimum value, which ever has

[3]The P calibration only rotates the parallactic angle dependence out of Measurement Equation terms containing the nominal interferometer response; the corrupting terms (in particular, those containing D) still have a parallactic angle dependence. It is this dependence which makes the D calibration separable and thus solvable. Refer to the Synthesis Calibration chapter of *G*etting Results for more information.

the largest absolute value. Use the viewer arrow buttons on the right to display Q and U images – record statistics for each image. Assume V = 0. Use the Stokes I value obtained from the Imager `tool` setjy function run earlier. For this example:

```
I = 7.462  Q = -0.621  U = -0.593  V = 0
```

If your data set has two spectral windows, create an image with spwid=2 and repeat this procedure to find I, Q, and U.

Now, solve for the D-term calibration:

```
stokes := [7.462, -0.621, -0.593, 0.0]       # Define polarization model
                                             #  determined from imaging

imgrC.setjy(fieldid=11,                      # Set model for IQUV
            fluxdensity=stokes);

calC.setdata(msselect='FIELD_ID==11');       # Select data for polarization
                                             #  calibrator, 3C286

calC.reset();                                # Reset setapply/setsolve

calC.setapply(type='P',                      # Arrange to apply parallactic angle
              t=5.0);                        #  correction

calC.setapply(type='G',                      # Arrange to apply flux-scaled
              t=0.0,                         #  G solutions
              table='ap366.fluxcal');

calC.setsolve(type='D',                      # Arrange to solve for D over a long
              t=86400.0,                     #  time scale, average data within
              preavg=600.0,                  #  the solution to no more than 600
              table='ap366.dcal');           #  sec per chunk.
                                             #  Write the solutions to the table
                                             #  ap366.dcal located on disk.

calC.state();                                # Review setapply/setsolve settings

calC.solve();                                # Solve
```

Examine the D-term or "leakage" solutions with *plotcal* plot type DRI (D-term real versus imaginary components):

```
calC.plotcal(plottype="DRI" ,
             tablename="ap366.dcal");
```

For VLA data like this example, see http://www.aoc.nrao.edu/~gtaylor/calman/polcal.html for a description on how to plan for good polarization calibration and what to expect when solving for D-terms.

Now, the data can be corrected for both G and D as follows:

```
calC.setdata(msselect='FIELD_ID IN [9:11]'); # Select fields 9, 10, & 11 to
                                             #  which calibration will be applied:
                                             #  9  = gain calibrator 0917+642
                                             #  10 = target source 0957+561
                                             #  11 = polarization cal 3C286

calC.reset();                                # Reset setapply/setsolve

calC.setapply(type='P',                      # Arrange to apply parallactic
              t=5.0);                        #  angle correction

calC.setapply(type='G',                      # Arrange to apply flux-scaled
              t=0.0,                         #  G solutions
              table='ap366.fluxcal');

calC.setapply(type='D',                      # Arrange to apply D solutions
              t=0.0,
              table='ap366.dcal');

calC.state();                                # review setapply settings

calC.correct()                               # Write the corrected the data
                                             #  to the CORRECTED_DATA column
                                             #  in the MS
```

The polarization calibration is almost complete – the final step, calibrating the electric vector position angle, will be done during the final imaging steps below.

10.11 Imaging

Imaging is done using the Imager `tool`, which supports a broad range of imaging modes and deconvolution algorithms, as described in the synthesis imaging chapter and in the User Reference Manual. This section describes single-field imaging/deconvolution for the continuum and spectral-line datasets using a CLEAN region which is defined interactively.

continuum polarimetry case

For the continuum dataset, make a single-field, full-polarization (I, Q, U, & V), continuum image of the target source, 0957+561. This image is made with the Clark CLEAN algorithm as follows:

First, use the Imager.advise function to get an estimate of what the image properties should be. Then set the image parameters and weight, and CLEAN the image using an interactive cleaning algorithm:

```
imgrC.setdata(mode='none',           # Select continuum data for the target
```

```
             fieldid=10);           #  source in field 10 (0957+561)

imgrC.advise(takeadvice=F,          # Determine image and cell size using
             fieldofview='1arcmin'); #  advise function.  Do not take advice,
                                    #  note parameters in logger
                                    #  window and use as a basis for setimage.

                    # ADVISE FUNCTION RESULTS:
                    #  Maximum uv distance = 585844 wavelengths
                    #  Recommended cell size < 0.176041 arcsec
                    #  Recommended number of pixels = 360
                    #  Dispersion in uv, w distance = 293779, 94533.3 wavelengths
                    #  Best fitting plane is w = 0.340419 * u + 0.340608 * v
                    #  Dispersion in fitted w = 30797.4 wavelengths
                    #  Wide field cleaning is not necessary


imgrC.setimage(nx=512,              # Set image plane parameters
               ny=512,
               cellx='0.1arcsec',
               celly='0.1arcsec',
               stokes='IQUV',       #   (full polarization)
               fieldid=10);

imgrC.weight(type='uniform');       # Set uniform weighting
```

At this point, you can examine the synthesized beam to determine the resolution resulting from the chosen weighting scheme and see what artifacts to expect from beam sidelobes during CLEANing, or you can proceed to CLEAN the image. To make an image of the synthesized beam (e.g. a PSF image):

```
imgrC.makeimage(type='psf',           # Form the PSF image
                image='0957+561.psf');

imgrC.fitpsf(psf='0957+561.psf');     # Measure the beam size
```

The beam size and position angle will be reported in the logger window. To CLEAN the image:

```
imgrC.clean(algorithm='clark',      # Image, and deconvolve using
            niter=5000,             #  the Clark CLEAN
            gain=0.1,               #  Write the cleaned image to the file
            model='0957+561.mod',   #  0957+561.im on disk.
            image='0957+561.im',
            residual='0957+561.resid',
            mask='0957+561.mask',   # Clean mask file name.
            interactive=T,          # Clean interactively, with the option
            npercycle=500);         #  to choose a new clean region every
                                    #  500 cleaning cycles.
```

During the CLEAN run, you will be asked to interactively define CLEAN regions (polygons or boxes) in a pop-up viewer. How to do this is described at the end of this section.

Note that the CLEAN component model is written as an image file. If necessary or desired, CLEANing may be restarted by specifying an existing (non-zero) model image, which will be subtracted before the deconvolution is resumed. This provides a simple means of monitoring progress of the deconvolution. Note that the `setdata` and `setimage` functions need not be run again when restarting a CLEAN with the previously setup Imager `tool`.

When you are finished CLEANing, use the Viewer to display the image 0957+561.im and obtain statistics. The RMS should be $\sim$ 0.5 mJy beam^{-1}. The peak in the image is $\sim$ 26.2 mJy beam^{-1}.

spectral-line case

For the spectral-line dataset, total intensity images must be generated for NGC5921 for each spectral channel to form a spectral line cube. In this example, the CLEANing process is done on the line+continuum data and then the continuum is subtracted in the image plane.

Cleaning a spectral line cube with the continuum in each channel can be inefficient if most of the clean cycles are spent cleaning the same structure (e.g. continuum emission) in each plane. Thus, it is often better to subtract the continuum emission before the CLEAN process. A *Glish* script along with a description of how to do continuum subtraction in the *uv* plane, before imaging, will be available soon.

```
imgrS.setdata(mode='channel',        # Select channel data for field 3
              nchan=63,
              start=1,
              step=1,
              fieldid=3);

imgrS.advise(takeadvice=F,           # Determine image and cell size using
            fieldofview='60arcmin'); #  advise function.  Do not take advice,
                                     #  note parameters in logger
                                     #  window and use for setimage.

                    # ADVISE FUNCTION RESULTS:
                    #  Maximum uv distance = 4798.4 wavelengths
                    #  Recommended cell size < 21.4931 arcsec
                    #  Recommended number of pixels = 180
                    #  Dispersion in uv, w distance = 1858.4, 759.892 wavelengths
                    #  Best fitting plane is w = -0.253496 * u + -0.574818 * v
                    #  Dispersion in fitted w = 172.198 wavelengths
                    #  Wide field cleaning is not necessary

imgrS.setimage(nx=256,               # Imaging parameters
               ny=256,
               cellx='10arcsec',
```

```
                celly='10arcsec',
                stokes='I',
                mode='channel',
                start=1,
                step=1,
                nchan=63,
                fieldid=3);

imgrS.weight(type='uniform');        # Uniform weighting

or

imgrS.weight(type='briggs',          # Robust weighting
             rmode='norm',
             robust=0.5);
```

At this point, you can examine the synthesized beam to see if you are happy with the resolution resulting from the chosen weighting scheme and see what artifacts to expect from beam sidelobes during CLEANing, or you can proceed to CLEAN the image.

To make an image of the synthesized beam (e.g. a PSF image):

```
imgrS.makeimage(type='psf',            # Form the PSF image
                image='ngc5921.psf');

imgrS.fitpsf(psf='ngc5921.psf');       # Measure the beam size
```

The beam size and position angle will be reported in the logger window.

You can also use the `Imager.Makeimage` function to create a dirty map:

```
imgrS.makeimage(type='observed',
                image='ngc5921.map');
```

To CLEAN the image:

```
imgrS.clean(algorithm='clark',       # Image and deconvolve
            niter=3000,              #  with Clark CLEAN
            gain=0.1,                #  Write the cleaned image to the file
            threshold='0Jy',         #  ngc5921.im on disk.
            model='ngc5921.mod',
            image='ngc5921.im',
            residual='ngc5921.resid');
```

When you are finished CLEANing, use the Viewer to display the image ngc5921.im and obtain statistics. The RMS in each channel should be $\sim$ 1.7 mJy beam^{-1}.

You should see 2 continuum source and the galaxy as you move through the cube. Note, in this example the entire inner quarter of the image is CLEANed. If desired, you can also use the interactive mask function as in the continuum polarimetry example.

To interactively define one or more CLEAN regions

1. To zoom on a region, click on the Zoom function on the top left of the pop-up Viewer. Use the left mouse button to draw a region around the source emission. Note: once the box is finished, you can move each point defining the region until it is the size you desire. Double click in the region to zoom in.

2. To make a box-mask, click on the box region function. Use the mouse button to create a box. To make a polygon-mask, click on the polygon region function on the left of the pop-up Viewer. Use the mouse button to create a polygon. Note: once the polygon is finished, you can move each point defining the region until it is the shape you desire.

3. Click on 'Add Region'

4. Double click in the region you just made (this will add the region to the mask file). The region will be reported on the glish command prompt.

5. Click on 'Refresh mask' to see the clean region(s) you have specified displayed on the image.

6. Hit the <ESC> key on the keyboard to remove the polygon or box outline.

7. Define more regions and add if desired.

8. To delete all or part of the mask:

   - Click on 'Remove region'
   - Draw a box or polygon around the mask area you want to remove,
   - Double click in the box or polygon.
   - Click on 'Refresh mask' to see your new region.

9. When finished, click on 'Done with Masking'

10. The CLEAN algorithm will continue using the region(s) to restrict the location of clean components.

11. You will be given the option to examine the results every npercycle time selected. If you want to stop the cleaning process, click on the 'Stop' button.

Note: at this time, if you want to interactively define the CLEAN region or mask and then CLEAN down to a specified threshold level without continuing to set or change the mask, run CLEAN with `niter` set to a small number and `interactive=T` to define the mask. Then re-start CLEAN with `niter` set to a large number, `threshold` set to the level you want, `mask` set to your newly made mask, and `interactive=F`.

10.12 Self-Calibration

If you have sufficient signal-to-noise on your target source or any source in the field, then you may now use the Calibrater `tool` to improve the dynamic range of your images. This improved calibration or "self-calibration" technique assumes that your images have been degraded by complex gain errors which vary too rapidly with time or direction to have been fully calibrated with the calibrator sources. The Calibrater `tool` compares the input *uv* DATA with a model of the data that you created during the initial CLEANing and creates amplitude and/or phase corrections which bring the CORRECTED_DATA into better agreement with the model.

Do not self-calibrate unless your target source has enough signal-to-noise to warrant improvement. Ask yourself whether your externally-calibrated CLEAN images contain artifacts well above the noise. If so, self-calibration may improve the image. If your images are limited by receiver noise, self-calibration may produce erroneous results, including the fabrication of weak sources!

continuum polarimetry case

Form the model using the Imager `tool` and fill the MODEL_DATA column in the MS for target source:

```
imgrC.setdata(mode='none',           # Select continuum data for the target
              fieldid=10);           #  source in field 10 (0957+561)

imgrC.ft(model='0957+561.mod');      # Fill MODEL_DATA column in the MS
```

Note: if you want to determine a new model (e.g. re-image with different imaging weights or different calibration solutions), or just start over from scratch, use the Imager.setjy to reset the MODEL_DATA column.

Calibrate the data using the new model. We want an incremental solution over the current G, so do an "atmospheric phase" T calibration (phase only, polarization-independent solutions): Note: the MODEL_DATA column is read automatically when solving for T.

```
calC.setdata(msselect='FIELD_ID==10'); # Set data for target source

calC.reset();                          # Reset apply/solve state

calC.setapply(type='P',                # Arrange to apply P corrections
              t=5.0);

calC.setapply(type="G",                # Arrange to apply flux-scaled
              t=0,                     #  G solutions
              table="ap366.fluxcal");

calC.setapply(type="D",                # Arrange to apply polarization
              t=0,                     #  D-term solutions
```

```
             table="ap366.dcal");

calC.setsolve(type="T",                 # Arrange to solve for T to
             t=10,                      #  get an incremental solution to
             preavg=0,                  #  G with a phase-only correction.
             phaseonly=T,
             table="ap366.tcal");

calC.state();                           # Review the setapply/setsolve settings

calC.solve();                           # Solve for T
                                        #  Write the output to a table
                                        #  called ap366.tcal1 on disk

calC.plotcal(plottype="PHASE",          # Examine solutions
            tablename="ap366.tcal",
            fields=10);
```

Now apply the P, G, D, and T solutions and write a new CORRECTED_DATA column for the target source in the MS:

```
calC.setapply(type="T",                 # Arrange to apply T solutions
             t=0,                       #  note: P, G, & D solutions are
             table="ap366.tcal");       #  already set to apply

calC.correct();                         # Correct the data
```

CLEAN the self-calibrated image of 0957+561 to see how the self-calibration process improved the image:

```
imgrC.setdata(mode='none',          # Select continuum data for the target
             fieldid=10);           #  source in field 10 (0957+561)

imgrC.setimage(nx=512,              # Set image plane parameters
              ny=512,
              cellx='0.1arcsec',
              celly='0.1arcsec',
              stokes='IQUV',        #   (full polarization)
              fieldid=10);

imgrC.weight(type='uniform');       # Set uniform weighting

imgrC.clean(algorithm='clark',      # Image, and deconvolve using
            niter=5000,             #  the Clark CLEAN
            gain=0.1,               #  Write the cleaned image to the file
            model='0957+561.mod2',  #  0957+561.im2 on disk.
            image='0957+561.im2',
            residual='0957+561.resid2',
            mask='0957+561.mask2',
            interactive=T,          # Clean interactively, have the option
```

```
            npercycle=500);          #  to choose a new clean region every
                                     #  500 cleaning cycles.
```

Use the viewer to examine the final source image 0957+561.im2. The RMS should be ~ 0.24 mJy beam^{-1} (versus 0.5 mJy beam^{-1} before self-calibration). The peak in the image is now ~ 29.4 mJy beam^{-1} (versus 26.2 mJy beam^{-1} before self-calibration). Self-calibration improved the dynamic range by a factor of 2.3 in this example. Repeat the self-calibration as necessary. Note: Self-calibration can be applied to any solvable calibration component (G, and/or D in the present example).

spectral-line case

In this example data set, there is an 80 mJy continuum source just south of the main line emission in NGC 5921. We can use this source to self-calibrate the line emission as shown below.

Create an image with line-free continuum channels only. First, create an image from line-free channels 5–8:

```
imgrS.setdata(mode='channel',        # Select channel data for 4
              nchan=4,               #  continuum-only channels in
              start=5,               #  field 3 (target field)
              step=1,
              spwid=1,
              fieldid=3);

imgrS.setimage(nx=256,               # Imaging parameters
               ny=256,
               cellx='10arcsec',
               celly='10arcsec',
               stokes='I',
               mode='channel',       # Choose channel mode
               start=5,              #  Grid selected channels into one.
               step=4,
               nchan=1,
               spwid=1,
               fieldid=3);

imgrS.weight(type='uniform');        # Uniform weighting

or

imgrS.weight(type='briggs',          # Robust weighting
             rmode='norm',
             robust=0.5);

imgrS.clean(algorithm='clark',       # Image and deconvolve
            niter=1000,              #  with Clark CLEAN
            gain=0.1,                #  Write the cleaned image,
```

```
            threshold='0Jy',          #  ngc5921.c1.im and the model,
            model='ngc5921.c1.mod',   #  image ngc5921.c1.mod to disk
            image='ngc5921.c1.im',
            residual='ngc5921.c1.resid');
```

Form the model using the Imager `tool` and fill the MODEL_DATA column in the MS for target source using only continuum channels that were used to create the model:

```
imgrS.setdata(mode='channel',         # Select channel data for 4
              nchan=4,                #  continuum-only channels in
              start=5,                #  field 3 (target field)
              step=1,
              spwid=1,
              fieldid=3);

imgrS.ft(model='ngc5921.c1.mod');  # Fill MODEL_DATA column in the MS
```

Note: if you want to determine a new model (e.g. re-image with different imaging weights or different calibration solutions), or just start over from scratch, use the Imager.setjy to reset the MODEL_DATA column.

We want an incremental solution over the current G, so do an "atmospheric phase" T calibration which will be derived from the continuum channels which were used to create the model (phase only, polarization-independent solutions): Note: the MODEL_DATA column is read automatically when solving for T. Set the time interval for the T solutions to be roughly the same as the solution intervals to get scan-based solution intervals.

```
calS.setdata(mode='channel',           # Select channel data for 4
              nchan=4,                 #  continuum-only channels in
              start=5,                 #  field 3 (target field)
              step=1,
              msselect='FIELD_ID==3');

calS.reset();                          # Reset apply/solve state

calS.setapply(type="G",                # Arrange to apply flux-scaled
              t=0,                     #  G solutions
              table="ngc5921.fluxcal");

calS.setapply(type="B",                # Arrange to apply
              t=0,                     #  Bandpass solutions
              table="ngc5921.bcal");

calS.setsolve(type="T",                # Arrange to solve for T to
              t=100,                   #  get an incremental solution to
              preavg=0,                #  G with a phase-only correction.
              refant=15,
              phaseonly=T,             #  Write the output to a table
      table="ngc5921.tcal");   #  called ngc5921.tcal on disk
```

```
calS.state();                          # Review the setapply/setsolve settings

calS.solve();                          # Solve for T

calS.plotcal(plottype="PHASE",         # Examine solutions
             tablename="ngc5921.tcal");
```

Now apply the G, B, and T solutions to all the data and write a new CORRECTED_DATA column for the target source in the MS:

```
calS.setdata(mode='channel',           # Select channel data for 4
             nchan=63,                 #  continuum-only channels in
             start=1,                  #  field 3 (target field)
             step=1,
             msselect='FIELD_ID==3');

calS.reset();                          # Reset apply/solve state

calS.setapply(type="G",                # Arrange to apply flux-scaled
              t=0,                     #  G solutions
              table="ngc5921.fluxcal");

calS.setapply(type="B",                # Arrange to apply
              t=0,                     #  Bandpass solutions
              table="ngc5921.bcal");

calS.setapply(type="T",                # Arrange to apply T solutions
              t=0,
              table="ngc5921.tcal");

calS.correct();                        # Correct the data
```

CLEAN the self-calibrated image of ngc5921 to see if the image is improved:

```
imgrS.setdata(mode='channel',        # Select channel data for field 3
              nchan=63,
              start=1,
              step=1,
              fieldid=3)

imgrS.setimage(nx=256,               # Select imaging parameters
               ny=256,
               cellx='10arcsec',
               celly='10arcsec',
               stokes='I',
               mode='channel',
               start=1,
               step=1,
               nchan=63,
```

```
                fieldid=3);

imgrS.weight(type='uniform');         # Uniform weighting

or

imgrS.weight(type='briggs',           # Robust weighting
             rmode='norm',
             robust=0.5);

imgrS.clean(algorithm='clark',        # Image and deconvolve
            niter=3000,               #  with Clark CLEAN
            gain=0.1,                 #  Write the cleaned image to the file
            model='ngc5921.mod2',     #  ngc5921.im2 on disk.
            image='ngc5921.im2',
            residual='ngc5921.resid2'
```

Use the viewer to examine the final source image ngc5921.im2. The RMS is ~ 1.4 mJy beam^{-1} (versus 1.6 mJy beam^{-1} before self-calibration). The peak of the continuum source is now 89.2 mJy beam^{-1} (versus 81.8 mJy beam^{-1} before self-calibration). Thus, self-calibration provided a marginal improvement to the image (the dynamic range improved by $\sim 20\%$.

If you have a data set in which there is a strong continuum source or strong line emission (e.g. a maser feature), and self-calibration improved the image significantly, then you can repeat the self-calibration as necessary. Note: Self-calibration can be applied to any solvable calibration component (G, T, and/or B in the present example).

10.13 Image analysis

Extensive display and image analysis tools are available in AIPS++. See the Display and Image analysis chapters of *G*etting Results for a detailed discussion on the options available. A brief orientation on the use of the Viewer is given below.

The Viewer

Start the Viewer by typing at the *Glish* command prompt:

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

Viewer Data Manager and Viewer Display Panel GUIs will appear. Using the Viewer Data Manager, click on an image you would like to see in the Viewer Display and choose the display type (e.g. Raster Image, Contour Map). The image will be loaded into the Viewer window. You can load as many images and display types as you want.

If you are using the GUI interface, you can also click on the "Wrench" icon from an image line in a `tool` window and select `view` to view a single image.

The Viewer is described in detail in the chapter on Display of Data while the 'Adjust' options are described at Adjust

To zoom in on a region, click on the 'zoom' button in the upper left of the Viewer display, use the left mouse button to make a region, double-click in the region to zoom. To un-zoom, hit the 'Unzoom' button at the bottom of the Display.

How the image, axes, and labels are displayed is controlled via the 'Adjustment' panel. Click on the 'Adjust' button to bring up this panel. There are a wide range of options to choose from: e.g. plot the beam, change the color map or stretch of an image, change contour lines, change the line width or color, add or delete labels. Once you have the image the way you like it, you can save the state of the image by clicking on the 'Save' button in the 'Adjustment' panel. The displaydata state can then be restored later by clicking on the 'Restore' button.

To obtain statistics on all or part of an image, click on the `Tools: ImageAnalysis` button at the top of the Viewer Display. An `Image Analysis` GUI will be created. Click on 'Statistics' to show the statistics panel. The 'Full' button will produce statistics on the full image (all planes). The 'Plane' button will produce statistics on only the region being displayed on the Viewer panel. Thus, to estimate the RMS in an image, zoom in on a region that has no emission and click on the Statistics panel 'Plane' button.

Use the 'DisplayData: Register' Panel to choose which images you want to register and display together. Use the 'DisplayData: Remove' Panel to delete images from the stack.

To print the image click on the 'Print' button at the bottom of the Viewer Panel. You will get a 'Canvas Print Manager' GUI to specify the output postscript file name or send the image to a local printer. Postscript files will be written to the directory you started AIPS++ in.

Below are straight-forward examples of a few image analysis tools that are useful for analyzing the example data sets:

continuum polarimetry case

Form a linear polarization intensity image, e.g. $P = \sqrt{Q^2 + U^2}$ and a vector image of the electric vectors of the polarization, e.g. $(Q + iU)$ as described in the Image analysis chapters of *G*etting Results

```
impolC:=imagepol(infile="0957+561.im2");    # Start the image polarization tool
                                            #  for the image 0957+561.im2

linpolintC:=impolC.linpolint();             # Form linear polarization intensity
                                            #  image tool, linpolint

linpolintC.subim(outfile='0957+561.pint'); # Form the subim tool and write
                                            #  the image 0957+561.pint
                                            #  to a file on disk
```

```
subim.done();                                # Close the tool (no longer needed)
linpolintC.done();                           # Close the tool (no longer needed)

impol.complexlinpol(outfile="0957+561.cmplx");
                                             # Create a vector image of the
                                             # electric vectors of the polarization

impol.done();                                # Close the tool (no longer needed)
```

Load the relevant images into the Viewer `tool` using the Tool Manager window (that you created early in the session):

- Load 0957+561.im2 as a raster image
- Load 0957+561.pint as a contour image
- Load 0957+561.cmplx as a vector map

Bring up the 'Image Analysis' tool and determine the RMS of the polarization intensity image using the 'Statistics' panel (for the 0957+561.im2 image, the RMS $\sim 1.5 \times 10^{-4}$). value yet. You will use this to set a mask expression in the 0957+561.cmplx vector image to get rid of polarization vectors that are noise.

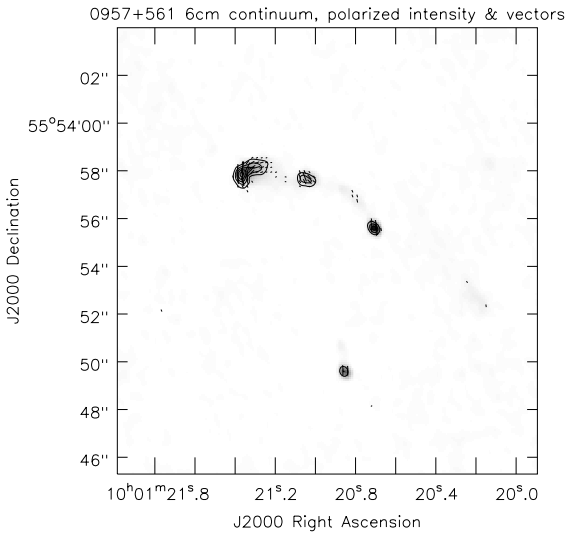
Click on the 'Adjust' button to bring up 3 Adjustment panels, one for each image. Then:

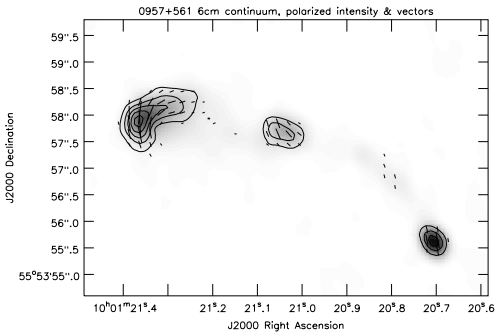
- Adjust the color map for 0957+561.im2 (Color palette of choice, Greyscale 1 for black & white Viewer, Greyscale 2 for output to a psfile).
- Adjust the size of the polarization vectors in 0957+561.cmplx using the 'amplitude scale factor', 'x-increment', and 'y-increment'.
- Set the mask expression in the 0957+561.cmplx image to be: '0957+561.pint' $>$ 0.00045 (3σ in the polarization intensity image). Note, the name 0957+561.pint must be in quotes because it contains a + sign which will be interpreted as an expression.

Now, calibrate the electric-vector positions angles, EVPA $= 0.5 \tan^{-1}(U/Q) =$ $0.5 * (\mathrm{R} - \mathrm{L}$ phase difference). First, it is necessary to specify the true ratio U/Q found for the polarization calibrator, 3C286. During the polarization calibration, we found for 3C286 ($I = 7.462 \;\; Q = -0.621 \;\; U = -0.593 \;\; V = 0$), thus, the measured R–L phase difference is: $\tan^{-1}(-0.593/-0.621) = 0.762338001$ radians or $\tan^{-1}(-0.593/-0.621) * 180/\pi = 43.67875°$. Since U and Q are both negative, this is in the 3^{rd} quadrant. Thus, the angle is really $(-180 + 43.68) = -136.32°$.

From the VLA calibration web page (http://www.aoc.nrao.edu/~gtaylor/calman/polcal.html) we find the true R-L phase difference for 3c286 is 66°. Thus, our phases should be rotated by $66 - 136.32 = 70.32°$ or $EVPA = 0.5 * 70.32°$. In the 'Adjust' panel for the 0957+561.cmplx image, enter `Extra rotation = 70.32/2`. To plot the Magnetic field vectors, enter `Extra rotation =` $70.32/2 + 90$.

Create postscript files of the resulting images:





spectral-line case

Now, use the Image `tool` to subtract the continuum emission in the image plane using the line-free channels (5-8) and (55-58).

First, the image coordinate system is stored so that regions can be made in world coordinates. Then make two regions with the line-free emission and union them together. Note, a union of regions is a compound region that has to be made from two *world* regions, not pixel regions, so it must be constructed from regions made with the `wrange` function from the default regionmanager (drm). Once the line-free channels are unioned together, an integrated continuum image is made and subtracted from the line-cube.

```
imS:=image('ngc5921.im2');          # Start Image tool

cs:=imS.coordsys();                 # Store image coordinate system
                                    #  in the variable 'cs'

range1:=dq.quantity([5,8],'pix');
                                    # Create variable 'range1'

cont1:=drm.wrange(pixelaxis=4,      # Make first region for lower
                  range=range1,     #  continuum channels,
                  csys=cs);         #  Pixelaxis 4 = frequency axis

range2:=dq.quantity([55,58],'pix');
                                    # Create variable 'range2'

cont2:=drm.wrange(pixelaxis=4,      # Make second region for upper
                  range=range2,     #  continuum channels,
                  csys=cs);
```

```
cont3:=drm.union(cont1,cont2);    # Union the two chunks of data together

imS.moments(outfile='ngc5921.cont',
            moments=-1,           # Create an integrated
            axis=4,               #  continuum image, ngc5921.cont,
            region=cont3);        #  and write the file to disk.

nocont:=imagecalc(outfile='ngc5921.nocont',
                  pixels='ngc5921.im2 - ngc5921.cont');
                                  # Subtract the continuum from the
                                  #  line cube and write the line-only
                                  #  cube, ngc5921.nocont, to disk.

imS.done();                       # close the tool, no longer needed
cs.done();                        # close the tool, no longer needed
nocont.done();                    # close the tool, no longer needed
```

View the final image in the Viewer. The continuum sources should now be gone and you can see the rotation curve of the HI in the galaxy as you move through the line cube.

Now, take a spectrum through the cube. There are two ways to pick positions for taking spectra: 1) view the continuum image and take spectra at the locations of the continuum sources to make sure that continuum subtraction has gone OK; 2) view the 0^{th} moment image (generated by the imageprofilefitter `tool` automatically) and use this to select spectra in regions with line emission. The latter is the default behavior. However spectral line cubes can have noisy beginning and end channels, so it is often better to view your own 0^{th} moment image when selecting positions for spectra. Both of these ways are shown below:

Use the continuum image to check that the continuum subtraction has gone well.

```
include 'imageprofilefitter.g';   # Create image profile fitter tool

pf:=imageprofilefitter(infile='ngc5921.nocont',
                       infile2='ngc5921.cont');
                                  # Get a spectrum from ngc5921.nocont
                                  #  But use the image of ngc5921.cont
                                  #  To see where to place the cursor
```

You will get a Viewer window labeled 'Image Profile Fitter' which shows the line+continuum image. To get a spectrum:

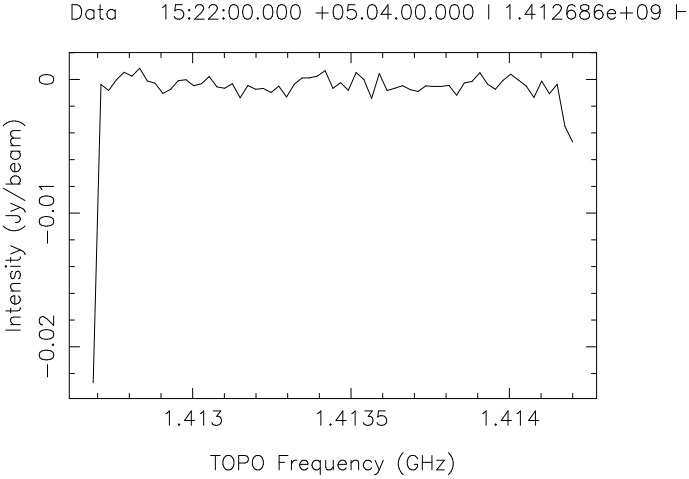
- Use the 'Adjust' button to adjust the stretch so you can see where the line emission is.
- Click on the 'Positioning' button (bulls-eye) to the left of the image.
- Use the left mouse button to click on a position. A line-only spectrum will be displayed in a Profile Fitter GUI. In the Image GUI, adjust the

'Width' parameter which specifies the number of pixels to obtain an average spectrum, and the spectrum position until you get the spectrum you want.

- Alternatively, you can generate a region around the source with the Box-region or polygon-region buttons, double click in the region to get a spectrum.

- In the Profile Fitter window, click on 'Print' to print the spectral line plot to the printer or postscript file.

The spectrum below, taken at the position of the brightest continuum source, shows that the continuum subtraction appears to have done a good job – the spectrum is flat and amplitude is zero.



Create a line-only 0^{th} moment image and use this to select regions regions of line emission to take spectra: Moment analysis for line-emission cubes. Viewing the image we find that there is line emission in channels 7–49.

```
imS:=image('ngc5921.nocont');                  # Start Image tool

cs:=imS.coordsys();                          # Store image coordinate system

line_range:=dq.quantity([7,49],'pix');       # There is line emission in
                                             #  channels 7-49

line_region:=drm.wrange(pixelaxis=4,         # Define the line region in
                        range=line_range,    #  world coordinates
                        csys=cs);

imS.moments(moments=[0, 1, 2] ,
          smoothaxes=[1,2,4] ,               # Pixels to be included in
                                             # moment analysis will be
```

```
                                    # decided by examing the
                                    # flux in a smoothed version
                                    # of the data.
                                    # We will smooth in ra, dec
                                    # and velocity (axis=1,2,4)

        region = line_region,       # Only use channels with
                                    # line emission

        mask='$imS>0',              # Only include pixels with
                                    # positive flux in original map

        excludepix=[0.003],         # Only sum flux from pixels
                                    # in smoothed map with flux
                                    # greater than abs(0.003 Jy)

        smoothtypes="gauss gauss hann" ,# Gaussian smoothing in ra
                                    # and dec; Hanning smoothing
        smoothwidths=[5,5,3] ,      # Width of smoothing kernals

        doppler="OPTICAL" ,         # optical velocity definition

        outfile='ngc5921.nocont2');
```

The moments command will create 3 images:

```
ngc5921.nocont2.integrated               # Moment 0 image
ngc5921.nocont2.weighted_coord           # Moment 1 image
ngc5921.nocont2.weighted_dispersion_coord # Moment 2 image
```

Now take a spectra using the clipped, integrated image to isolate the area of line-emission:

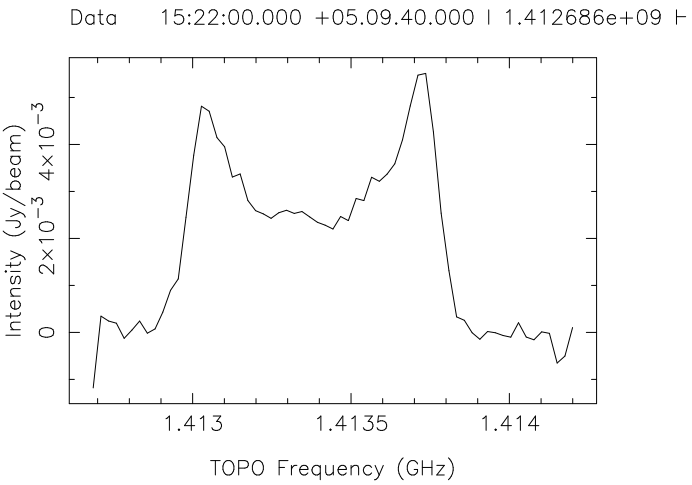
```
include 'imageprofilefitter.g';
pf2:=imageprofilefitter(infile='ngc5921.nocont',
                        infile2='ngc5921.nocont2.integrated');
```

The spectrum below, illustrating a classic double-peaked line profile showing the galactic rotation, a was made with the 'Polygon-Region' option in the Viewer window to circle the emission in NGC 5921, then double clicking on the region to create the spectrum.

The `Image tool` can be used to, e.g. create moment maps, regrid an image, create subimages, obtain statistics, Fourier transform an image.

10.14 Cleanup

After completing work with AIPS++ tools, it is desirable to close them out by issuing the `done` function for each, as follows:



continuum polarimetry case

```
calC.done();                    # Finish imager tool
imgrC.done();                   # Finish calibrater tool
imC.done();                     # Finish image tool
```

spectral-line case

```
calS.done();                    # Finish imager tool
imgrS.done();                   # Finish calibrater tool
imS.done();                     # Finish image tool
```

10.15 References

Hamaker, J.P., Bregman, J.D., and Sault, R.J., 1996a, *A&AS*, **117**, 137.
Sault, R.J., Hamaker, J.P., and Bregman, J.D., 1996b, *A&AS*, **117**, 149.

10.16 Appendix 1

Example .aipsrc file

```
#
# Work in the current directory:
#
user.aipsdir:   .
#
# Put scratch files here: can be a list of directories
#
user.directories.work:  /export/home/your_computer/aips++/scratch
#
# Put the aips++ cache files in one place: it shouldn't be
# over an NFS mount or it will slow aips++ down to a crawl!
```

```
#
user.cache:     /export/home/your_computer/aips++/cache
#
# Set these so that bug(), ask(), etc. know who you are
#
userinfo.name:  Your_name
userinfo.email: Your_email@institution.edu
userinfo.org:   Your Organization
#
# Switch on the display of memory used by each server
#
user.display.memory:    True
#
# Uncomment this to send log messages to the deep blue yonder
#
#logger.file:           none
#
# plot axis labels as the default in the viewer
# and set initial character size (normal default size = 1.2):
#
display.axislabels: on
display.axislabels.charsize: 1.2
#
#
system.aipscenter:       NorthAmerica
logger.file:             aips++.log
logger.height:           15
catalog.gui.auto:        T
catalog.confirm:         T
catalog.view.PostScript: ghostview
toolmanager.gui.auto:    T
toolmanager.refresh:     10s
```

10.17 Appendix 2

Detailed summary of the continuum polarimetry dataset (created with the command `msetC.summary(verbose=T)`):

```
           MeasurementSet Name:  ap366.ms      MS Version 2

   Observer: AP366      Project:
Observation: VLA
Data records: 93239       Total integration time = 21935 seconds
   Observed from   22-May-1998/22:40:05   to   23-May-1998/04:45:40

   ObservationID = 1         ArrayID = 1
   Date        Timerange                 Field          DataDescIds
   22-May-1998/22:40:05.0 - 22:41:10.0   0917+624       [1]
               23:05:30.0 - 23:14:00.0   0957+561       [1]
```

```
              23:26:55.0 - 23:47:30.0   0917+624       [1]
   23-May-1998/00:10:50.0 - 00:18:20.0   0957+561       [1]
              00:30:15.0 - 00:31:20.0   0917+624       [1]
              00:53:15.0 - 04:45:40.0   1328+307       [1]
Fields: 3
   ID   Name      Right Ascension  Declination   Epoch
   9    0917+624  09:21:36.23      +62.15.52.18  J2000
   10   0957+561  10:01:20.70      +55.53.52.65  J2000
   11   1328+307  13:31:08.28      +30.30.32.94  J2000
Data descriptions: 1 (1 spectral windows and 1 polarization setups)
   ID    Ref.Freq    #Chans  Resolution  TotalBW     Correlations
   1     4885.1 MHz  1       50000  kHz  50000  kHz  RR  RL  LR  LL
Feeds: 28: printing first row only
   Antenna   Spectral Window     # Receptors    Polarizations
   1         -1                  2              [         R, L]
Antennas: 27:
   ID   Name  Station   Diam.    Long.         Lat.
   1    1     VLA:W16   25.0 m   -107.37.57.4  +33.53.33.0
   2    2     VLA:N16   25.0 m   -107.37.10.9  +33.54.48.0
   3    3     VLA:N48   25.0 m   -107.37.38.1  +33.59.06.2
   4    4     VLA:W8    25.0 m   -107.37.21.6  +33.53.53.0
   5    5     VLA:N56   25.0 m   -107.37.47.9  +34.00.38.4
   6    6     VLA:E16   25.0 m   -107.36.09.8  +33.53.40.0
   7    7     VLA:E24   25.0 m   -107.35.13.4  +33.53.18.1
   8    8     VLA:N8    25.0 m   -107.37.07.5  +33.54.15.8
   9    9     VLA:E8    25.0 m   -107.36.48.9  +33.53.55.1
   10   10    VLA:W40   25.0 m   -107.41.13.5  +33.51.43.1
   11   11    VLA:N24   25.0 m   -107.37.16.1  +33.55.37.7
   13   13    VLA:W56   25.0 m   -107.44.26.7  +33.49.54.6
   14   14    VLA:E64   25.0 m   -107.27.00.1  +33.50.06.7
   15   15    VLA:N64   25.0 m   -107.37.58.7  +34.02.20.5
   16   16    VLA:W32   25.0 m   -107.39.54.8  +33.52.27.2
   17   17    VLA:W64   25.0 m   -107.46.20.1  +33.48.50.9
   18   18    VLA:E48   25.0 m   -107.30.56.1  +33.51.38.4
   19   19    VLA:N40   25.0 m   -107.37.29.5  +33.57.44.4
   20   20    VLA:E40   25.0 m   -107.32.35.4  +33.52.16.9
   21   21    VLA:E56   25.0 m   -107.29.04.1  +33.50.54.9
   22   22    VLA:E72   25.0 m   -107.24.42.3  +33.49.18.0
   23   23    VLA:W24   25.0 m   -107.38.49.0  +33.53.04.0
   24   24    VLA:N32   25.0 m   -107.37.22.0  +33.56.33.6
   25   25    VLA:W72   25.0 m   -107.48.24.0  +33.47.41.2
   26   26    VLA:W48   25.0 m   -107.42.44.3  +33.50.52.1
   27   27    VLA:N72   25.0 m   -107.38.10.5  +34.04.12.2
   28   28    VLA:E32   25.0 m   -107.34.01.5  +33.52.50.3


Tables:
   MAIN                   93239 rows
   ANTENNA                   28 rows
   DATA_DESCRIPTION           1 row
```

```
   DOPPLER             <absent>
   FEED                     28 rows
   FIELD                    11 rows
   FLAG_CMD             <empty>
   FREQ_OFFSET         <absent>
   HISTORY                 108 rows
   OBSERVATION               1 row
   POINTING                168 rows
   POLARIZATION              1 row
   PROCESSOR            <empty>
   SOURCE              <absent> (see FIELD)
   SPECTRAL_WINDOW           1 row
   STATE                <empty>
   SYSCAL              <absent>
   WEATHER             <absent>
```

Detailed summary of the spectral line dataset (created with the command `msetS.summary(verbose=T)`**):**

```
         MeasurementSet Name:  ngc5921.ms      MS Version 2

   Observer: TEST      Project:
Observation: VLA
Data records: 22653       Total integration time = 5280 seconds
   Observed from   13-Apr-1995/09:19:00   to   13-Apr-1995/10:47:00

   ObservationID = 1         ArrayID = 1
   Date        Timerange                 Field          DataDescIds
   13-Apr-1995/09:19:00.0 - 09:24:30.0   1331+30500002   [1]
               09:27:30.0 - 09:29:30.0   1445+09900002   [1]
               09:33:00.0 - 09:48:00.0   N5921          [1]
               09:50:30.0 - 10:23:00.0   1445+09900002   [1]
               10:26:00.0 - 10:43:00.0   N5921          [1]
               10:45:30.0 - 10:47:00.0   1445+09900002   [1]
Fields: 3
   ID   Name      Right Ascension  Declination   Epoch
   1    1331+3050000213:31:08.29      +30.30.32.96  J2000
   2    1445+0990000214:45:16.47      +09.58.36.07  J2000
   3    N5921     15:22:00.00      +05.04.00.00  J2000
Data descriptions: 1 (1 spectral windows and 1 polarization setups)
   ID    Ref.Freq    #Chans  Resolution  TotalBW     Correlations
   1     1413.45MHz  63      24.4141kHz  1550.2 kHz  RR  LL
Feeds: 28: printing first row only
   Antenna   Spectral Window     # Receptors    Polarizations
   1         -1                  2              [         R, L]
Antennas: 27:
   ID   Name  Station   Diam.    Long.         Lat.
   1    1     VLA:N7    25.0 m   -107.37.07.2  +33.54.12.9
   2    2     VLA:W1    25.0 m   -107.37.05.9  +33.54.00.5
   3    3     VLA:W2    25.0 m   -107.37.07.4  +33.54.00.9
```

```
   4    4     VLA:E1    25.0 m   -107.37.05.7  +33.53.59.2
   5    5     VLA:E3    25.0 m   -107.37.02.8  +33.54.00.5
   6    6     VLA:E9    25.0 m   -107.36.45.1  +33.53.53.6
   7    7     VLA:E6    25.0 m   -107.36.55.6  +33.53.57.7
   8    8     VLA:W8    25.0 m   -107.37.21.6  +33.53.53.0
   9    9     VLA:N5    25.0 m   -107.37.06.7  +33.54.08.0
   10   10    VLA:W3    25.0 m   -107.37.08.9  +33.54.00.1
   11   11    VLA:N4    25.0 m   -107.37.06.5  +33.54.06.1
   12   12    VLA:W5    25.0 m   -107.37.13.0  +33.53.57.8
   13   13    VLA:N3    25.0 m   -107.37.06.3  +33.54.04.8
   14   14    VLA:N1    25.0 m   -107.37.06.0  +33.54.01.8
   15   15    VLA:N2    25.0 m   -107.37.06.2  +33.54.03.5
   16   16    VLA:E7    25.0 m   -107.36.52.4  +33.53.56.5
   17   17    VLA:E8    25.0 m   -107.36.48.9  +33.53.55.1
   18   18    VLA:W4    25.0 m   -107.37.10.8  +33.53.59.1
   19   19    VLA:E5    25.0 m   -107.36.58.4  +33.53.58.8
   20   20    VLA:W9    25.0 m   -107.37.25.1  +33.53.51.0
   21   21    VLA:W6    25.0 m   -107.37.15.6  +33.53.56.4
   22   22    VLA:E4    25.0 m   -107.37.00.8  +33.53.59.7
   24   24    VLA:E2    25.0 m   -107.37.04.4  +33.54.01.1
   25   25    VLA:N6    25.0 m   -107.37.06.9  +33.54.10.3
   26   26    VLA:N9    25.0 m   -107.37.07.8  +33.54.19.0
   27   27    VLA:N8    25.0 m   -107.37.07.5  +33.54.15.8
   28   28    VLA:W7    25.0 m   -107.37.18.4  +33.53.54.8


Tables:
   MAIN                   22653 rows
   ANTENNA                   28 rows
   DATA_DESCRIPTION           1 row
   DOPPLER             <absent>
   FEED                      28 rows
   FIELD                      3 rows
   FLAG_CMD            <empty>
   FREQ_OFFSET         <absent>
   HISTORY                  270 rows
   OBSERVATION                1 row
   POINTING                 168 rows
   POLARIZATION               1 row
   PROCESSOR           <empty>
   SOURCE              <absent> (see FIELD)
   SPECTRAL_WINDOW            1 row
   STATE               <empty>
   SYSCAL              <absent>
   WEATHER             <absent>
```